



ORACLE
NETSUITE

SuiteScript 2.0

2021.1

July 28, 2021



Copyright © 2005, 2021, Oracle and/or its affiliates.

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

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

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

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

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

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

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

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

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

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

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

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

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

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

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

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

Sample Code

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

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

No Excessive Use of the Service

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

Beta Features

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

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

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

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any

operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

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

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

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

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

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

The information contained in this document is for informational sharing purposes only and should be considered in your capacity as a customer advisory board member or pursuant to your pre-General Availability trial agreement only. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

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

Send Us Your Feedback

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

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

- Did you find the information you needed? If not, what was missing?

- Did you find any errors?
- Is the information clear?
- Are the examples correct?
- Do you need more examples?
- What did you like most about this document?

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

To report software issues, contact NetSuite Customer Support.

Table of Contents

| | |
|---|-----|
| SuiteScript 2.x API Introduction | 1 |
| SuiteScript 2.x Hello World | 1 |
| SuiteScript 2.x Script Basics | 10 |
| SuiteScript 2.x Anatomy of a Script | 12 |
| SuiteScript 2.x Script Creation Process | 14 |
| SuiteScript 2.x Advantages | 15 |
| SuiteScript 2.x Terminology | 19 |
| SuiteScript 2.x Developer Resources | 20 |
| SuiteScript Reserved Words | 21 |
| SuiteScript Versioning Guidelines | 22 |
| SuiteScript 2.1 | 24 |
| Executing Scripts Using SuiteScript 2.1 | 24 |
| Executing a Single Script Using SuiteScript 2.1 | 25 |
| Enabling SuiteScript 2.1 at the Account Level | 26 |
| SuiteScript 2.1 Language Examples | 27 |
| Differences Between SuiteScript 2.0 and SuiteScript 2.1 | 30 |
| SuiteScript Analytic APIs | 37 |
| Workbook API (Beta) | 39 |
| Enabling the Workbook API (Beta) Feature | 40 |
| Creating a Dataset Using the Workbook API (Beta) Tutorial | 40 |
| Workbook API (Beta) UI Guide | 52 |
| Workbook API (Beta) Limitations | 87 |
| SuiteScript 2.x Script Types | 88 |
| SuiteScript 2.x Bundle Installation Script Type | 90 |
| SuiteScript 2.x Bundle Installation Script Reference | 91 |
| SuiteScript 2.x Bundle Installation Script Entry Points | 94 |
| SuiteScript 2.x Client Script Type | 97 |
| SuiteScript 2.x Client Script Reference | 101 |
| SuiteScript 2.x Client Script Entry Points and API | 105 |
| SuiteScript 2.x Map/Reduce Script Type | 114 |
| SuiteScript 2.x Map/Reduce Script Reference | 123 |
| SuiteScript 2.x Map/Reduce Script Entry Points and API | 155 |
| SuiteScript 2.x Mass Update Script Type | 195 |
| SuiteScript 2.x Mass Update Script Entry Points | 196 |
| SuiteScript 2.x Portlet Script Type | 197 |
| SuiteScript 2.x Portlet Script Reference | 201 |
| SuiteScript 2.x Portlet Script Entry Points and API | 203 |
| SuiteScript 2.x RESTlet Script Type | 213 |
| SuiteScript 2.x Getting Started with RESTlets | 213 |
| SuiteScript 2.x RESTlet Reference | 220 |
| SuiteScript 2.x RESTlet Script and Request Samples | 223 |
| SuiteScript 2.x RESTlet Script Entry Points | 230 |
| SuiteScript 2.x Scheduled Script Type | 232 |
| SuiteScript 2.x Scheduled Script Reference | 235 |
| SuiteScript 2.x Script Entry Points and API | 245 |
| SuiteScript 2.x Suitelet Script Type | 247 |
| SuiteScript 2.x Suitelet Script Reference | 248 |
| SuiteScript 2.x Suitelet Script Entry Points and API | 256 |
| SuiteScript 2.x Suitelet Script Type Code Samples | 257 |
| SuiteScript 2.x User Event Script Type | 285 |
| SuiteScript 2.x User Event Script Reference | 287 |
| SuiteScript 2.x User Event Script Entry Points and API | 298 |
| SuiteScript 2.x Workflow Action Script Type | 303 |

| | |
|---|-----|
| Creating and Using Workflow Action Scripts | 305 |
| SuiteScript 2.x Workflow Action Script Entry Points and API | 306 |
| SuiteScript 2.x SDF Installation Script Type | 306 |
| SuiteScript 2.x SDF Installation Script Entry Points and API | 308 |
| SuiteScript 2.x Record Actions and Macros | 310 |
| Overview of Record Action and Macro APIs | 310 |
| Supported Record Actions | 311 |
| Supported Record Macros | 312 |
| SuiteScript 2.x JSDoc Validation | 313 |
| Controlling Access to Scripts and Custom Modules | 315 |
| SuiteScript 2.x Entry Point Script Creation and Deployment | 318 |
| Record-Level and Form-Level Scripts | 318 |
| SuiteScript 2.x Entry Point Script Validation | 319 |
| Entry Point Script Validation Guidelines | 319 |
| Entry Point Script Validation Examples | 322 |
| Entry Point Script Validation Error Reference | 324 |
| SuiteScript 2.x Record-Level Scripts | 328 |
| Script Record Creation | 328 |
| Script Deployment | 332 |
| Viewing System Notes | 337 |
| SuiteScript 2.x Form-Level Scripts | 338 |
| Attaching a Client Script to a Form | 338 |
| Configuring a Custom Action | 339 |
| SuiteScript 2.x Custom Modules | 341 |
| SuiteScript 2.x Anatomy of a Custom Module Script | 342 |
| SuiteScript 2.x Custom Module Tutorial | 343 |
| Module Dependency Paths | 352 |
| Naming a Custom Module | 353 |
| Custom Module Examples | 354 |
| Troubleshooting Errors | 358 |
| Custom Modules Frequently Asked Questions | 358 |
| SuiteScript 2.x Scripting Records and Subrecords | 360 |
| SuiteScript 2.x Scripting Records | 360 |
| SuiteScript 2.x Standard and Dynamic Modes | 360 |
| SuiteScript 2.x Record Modules | 361 |
| SuiteScript 2.x Scripting Subrecords | 362 |
| About Subrecords | 362 |
| Subrecord Scripting in SuiteScript 2.x Compared With 1.0 | 376 |
| Scripting Subrecords that Occur on Sublist Lines | 377 |
| Scripting Subrecords that Occur in Body Fields | 396 |
| SuiteScript 2.x Custom Pages | 413 |
| SuiteScript 2.x Custom Forms | 413 |
| Sample Custom Form Script | 418 |
| SuiteScript 2.x Custom List Pages | 430 |
| Sample Custom List Page Script | 432 |
| SuiteScript 2.x Working with UI Objects | 435 |
| Custom UI development | 436 |
| UI Component Overview | 437 |
| SuiteScript 2.x UI Modules | 438 |
| Using HTML | 439 |
| Creating Custom Assistants | 443 |
| Transitioning from SuiteScript 1.0 to SuiteScript 2.x | 447 |
| Overview of the Differences Between SuiteScript 1.0 and SuiteScript 2.x | 448 |
| Differences Between SuiteScript 1.0 and SuiteScript 2.x Script Types | 451 |
| Differences in Similar SuiteScript 1.0 and SuiteScript 2.x Capabilities | 455 |

| | |
|--|-----|
| SuiteScript 1.0 APIs Not Directly Mapped to a SuiteScript 2.x Module | 457 |
| Converting a SuiteScript 1.0 Script to a SuiteScript 2.x Script | 458 |
| Sample SuiteScript 1.0 to SuiteScript 2.x Conversions | 459 |
| Confirm bundle installation, enable features, create account | 460 |
| Set default fields values on a record | 461 |
| Disable fields on a record | 462 |
| Display user profile information | 463 |
| Update fields on sales order and estimate records | 463 |
| Build a simple portlet that posts data to a servlet | 464 |
| GET, POST, PUT, and DELETE methods | 465 |
| Fulfill and bill sales orders each day | 467 |
| Create a simple form | 468 |
| Implement end of month sales order promotions | 469 |
| Set the sales rep on the record in a workflow | 471 |
| Create an item receipt from a purchase order | 471 |
| Create and run a joined search | 472 |
| Create CSV imports | 473 |

SuiteScript 2.x API Introduction

You can use NetSuite's scripting language, SuiteScript, to extend and customize, search, and process your NetSuite data. SuiteScript enables full-featured application-level scripting capabilities that support sophisticated procedural logic for both client and server scripts, as well as robust debugging.

The latest major version, SuiteScript 2.0, extends the capabilities provided by the initial SuiteScript version with an API architecture familiar to JavaScript developers. SuiteScript 2.x refers to the latest minor version of SuiteScript. As of this release, the most recent minor version of SuiteScript is 2.1. For more information on SuiteScript 2.1 and SuiteScript versioning, see [SuiteScript 2.1, Executing Scripts Using SuiteScript 2.1](#), and [SuiteScript Versioning Guidelines](#).

SuiteScript 2.x's modularity supports encapsulation, provides intuitive code organization, and ensures there are no global variable or method naming conflicts. Automatic dependency management enables you to concentrate on logic instead of dependencies and load order. SuiteScript 2.x is designed to support all standard JavaScript. The supplied SuiteScript 2.x APIs give you programmatic access to NetSuite functionality. For generic logic, you can use custom modules to load your preferred third party JavaScript APIs.

SuiteScript 2.x API enhances the APIs supported by the previous SuiteScript version with APIs for SFTP file transfer, data caching, search pagination, flat file streaming, and enhanced encryption, decryption, and hashing. SuiteScript 2.x also provides asynchronous processing through promises, map/reduce scripts that provide a structured framework for server scripts processing a large number of records, and support for all HTTP content types.

The following help topics show how to write scripts using the SuiteScript 2.x API:

- [SuiteScript 2.x Hello World](#)
- [SuiteScript 2.x Script Basics](#)
- [SuiteScript 2.x Anatomy of a Script](#)
- [SuiteScript 2.x Script Creation Process](#)
- [SuiteScript 2.x Advantages](#)
- [SuiteScript 2.x Terminology](#)
- [SuiteScript 2.x Developer Resources](#)
- [SuiteScript Reserved Words](#)

SuiteScript 2.x Hello World

SuiteScript 2.x is a JavaScript API that offers a broad range of options for enhancing and extending NetSuite. You can use SuiteScript to customize the behavior of a page, create custom workflows, schedule tasks, and much more.

To help you understand how SuiteScript 2.x works, this topic walks you through the implementation of a basic customization. After you complete the steps in this topic, the system displays a "Hello, World!" message whenever you load a NetSuite task record.

This topic contains the following sections:

- [Key Concepts](#)
- [Step 1 Enable the Feature](#)
- [Step 2 Create the Script File](#)
- [Step 3 Upload the Script File to NetSuite](#)

- [Step 4 Create a Script Record and Script Deployment Record](#)
- [Step 5 Test the Script](#)
- [Next Steps](#)

Key Concepts

Before proceeding with the steps in this topic, it may be useful to consider a few concepts and terms that are central to SuiteScript 2.x development. Some of these terms are referenced in the steps that appear later in this topic. You can read about these concepts now, or you can go straight to [Step 1 Enable the Feature](#).

SuiteScript Versions

The sample script in this topic uses SuiteScript 2.0. A newer version, SuiteScript 2.1, is also available and supports new language features that are included in the ES2019 specification. You can write your scripts using either SuiteScript 2.0 or SuiteScript 2.1.

- For help with writing scripts in SuiteScript 2.x, see [SuiteScript 2.x Entry Point Script Creation and Deployment](#).
- For more information about SuiteScript versions and SuiteScript 2.1, see [SuiteScript Versioning Guidelines](#) and [SuiteScript 2.1](#).

SuiteScript Reserved Words

Like most programming languages, SuiteScript includes a list of reserved words that you cannot use as variable names or function names in your scripts. For example, the `var` reserved word indicates a variable declaration, and the `try` reserved word indicates the start of a try-catch block. For more information, see [SuiteScript Reserved Words](#).

SuiteScript 2.x Script Types and Entry Points

If you have written scripts before, you are probably used to thinking through basic questions prior to coding. These questions might include: What does the script need to accomplish? Should the script run in the browser or on the server? What specific event should trigger the script?

With SuiteScript 2.x, part of this process involves deciding which of the system's predefined **script types** you should use. Each script type is designed for a specific type of situation and specific types of triggering events. The following are some of the available [SuiteScript 2.x Script Types](#):

- The [SuiteScript 2.x Client Script Type](#) is for scripts that should run in the browser.
- The [SuiteScript 2.x Scheduled Script Type](#) is for server scripts that should run at a specific time or on a recurring schedule.
- The [SuiteScript 2.x RESTlet Script Type](#) is for server scripts that should execute when called over HTTP by an application external to NetSuite.

Each script type includes one or more **entry points** that are exclusive to that type. The entry point represents the juncture at which the system grants control of the NetSuite application to the script. When you include an entry point in your script, you tell the system that it should do something when that entry point is invoked. Specifically, you tell the system that it should execute a function defined within the script. This function is called an **entry point function**.

With many script types, the available entry points are analogous to types of events — various things that can happen — to trigger your script. For example, the client script type's entry points represent events that can occur during a browser session. These entry points include `fieldChanged`, which represents a change to the value of a field, and `pageInit`, which represents the loading of a page. In comparison, the

scheduled script type has only one entry point, called [execute](#). It represents the point at which a schedule executes the script or a user acts manually to execute the script.

In the example used in this topic, you want a dialog alert to appear when a user working in a browser loads the NetSuite task record page. For that reason, this example uses the client script type and the `pageInit` entry point.

SuiteScript 2.x Modules

SuiteScript 2.x has a [Modular Architecture](#). As one indication of this, all SuiteScript 2.x APIs are organized into a series of standard modules. Each module's name reflects its functionality. For example, the [N/record Module](#) lets you interact with NetSuite records. The [N/https Module](#) lets you make https requests to external web services.

Most modules must be explicitly loaded by a script before the script can access that module's APIs. At a very high level, loading a JavaScript module is similar to importing a library in Java. It is a way of providing access to logic that is defined elsewhere. In an entry point script, you load a module by using the [define Object](#). You list the modules that you want to load as an argument of the `define` function.

In contrast, some APIs are globally available. When an object, method, or function is globally available, it can be used even when you do not explicitly load the module to which it belongs. Globally available APIs are listed in [SuiteScript 2.x Global Objects](#).

The example in this topic uses both approaches: It uses globally available APIs. It also uses a method that becomes available only after the appropriate module is loaded.

Entry Point Scripts Versus Custom Module Scripts

The example script in this topic is relatively simple. All of its logic is contained within one script file. However, you might want to create scripts that rely on logic defined in other script files. In SuiteScript 2.x, these supporting script files are known as **custom module scripts**.

In contrast, the primary script file — the one that identifies the script type, entry point, and entry point function — is known as an **entry point script**. The system imposes formatting requirements on entry point scripts that are different from those of custom module scripts. The remaining steps in this topic highlight some of the requirements that exist for entry point scripts.

Custom module scripts are not covered in this topic. For information about custom module scripts, see [SuiteScript 2.x Custom Modules](#).

Step 1 Enable the Feature

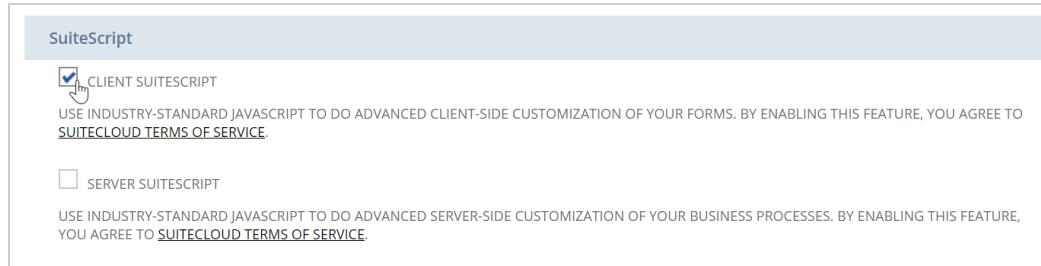
Before you can complete the rest of the steps in this topic, the Client SuiteScript feature must be enabled in your NetSuite account.

If you are not sure whether the feature is enabled, you can check by navigating to Customization > Scripting > Scripts. If you can access that path, the feature is enabled. If the option does not appear, the reason could be that you do not have permission to access it, or that the menu path has been customized. If you are not sure, check with your account administrator.

The feature can be enabled by an account administrator or by a user with the Enable Features permission. To enable the feature, an authorized user should complete the following steps.

To enable the Client SuiteScript feature:

1. Select Setup > Company > Enable Features.
2. Click the **SuiteCloud** subtab.
3. Locate the **Client SuiteScript** option. If it the box is already checked, skip ahead to [Step 2 Create the Script File](#). If it is not, check the box.



The system displays a window listing the terms of service.

4. If you agree to the terms, scroll to the bottom of the window and click **I Agree**.
5. Server SuiteScript is not required for the steps described in this topic. However, if you plan to do further SuiteScript development, consider checking the **Server SuiteScript** box. If you check the box, the system displays another window listing the terms of service. Click **I Agree**.
6. Click **Save**.

Step 2 Create the Script File

Before proceeding, you must create a script file called `helloWorld.js`. To create this file, you can use either of the following approaches:

- If you want to copy and paste the completed script directly from this document, skip ahead to [Copy the Full Script](#).
- If you want to read about how to construct the script yourself, refer to [Create the Script Step by Step](#).

Create the Script Step by Step

The following steps walk you through the process of creating the `helloWorld.js` sample script. Many of the steps in this process are required for any entry point script.

To create the script file:

1. Open a new file in your text editor of choice.
2. Add two JSDoc tags `@NApiVersion`, and `@NScriptType`. After each tag, add the appropriate value, as shown in the following snippet.

```

1  /**
2  * @NApiVersion 2.0
3  * @NScriptType ClientScript
4  */
5
6 ...

```

These tags reflect the following:

- The version of SuiteScript you are using.
- The script type you are using.

Every SuiteScript 2.x entry point script must include these tags. For each tag, you can include only one value. For more information and a list of valid values, see [SuiteScript 2.x JSDoc Tags](#).

3. Add the `define` Object. Every entry point script must use this function.

Use `['N/ui/dialog']` as the `define` function's first argument. This first argument is an array of string values representing the modules that the function should load.

```

1 ...
2 define(['N/ui/dialog'],
3
4     // In Step 4, you put additional code here.
5
6 );

```

The [N/ui/dialog Module](#) includes methods that display various types of dialogs. You load this module so that the script can use one of these methods.

4. Declare a callback function. The callback function is the define function's second argument. Give this function one argument called **dialog**.

```

1 ...
2     function(dialog) {
3
4         // In Step 5, you put additional code here.
5
6     }
7 ...

```

If you are not familiar with callback functions, just remember that this function will contain all of the script's other logic. Additionally, remember that the number of arguments used by the callback function must equal the number of modules loaded by the define function. Each argument is an object that lets you access the module it represents. As a best practice, give each argument a name similar to that of the corresponding module.

In this example, the define function loads only one module, so the callback function has only one argument.

5. Within the callback function, declare a function called **helloWorld**.

```

1 ...
2     function helloWorld() {
3
4         // In steps 6-10, you put additional code here.
5
6     }
7 ...

```

Later, you designate the helloWorld function as the script's entry point function. Every entry point script must have an entry point function.

A function is considered an entry point function only if it is linked to an entry point. You create this link in Step 11.

6. Within the entry point function, create an object named **options**.

Many SuiteScript 2.x methods either require or can accept a plain JavaScript object as their argument. The method that you use to create the "Hello, World!" dialog falls into this category. This method accepts an object that has two parameters: title and message.

```

1 ...
2     var options = {
3         title: 'Hello!',
4         message: 'Hello, World!'
5     };
6 ...

```

7. Add a [try/catch statement](#). This statement is not required. However, this approach lets your script handle errors gracefully. That is, if an error occurs and is handled by a try/catch statement, your script — and any others that are deployed on the page — can continue executing. Using a try/catch statement can also help you understand why problems occur. Note that the try/catch keywords are part of JavaScript and not specific to SuiteScript 2.x.

A basic try/catch statement consists of two parts. The try block holds code that you want to execute. The catch block holds logic that should execute if JavaScript errors are encountered during the try block.

Add the try/catch statement after the object that you created in Step 6.

```

1  ...
2   try {
3     // In steps 8 and 9, you put additional code here.
4   } catch (e) {
5     // In Step 10, you put additional code here.
6   }
7 ...
8
9
10
11 ...

```

- For the first action of the try block, invoke the [N/ui/dialog Module](#)'s alert() method. This method creates a dialog with a title, a message, and an OK button.

You invoke this method by using the object which, in Step 4, you named **dialog**. You also use the method's name, **alert**. To define the dialog's title and message, pass in the object titled **options**.

```

1  ...
2   dialog.alert(options);
3 ...

```

For more details about this method, see the [dialog.alert\(options\)](#) reference page. Note that the title of the [dialog.alert\(options\)](#) reference page matches the code you have added to your script. However, be aware that the reference pages for standard module methods may not always reflect your naming conventions. For example, if you had specified an argument named message when you declared your callback function, you would invoke the alert method using the expression message.alert(options).

Similarly, note that within the documentation, each method's argument is typically referenced as an object titled options. However, in your script, you can give the object any name. You can also create the object directly, as part of the process of invoking the method.

- Add logic to create a log entry when the dialog displays successfully. To do this, use the globally available [log.debug\(options\)](#) method.

The debug() method is called on the log object. Unlike the dialog object, which you had to take steps to access, the log object is made available to every script. For that reason, you don't give this object a name, so you can always write **log.debug** to call this method.

This method takes an object that has two properties: a title and a detailed message. This time, create the object directly. Contrast this style with the way you created an object in Step 5, then passed that object by name to the alert() method.

```

1  ...
2   log.debug ({
3     title: 'Success',
4     details: 'Alert displayed successfully'
5   });
6 ...

```

The log entry is created in the UI when a user triggers the dialog alert. An explanation of how to find the log is covered in [Step 5 Test the Script](#).

- In the catch block, add logic to create a log entry if an error is thrown. Use the globally available [log.error\(options\)](#) method. The log.error() method is similar to the log.debug() method. The only difference is that, with log.error(), the log entry is classified as an entry of type error, instead of debug.

```

1 ...
2     log.error ({
3         title: e.name,
4         details: e.message
5     });
6 ...

```

11. Immediately after the entry point function, add a return statement. In every SuiteScript 2.x entry point script, the return statement must include at least one line that has two components:
- An entry point, which in this case is [pageInit](#).
 - A function, which in this case is [helloWorld](#).

```

1 ...
2     return {
3         pageInit: helloWorld
4     };
5 ...

```

Because of this reference, [helloWorld](#) is considered an entry point function.

Although this script uses only one entry point, a return statement can include multiple entry points. Using multiple entry points is permitted as long as they all belong to the script type identified by the `@NScriptType` tag at the top of the file. For example, in addition to [pageInit](#), the client script type also includes [saveRecord](#). So, if you wanted the script to take one action when the page loads and another action when the user clicks Save, you could use both entry points. For an example of a script that uses multiple entry points, see [SuiteScript Client Script Sample](#).

12. Save the file, naming it [helloWorld.js](#).

Copy the Full Script

This section shows the full sample script. If you haven't already created the script file by using the steps described in [Create the Script Step by Step](#), copy and paste the following code into a text file. Save the file and name it [helloWorld.js](#).

```

1 /**
2 * @NApiVersion 2.0
3 * @NScriptType ClientScript
4 */
5
6
7 define(['N/ui/dialog'],
8
9     function(dialog) {
10
11     function helloWorld() {
12
13         var options = {
14             title: 'Hello!',
15             message: 'Hello, World!'
16         };
17
18         try {
19
20             dialog.alert(options);
21
22             log.debug ({
23                 title: 'Success',
24                 details: 'Alert displayed successfully'
25             });
26
27         } catch (e) {
28
29             log.error ({

```

```

30         title: e.name,
31         details: e.message
32     });
33 }
34
35 return {
36     pageInit: helloWorld
37 };
38 });
39 });

```



Note: If the script sample splits across multiple pages, make sure that you copy all of the code.

Step 3 Upload the Script File to NetSuite

After you have created your entry point script file, upload it to your NetSuite File Cabinet.

To upload the script file:

1. In the NetSuite UI, go to Documents > File > SuiteScripts.
2. In the left pane, select the SuiteScripts folder and click **Add File**.
3. Follow the prompts to locate the helloWorld.js file in your local environment and upload it.

Be aware that even after you upload the file, you can edit it from within the File Cabinet, if needed. For details, see the help topic [Editing Files in the File Cabinet](#).

Step 4 Create a Script Record and Script Deployment Record

In general, before an entry point script can execute in your account, you must create a script record that represents the entry point script file. You must also create a script deployment record.

The script deployment record contains part of the logic that determines when the script executes. Some of that logic is contained within the script, by the entry point that the script uses. For example, by using the [pageInit](#) entry point, you tell the system that the script should execute when a page loads. However, you must also identify the specific page which, when loaded, causes the script to execute. Put another way, you must tell the system which record type this script should execute on. To do that, you use the script deployment record.

These records can be created programmatically. You can also create them in the UI, as described in the following procedure.

To create the script record and script deployment record:

1. Go to Customization > Scripting > Scripts > New.
2. In the **Script File** dropdown list, select helloWorld.js.

If you had not yet uploaded the file, as described in Step Three, you could upload the file from this page. With your cursor, point to the right of the dropdown list to display a plus icon. Clicking this icon opens a window that lets you upload a file.

3. After you have populated the dropdown list, click the **Create Script Record** button.

In response, the system displays a new script record, with the helloWorld.js file listed on the **Scripts** subtab.

4. Fill out the required body fields as follows:

- In the **Name** field, enter **Hello World Client Script**.
- In the **ID** field, enter **_cs_helloworld**.

5. Click the **Deployments** subtab. You use this subtab to create the deployment record.
6. Add a line to the sublist, as follows:
 - Set the **Applies to** dropdown list to **Task**. If you want the dialog to appear when a different type of record loads, select another record type from the list.
 - In the **ID** field, enter `_cs_helloworld`.

Leave the other fields set to their default values. Note that the **Status** field is set to **Testing**. This value means that the script does not deploy for other users. If you want to change the deployment later and make this customization available to all users, edit the deployment and set the status to **Released**.
7. Click **Save**.

The system creates the script and script deployment records.

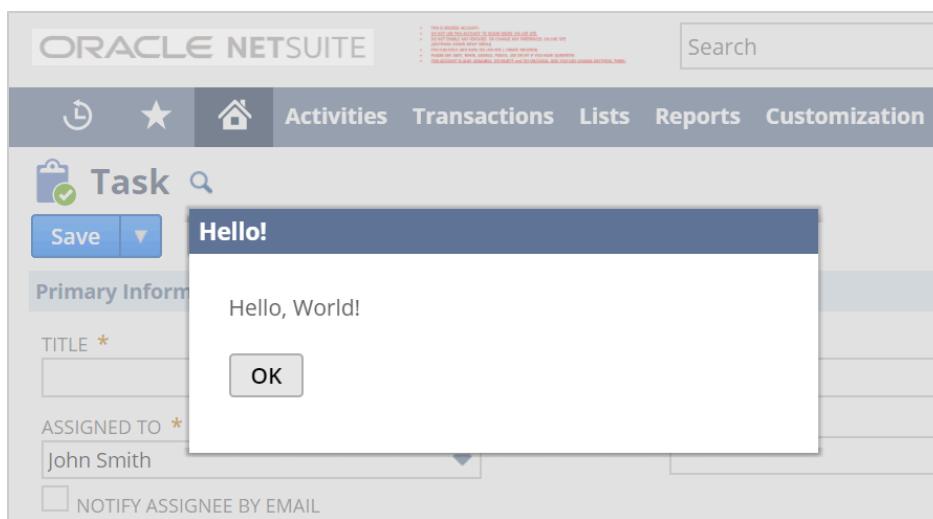
Step 5 Test the Script

Now that the script is deployed, you should verify that it executes as expected.

To test the script:

1. Verify that the dialog alert appears when it should:
 - a. Open a task record by going to Activities > Scheduling > Tasks > New.

If the script is working properly, the system displays a dialog alert.



- b. Confirm and close the dialog by clicking **OK**.
2. Verify that the expected log entry has been saved to the script deployment record:
 - a. Go to Customization > Scripting > Script Deployments.
 - b. Locate your deployment, and click **View**.

| Script Deployments | | | | | | | Scripts |
|---|---|----------------------------|---------------------------|---------|-------------|------------------|----------|
| New Deployment FILTERS | | | | | | | |
| <input type="checkbox"/> SHOW UNDEPLOYED | | | | | | | TOTAL: 1 |
| INTERNAL ID | EDIT VIEW | ID | SCRIPT | STATUS | RECORD TYPE | LAST MODIFIED | |
| 110 | Edit View | customdeploy_cs_helloworld | Hello World Client Script | Testing | Task | 8/3/2017 5:42 pm | |

- c. Click the **Execution Log** subtab.

The subtab should show an entry similar to the following.

A screenshot of the SuiteScript 2.x interface showing the 'Execution Log' subtab. The interface has a header with tabs: Audience, Scripts, Execution Log (which is selected and highlighted in blue), and System Notes. Below the header are two dropdown menus: 'TYPE' set to '- All -' and 'VIEW' set to 'Default Script Notes View'. There are three buttons at the top: 'Customize View', 'Remove All', and 'Refresh'. The main area is a table with the following columns: VIEW, TYPE, TITLE, DATE, TIME, USER, DETAILS, and REMOVE. A single row is present in the table:

| VIEW | TYPE | TITLE | DATE | TIME | USER | DETAILS | REMOVE |
|------|-------|---------|----------|----------|------------|------------------------------|--------|
| View | Debug | Success | 8/4/2017 | 11:06 am | John Smith | Alert displayed successfully | Remove |

The same entry also appears on the Execution Log of the script record.

If an error had been encountered, the error log entry you created would be displayed instead of the debug entry.

Next Steps

Now that you have deployed your first script, consider browsing other topics in [SuiteScript 2.x API Introduction](#). Or, if you want to experiment with other script samples, try the following:

- Another commonly used script type is the user event type. This type is designed for server scripts that should execute when users take certain actions with records. To try a simple user event script, see [SuiteScript 2.x User Event Script Tutorial](#).
- For a slightly more complex user event script, see [SuiteScript 2.x User Event Script Sample](#).
- For an example of a custom module script and a user event script that calls a custom module function, see [SuiteScript 2.x Custom Module Tutorial](#).

SuiteScript 2.x Script Basics

Certain components are common to all SuiteScript 2.x scripts. This topic describes some of these components. It includes the following sections:

- [Modular Architecture](#)
- [Objects as Arguments](#)

Note: If you have not already done so, review [SuiteScript 2.x Hello World](#), paying particular attention to the [Key Concepts](#) section.

Modular Architecture

SuiteScript 2.x is modular. As one indication of this, all of its APIs are organized into a series of standard modules. After your script loads a standard module, it can use any of that module's APIs.

But modularity has other implications, too. For example, each SuiteScript 2.x entry point and custom module script must be structured so that it creates a module. If you have used SuiteScript 2.x before, you may not have realized that your scripts were creating modules. However, every SuiteScript 2.x script, with the exception of scripts intended solely for on-demand debugging in the NetSuite Debugger or a browser console, must create a module. This guideline affects both entry point scripts and custom module scripts. For example, an entry point script must create a module that can be used by the NetSuite application. A custom module script must create a module that can be loaded and used by other scripts. In both cases, the module must be designed to return an object, which is usually a function.

You create a module by using the globally available [define Object](#). If you have read [SuiteScript 2.x Hello World](#), then you may recall that you can use the define function to load modules for use within your script. The ability to load modules is a significant part of the define function. However, an equally important aspect of the define function is its ability to create modules.

The define function can take between one and three arguments. However, a common technique, and the one used in most samples throughout this help center, is to provide two arguments, as follows:

- The first argument is a list of any dependencies, such as standard [SuiteScript 2.x Modules](#). These modules are loaded when the script executes.
- The second argument is a callback function. The callback function must include a return statement that identifies at least one standard or custom entry point. The entry point must be associated with an object that is returned when the entry point is invoked. In most use cases, this object is a function.

If you want to understand more about the modular architecture of SuiteScript 2.x, consider reviewing the [Asynchronous Module Definition \(AMD\) Specification](#), which SuiteScript 2.x uses. However, you do not have to be deeply familiar with AMD before you can use SuiteScript 2.x.



Note: Some samples that appear in this help center use the [require Function](#) instead of the define function. Samples that use require are intended for on-demand debugging, either in the NetSuite Debugger or in a browser console. If a script is to be used for anything other than on-demand debugging, it must use the [define Object](#).

Objects as Arguments

In SuiteScript 2.x, the arguments passed to methods are typically objects. For two examples of how this characteristic is significant, see the following sections:

- [Objects as Arguments for Standard Methods](#)
- [Context Objects Passed to Standard and Custom Entry Points](#)

Objects as Arguments for Standard Methods

Many SuiteScript 2.x methods require you to provide one or more pieces of information. But regardless of how many discrete pieces of information are needed, generally all methods have one thing in common: they expect all input to be provided in the form of a single object. This object can have one property or it can have many.

For example, when you set a value on a record, you use the [Record.setValue\(options\)](#) method. With this method, you must submit two pieces of information: the ID of the field, and the value that you want to set for that field. To pass this information to the method, you must create an object with a **fieldId** property and a **value** property. Then you pass this object to the method. For example:

```

1 var myObject = {
2   fieldId: 'Hello!',
3   value: 'Hello, World!'
4 };
5
6 myRecord.setValue(myObject);

```

Another way of passing in an object is to define it within the method call. For example:

```

1 myRecord.setValue({
2   fieldId: 'Hello!',
3   value: 'Hello, World!'

```

```
4 | });

```

In the documentation for standard SuiteScript 2.0 methods, each method's argument is typically referenced as an object titled **options**. However, in your script, you can give the object any name. For an example, see [SuiteScript 2.x User Event Script Tutorial](#).

Context Objects Passed to Standard and Custom Entry Points

Every SuiteScript 2.x script — every entry point script and every custom module script — must use an entry point. Further, each entry point must correspond with an object defined within the script. This object is usually a function. When the object is a function, it is referred to as an entry point function.

In most cases, an entry point function has access to an object provided by the system. Typically, this object can let your script access data and take actions specific to the context in which the script is executing. For that reason, this object is often referred to as a context object.

Depending on what your script needs to do, this object can be a critical part of your script. You can give this object any name you want, but most examples in this help center call these objects **context**.



Note: For an explanation of entry points, see the [Key Concepts](#) section of the [SuiteScript 2.x Hello World](#).

Context Objects in Entry Point Scripts

Every standard script type includes entry points specific to that type. Most of these standard entry points have access to a context object that provides access to data or methods. The properties of this object vary depending on the entry point being used. For details about the context object available to each entry point, refer to the documentation for that entry point.

For an example of context objects being used in a user event script, see [SuiteScript 2.x User Event Script Tutorial](#).

Context Objects in Custom Module Scripts

Just like an entry point script, in a custom module script, an entry point function can receive a context object. However, in the case of a custom module, the object is not provided by the NetSuite application. The object is provided by the SuiteScript that is calling the module.

If you want a custom entry point function to receive an object, then the SuiteScript that calls the function must be written in a way that supports that behavior: The calling script must create the object, set whatever properties are needed, and pass the object to the custom module script. The custom module can then use the object.

For an example, see [SuiteScript 2.x Custom Module Tutorial](#).



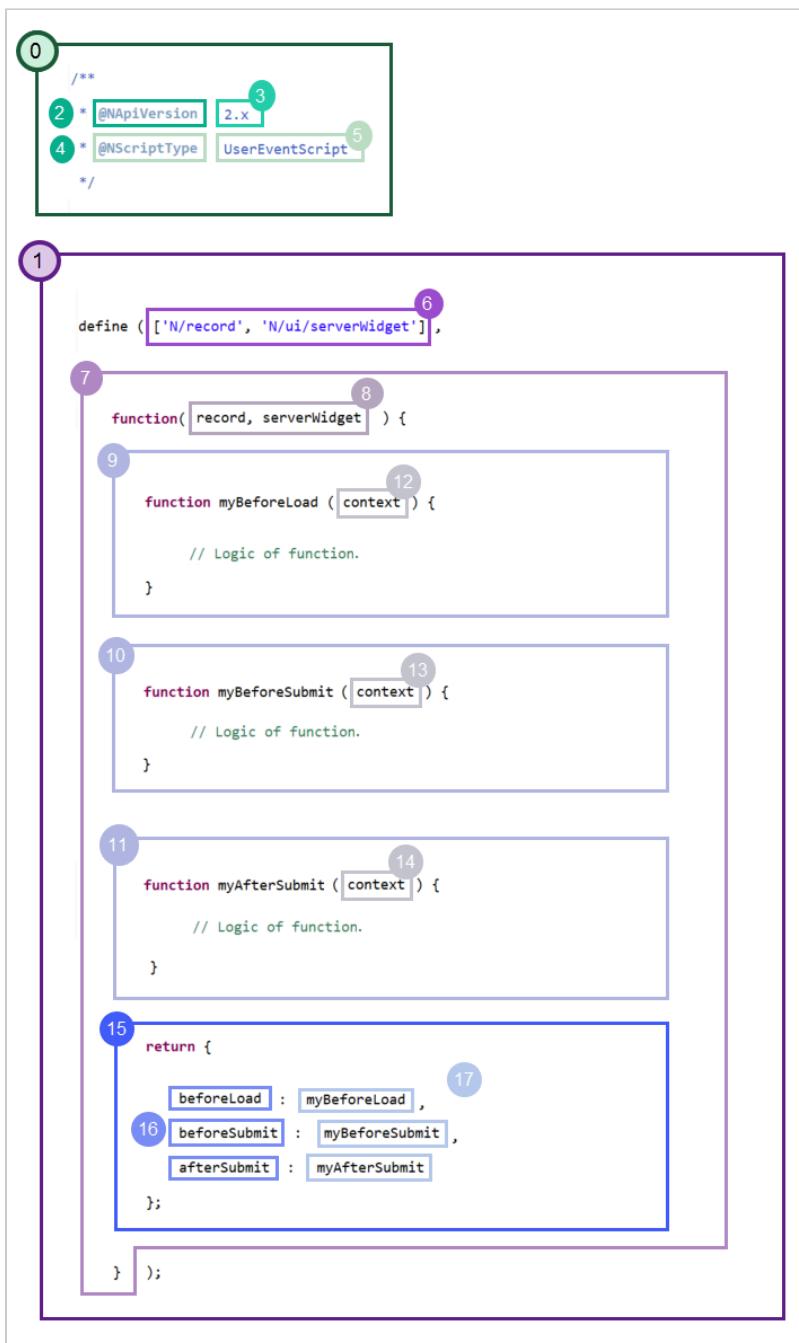
Note: To see an example of how the components described in this topic appear in scripts, review [SuiteScript 2.x Anatomy of a Script](#) and [SuiteScript 2.x Anatomy of a Custom Module Script](#).

SuiteScript 2.x Anatomy of a Script

All SuiteScript 2.x entry point scripts must conform to the same basic structure. The following diagram illustrates that structure. For an explanation of the numbered components of the script, refer to the table that follows the diagram.



Note: For help with the terms defined in this topic, review SuiteScript 2.x Hello World.



| General Area | Callout | Description |
|----------------|---------|---|
| 0 — JSDoc tags | 2 and 3 | The @NApiVersion tag, which is required in an entry point script, and its value. Valid values are 2.0, 2.x, and 2.X. |
| | 4 and 5 | The @NScriptType tag, which is required in an entry point script, and its value. The value is not case sensitive, but using Pascal case, as shown in this example, is recommended for better readability. |

| General Area | Callout | Description |
|----------------------|----------------|---|
| 1 — define statement | 6 | The define function's first argument, which is a list of dependencies, or a list of modules that the script loads. This script uses the N/record Module , which lets the script interact with records, and the N/ui/serverWidget Module , which lets the script interact with forms. |
| | 7 | The define function's second argument, which is a callback function. |
| | 8 | The arguments of the callback function. The first argument is an object that represents the N/record Module . The second represents the N/ui/serverWidget Module . The sequence of these objects matches the sequence of the define function's list of dependencies (Callout 6). These objects can be used anywhere in the callback function to access the APIs of those modules. You can give these objects any names you prefer. As a best practice, use names that are similar to the module names. |
| | 9, 10, and 11 | Entry point functions. For any function to be used, it must be named in the return statement alongside an entry point, as shown in Callout 17. |
| | 12, 13, and 14 | The context object provided to each entry point function. The characteristics of these objects vary depending on the entry point. For an explanation of these objects, see Context Objects Passed to Standard and Custom Entry Points . |
| | 15 | The callback function's return statement. |
| | 16 | The entry points used by the script. At least one entry point must be used. In an entry point script, any entry points used must belong to the script type identified by the @NScriptType tag (callouts 4 and 5). |
| | 17 | References to the script's entry point functions. For each entry point used, your script must identify an entry point function defined elsewhere in the script. |

SuiteScript 2.x Script Creation Process

The following is a very basic process flow for SuiteScript 2.x script creation. For a more detail explanation with a sample script, see [SuiteScript 2.x Hello World](#).



Note: Your specific process may vary, depending on the content of your script.

| Stage | Description | Additional Information |
|-------|--|--|
| 1 | Use the define() function to load SuiteScript 2.0 modules in your entry point script. Your entry point script is the script you attach to the script record. | Modular Architecture SuiteScript 2.x Global Objects |
| 2 | Add required JSDoc tags to your entry point script. | SuiteScript 2.x JSDoc Validation |
| 3 | Add at least one entry point function to your entry point script. An entry point function is a named function that is executed when an entry point is triggered. | SuiteScript 2.x Script Types |

| Stage | Description | Additional Information |
|-------|--|--|
| |  Important: Your entry point script can implement only one script type. For example, your entry point script cannot return both a beforeLoad entry point and an onRequest entry point. | |
| 4 | Organize your supporting code into custom modules (as a replacement for SuiteScript 1.0 libraries). Create these modules with the define() function and then load them in your entry point script. | SuiteScript 2.x Global Objects |
| 5 | Upload and deploy your script to NetSuite. | SuiteScript 2.x Entry Point Script Creation and Deployment |

SuiteScript 2.x Advantages

SuiteScript 2.x is a complete redesign of the SuiteScript model used in SuiteScript 1.0. This topic discusses several of the advantages SuiteScript 2.x has over SuiteScript 1.0.

 **Note:** Use SuiteScript 2.x for new scripts that you develop, and consider converting your SuiteScript 1.0 scripts to SuiteScript 2.0 or SuiteScript 2.1. SuiteScript 1.0 is no longer being updated, and no new feature development or enhancement work is being done for this version. SuiteScript 1.0 scripts continue to be supported, but you should use SuiteScript 2.0 or SuiteScript 2.1 for any new or substantially revised scripts to take advantage of new features, APIs, and functionality enhancements.

- Complexity Management and Intuitive Code Organization
- Automatic Dependency Management
- Modern Programming Syntax and Behavior
- Functionality Enhancements

Complexity Management and Intuitive Code Organization

SuiteScript 2.x is built on modularity. Modern SuiteApps require complex scripts that typically consist of many lines of code and many files. Modularity gives users built-in complexity management. It also adds encapsulation, provides intuitive code organization, and ensures there are no global variable or method naming conflicts.

SuiteScript 2.x comes with a complete set of new APIs, contained within modules. These modules are organized and named based on behavior. For example, you use the N/file module when you need to work with files in NetSuite. Your script loads only those modules that it needs. With SuiteScript 1.0, all APIs are contained in the same global namespace. Each SuiteScript 1.0 script utilizes the entire namespace, regardless of which APIs it actually uses.

SuiteScript 2.x also enables you to create your own custom modules. You can use these custom modules to organize your code (as a replacement for SuiteScript 1.0 libraries). Additionally, you can add custom modules to SuiteApps and expose those modules to third parties.

For additional information, see the following help topics:

- [SuiteScript 2.x Hello World](#)
- [SuiteScript 2.x Script Basics](#)

- SuiteScript 2.x Modules
- SuiteScript 2.x Entry Point Script Creation and Deployment
- SuiteScript 2.x Custom Modules



Note: SuiteScript 2.x implements the Asynchronous Module Definition (AMD) specification. AMD is used to define and load JavaScript modules and their dependencies. For additional information regarding AMD, see <http://requirejs.org/docs/whyamd.html> and <https://github.com/amdjs/amdjs-api/blob/master/AMD.md>.

Automatic Dependency Management

SuiteScript 2.x gives you built-in dependency management. With SuiteScript 2.x, you define the SuiteScript 2.x modules and custom modules that must load prior to script execution. The module loader automatically loads the dependencies of those modules, the dependencies of the dependencies, and so forth. Automatic dependency management enables you to concentrate on logic instead of dependencies and load order.

For additional information, see the following help topics:

- SuiteScript 2.x Hello World
- SuiteScript 2.x Script Basics
- SuiteScript 2.x Entry Point Script Creation and Deployment
- SuiteScript 2.x Custom Modules

Modern Programming Syntax and Behavior

The underlying design principle of SuiteScript 2.x is that SuiteScript 2.x === JavaScript. This results in a decreased learning curve for experienced JavaScript developers. The syntax is straightforward JavaScript, and unlike SuiteScript 1.0, the behavior is consistent. Enhancements to syntax and behavior include the following:

- **Third party JavaScript API support:** SuiteScript 2.x is designed to support all standard JavaScript. The supplied SuiteScript 2.x APIs give you programmatic access to NetSuite functionality. For generic logic, use custom modules to load your preferred third party JavaScript APIs.
- **SuiteScript 1.0 nlapi/nlobj prefix retirement:** SuiteScript 2.x is modeled to look and behave like modern JavaScript. To meet that objective, SuiteScript 2.x methods and objects are not prefixed with nlapi and nlobj. This change also reflects the modular organization of SuiteScript 2.x. SuiteScript 1.0 methods and objects respectively belong to the nlapi and nlobj namespaces. SuiteScript 2.x methods and objects are encapsulated within various modules.
- **Usage of properties and enumerations:** SuiteScript 2.x adopts the usage of properties and enumerations. Most SuiteScript 1.0 getter and setter methods are replaced with properties in SuiteScript 2.x, and enumerations encapsulate common constants (for example, standard record types).



Note: JavaScript does not include an enumeration type. The SuiteScript 2.x documentation uses the term enumeration (or enum) to describe the following: a plain JavaScript object with a flat, map-like structure. Within this object, each key points to a read-only string value.

- **Updated sublist and column indexing:** The standard practice in the development world is to start indexing at 0. This behavior is observed in the majority of programming languages. SuiteScript 1.0

starts sublist and column indexing at 1. To bring SuiteScript into alignment with modern JavaScript, sublist and column indexing within SuiteScript 2.x begins at 0.

- A new version of SuiteScript, SuiteScript 2.1, is also available. This version is the latest minor version of SuiteScript. It extends SuiteScript 2.x by supporting additional ECMAScript language features and syntax. For more information, see [SuiteScript 2.1](#).

For additional information, see the following help topics:

- [SuiteScript 2.x Hello World](#)
- [SuiteScript 2.x Entry Point Script Creation and Deployment](#)
- [SuiteScript 2.x Custom Modules](#)

Functionality Enhancements

The following enhancements are exclusive to SuiteScript 2.x:

- [Map/Reduce Script Type](#)
- [Asynchronous Processing \(Promises\)](#)
- [SFTP File Transfer API](#)
- [Cache API](#)
- [Search Pagination API](#)
- [Flat File Streaming API](#)
- [Expanded Support for HTTP Content Type Headers](#)
- [New Encryption/Encoding Functionality](#)

Map/Reduce Script Type

SuiteScript 2.x introduces a new server script type based on the map/reduce model. Map/ reduce scripts provide a structured framework for server scripts that process a large number of records. In addition, SuiteCloud Plus users can also use map/reduce scripts to process records in parallel across multiple processors. Users manually select the number of processors to utilize from the script deployment record.

For additional information, see [SuiteScript 2.x Map/Reduce Script Type](#).

Asynchronous Processing (Promises)

Promises are JavaScript objects that represent the eventual result of an asynchronous process. After these objects are created, they serve as placeholders for the future success or failure of an operation. During the period of time that a Promise object is waiting in the background, the remaining segments of the script can execute.

In SuiteScript 2.x, all client scripts, and a subset of server scripts, support the use of promises. With promises, developers can write asynchronous code that is intuitive and efficient. SuiteScript 2.x provides promise APIs for selected modules. In addition, you can create custom promises in all client scripts.

For additional information see the help topic [Promise Object](#).

SFTP File Transfer API

SuiteScript 2.x provides support for SFTP (Secure File Transfer Protocol). This feature enables you to securely transfer files between NetSuite and external FTP (File Transfer Protocol) servers. SFTP is a protocol packaged with SSH (Secure Shell). It is similar to FTP, but files are transferred over a secure connection. Server authorization requires a password GUID (Globally Unique Identifier) and a DSA (Digital Signature Algorithm), ECDSA (Elliptical Curve Digital Signature Algorithm), or RSA (cryptosystem) host key.

For additional information, see the help topic [N/sftp Module](#).

Cache API

The SuiteScript 2.x Cache API enables you to load data into a cache and make it available to one or more scripts. This feature reduces the amount of time required to retrieve data.

For additional information, see the help topic [N/cache Module](#).

Search Pagination API

The SuiteScript 2.x Search Pagination API enables you to page through search results. This enhancement increases script performance and gives you an intuitive means to efficiently traverse search result data.

For additional information, see the help topic [N/search Module](#).

Flat File Streaming API

With SuiteScript 1.0, you cannot easily access the contents of files that are over 10MB. You must partition your files into smaller, separate files to read, write, and append file contents in memory.

The SuiteScript 2.x Flat File Streaming API enables you to efficiently process and stream large CSV and plain text files. With this enhancement, you can load and edit each line of content into memory, and then append the lines back together. The Flat File Streaming API enforces the 10MB limit only on individual lines of content.

For additional information, see the help topic [N/file Module](#).

Expanded Support for HTTP Content Type Headers

SuiteScript 2.x adds support for all HTTP content types. This enhancement applies to both client request and server response HTTP headers.

For additional information, see the help topics [N/http Module](#) and [N/https Module](#).

New Encryption/Encoding Functionality

SuiteScript 2.x adds enhanced encryption, decryption, and hashing functionality with the [N/crypto Module](#). Additional encoding functionality is exposed in the [N/encode Module](#).

SuiteScript 2.x Terminology



Note: These definitions reflect the usage of the terms as they apply to SuiteScript 2.0 and SuiteScript 2.1. Some terms may have alternate meanings outside of the SuiteScript context.

| Term | Definition |
|-----------------------|---|
| Class | A programming template that allows you to create complex objects; classes encapsulate data with code to work on that data. This applies only to SuiteScript 2.1. |
| Constants | A constant reference to a value. This applies only to SuiteScript 2.1. |
| Custom Module | <p>A custom module is a user-defined module that serves as a JavaScript library or supporting logic. This module is separate from your entry point script. Your entry point script loads this module as a dependency.</p> <p>For additional information on custom modules, see SuiteScript 2.x Custom Modules.</p> |
| Deferred Dynamic Mode | See the definition for standard mode. |
| Destructuring | A JavaScript expression that allows us to extract data from arrays, objects, and maps and set them into new, distinct variables. This applies only to SuiteScript 2.1. |
| Dynamic Mode | See the definition for standard mode. |
| Entry Point | <p>An entry point represents the juncture at which the system grants control of the NetSuite application to the script. Each script type includes one or more entry points that are exclusive to that type. When that entry point is invoked, the system knows to execute its corresponding entry point function.</p> <p>For additional information on entry points, see SuiteScript 2.x Entry Point Script Validation, SuiteScript 2.x Hello World, and SuiteScript 2.x Script Types.</p> |
| Entry Point Function | <p>A function that executes when an entry point is invoked.</p> <p>For additional information on entry point functions, see SuiteScript 2.x Entry Point Script Validation and SuiteScript 2.x Hello World.</p> |
| Entry Point Script | <p>Your entry point script is the primary script that you attach to the script record. This script identifies the script type, entry points, and entry point functions. Each entry point script must include at least one entry point and entry point function.</p> <p>For additional information on entry point scripts, see SuiteScript 2.x Entry Point Script Validation and SuiteScript 2.x Hello World.</p> |
| Enum | JavaScript does not include an enumeration type. The SuiteScript 2.x documentation utilizes the term enumeration (or enum) to describe the following: a plain JavaScript object with a flat, map-like structure. Within this object, each key points to a read-only string value. |
| Governance | <p>Governance ensures that an individual script does not consume an unreasonable amount of resources. The SuiteScript governance model is based on usage units. Certain API calls cost a specific number of usage units. Each script type has a maximum number of usage units that it can expend per execution. If the number of allowable usage units is exceeded, the script is terminated and an error is thrown.</p> <p>For additional information on governance, see the help topic SuiteScript Governance and Limits.</p> |
| Script | <p>A script refers to an aggregate of</p> <ul style="list-style-type: none"> ■ An entry point script file ■ All custom modules used by the entry point script |

| Term | Definition |
|-------------------|---|
| | <ul style="list-style-type: none"> ■ The script record associated with the entry point script <p>For additional information on the components that make up a script, see SuiteScript 2.x Entry Point Script Validation, SuiteScript 2.x Custom Modules, Script Record Creation, and SuiteScript 2.x Hello World.</p> |
| Script Deployment | <p>A script deployment determines a portion of the associated script's runtime behavior. The settings you can define on a script deployment record include:</p> <ul style="list-style-type: none"> ■ The record types the script executes against ■ When the script is executed ■ The audience and role restrictions ■ The script log levels <p>The settings found on the deployment record vary based on script type. For additional information on script deployments, see SuiteScript 2.x Entry Point Script Creation and Deployment and SuiteScript 2.x Hello World.</p> |
| Script Type | <p>SuiteScript 2.x scripts consist of several script types. Each script type is designed for a specific type of situation and specific types of triggering events.</p> <p>For additional information on script types, see SuiteScript 2.x Script Types.</p> |
| Standard Mode | <p>Standard mode is also referred to as deferred dynamic mode.</p> <p>There are two modes you can operate in when you work with a record in SuiteScript: standard mode and dynamic mode.</p> <ul style="list-style-type: none"> ■ In standard mode, the record's body fields and sublist line items are not sourced, calculated, and validated until the record is saved. ■ In dynamic mode, the record's body fields and sublist line items are sourced, calculated, and validated in real-time. A record in dynamic mode emulates the behavior of a record in the UI. <p>For additional information on standard and dynamic mode, see the help topic record.Record.</p> |
| Usage Units | See the definition for Governance. |

SuiteScript 2.x Developer Resources

SuiteScript 2.x Help

See the following help sections for information about developing with SuiteScript 2.x:

- [SuiteScript 1.0 to SuiteScript 2.x API Map](#)
- [SuiteScript 2.x Script Types](#)
- [SuiteScript 2.x Global Objects](#)
- [SuiteScript 2.x Modules](#)
- [SuiteScript 2.x JSDoc Validation](#)
- [SuiteScript 2.x Entry Point Script Creation and Deployment](#)
- [SuiteScript 2.x Custom Modules](#)
- [SuiteScript 2.x Scripting Records and Subrecords](#)
- [SuiteScript 2.x Custom Pages](#)

SuiteScript 2.x Internal Resources

The following internal resources are available in addition to the SuiteScript 2.x help.

| Resource | Description |
|---|--|
| NetSuite User Group | Official forum for the NetSuite community. Use the search field to find the SuiteScript 2.x board. |
| SuiteScript 2.0: Extend NetSuite with JavaScript | Oracle Training Course |
| SuiteScript 2.0 for Experienced Developers | Oracle Training Course |
|  SuiteScript Help Overview video | Video from the Oracle Video Hub |

SuiteScript 2.x External Resources

The following external resources are available in addition to the SuiteScript 2.x help.

| Resource | Description |
|---|--|
| AMD Specification | SuiteScript 2.x implements its modular architecture with the Asynchronous Module Definition (AMD) specification. AMD is used to define and load JavaScript modules and their dependencies. |
| RequireJS | RequireJS also implements the AMD specification. |
| https://www.promisejs.org/ | Tutorials on JavaScript promises |
| Eloquent JavaScript | Free online JavaScript book |
| You Don't Know JS | Free online JavaScript book series |
| Object-oriented JavaScript for Beginners | MDN article on object oriented JavaScript |
| StackOverflow – SuiteScript 2.0 | Online community for developers |

SuiteScript Reserved Words

In the ECMAScript specification, reserved words are identifiers that have special meaning. For example, the `var` reserved word indicates a variable declaration, and the `try` reserved word indicates the start of a try-catch block. You cannot use reserved words as variable names, labels, or function names in languages based on the ECMAScript specification, such as JavaScript. For more information about the ECMAScript specification, see [JavaScript language resources](#).

SuiteScript is based on the ECMAScript specification, and you cannot use ECMAScript reserved words as variable names or function names in your SuiteScript scripts. This restriction applies to both SuiteScript 2.0 and SuiteScript 2.1. If you use a reserved word as a variable name or function name, your script may produce a syntax error when it runs.

To view the list of reserved words, visit the following link:

[JavaScript Reserved Words](#)

You should not use any of the reserved words that are included in ECMAScript 2015. For best compatibility, you should also avoid using any reserved words that may be supported in future editions of ECMAScript.

SuiteScript Versioning Guidelines

See the following sections for information about SuiteScript versioning:

- [SuiteScript 2.1](#)
- [SuiteScript Versioning](#)
- [Version Cohabitation Rules](#)



Important: If you are using SuiteScript 1.0 for your scripts, consider converting these scripts to SuiteScript 2.0 or 2.1. Use SuiteScript 2.0 and 2.1 to take advantage of new features, APIs, and functionality enhancements. For more information, see [SuiteScript 2.x Advantages](#) and [SuiteScript 2.1](#).

SuiteScript 2.1

A new version of SuiteScript is now available. This new version, SuiteScript 2.1, includes the existing features and functionality that SuiteScript 2.0 offers, and it also supports new language capabilities that were introduced in the ECMAScript 2019 (ES2019) edition of the ECMAScript specification. SuiteScript 2.1 supports all server script types, such as user event scripts, scheduled scripts, and Suitelets, and it is backward compatible with SuiteScript 2.0.

This latest version of SuiteScript is separate from previous SuiteScript versions (1.0 and 2.0), and you can create and run SuiteScript 2.1 scripts alongside SuiteScript 1.0 and 2.0 scripts in your account. Your existing scripts are not affected.

To learn more about SuiteScript 2.1, see [SuiteScript 2.1](#).

SuiteScript Versioning

SuiteScript 2.0 and all future releases of SuiteScript will maintain the following versioning system.

| Version Type | Numbering Pattern | Description |
|--------------|-----------------------|--|
| Major | Version 2.0, 3.0, 4.0 | <p>Major versions of SuiteScript include significant functionality changes and improvements.</p> <p>Major versions are not backward compatible with previously released versions.</p> |
| Minor | Version 2.0, 2.1, 2.2 | <p>Minor versions of SuiteScript include enhancements to existing features.</p> <p>Minor versions are backward compatible with all versions released since the last major version. For example, SuiteScript 2.1 is backward compatible with SuiteScript 2.0, but it is not backward compatible with SuiteScript 1.0.</p> |
| Patch | Does not apply | <p>Patch versions of SuiteScript are included with regular NetSuite bug fix releases.</p> <p>Patch versions are backward compatible with all versions released since the last major version.</p> |

You can also specify a SuiteScript version of 2.x in your scripts. The 2.x value usually represents the latest version of SuiteScript that is generally available and does not represent any versions that are released as beta features. However, this does not apply to SuiteScript 2.1. In this release, the 2.x value indicates that a script uses SuiteScript 2.0, not SuiteScript 2.1. You can still use SuiteScript 2.1 and all of its features in your server scripts, but your 2.x scripts will not run as SuiteScript 2.1 scripts until a future release. For more information about specifying a version for your script, see [SuiteScript 2.x JSDoc Validation](#) and [SuiteScript Versions](#).

You can use an account-level preference, Execute SuiteScript 2.x Server Scripts As, to specify that the 2.x value should represent SuiteScript 2.1 instead of SuiteScript 2.0. Setting this preference to 2.1 applies to all scripts in your account that use the 2.x value. For more information about this preference and how to use it, see [Enabling SuiteScript 2.1 at the Account Level](#).

Version Cohabitation Rules

You cannot use APIs from different major versions of SuiteScript in one script. For example, you cannot use SuiteScript 1.0 APIs and SuiteScript 2.0 APIs in the same script. However, you can have multiple scripts in your account that use different SuiteScript versions. These scripts can be deployed in the same account, in the same SuiteApp, and on the same record.

The Map/Reduce script type is a new functionality introduced with SuiteScript 2.0. You cannot use `nlapiScheduleScript(scriptId, deployId, params)` to schedule a Map/Reduce script. See [SuiteScript 2.x Map/Reduce Script Type](#) for more information.

SuiteScript 2.1

A new version of SuiteScript, SuiteScript 2.1, is now available. This new version uses a different runtime engine than SuiteScript 2.0, includes new language capabilities and functionality, and supports all server script types.

SuiteScript 2.1 supports features planned for future editions of the ECMAScript specification using ES.Next. ES.Next introduces several new language features that you can use in your SuiteScript 2.1 scripts. For example, ES.Next introduces the spread operator, which lets an iterable element spread or expand inside a receiver. You can use the spread operator to spread the values from two Objects into a new Object. For more information on other powerful new language features, see [SuiteScript 2.1 Language Examples](#).

SuiteScript 2.0 is based on the ECMAScript 5.1 (ES5.1) edition of the specification, and it does not support the language capabilities included in the current edition or future editions using ES.Next. For more information about ECMAScript, see [JavaScript language resources](#).

A few notes:

- SuiteScript 2.1 uses the same reserved words as SuiteScript 2.0. For more information, see [SuiteScript Reserved Words](#).
- The functionality provided by the SuiteScript API is the same for SuiteScript 2.0 and SuiteScript 2.1. For more information on this API, see the help topic [SuiteScript 2.x API Reference](#).
- SuiteScript 2.1 client scripts are not currently supported in the Scriptable Cart. For more information on the Scriptable Cart, see the help topics [Scriptable Cart](#) and [SuiteScript for Scriptable Cart](#).

For more information on how to execute your script as a SuiteScript 2.1 script, see [Executing Scripts Using SuiteScript 2.1](#).

Executing Scripts Using SuiteScript 2.1

You can use an account-level preference, Execute SuiteScript 2.x Server Scripts As, to specify that the 2.x value in the @NApiVersion JSDoc tag should represent SuiteScript 2.1 and not default to SuiteScript 2.0. With this preference, you can execute all of your SuiteScript 2.x scripts as SuiteScript 2.1 scripts without having to make any changes to any script. This preference lets you specify which version of SuiteScript the 2.x value maps to (SuiteScript 2.0 or SuiteScript 2.1). For more information about this preference, see [Enabling SuiteScript 2.1 at the Account Level](#).

For this release, a value of 2.x for the @NApiVersion JSDoc tag will, by default, resolve to 2.0 when the script is uploaded and executed. This means that all scripts that include @NApiVersion 2.x are executed as SuiteScript 2.0 scripts. In a future release, this will change so that scripts annotated with 2.x will automatically execute as SuiteScript 2.1 scripts, by default. When this happens, account administrators will receive notifications that 2.x annotated scripts will begin executing as SuiteScript 2.1 scripts. Notifications will be sent to account administrators if there is at least one script in their account that is annotated with the @NApiVersion JSDoc tag value of 2.x.



Note: There are some considerations when annotating your browser-executed client scripts as 2.1 (or 2.x with the Execute SuiteScript 2.x Server Scripts As preference set to 2.1):

- Client scripts annotated with the @NApiVersion JSDoc tag value of 2.1 may not work in the Internet Explorer 11 (IE11)(or below) browser, particularly if the script includes ES6+ syntax. If your client script is to be executed specifically on the IE11 (or below) browser, you will need to use the 1.0 or 2.0 value for the @NApiVersion JSDoc tag. If your script is not required to execute on the IE11 (or below) browser, you can safely use the 2.1 value for the @NApiVersion JSDoc tag.
- Client scripts annotated with the @NApiVersion JSDoc tag value of 1.0 or 2.0 will work as expected in the Internet Explorer 11 browser.

For more information on which browsers are supported in NetSuite, see the help topic [Supported Browsers for NetSuite](#).

There are two ways you can execute your scripts using SuiteScript 2.1:

- **Execute a single script using SuiteScript 2.1** — You can use the @NApiVersion JSDoc tag in your script file to indicate explicitly that the script should execute as a SuiteScript 2.1 script. For more information, see [Executing a Single Script Using SuiteScript 2.1](#).
- **Execute all SuiteScript 2.x scripts using SuiteScript 2.1** — You can use the account-level preference, Execute SuiteScript 2.x Server Scripts As, to specify that all scripts that specify 2.x for the @NApiVersion JSDoc tag value should execute as SuiteScript 2.1 scripts. For more information, see [Enabling SuiteScript 2.1 at the Account Level](#).

For more information about the @NApiVersion JSDoc tag, see [SuiteScript 2.x JSDoc Validation](#).

Executing a Single Script Using SuiteScript 2.1

To execute a single script as a SuiteScript 2.1 script, you must annotate the script to explicitly use SuiteScript 2.1. This will let you execute a single script as a SuiteScript 2.1 script without affecting any other scripts in your account. This approach is useful if you want to use SuiteScript 2.1 features in only a few of your scripts. It is also useful if you are not ready to execute all SuiteScript 2.x annotated scripts in your account as SuiteScript 2.1 scripts.

To execute a single script using SuiteScript 2.1:

1. In your script file, include the following JSDoc comment block at the beginning of the file:

```
1 | /**
2 | * @NApiVersion 2.1
3 | */
```

2. Add other JSDoc tags (such as @NScriptType) as appropriate. For more information, see [SuiteScript 2.x JSDoc Validation](#).
3. Upload and deploy your script. Your script will be validated as a SuiteScript 2.1 script. For more information, see [SuiteScript 2.x Entry Point Script Creation and Deployment](#).

Enabling SuiteScript 2.1 at the Account Level

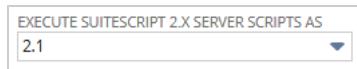
When you use the `@NApiVersion` JSDoc tag in a script file, you can specify a value of `2.x`. This value usually represents the latest version of SuiteScript that is generally available. However, in this release, this value represents SuiteScript 2.0 by default. You can use an account-level preference, **Execute SuiteScript 2.x Server Scripts As**, to specify that the `2.x` value should represent SuiteScript 2.1 instead of SuiteScript 2.0. Setting this preference to 2.1 applies to all scripts in your account that use the `2.x` value. Any scripts that use the 2.0 or 2.1 value for the `@NApiVersion` JSDoc tag are not affected.

You can use the **Execute SuiteScript 2.x Server Scripts As** preference to preview how your SuiteScript 2.x scripts will execute using SuiteScript 2.1. This approach is useful if you want to test all of the 2.x scripts in your account as SuiteScript 2.1 scripts without having to specifically annotate each script with the `@NApiVersion` JSDoc tag value of 2.1. Using this preference, you do not need to alter any script files to change the SuiteScript version they use to execute.

Account-Level Preference for SuiteScript 2.x Scripts

To enable SuiteScript 2.1 at the account level:

1. Go to Setup > Company > Preferences > General Preferences.
2. In the **Execute SuiteScript 2.x Server Scripts As** dropdown list, select **2.1**.



3. Click **Save**.

When you use this account-level preference, you should be aware of how validation of the script occurs. If you have a SuiteScript 2.x script in your account and you set the **Execute SuiteScript 2.x Server Scripts As** preference to 2.1, any syntax changes to the script are validated as if the script was a SuiteScript 2.0 script. If you add syntax that is supported only in SuiteScript 2.1, you receive a syntax error when you save the script file. This approach is designed to ensure that SuiteScript 2.x scripts in your account continue to work even if the **Execute SuiteScript 2.x Server Scripts As** preference is changed from 2.1 back to 2.0. To test new SuiteScript 2.1 features and syntax, you should update the `@NApiVersion` tag in your script files from 2.x to 2.1 explicitly. You can also upload new scripts as SuiteScript 2.1 scripts. The following table summarizes when errors can occur with SuiteScript 2.1 scripts:

| Scenario | Outcome |
|---|--|
| <ol style="list-style-type: none"> 1. Set the Execute SuiteScript 2.x Server Scripts As preference to 2.1. 2. Open the script record of an executing script in your account that uses an <code>@NApiVersion</code> value of <code>2.x</code>. 3. Edit the script and add code that is supported only in SuiteScript 2.1 (such as using the Spread operator). 4. Save the updated script. | <p>A syntax error occurs when you save the script. You cannot add SuiteScript 2.1 syntax to existing 2.x scripts, even if the Execute SuiteScript 2.x Server Scripts As preference is set to 2.1.</p> <p>To resolve this issue, change the <code>@NApiVersion</code> value to 2.1 to explicitly execute the script as a SuiteScript 2.1 script.</p> |
| <ol style="list-style-type: none"> 1. Upload a script includes an <code>@NApiVersion</code> value of <code>2.x</code> and also include SuiteScript 2.1 syntax (such as using the Spread operator). 2. Change the value of the Execute SuiteScript 2.1 Server Scripts As preference to 2.0. | <p>The script no longer works. When the account-level preference is set to SuiteScript 2.0, all 2.x scripts execute as SuiteScript 2.0 scripts. You cannot include SuiteScript 2.1 syntax in a SuiteScript 2.0 script.</p> <p>To resolve this issue, do one of the following:</p> <ul style="list-style-type: none"> ■ Change the <code>@NApiVersion</code> value to 21.1 to explicitly execute the script as a SuiteScript 2.1 script. |

| Scenario | Outcome |
|----------|--|
| | <ul style="list-style-type: none"> ■ Change the value of the Execute SuiteScript 2.x Server Scripts As preference to 2.1. ■ Remove the SuiteScript 2.1 syntax from the script. |

You can verify the version of SuiteScript used to execute a script by checking the Execute As Version field on the script record. For more information, see [Version Information on the Script Record](#).

Version Information on the Script Record

Using the SuiteScript 2.1 value for the Execute SuiteScript 2.x Server Scripts As preference overrides the script version specified using the `@NApiVersion` JSDoc tag within the script . Because of this, it can be useful to verify the version of SuiteScript used to execute the script. The script record includes two fields that provide information about SuiteScript version:

- **API Version** — This field indicates the version of SuiteScript that is specified in the script file. It always represents a concrete version of SuiteScript. For example, if you specify a version of 2.x in the script file, the value of this field is set to 2.0 (the value that the 2.x value maps to in this release). For more information about this field, see [Script Record Creation](#).



- **Execute As Version** — This field indicates the version of SuiteScript that is being used to execute the script. This field only appears on the script record if the Execute SuiteScript 2.x Server Scripts As preference is set to 2.1, and is the only indication that the script is executed as a SuiteScript 2.1 script based on the Execute SuiteScript 2.x Server Scripts As preference. The original SuiteScript version (as indicated in the API Version field) is displayed when the script is listed in NetSuite (for example, in script lists, script deployment lists, scripted records lists, and so on).



SuiteScript 2.1 Language Examples

SuiteScript 2.1 supports ES.Next, which includes the latest published edition of the ECMAScript specification (SuiteScript 2.0 supports ES5.1 only). New editions include many improvements to previous editions of ECMAScript, as well as new language features. For example, ES.Next includes support for constants, block-scoped variables and functions, customizable iterator behavior, typed arrays, and more. You can take advantage of these features when you write SuiteScript 2.1 server scripts. You can also consider converting your existing SuiteScript 2.0 server scripts to SuiteScript 2.1, since some of these new features may help you improve performance or refactor your code.

In future releases, SuiteScript will continue to support the latest published edition of ECMAScript using ES.Next. ES.Next does not represent a published edition of the ECMAScript specification. Instead, ES.Next refers to future editions of ECMAScript that have not been released yet. ES.Next includes proposed features that have not been approved for inclusion in an official edition. When one of these features is approved, it will be included in the next published edition of the ECMAScript specification. SuiteScript will support all future published editions of ECMAScript. Future minor versions of SuiteScript will also be backward compatible with previous versions

The following sections include examples of using ES.Next features in SuiteScript 2.1:

- [Spread Operator](#)

- Classes
- Destructuring
- Rest Operator
- Asynchronous Server-Side Promises

For more information about new language features, see [JavaScript language resources](#).

Spread Operator

You can use the spread operator to spread the values from two Objects into a new Object:

```

1 const createDynamicRecord = (options) => {
2   const defaultOptions = {
3     isDynamic: true
4   };
5
6   // Spread values from two Objects into a new Object, and create a new record
7   return record.create({
8     ...defaultOptions,
9     ...options
10   });
11 }
12
13 const cashSale = createDynamicRecord({
14   type: record.Type.CASH_SALE,
15   defaultValues: {
16     'subsidiary' : '1'
17   }
18 });

```

In this example, an arrow function called `createDynamicRecord()` is created. This function uses a set of default parameter values (represented by `defaultOptions`) that apply to all records created using this function. When this function is called, additional parameter values can be specified, and these additional values are merged with the default parameter values when `record.create(options)` is called.

Classes

You can create classes to represent common structures:

```

1 /**
2  * @NApiVersion 2.1
3 */
4 define(['N/currency', 'N/query'],
5       (currency, query) => {
6         // Define a class called Transaction
7         class Transaction {
8           constructor(amount, currency, date, id) {
9             this.amount = amount;
10            this.currency = currency;
11            this.date = date;
12            this.id = id;
13          }
14
15          // Create a helper function to check if the transaction exists
16          exists() {
17            return this.id && query.create({
18              type: query.Type.TRANSACTION,
19              condition: query.createCondition({
20                fieldId: 'id',
21                operator: query.Operator.IS,
22                values: this.id
23              })
24          );
25        }
26      }
27    );
28  );

```

```

24     }).run().results.length > 0;
25   }
26
27   // Create a helper function to convert transaction amount to a desired currency
28   toCurrency(targetCurrency) {
29     return this.amount * currency.exchangeRate({
30       date: this.date,
31       source: this.currency,
32       target: targetCurrency
33     });
34   }
35 }
36
37 // Define Sale and Purchase classes that inherits helper methods from the Transaction class
38 class Sale extends Transaction {
39   constructor(amount, currency, date, customer, id) {
40     super(amount, currency, date, id)
41     this.customer = customer;
42   }
43 }
44
45 class Purchase extends Transaction {
46   constructor(amount, currency, date, vendor, id) {
47     super(amount, currency, date, id)
48     this.vendor = vendor;
49   }
50 }
51
52 // Use case: A user attempts to create a Sale or Purchase object from a search or query
53 // result set. The user can use the createSale() and createPurchase() functions
54 // to create the desired objects.
55 return {
56   createSale(amount, currency, date, customer, id) {
57     return new Sale(amount, currency, date, customer, id);
58   },
59   createPurchase(amount, currency, date, vendor, id) {
60     return new Purchase(amount, currency, date, vendor, id);
61   }
62 }
63 }
64 );

```

Destructuring

You can use destructuring to bind variables to different properties of an Object:

```

1 const name = 'data.csv';
2 const isOnline = true;
3 const fileType = file.Type.CSV;
4 const contents = "Alpha,Bravo,Charlie,Delta";
5
6 // Create a file using destructured properties
7 const dataFile = file.create({name, fileType, isOnline, contents});

```

Rest Operator

You can use destructured assignment to obtain a single property from an Object, and you can use the rest operator to create a new Object with the remaining properties:

```

1 const {fileType, ...fileProps} = file.load({
2   id: 1234
3 });
4 log.debug("fileType", fileType);
5 log.debug("isOnline?", fileProps.isOnline);

```

Asynchronous Server-Side Promises

SuiteScript 2.1 fully supports non-blocking asynchronous server-side promises expressed using the `async`, `await`, and `promise` keywords for a subset of modules: N/http, N/https, N/query, N/search, and N/transaction. See the help topic [Promise Object](#) for a list of promise methods supported in server scripts.

When using server-side promises in SuiteScript, you should consider the following:

- This capability is not for bulk processing use cases where an out-of-band solution, such as a work queue, may suffice.
- This capability is mainly for in-process and distinct operations, such as business functions on a list of transactions when ordering does not matter.

Sample

The following code sample shows the use of the `async` and `await` keywords:

```

1  async function() {
2    let myQuery = query.load ({
3      id: myQueryId
4    });
5    let myResult = await myQuery.run().then (function(result) {
6      // do stuff
7    });
8  }

```

For more information on JavaScript promises, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

For more information on asynchronous JavaScript programming, see <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous>.

For more information on the Promise object in SuiteScript, see the help topic [Promise Object](#).

Differences Between SuiteScript 2.0 and SuiteScript 2.1

SuiteScript 2.1 uses a different runtime engine than SuiteScript 2.0, also supports ECMAScript language features that are not supported in SuiteScript 2.0. This causes some capabilities in SuiteScript 2.0 scripts to function differently when the script is executed using SuiteScript 2.1.

The following sections describe important differences between SuiteScript 2.0 and SuiteScript 2.1:

- Reserved Words as Identifiers
- Error Object Properties
- Invalid JSON Parsing
- Strict Mode
- Reassignment of `const` Variables
- Behavior of `for...each...in` Statement
- Formats for Converting Dates to Local Date Strings
- Conditional Catch Blocks
- The `toSource` Method
- Set Decimal Number with Trailing Zeros
- String Differences for RESTlet Post Method

- RESTlet Return Type Difference
- parseInt Difference

Reserved Words as Identifiers

In the ECMAScript specification, reserved words are identifiers that have special meaning. For example, the `var` reserved word indicates a variable declaration. You cannot use reserved words as variable names, labels, or function names in JavaScript/ECMAScript. Because SuiteScript 2.1 supports a later edition of the ECMAScript specification (ECMAScript 2019) than SuiteScript 2.0 (ECMAScript 5.1), the list of reserved words is different. SuiteScript 2.1 is stricter about not including reserved words in scripts.

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|--|--|--|
| You use <code>extends</code> in one of the following ways in a script: | <p>The script executes and does not generate an error. The <code>extends</code> identifier is not a reserved word in ECMAScript 5.1.</p> <pre> 1 myObj({extends: 'test'}); 2 function myFunction(extends) {} 3 function extends() {} 4 var extends = 1; </pre> | <p>The script generates a syntax error. The <code>extends</code> identifier is a reserved word in ECMAScript 2019.</p> |

To avoid this issue, do not use any reserved words from any edition of ECMAScript (including those planned for future editions using ES.Next). For more information, see [SuiteScript Reserved Words](#).

Error Object Properties

SuiteScript supports the `Error` object, which is provided by ECMAScript, to represent errors that can occur in scripts. You can call `JSON.stringify(obj)` on an Error object to obtain a string representation of the error. If you pass an argument to the constructor when you create the Error object, the string representation in SuiteScript 2.0 includes this argument as the value of the `message` property. The string representation in SuiteScript 2.1 does not include the `message` property.

This difference is because of how `JSON.stringify(obj)` handles enumerable and non-enumerable properties. In SuiteScript 2.0, the output of `JSON.stringify(obj)` includes the content of both enumerable and non-enumerable properties. In SuiteScript 2.1, the output of `JSON.stringify(obj)` includes only the content of enumerable properties and does not include the content of non-enumerable properties, as per the ECMAScript specification.

When you pass an argument to the `Error` object constructor, a non-enumerable `message` property is defined. If you change the value of this property after the `Error` object is created, the property remains non-enumerable. So, when an `Error` object is created in this way, the `message` property is not included in the `JSON.stringify(obj)` output in SuiteScript 2.1 scripts.

By contrast, when you do not pass an argument to the `Error` object constructor, the `message` property is not defined. If you try to set the value of this property after the `Error` object is created, a new `message` property is defined. Properties that are defined in this way are always enumerable. So, when an `Error` object is created in this way, the `message` property is included in the `JSON.stringify(obj)` output in SuiteScript 2.1 scripts.

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|---|---|---|
| You use the following code to create an <code>Error</code> object with no argument in a script: | <p>The <code>firstErrorJson</code> variable contains the following output:</p> <pre> 1 var myFirstError = new Error(); 2 { 3 'message' : 'A message' </pre> | <p>The <code>firstErrorJson</code> variable contains the following output:</p> <pre> 1 { 2 'message' : 'A message' </pre> |

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|---|---|---|
| <pre> 2 myFirstError.message = 'A message'; 3 4 var firstErrorJson = JSON.stringify(myFirstError); </pre> | 3 } | 3 } |
| You use the following code to create an Error object with an argument in a script: | The secondErrorJson variable contains the following output: | The secondErrorJson variable contains the following output: |
| <pre> 1 var mySecondError = new Error(''); 2 mySecondError.message = 'A message'; 3 4 var secondErrorJson = JSON.stringify(mySecondError); </pre> | <pre> 1 { 2 'message' : 'A message' 3 } </pre> | 1 {} |

To avoid this issue, use only enumerable properties for objects. For error messages, create Error objects with no arguments. For more information, see the help topics [JSON.stringify\(obj\)](#) and [Enumerability and ownership of properties](#).

Invalid JSON Parsing

SuiteScript can parse JSON strings that you use in your scripts. When invalid JSON is encountered, a syntax error is generated. The type and format of the generated error message is different in SuiteScript 2.1 than in SuiteScript 2.0. Also, including trailing commas in JSON strings is accepted in SuiteScript 2.0 but generates an error in SuiteScript 2.1.

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|---|--|---|
| You try to parse the following JSON string in a script: | The following error is generated: <i>org.mozilla.javascript.EcmaError: SyntaxError: Unexpected token: b</i> | The following error is generated: <i>SyntaxError: Invalid JSON: <json>:1:0 Expected json literal but found b</i> |
| You try to parse the following JSON string in a script: | The script executes successfully. | The following error is generated: <i>SyntaxError: Invalid JSON: <json>:1:0 Expected json literal but found ,</i> |

To avoid this issue, make sure to use the JSON format that is compatible with the ECMAScript specification. For more information, see the help topic [JSON object](#).

Strict Mode

ECMAScript 5 introduced strict mode. If a SuiteScript 2.0 script assigns a value to a variable without first declaring the variable using the `var` or `const` keyword, the script continues executing and no error is thrown. If a SuiteScript 2.1 script assigns a value to a variable without using the `var`, `let`, or `const` keyword, an error is thrown. For more information about strict mode, see [Strict Mode](#).

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|---|---|-----------------------------------|
| You place a portion of your script (or your entire script) into strict mode and assign a value to an undeclared variable: | Script execution completes without throwing an error. | The following error is generated: |

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|---|--------------------------|---|
| <pre> 1 'use strict'; 2 x = 122; 3 log.debug('x = ', x); </pre> | | <i>ReferenceError: x is not defined</i> |

Reassignment of const Variables

JavaScript const variables create a read-only reference to a value. Once a const variable is assigned a value, the variable identifier cannot be used again (reassigned). In SuiteScript 2.0, no error is thrown when you reassign a const variable. In SuiteScript 2.1, a TypeError is thrown.

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|---|--|---|
| You set a value for a const variable and then try to assign it a new value: | The script completes execution successfully, however the value of recordId is not changed by the recordId = 2; line. The log statement will write: recordId 1. | The execution stops at the recordId = 2; line, and the following error is generated: <i>TypeError: Assignment to constant variable</i> |

Behavior of for...each...in Statement

The for...each...in statement was deprecated as part of ECMA-357 (4x) specifications and should no longer be used. In SuiteScript 2.0, the for...each...in statement is accepted. In SuiteScript 2.1, a SyntaxError is thrown if you try to use the for...each...in statement in your script. For more information on the deprecation of for...each...in, see [for..each..in](#).

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|---|---|---|
| You use a for...each...in statement to process values in an object: | The for...each...in statement is accepted and allows you to process the values in the object. | The following error is generated: <i>SyntaxError: SyntaxError: <eval>:3:4 Expected (but found each for each (var value in obj) {</i> |

Formats for Converting Dates to Local Date Strings

JavaScript supports several date formats and ways to create a date. After you create a date, you can convert it to a local date string. In SuiteScript 2.0, the default format for the converted date string is the long format. In SuiteScript 2.1, the default format for the converted date string is the short format.

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|--|--|---|
| You create a date and log two different formats for that date: | The default format is the long format, and additional properties passed are ignored. The output is: | The default format is the short format, and additional properties passed are ignored. The output is: |

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|--|--|--|
| <pre> 1 var event 2 = new Date(Date.UTC(2012, 12, 21, 12)); 3 log.debug("date1 = ", event.toLocal DateString()); 4 log.debug("date2 = ", event.toLocal DateString('de-De', { year: 'numer ic', month: 'long', day: 'numeric' })); </pre> | <ul style="list-style-type: none"> ■ date1 = December 21, 2012 ■ date2 = December 21, 2012 | <ul style="list-style-type: none"> ■ date1 = 2012-12-21 ■ date2 = 2012-12-21 |

Conditional Catch Blocks

In JavaScript, a try-catch statement can include an optional if statement within the catch block, however it is not ECMAScript specification compliant. In SuiteScript 2.0, a script that includes a conditional catch statement executes without error. In SuiteScript 2.1, a SyntaxError is thrown for any script that includes a conditional catch statement.

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|---|---|--|
| You create a try-catch statement with an if statement in the catch block: | The script completes execution without throwing an error. | <p>The following error is generated:</p> <pre>SyntaxError: SyntaxError: <eval>:2:11 Expected) but found if } catch (e if e instanceof TypeError) { // Don't use</pre> |

The toSource Method

In JavaScript, the toSource method is used to return a string representing the source code of the object. However, this feature is obsolete and should not be used. In SuiteScript 2.0, a script that includes the toSource method for an object executes without error. In SuiteScript 2.1, an error is thrown for any script that includes the toSource method for an object. For more information on the toSource method, see [toSource](#).

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|--|---|---|
| You try to log a transaction's source: | The script completes execution without throwing an error. | An error is thrown indicating toSource is not a valid function. |

Set Decimal Number with Trailing Zeros

When you set a decimal number value in JavaScript, behavior depends on the format of the numerical value. In SuiteScript 2.0, if there are trailing zeros in a decimal value you set, the value is set as a double precision floating point number (double). In SuiteScript 2.1, if there are trailing zeros in a decimal value

you set, the value is set as an integer value. Note that in both SuiteScript 2.0 and SuiteScript 2.1, a value is always set as a double value if there are no trailing zeros in the decimal value you specify.

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|---|---|--|
| You set a decimal value with trailing zeros: | record.setValue() saves a number as a double number: 616.0. 1 record.setValue('memo', 616.00) | record.setValue() saves a number with trailing zeros as an Integer: 616. |
| You set a decimal value with no trailing zeros: | record.setValue() saves a number as a double number: 616.01. 1 record.setValue('memo', 616.01) | record.setValue() saves a number as a double number: 616.01. |

String Differences for RESTlet Post Method

In SuiteScript 2.0, a `JSON.stringify()` call is added internally to whatever is passed in the `post()` method of a RESTlet. This affects the value passed. In SuiteScript 2.1, a `JSON.stringify()` call is not added to the `post()` method for a RESTlet.

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|--|--|--|
| You return <code>JSON.stringify</code> in a post function: | Returns "\\"flower\\\" and a string length of 12. If you specify "flower" in the return statement, "flower" is returned with a string length of 8. 1 //the following is included in a 2.x RESTlet 2 ... 3 function post() { 4 return JSON.stringify("flower"); 5 } | Returns "flower" and a string length of 8. If you specify "flower" in the return statement, flower is returned with a string length of 6. |
| | The log statements are: xmlRequest.responseText="\\\"ok\\\"" xmlRequest.responseText.length=8 1 let xmlHttpRequest = new XMLHttpRequest(); 2 var url = "/app/site/hosting/restlet.nl?script=RESTLET_SCRIPT_ID&deploy=RESTLET_DEPLOYMENT_ID"; 3 4 xmlHttpRequest.onreadystatechange = () => { 5 if (xmlHttpRequest.readyState === 4) { 6 if (xmlHttpRequest.status === 200) { 7 log.debug("xmlRequest.responseText=" + xmlHttpRequest.responseText); 8 log.debug("xmlRequest.responseText.length=" + xmlHttpRequest.responseText.length); 9 } else { 10 log.debug("xmlRequest.status=" + xmlHttpRequest.responseText); 11 } 12 }; 13 14 xmlHttpRequest.open("POST", url, true /* async */); 15 xmlHttpRequest.setRequestHeader("Content-Type", "application/json"); 16 xmlHttpRequest.send(JSON.stringify({})); //log statements appear after request is sent 17 | The log statements are: xmlRequest.responseText="ok" xmlRequest.responseText.length=4 |

RESTlet Return Type Difference

SuiteScript 2.1 changes the way that RESTlet responses are formatted for certain content types. When the Content-Type in the RESTlet header does not match the actual type that the RESTlet is returning, the results returned may not be what you are expecting.

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|---|---|---|
| You set the Content-Type in a RESTlet header to application/json. For example, see the following sample RESTlet script and sample code for calling the RESTlet. | <pre> 1 /** 2 * @NApiVersion 2.x 3 * @NScriptType restlet 4 */ 5 6 define([], function() { 7 function get() { 8 var x = {name:"jane", age:20}; 9 return x; 10 } 11 return { 12 get: get 13 } 14 }); </pre> <pre> 1 var host = window.location.host; 2 var url = 'https://' + host + '/app/site/hosting/restlet.nl? script=customscript296&deploy=customdeploy1'; 3 var xhttp = new XMLHttpRequest(); 4 xhttp.onreadystatechange = function() { 5 if (this.readyState == 4 && this.status == 200) { 6 alert(typeof xhttp.response); 7 alert(typeof JSON.parse(xhttp.response)); 8 } 9 }; xhttp.open("GET", url, true); xhttp.setRequestHeader('Content-Type', 'application/json'); xhttp.send(); </pre> | <p>The response is automatically converted to a string and that string is returned instead of an object.</p> <p>In the example scenario, the alert output is a string, SS2.0 gives string; but in SS2.1 it comes as object.</p> <p>The return type for JSON will be an object.</p> <p>In the example scenario, the alert output is an object.</p> |

parseInt Difference

The parseInt function operates differently in SuiteScript 2.1 than it did in SuiteScript 2.0, for certain calls to parseInt. To avoid this difference, we recommend best practices for using parseInt function, which can be found at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt.

| Scenario | SuiteScript 2.0 Behavior | SuiteScript 2.1 Behavior |
|---|---|-------------------------------------|
| You set the Content-Type in a RESTlet header to application/json. For example, see the following sample RESTlet script and sample code for calling the RESTlet. | <p>There is no value assigned to x.</p> | <p>The return value for x is 8.</p> |

SuiteScript Analytic APIs

SuiteScript Analytic APIs help you work with analytical data in NetSuite using SuiteScript. These APIs support advanced queries and operations that enable you to load, analyze, and update your data programmatically in server scripts.

The following table describes the APIs that are included in SuiteScript Analytic APIs, including links to relevant help topics for each API.

| API | Description | Help Topics |
|---------------------|---|--|
| Query API | The Query API lets you create constructed queries and run them using the N/query module. You can create query conditions, use multilevel joins to different record types, and run queries as paged or non-paged queries. | N/query Module |
| SuiteQL API | The SuiteQL API lets you run queries using the SuiteQL query language. SuiteQL is based on the SQL-92 revision of the SQL database query language and provides advanced query capabilities you can use to access your NetSuite records and data. | SuiteQL SuiteQL in the N/query Module |
| Workbook API (Beta) | <p>The Workbook API lets you manage datasets and workbooks programmatically as part of SuiteAnalytics Workbook. You can create datasets to define your data, create conditions and joins with other record types, define workbook visualizations such as charts and pivots, and more.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> i Note: SuiteQL API refers only to running SuiteQL queries using the N/query module. SuiteQL is available for other channels (such as SuiteAnalytics Connect), but these channels are not included in SuiteQL API. </div> | Workbook API (Beta) |

| API | Description | Help Topics |
|-----|--|---|
| | <p> Warning: The Workbook API is a beta feature. The contents of this feature are preliminary and may be changed or discontinued without prior notice. Any changes may impact the feature's operation with the NetSuite application. NetSuite warranties and product service levels shall not apply to the feature or to the impact of the feature on other portions of the NetSuite application. NetSuite may review and monitor the performance and use of this feature. The documentation for this feature is also considered a beta version and is subject to revision. Please review Beta Software Legal Notices. ORACLE CONFIDENTIAL. For authorized use only. Do not distribute to third parties. Pre-General Availability Draft: September 25, 2020.</p> <p> Note: To access this beta feature, you must complete the Oracle NetSuite Umbrella Beta Program Recruitment form. A link to this form is available from an informational banner on the NetSuite Enable Features page. After you click this link, you can click the Request Login button to access the Recruitment form through Oracle SSO.</p> <p>On the Recruitment form, you are required to identify an authorized user who can sign to accept the Oracle Cloud Services Beta Trial License Agreement. After this Agreement is fully executed by an authorized user through completion of this form, the requested feature is available for enablement on the Enable Features page. For more information, see Accessing Oracle NetSuite Umbrella Beta Program Features.</p> | N/dataset Module N/workbook Module |

Analytics Data Source and Supported Browsers

SuiteScript Analytic APIs are based on the analytics data source. This data source is a collection of NetSuite data that is grouped according to record types and fields. The structure of the analytics data source provides many capabilities and options to analyze your NetSuite data. For more information, see the help topic [Analytics Data Source Overview](#).

The analytics data source provides different information than previous data sources. For example, Search and Report features (including SuiteScript modules such as the N/search module) use a different data source that was available before the introduction of the analytics data source. The record types and fields supported in this data source may be different than those supported in the analytics data source.

You can use a browser to view a summary of the data that a data source includes. A browser provides information about all record types and fields that are available for a data source. The information displayed in the browser depends on the underlying data source. For example, the SuiteScript Records Browser uses the Search and Report data source, so you can use this browser to find record type names and field names for Search and Report functionality. However, this browser does not use the analytics data source, so you cannot use this browser to find record type names and field names for SuiteScript Analytic APIs.

The following browsers are available for the analytics data source:

- **Analytics Browser** — For more information, see the help topic [Analytics Browser](#).
- **Records Catalog** — For more information, see the help topic [Records Catalog Overview](#).

When working with SuiteScript Analytic APIs, you can use either the Analytics Browser or the Records Catalog to find record type names and field names.

Workbook API (Beta)



Note: The content in this help topic pertains to SuiteScript 2.0.



Warning: The Workbook API is a beta feature. The contents of this feature are preliminary and may be changed or discontinued without prior notice. Any changes may impact the feature's operation with the NetSuite application. NetSuite warranties and product service levels shall not apply to the feature or to the impact of the feature on other portions of the NetSuite application. NetSuite may review and monitor the performance and use of this feature. The documentation for this feature is also considered a beta version and is subject to revision. Please review [Beta Software Legal Notices](#). ORACLE CONFIDENTIAL. For authorized use only. Do not distribute to third parties. Pre-General Availability Draft: September 25, 2020.



Note: To access this beta feature, you must complete the Oracle NetSuite Umbrella Beta Program Recruitment form. A link to this form is available from an informational banner on the NetSuite Enable Features page. After you click this link, you can click the Request Login button to access the Recruitment form through Oracle SSO.

On the Recruitment form, you are required to identify an authorized user who can sign to accept the Oracle Cloud Services Beta Trial License Agreement. After this Agreement is fully executed by an authorized user through completion of this form, the requested feature is available for enablement on the Enable Features page. For more information, see [Accessing Oracle NetSuite Umbrella Beta Program Features](#).

The Workbook API is available as a beta feature. The Workbook API lets you manage your datasets and workbooks programmatically as part of SuiteAnalytics Workbook. You can do the following:

- Create new datasets to define your data
- Create dataset conditions and joins with other record types
- Create new workbooks
- Define workbook visualizations, such as charts, pivots, and table views
- Run table views and obtain results
- Load existing datasets or workbooks and manage them

To use the Workbook API in this release, you must enable the Workbook API (Beta) feature on the Enable Features page in your NetSuite account. This feature controls access to all Workbook API components, including SuiteScript modules and core plug-ins. For more information, see [Enabling the Workbook API \(Beta\) Feature](#).

The Workbook API includes the following components:

- **SuiteScript modules** – The N/dataset and N/workbook modules include all of the methods and objects you need to create and manage datasets and workbooks. The N/dataset module lets you define datasets to use for your workbooks. The N/dataset module is similar to the N/query module but is designed to work alongside the N/workbook module. You should use the N/dataset module to create all datasets you plan to use in the Workbook API. The N/workbook module lets you create workbooks, define supporting workbook components, and run table views. For more information, see the help topics [N/dataset Module](#) and [N/workbook Module](#).
- **Granular Workbook elements** – The N/dataset and N/workbook modules let you work with granular components that represent different areas in the SuiteAnalytics Workbook UI. For example, when you use the N/workbook module to create a pivot definition, you can specify pivot-related properties such as name, pivot axes, and sorting behavior. You can also define limiting filters (which filter results based

on a limiting number) and conditional filters (which filter results based on simple or complex criteria). For more information, see [Workbook API \(Beta\) UI Guide](#).

- **Core plug-ins** – The new Dataset Builder Plug-in and Workbook Builder Plug-in let you provide custom implementations for creating datasets and workbooks. You can provide your own implementations with custom logic that meets your business needs.

Getting Started with the Workbook API (Beta)

The following table lists tasks and links to help you get started with the Workbook API.

| Task | Description | Links |
|---|--|---|
| Enable the Workbook API (Beta) feature | Before you can start using the Workbook API, you must enable the Workbook API (Beta) feature on the Enable Features page in your NetSuite account. This feature controls access to all Workbook API components, including SuiteScript modules and core plug-ins. | Enabling the Workbook API (Beta) Feature |
| Follow a tutorial to create your first dataset | A tutorial is available to help you create a basic dataset using the Workbook API. You can use this tutorial as a starting point to explore the features of the Workbook API. | Creating a Dataset Using the Workbook API (Beta) Tutorial |
| Learn how objects and properties in the Workbook API relate to the SuiteAnalytics Workbook UI | The Workbook API lets you manage datasets and workbooks as part of SuiteAnalytics Workbook. Most objects and properties in the Workbook API are represented in the SuiteAnalytics Workbook UI. | Workbook API (Beta) UI Guide |
| Explore the objects and methods in the Workbook API | The N/dataset and N/workbook modules include all of the methods and objects you need to create and manage datasets and workbooks. | N/dataset Module N/workbook Module |

Enabling the Workbook API (Beta) Feature

 **Note:** The content in this help topic pertains to SuiteScript 2.0.

To use the Workbook API in this release, you must enable the Workbook API (Beta) feature on the Enable Features page in your NetSuite account. This feature controls access to all Workbook API components, including SuiteScript modules and core plug-ins.

To enable the Workbook API (Beta) feature:

1. Go to Setup > Company > Enable Features.
2. Click the **SuiteCloud** subtab.
3. In the **SuiteScript** section, check the **Workbook API (Beta)** box.
4. Read and accept the SuiteCloud terms of service and Oracle Cloud Services Beta license agreement.

Creating a Dataset Using the Workbook API (Beta) Tutorial

 **Note:** The content in this help topic pertains to SuiteScript 2.0.

In this tutorial, you'll learn how to:

- Load the N/dataset module
- Create an initial dataset
- Create dataset components (joins, columns, and conditions)
- Add dataset components to the initial dataset
- Save and run the dataset

You will write your script in a text editor of your choice. When the script is complete, you will use the SuiteScript Debugger to run the script and create the dataset. Finally, you will use the SuiteAnalytics Workbook UI to verify that the dataset was created successfully.

Navigating the Tutorial

This tutorial includes navigation links at the bottom of each page. You can use these links to go forward to the next step in the tutorial or go back to the previous step.

Use the following table to go to a specific step in the tutorial.

| Step | Description |
|--|--|
| Prerequisites | This topic lists the prerequisites for this tutorial, including required NetSuite features and permissions. |
| Step One: Create an initial dataset | This step describes how to create an initial dataset with no components. |
| Step Two: Create joins with other record types | This step describes how to create joins with other record types so you can include more fields in the dataset. |
| Step Three: Create columns | This step describes how to create columns, which represent the fields to include in the dataset. |
| Step Four: Create conditions | This step describes how to create conditions to filter the dataset. |
| Step Five: Add components to the initial dataset | This step describes how to add the joins, conditions, and columns to the dataset. |
| Step Six: Save and run the dataset | This step describes how to save the dataset and run it to obtain the results. |

Related Help Topics

The following help topics provide more information about the concepts discussed in this tutorial:

- [Defining a Dataset](#)
- [Available Record Types](#)
- [Guidelines for Joining Record Types in SuiteAnalytics Workbook](#)
- [Analytics Browser](#)
- [SuiteAnalytics Workbook Tutorial](#)
- [Navigating SuiteAnalytics Workbook](#)
- [SuiteScript Debugger](#)
- [N/dataset Module](#)

Next: [Prerequisites](#)

Prerequisites

 **Note:** The content in this help topic pertains to SuiteScript 2.0.

Before you start this tutorial, review the following prerequisites. If you are not sure whether your NetSuite account or role meet these prerequisites, contact your administrator.

Required Features

The following features must be enabled in your NetSuite account:

- **SuiteAnalytics Workbook** — This feature lets you access SuiteAnalytics Workbook. It is required both to access the SuiteAnalytics Workbook UI and to use the N/dataset and N/workbook modules in a script.
- **Server SuiteScript** — This feature lets you create and run server scripts in SuiteScript.

For more information about enabling features, see the help topic [Enabling Features](#).

Required Roles or Permissions

Your role and role permissions must let you do the following:

- Access the Analytics tab in the NetSuite UI
- View saved datasets and workbooks
- Create new datasets and workbooks
- Run server scripts using the SuiteScript Debugger

For more information about roles and permissions, see the help topic [NetSuite Users & Roles](#).

Previous: [Creating a Dataset Using the Workbook API \(Beta\) Tutorial](#)

Next: [Create an initial dataset](#)

Create an initial dataset

 **Note:** The content in this help topic pertains to SuiteScript 2.0.

Datasets are the basis for all workbooks in your account. In a dataset, you combine the fields of a root record type and any joined related record types to create a query. You can also create custom formula fields to calculate values that are not available through standard record type fields.

In this step, you will create an initial dataset object that uses the customer record type as the root record type.

1. Create a new file in a text editor of your choice. If your text editor is able to provide syntax highlighting based on file type, select JavaScript (.js) file highlighting.
2. Add the following code to load the N/dataset module:

```
1 | require(['N/dataset'], function(dataset) {
2 |
3 |});
```

In this tutorial, you load only the N/dataset module. However, when you work with datasets and workbooks in your production account, you should always load both the N/dataset and N/workbook modules in your scripts. This approach ensures that you can access all workbook-related objects and methods.

3. Create a dataset using [dataset.create\(options\)](#). Use the type parameter to specify the root record type for the dataset. Add the following code inside the require function definition:

```

1 var tutorialDataset = dataset.create({
2   type: 'customer'
3 });

```

This method returns a [dataset.Dataset](#) object. The string `customer` is the name of the customer record type. When you create a dataset, you must specify the name of the root record type to use. To obtain this name (as well as other record and field names), you can use the Analytics Browser. For more information, see the help topic [Analytics Browser](#).

- Add a name and description for the dataset using the `name` and `description` properties of the [dataset.Dataset](#) object. Add the following code after the [dataset.create\(options\)](#) call:

```

1 tutorialDataset.name = 'Tutorial Dataset';
2 tutorialDataset.description = 'This is a tutorial dataset.';

```

You can also set the name and description of the dataset when you create it using [dataset.create\(options\)](#).

At this point, your script file should look similar to the following:

```

1 require(['N/dataset'], function(dataset) {
2   var tutorialDataset = dataset.create({
3     type: 'customer'
4   });
5
6   tutorialDataset.name = 'Tutorial Dataset';
7   tutorialDataset.description = 'This is a tutorial dataset.';
8 });

```

SuiteAnalytics Workbook UI

In the SuiteAnalytics Workbook UI, the root record type for a dataset appears in the info popup beside the dataset name.

The screenshot shows the Oracle NetSuite interface with the 'Analytics' tab selected. A dataset named 'Cheps' Test Dataset' is selected. An info popup is open over the dataset list, showing details about the dataset. The 'Root Record' field is highlighted with a red box and contains the value 'Transaction'. The main pane displays a table of transaction data with columns: DATE, SALES REP, ENTITY, and ENTITY (CUSTOMER) EMAIL. The data rows are:

| DATE | SALES REP | ENTITY | ENTITY (CUSTOMER) EMAIL |
|-----------|------------------|--------|-----------------------------|
| 1/15/2002 | Eric Schmidt | | eschmidt@freeversion.com |
| 1/4/2002 | Amy Boughton | | boughton751@freeversion.com |
| 1/28/2002 | Tony Matsuda | | tonymat123@freeversion.com |
| 2/13/2002 | Phillip Van Hook | | vanhook@freecheckbook.net |

The name and description also appear in the same info popup.

Dataset Name: Cheps' Test Dataset

Dataset Description: This is Cheps' test dataset.

| DATE | SALES REP | ENTITY | ENTITY (CUSTOMER): EMAIL |
|-----------|------------------|-----------------------------|--------------------------|
| 1/15/2002 | Eric Schmidt | eschmidt@freeversion.com | |
| 1/4/2002 | Amy Boughton | boughton751@freeversion.com | |
| 1/28/2002 | Tony Matsuda | tonymat123@freeversion.com | |
| 2/13/2002 | Phillip Van Hook | vanhook@freecheckbook.net | |

Dataset Name: Cheps' Test Dataset

Dataset Description: This is Cheps' test dataset.

| DATE | SALES REP | ENTITY | ENTITY (CUSTOMER): EMAIL |
|-----------|------------------|-----------------------------|--------------------------|
| 1/15/2002 | Eric Schmidt | eschmidt@freeversion.com | |
| 1/4/2002 | Amy Boughton | boughton751@freeversion.com | |
| 1/28/2002 | Tony Matsuda | tonymat123@freeversion.com | |
| 2/13/2002 | Phillip Van Hook | vanhook@freecheckbook.net | |

[Previous: Prerequisites](#)

[Next: Create joins with other record types](#)

Create joins with other record types

Note: The content in this help topic pertains to SuiteScript 2.0.

SuiteAnalytics Workbook enables you to add fields from multiple record types to a single dataset. This includes record types that are more than one join away from the root record type of a dataset, enabling you to compile workbook source data from a diverse set of record types.

In this step, you will create objects to represent joins with other record types from the root record type.

1. In your script file, create a join using [dataset.createJoin\(options\)](#). This join connects the root record type (customer) with the employee record type using the salesrep field. The salesrep field is located on the customer record. Add the following code after you define the dataset name and description in [Create an initial dataset](#):

```
1 var salesRepJoin = dataset.createJoin({
2   fieldId: 'salesrep'
3 });
```

The [dataset.createJoin\(options\)](#) method returns a [dataset.Join](#) object.

2. Create a second join to connect the root record type (customer) with the transaction record type using the entity field. The entity field on the transaction record type refers to the customer associated with that transaction. However, there is no field on the customer record type that refers to the transactions for that customer. To represent this relationship as a join object, you must specify the source parameter as well as the fieldId parameter. The source parameter refers to the record type to join from (transaction), and the fieldId parameter refers to the field on that record type to use for the join (entity). Add the following code after the first [dataset.createJoin\(options\)](#) call:

```
1 var transactionJoin = dataset.createJoin({
2   fieldId: 'entity',
3   source: 'transaction'
4 });
```

At this point, your script file should look similar to the following:

```
1 require(['N/dataset'], function(dataset) {
2   var tutorialDataset = dataset.create({
3     type: 'customer'
4   });
5
6   tutorialDataset.name = 'Tutorial Dataset';
7   tutorialDataset.description = 'This is a tutorial dataset.';
8
9   var salesRepJoin = dataset.createJoin({
10    fieldId: 'salesrep'
11  });
12   var transactionJoin = dataset.createJoin({
13    fieldId: 'entity',
14    source: 'transaction'
15  });
16});
```

SuiteAnalytics Workbook UI

In the SuiteAnalytics Workbook UI, joined fields in a dataset are listed when you click a joined record type in the Records list. They are highlighted and appear at the top of the Fields list.

| TASTIC DATE | ENTITY | ENTITY (CUSTOMER): EMAIL | CONTACTS: ENTITY ID |
| --- | --- | --- | --- |
| 002 | Eric Schmidt | eschmidt@freeversion.com | |
| 02 | Amy Boughton | boughton751@freeversion.com | |
| 002 | Tony Matsuda | tonymat123@freeversion.com | |
| 002 | Phillip Van Hook | vanhook@freecheckbook.net | |

 A note at the bottom says 'Showing 4 rows out of 4 as a statistical sample overview of the entire dataset.'"/>

[Previous: Create an initial dataset](#)

[Next: Create columns](#)

Create columns



Note: The content in this help topic pertains to SuiteScript 2.0.

Columns represent the record fields that are included in the dataset. Column definitions are used to create conditions that apply to the corresponding fields in the dataset.

In this step, you will create objects to represent columns for the dataset.

1. In your script file, create a column using `dataset.createColumn(options)`. This column represents the entityid field on the root customer record type. Add the following code after the two `dataset.createJoin(options)` calls in [Create joins with other record types](#):

```

1 | var entityIdColumn = dataset.createColumn({
2 |   fieldId: 'entityid'
3 | });

```

The `dataset.createColumn(options)` method returns a `dataset.Column` object.

2. Create a second column to represent the email field on the joined employee record type. Add the following code after the first `dataset.createColumn(options)` call:

```

1 | var emailColumn = dataset.createColumn({
2 |   fieldId: 'entityid',
3 |   join: salesRepJoin
4 | });

```

3. Create a third column to represent the trandate field (transaction date) on the joined transaction record type. Add the following code after the second `dataset.createColumn(options)` call:

```

1 | var trandateColumn = dataset.createColumn({
2 |   fieldId: 'trandate',
3 |   join: transactionJoin
4 | });

```

```
4 | }));
```

At this point, your script file should look similar to the following:

```

1 require(['N/dataset'], function(dataset) {
2     var tutorialDataset = dataset.create({
3         type: 'customer'
4     });
5
6     tutorialDataset.name = 'Tutorial Dataset';
7     tutorialDataset.description = 'This is a tutorial dataset.';
8
9     var salesRepJoin = dataset.createJoin({
10        fieldId: 'salesrep'
11    });
12    var transactionJoin = dataset.createJoin({
13        fieldId: 'entity',
14        source: 'transaction'
15    });
16
17    var entityIdColumn = dataset.createColumn({
18        fieldId: 'entityid'
19    });
20    var emailColumn = dataset.createColumn({
21        fieldId: 'entityid',
22        join: salesRepJoin
23    });
24    var trandateColumn = dataset.createColumn({
25        fieldId: 'trandate',
26        join: transactionJoin
27    });
28});
```

SuiteAnalytics Workbook UI

In the SuiteAnalytics Workbook UI, each included column (field) is highlighted at the top of the Fields list. You can also see each field as a column heading in the Data Grid.

| DATE | SALES REP | ENTITY | ENTITY (CUSTOMER): EMAIL |
|-----------|------------------|-----------------------------|--------------------------|
| 1/15/2002 | Eric Schmidt | eschmidt@freeversion.com | |
| 1/4/2002 | Amy Boughton | boughton751@freeversion.com | |
| 1/28/2002 | Tony Matsuda | tonymat123@freeversion.com | |
| 2/13/2002 | Phillip Van Hook | vanhook@freecheckbook.net | |

[Previous: Create joins with other record types](#)

[Next: Create conditions](#)

Create conditions

Note: The content in this help topic pertains to SuiteScript 2.0.

Conditions let you filter the query results in a dataset.

Note: A condition filters entire records and fields from a dataset, which affects all workbooks that are based on the dataset. For example, if you create a condition that filters invoice records from a dataset, workbooks based on that dataset will also not include invoice record data.

In this step, you will create objects to represent conditions for the dataset.

1. In your script file, create a condition using `dataset.createCondition(options)`. This condition filters any customer records that are inactive (that is, the Inactive box in the UI is checked). Because this condition applies to a field (`isinactive`) that does not have a corresponding `dataset.Column` object in your script, you must create a column for the field to use when creating the condition. Add the following code after the three `dataset.createColumn(options)` calls in [Create columns](#):

```

1 var inactiveColumn = dataset.createColumn({
2   fieldId: 'isinactive'
3 });
4 var inactiveCondition = dataset.createCondition({
5   column: inactiveColumn,
6   operator: 'IS',
7   values: [false]
8 });

```

The `dataset.createCondition(options)` method returns a `dataset.Condition` object. For a list of available operators, see the help topic [query.Operator](#).

2. Create a second condition that filters any transaction records with a transaction date on or before March 3, 2020. You already created a `dataset.Column` object for the `trandate` field in [Create columns](#), so you can use that object when creating the condition. Add the following code after the first `dataset.createCondition(options)` call:

```

1 var trandateCondition = dataset.createCondition({
2   column: trandateColumn,
3   operator: 'AFTER',
4   values: ['3/3/2020']
5 });

```

At this point, your script file should look similar to the following:

```

1 require(['N/dataset'], function(dataset) {
2   var tutorialDataset = dataset.create({
3     type: 'customer'
4   });
5
6   tutorialDataset.name = 'Tutorial Dataset';
7   tutorialDataset.description = 'This is a tutorial dataset.';
8
9   var salesRepJoin = dataset.createJoin({
10     fieldId: 'salesrep'
11   });
12   var transactionJoin = dataset.createJoin({
13     fieldId: 'entity',
14     source: 'transaction'
15   });
16
17   var entityIdColumn = dataset.createColumn({
18     fieldId: 'entityid'

```

```

19 });
20 var emailColumn = dataset.createColumn({
21   fieldId: 'entityid',
22   join: salesRepJoin
23 });
24 var trandateColumn = dataset.createColumn({
25   fieldId: 'trandate',
26   join: transactionJoin
27 });
28
29 var inactiveColumn = dataset.createColumn({
30   fieldId: 'isinactive'
31 });
32 var inactiveCondition = dataset.createCondition({
33   column: inactiveColumn,
34   operator: 'IS',
35   values: [false]
36 });
37 var trandateCondition = dataset.createCondition({
38   column: trandateColumn,
39   operator: 'AFTER',
40   values: ['3/3/2020']
41 });
42 });

```

SuiteAnalytics Workbook UI

In the SuiteAnalytics Workbook UI, conditions appear in the Criteria summary area above the Data Grid.

The screenshot shows the SuiteAnalytics Workbook UI interface. On the left, there's a sidebar with a search bar and a list of columns categorized under 'Formulas' and 'Transaction'. The 'Transaction' category is expanded, showing sub-options like 'Transaction Line', 'Billing Address', etc. In the center, the 'Criteria summary' section is highlighted with a red box. It displays two conditions: 'Status is Sales Order : Billed' and 'Entity (Customer): Email contains free', connected by an 'AND' operator. Below this is a 'Data Preview' section with a table showing four rows of sample data:

| DATE | SALES REP | ENTITY | ENTITY (CUSTOMER): EMAIL |
|-----------|-----------|------------------|------------------------------|
| 1/15/2002 | | Eric Schmidt | eschmidt@freeversion.com |
| 1/4/2002 | | Amy Boughton | boughton751@freeversion.com |
| 1/28/2002 | | Tony Matsuda | tonymat123@freecheckbook.net |
| 2/13/2002 | | Phillip Van Hook | vanhook@freecheckbook.net |

At the bottom of the preview, it says 'Showing 4 rows out of 4 as a statistical sample overview of the entire dataset.'

[Previous: Create columns](#)

[Next: Add components to the initial dataset](#)

Add components to the initial dataset

Note: The content in this help topic pertains to SuiteScript 2.0.

After you create your dataset components, you can add them to the dataset you created in [Create an initial dataset](#).

In this step, you will add your dataset components to the initial dataset.

1. In your script file, assign the columns you created to the `Dataset.columns` property. This property accepts an array of `dataset.Column` objects. Add the following code after the two `dataset.createCondition(options)` calls in [Create conditions](#):

```
1 | tutorialDataset.columns = [entityIdColumn, emailColumn, trandateColumn];
```

2. Assign the conditions you created to the `Dataset.condition` property. This property accepts a single `dataset.Condition` object or an array of `dataset.Condition` objects. If you provide an array of objects, the conditions are combined using the AND operator to produce a final condition for the dataset. Add the following code after you assign the columns above:

```
1 | tutorialDataset.condition = [inactiveCondition, trandateCondition];
```

At this point, your script file should look similar to the following:

```
1 | require(['N/dataset'], function(dataset) {
2 |   var tutorialDataset = dataset.create({
3 |     type: 'customer'
4 |   });
5 |
6 |   tutorialDataset.name = 'Tutorial Dataset';
7 |   tutorialDataset.description = 'This is a tutorial dataset.';
8 |
9 |   var salesRepJoin = dataset.createJoin({
10 |     fieldId: 'salesrep'
11 |   });
12 |   var transactionJoin = dataset.createJoin({
13 |     fieldId: 'entity',
14 |     source: 'transaction'
15 |   });
16 |
17 |   var entityIdColumn = dataset.createColumn({
18 |     fieldId: 'entityid'
19 |   });
20 |   var emailColumn = dataset.createColumn({
21 |     fieldId: 'entityid',
22 |     join: salesRepJoin
23 |   });
24 |   var trandateColumn = dataset.createColumn({
25 |     fieldId: 'trandate',
26 |     join: transactionJoin
27 |   });
28 |
29 |   var inactiveColumn = dataset.createColumn({
30 |     fieldId: 'isinactive'
31 |   });
32 |   var inactiveCondition = dataset.createCondition({
33 |     column: inactiveColumn,
34 |     operator: 'IS',
35 |     values: [false]
36 |   });
37 |   var trandateCondition = dataset.createCondition({
38 |     column: trandateColumn,
39 |     operator: 'AFTER',
40 |     values: ['3/3/2020']
41 |   });
42 |
43 |   tutorialDataset.columns = [entityIdColumn, emailColumn, trandateColumn];
44 |   tutorialDataset.condition = [inactiveCondition, trandateCondition];
45 | });
```

[Previous: Create conditions](#)

[Next: Save and run the dataset](#)

Save and run the dataset



Note: The content in this help topic pertains to SuiteScript 2.0.

When you are done creating your dataset, you can save it and run it. You must save the dataset before you can view it in the SuiteAnalytics Workbook UI. When you run the dataset, you receive a query result set that includes all of the rows in the dataset.

In this step, you will save the dataset and run it to obtain the query result set.

1. In your script file, save the dataset using [Dataset.save\(\)](#). Add the following code after you add dataset columns and conditions in [Add components to the initial dataset](#):

```
1 | tutorialDataset.save();
```

2. Run the dataset using [Dataset.run\(\)](#). Add the following code after you save the dataset above:

```
1 | var results = tutorialDataset.run();
```

The [Dataset.run\(\)](#) method returns a [query.ResultSet](#) object (which is located in the N/query module). You can also run the dataset as a paged query using [Dataset.runPaged\(\)](#), which returns a [query.PagedData](#) object (also located in the N/query module).

Your final script file should look similar to the following:

```

1 | require(['N/dataset'], function(dataset) {
2 |   var tutorialDataset = dataset.create({
3 |     type: 'customer'
4 |   });
5 |
6 |   tutorialDataset.name = 'Tutorial Dataset';
7 |   tutorialDataset.description = 'This is a tutorial dataset.';
8 |
9 |   var salesRepJoin = dataset.createJoin({
10 |     fieldId: 'salesrep'
11 |   });
12 |   var transactionJoin = dataset.createJoin({
13 |     fieldId: 'entity',
14 |     source: 'transaction'
15 |   });
16 |
17 |   var entityIdColumn = dataset.createColumn({
18 |     fieldId: 'entityid'
19 |   });
20 |   var emailColumn = dataset.createColumn({
21 |     fieldId: 'entityid',
22 |     join: salesRepJoin
23 |   });
24 |   var trandateColumn = dataset.createColumn({
25 |     fieldId: 'trandate',
26 |     join: transactionJoin
27 |   });
28 |
29 |   var inactiveColumn = dataset.createColumn({
30 |     fieldId: 'isinactive'
31 |   });
32 |   var inactiveCondition = dataset.createCondition({
33 |     column: inactiveColumn,
34 |     operator: 'IS',
35 |     values: [false]
36 |   });
37 |   var trandateCondition = dataset.createCondition({
38 |     column: trandateColumn,
39 |     operator: 'AFTER',

```

```

40     values: ['3/3/2020']
41   });
42
43   tutorialDataset.columns = [entityIdColumn, emailColumn, trandateColumn];
44   tutorialDataset.condition = [inactiveCondition, trandateCondition];
45
46   tutorialDataset.save();
47   var results = tutorialDataset.run();
48 });

```

Testing Your Solution

Typically, you run scripts in NetSuite by associating the script with an entry point. When the entry point is triggered, the associated script runs. For this tutorial, you can test your solution by using the SuiteScript Debugger. You can access the debugger in your NetSuite account and copy the contents of your script file into the debugger. When you run the script, it creates the dataset that you defined in your script. You can verify that the dataset was created successfully by viewing it in the SuiteAnalytics Workbook UI. For more information about the SuiteScript Debugger, see the help topic [SuiteScript Debugger](#).

Previous: [Add components to the initial dataset](#)

Workbook API (Beta) UI Guide

This guide describes how objects and properties in the Workbook API are represented in the SuiteAnalytics Workbook UI.

The following table describes the objects you can explore.

| Objects | Description | Example |
|--------------------------|--|--------------------------------------|
| Dataset Objects | These objects let you define columns and conditions for datasets. You can also define joins with other record types. | dataset.Column |
| Workbook Objects | These objects let you define visualizations (such as table views and pivots) for workbooks. | workbook.Workbook |
| Table Definition Objects | These objects let you define columns and filters for table views. | workbook.TableFilter |

Dataset Objects

A dataset in the N/dataset module corresponds to a dataset in the SuiteAnalytics Workbook UI. A dataset is a set of data that you can define to use as a basis for your workbook visualizations. Datasets let you define your data and workbook visualizations let you analyze the selected data. For more information about datasets in the SuiteAnalytics Workbook UI, see the help topic [Defining a Dataset](#).

To define your dataset, you can do the following:

- View the data of the underlying dataset displayed in columns
- Join record types
- Filter the data

The Workbook API includes the following objects that relate to datasets:

- [dataset.Dataset](#)
- [dataset.Column](#)
- [dataset.Condition](#)
- [dataset.Join](#)

dataset.Dataset

Datasets must contain dataset columns (represented by `dataset.Columns` or `dataset.Joins` objects) and can contain criteria criteria filters (represented by a `dataset.Condition` object).

- [dataset.Dataset](#)
- [Dataset.columns](#)
- [Dataset.condition](#)
- [Dataset.description](#)
- [Dataset.id](#)
- [Dataset.name](#)
- [Dataset.type](#)

dataset.Dataset

This object represents an entire dataset, including conditions, columns, and joins.

You can view a dataset by clicking Analytics, then clicking the Datasets tab. All available datasets are displayed, and you can click a dataset name to view that dataset.

The screenshot shows the NetSuite Analytics interface. At the top, it says "Dataset | Cheps' Test Dataset" and "Employee". Below that is a message: "This dataset is used in 1 Workbooks. Making a change here will affect these workbooks." On the right, there are buttons for "Create New Workbook", "Export", "Share", and "Save".

The main area has two sections: "Criteria summary" and "Data Preview".

Criteria summary: Shows a logical AND condition: "Status AND Entity (Customer): Email". It includes two filters: "Status is Sales Order : Billed" and "Entity (Customer): Email contains free". There is also a "New Group" button.

Data Preview: Shows a table with four rows of data:

| DATE | SALES REP | ENTITY | ENTITY (CUSTOMER): EMAIL |
|-----------|------------------|--------|-----------------------------|
| 1/15/2002 | Eric Schmidt | | eschmidt@freeversion.com |
| 1/4/2002 | Amy Boughton | | boughton751@freeversion.com |
| 1/28/2002 | Tony Matsuda | | tonymat123@freeversion.com |
| 2/13/2002 | Phillip Van Hook | | vanhook@freecheckbook.net |

At the bottom, it says "Showing 4 rows out of 4 as a statistical sample overview of the entire dataset."

Dataset.columns

This property represents the fields (columns) that have been added to the dataset. This property is an array of `dataset.Column` objects, each of which represents one field.

When viewing a dataset, included fields are highlighted at the top of the Fields list. You can also see them as column headings in the Data Grid.

The screenshot shows the NetSuite Dataset interface for 'Cheps' Test Dataset'. On the left, a sidebar lists various fields under 'Transaction'. Several fields are highlighted with a red border: Date, Entity, Sales Rep, Status, Total Amount (Transact...), and Type. In the main area, the 'Criteria summary' section shows a condition: Status AND Entity (Customer): Email. Below it is a 'Data Preview' section with a table showing four rows of data:

| DATE | SALES REP | ENTITY | ENTITY (CUSTOMER): EMAIL |
|-----------|------------------|--------|-----------------------------|
| 1/15/2002 | Eric Schmidt | | eschmidt@freeversion.com |
| 1/4/2002 | Amy Boughton | | boughton751@freeversion.com |
| 1/28/2002 | Tony Matsuda | | tonymat123@freeversion.com |
| 2/13/2002 | Phillip Van Hook | | vanhook@freecheckbook.net |

At the bottom, a note says 'Showing 4 rows out of 4 as a statistical sample overview of the entire dataset.'

Dataset.condition

This property represents the conditions applied to the dataset. This property is a single dataset.Condition object. This object includes a Condition.children property with the individual conditions (or groups of conditions) for the dataset. These conditions are then combined using the appropriate operator (AND, OR) to produce a single dataset.Condition object.

When viewing a dataset, conditions appear in the Criteria summary area above the Data Grid.

The screenshot shows the NetSuite Dataset interface for 'Cheps' Test Dataset'. The 'Criteria summary' section is highlighted with a red border. It displays the condition: Status AND Entity (Customer): Email. Below it is a 'Data Preview' section with a table showing four rows of data:

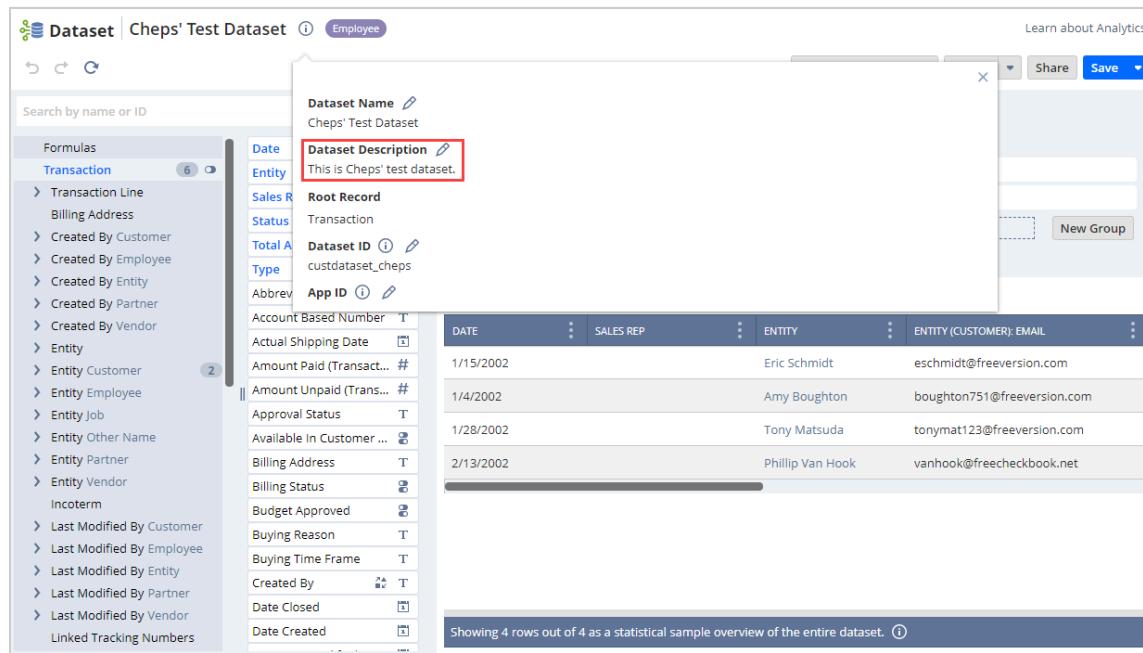
| DATE | SALES REP | ENTITY | ENTITY (CUSTOMER): EMAIL |
|-----------|------------------|--------|-----------------------------|
| 1/15/2002 | Eric Schmidt | | eschmidt@freeversion.com |
| 1/4/2002 | Amy Boughton | | boughton751@freeversion.com |
| 1/28/2002 | Tony Matsuda | | tonymat123@freeversion.com |
| 2/13/2002 | Phillip Van Hook | | vanhook@freecheckbook.net |

At the bottom, a note says 'Showing 4 rows out of 4 as a statistical sample overview of the entire dataset.'

Dataset.description

This property represents the description of the dataset. This property is a string.

When viewing a dataset, the description appears in the info popup beside the dataset name.



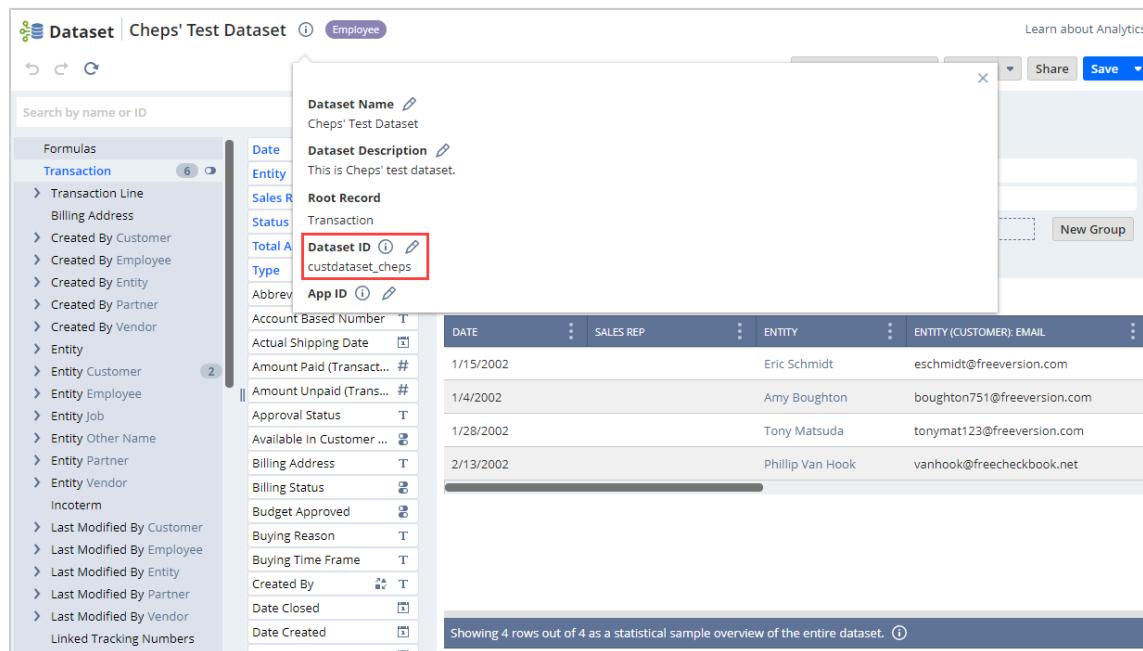
The screenshot shows the NetSuite Studio interface for creating a new dataset named "Cheps' Test Dataset". The "Dataset Description" field is highlighted with a red box and contains the text "This is Cheps' test dataset." Below the description, the "Root Record" is set to "Transaction". The "Dataset ID" is listed as "custdataset_cheps". A preview table shows four rows of transaction data:

| | DATE | SALES REP | ENTITY | ENTITY (CUSTOMER) EMAIL |
|----------------------------|-----------|------------------|--------|-----------------------------|
| Actual Shipping Date | 1/15/2002 | Eric Schmidt | | eschmidt@freeversion.com |
| Amount Paid (Transact... # | 1/4/2002 | Amy Boughton | | boughton751@freeversion.com |
| Approval Status | 1/28/2002 | Tony Matsuda | | tonymat123@freeversion.com |
| Billing Address | 2/13/2002 | Phillip Van Hook | | vanhook@freecheckbook.net |

Dataset.id

This property represents the ID of the dataset. This property is a string. This ID always starts with 'custdataset'. This ID is used when loading the dataset using `dataset.load(options)`.

When viewing a dataset, the ID appears in the info popup beside the dataset name.



The screenshot shows the NetSuite Studio interface for creating a new dataset named "Cheps' Test Dataset". The "Dataset ID" field is highlighted with a red box and contains the value "custdataset_cheps". Below the ID, the "Root Record" is set to "Transaction". The "Dataset Name" is listed as "Cheps' Test Dataset". A preview table shows four rows of transaction data:

| | DATE | SALES REP | ENTITY | ENTITY (CUSTOMER) EMAIL |
|----------------------------|-----------|------------------|--------|-----------------------------|
| Actual Shipping Date | 1/15/2002 | Eric Schmidt | | eschmidt@freeversion.com |
| Amount Paid (Transact... # | 1/4/2002 | Amy Boughton | | boughton751@freeversion.com |
| Approval Status | 1/28/2002 | Tony Matsuda | | tonymat123@freeversion.com |
| Billing Address | 2/13/2002 | Phillip Van Hook | | vanhook@freecheckbook.net |

Dataset.name

This property represents the name of the dataset. This property is a string.

When viewing a dataset, the name appears in the info popup beside the dataset name.

The screenshot shows the Oracle NetSuite Analytics workspace. A modal window titled 'Dataset' is open, showing details for 'Cheps' Test Dataset'. The 'Dataset Name' field is highlighted with a red box and contains 'Cheps' Test Dataset'. The 'Root Record' field is also highlighted with a red box and contains 'Transaction'. The modal also displays a description: 'This is Cheps' test dataset.', a table of sample data, and a message at the bottom: 'Showing 4 rows out of 4 as a statistical sample overview of the entire dataset.'

Dataset.type

This property represents the type of the root record used to create the dataset. This property is a string.

When viewing a dataset, the type appears in the info popup beside the dataset name.

The screenshot shows the Oracle NetSuite Analytics workspace. A modal window titled 'Dataset' is open, showing details for 'Cheps' Test Dataset'. The 'Root Record' field is highlighted with a red box and contains 'Transaction'. The modal also displays a description: 'This is Cheps' test dataset.', a table of sample data, and a message at the bottom: 'Showing 4 rows out of 4 as a statistical sample overview of the entire dataset.'

dataset.Column

Datasets columns correspond to the fields (columns) that you can add to the Data Grid in the Workbook UI. There are three types of columns that you can use to define your dataset:

- Field - Default fields (columns) based on the selected root record type
- Formula field - Fields (columns) that are defined using a formula
- Joined field - Fields (columns) that appear on a joined record type

You can define all of these column types using the `dataset.Column` object.

- `dataset.Column`
- `Column.alias`
- `Column.fieldId`
- `Column.formula`
- `Column.join`
- `Column.label`
- `Column.type`

dataset.Column

This object represents a single field (column) that has been added to the dataset.

When viewing a dataset, each included field is highlighted at the top of the Fields list. You can also see each field as a column heading in the Data Grid.

| DATE | SALES REP | ENTITY | ENTITY (CUSTOMER) EMAIL |
|-----------|------------------|--------|-----------------------------|
| 1/15/2002 | Eric Schmidt | | eschmidt@freeversion.com |
| 1/4/2002 | Amy Boughton | | boughton751@freeversion.com |
| 1/28/2002 | Tony Matsuda | | tonymat123@freeversion.com |
| 2/13/2002 | Phillip Van Hook | | vanhook@freecheckbook.net |

Column.alias

This property represents the alias of the column. This property is a string. You can use this property to get an expression for the column, which you can use in workbook visualizations.

Column.fieldId

This property represents the ID of the field that this column applies to. This property is a string.

When viewing a dataset, the field ID appears in the info popup beside the field name in the Fields list.

The screenshot shows the NetSuite Studio interface. On the left, the 'Fields' list is open, showing various fields like Date, Entity, Sales Rep, etc. A specific field, 'Date', is selected and highlighted with a red box. An info pop-up for this field displays its 'Field ID' as 'trandate'. On the right, the 'Criteria summary' section is visible, showing a query with two criteria: 'Status is Sales Order : Billed' and 'Entity (Customer): Email contains free'. Below the criteria is a 'Data Preview' table with four rows of sample data:

| FANTASTIC DATE | SALES REP | ENTITY | ENTITY (CUSTOMER): EMAIL |
|----------------|------------------|-----------------------------|--------------------------|
| 1/15/2002 | Eric Schmidt | eschmidt@freeversion.com | |
| 1/4/2002 | Amy Boughton | boughton751@freeversion.com | |
| 1/28/2002 | Tony Matsuda | tonymat123@freeversion.com | |
| 2/13/2002 | Phillip Van Hook | vanhook@freecheckbook.net | |

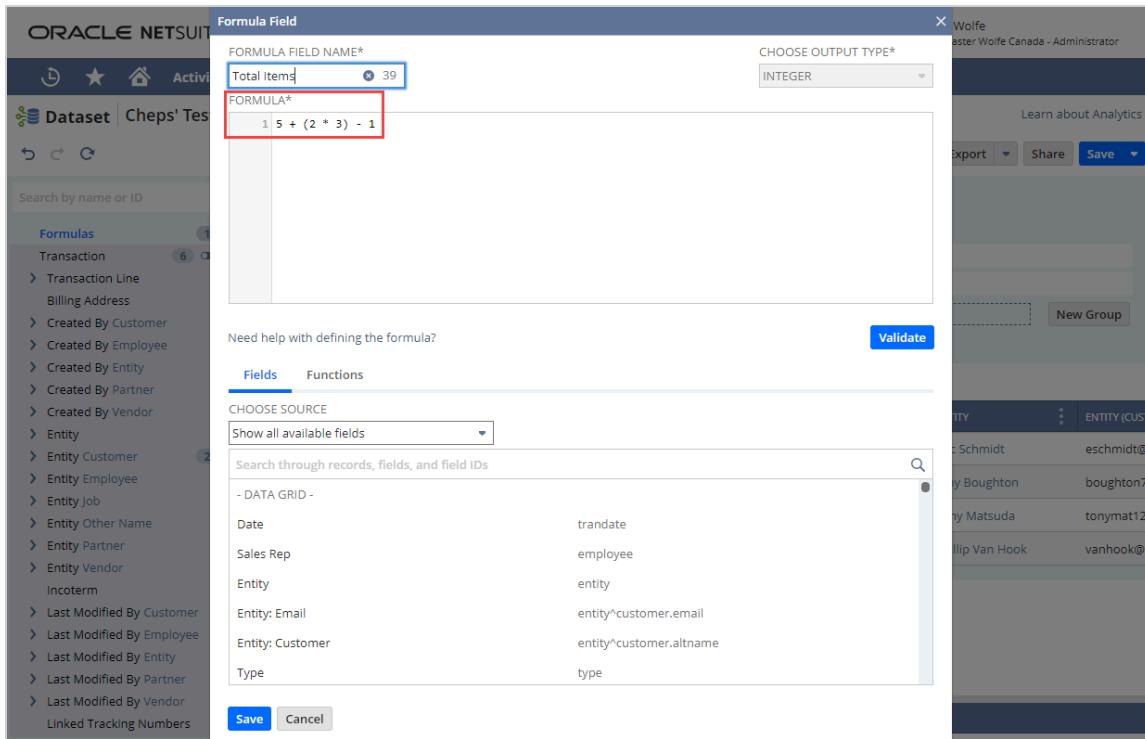
Column.formula

This property represents the formula used for the column. This property is a string. This property is populated only if the field/column is defined using a formula and the Column.fieldId property is not used.

When viewing a dataset, formula fields appear when you click Formulas at the top of the Records list. You can then click the pencil/edit icon beside the formula field in the Fields list to see the formula.

The screenshot shows the NetSuite Studio interface. On the left, the 'Fields' list is open, and the 'Formulas' section is selected and highlighted with a red box. A new formula button ('New Formula') is also highlighted with a red box. On the right, the 'Criteria summary' section is visible, showing a query with two criteria: 'Status is Sales Order : Billed' and 'Entity (Customer): Email contains free'. Below the criteria is a 'Data Preview' table with four rows of sample data:

| MY FANTASTIC DATE | TOTAL ITEMS | SALES REP | ENTITY | ENTITY (CUS |
|-------------------|-------------|------------------|-----------|-------------|
| 1/15/2002 | 10 | Eric Schmidt | eschmidt@ | |
| 1/4/2002 | 10 | Amy Boughton | boughton7 | |
| 1/28/2002 | 10 | Tony Matsuda | tonymat12 | |
| 2/13/2002 | 10 | Phillip Van Hook | vanhook@ | |



Column.join

This property represents the join for the column. This property is a dataset.Join object, and it contains information about the join target record. This property is populated only for fields that appear on a joined record type.

When viewing a dataset, joined fields are listed when you click a joined record type in the Records list. They are highlighted and appear at the top of the Fields list.

The screenshot shows the Oracle NetSuite Dataset interface for 'Cheps' Test Dataset'. The 'Employee' record type is selected. In the 'Fields' list on the left, the 'Email' field is highlighted with a red box. The 'Criteria summary' section shows a join condition for 'Email': 'Status is Sales Order : Billed' AND 'Entity (Customer): Email contains free'. The 'Data Preview' section shows a table with columns: MY FANTASTIC DATE, TOTAL ITEMS, SALES REP, ENTITY, and ENTITY (CUS). The table data is as follows:

| MY FANTASTIC DATE | TOTAL ITEMS | SALES REP | ENTITY | ENTITY (CUS) |
|-------------------|-------------|------------------|-----------|--------------|
| 1/15/2002 | 10 | Eric Schmidt | eschmidt@ | |
| 1/4/2002 | 10 | Amy Boughton | boughton7 | |
| 1/28/2002 | 10 | Tony Matsuda | tonymat12 | |
| 2/13/2002 | 10 | Phillip Van Hook | vanhook@ | |

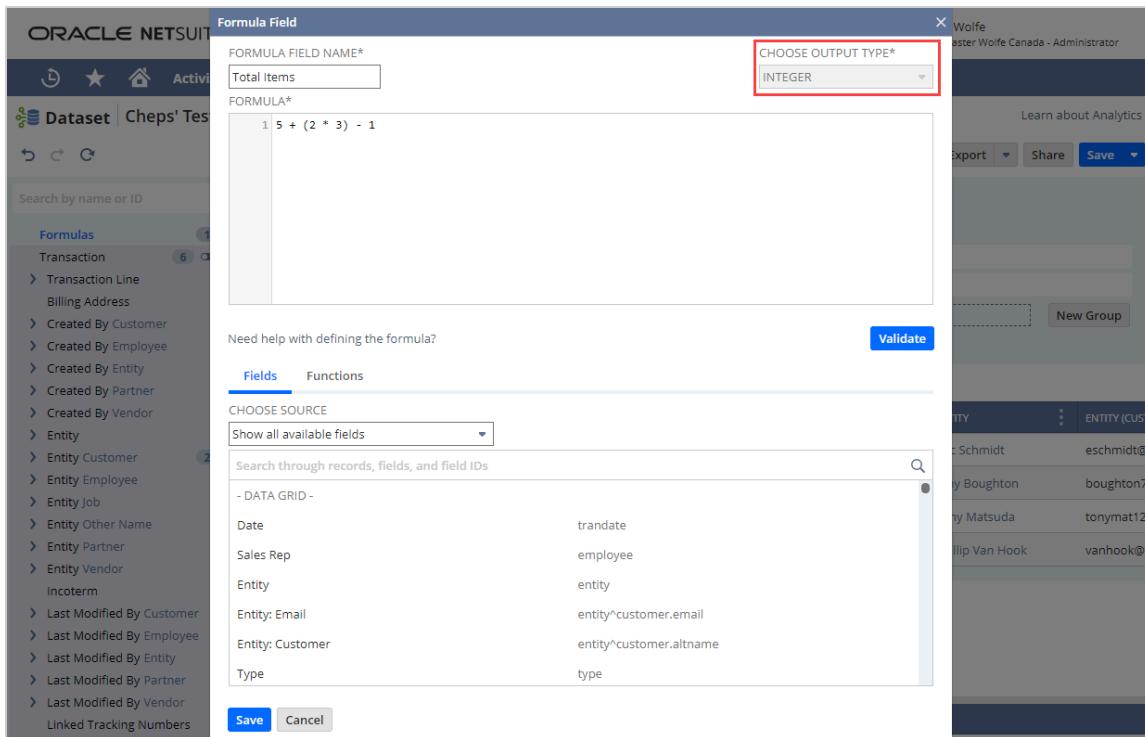
Column.label

This property represents the label for the column. This property is a string.

Column.type

This property represents the return type of the formula used for this column. This property is a string. This property is populated only if the field/column is defined using a formula and the Column.fieldId property is not used.

When viewing a dataset, formula fields appear when you click Formulas at the top of the Records list. You can then click the pencil/edit icon beside the formula field in the Fields list to see the formula.



dataset.Condition

Datasets conditions correspond to the criteria filters that apply to your dataset in the Workbook UI. Criteria filters appear on the Criteria Builder above the Data Grid in the Workbook UI. The criteria that you define in your dataset impacts all workbook visualizations that are based on the dataset.

- [dataset.Condition](#)
- [Condition.children](#)
- [Condition.column](#)
- [Condition.operator](#)
- [Condition.values](#)

dataset.Condition

This object represents the conditions applied to a dataset. This object includes a Condition.children property with the individual conditions (or groups of conditions) for the dataset. These conditions are then combined using the appropriate operator (AND, OR) to produce a single dataset.Condition object for the entire dataset.

When viewing a dataset, conditions appear in the Criteria summary area above the Data Grid.

Criteria summary Show details
Status • AND • Entity (Customer): Email

- :: Status is Sales Order : Billed
- :: Entity (Customer): Email contains free

AND

Reset Criteria

Data Preview

| DATE | SALES REP | ENTITY | ENTITY (CUSTOMER): EMAIL |
|-----------|------------------|-----------------------------|--------------------------|
| 1/15/2002 | Eric Schmidt | eschmidt@freeversion.com | |
| 1/4/2002 | Amy Boughton | boughton751@freeversion.com | |
| 1/28/2002 | Tony Matsuda | tonymat123@freeversion.com | |
| 2/13/2002 | Phillip Van Hook | vanhook@freecheckbook.net | |

Showing 4 rows out of 4 as a statistical sample overview of the entire dataset. ⓘ

Condition.children

This property represents the child conditions to be combined into a single condition. This property is an array of dataset.Condition objects. Conditions can be grouped. A group of conditions is represented as a single Condition object that combines the child conditions in the group using the AND operator. This Condition object can then be used as one of the children of a higher-level condition.

When viewing a dataset, child conditions appear as individual entries in the Criteria summary area above the Data Grid.

Criteria summary Show details
Status • AND • Entity (Customer): Email

- AND
- :: Status is Sales Order : Billed
- :: Entity (Customer): Email contains free

Reset Criteria

Data Preview

| MY FANTASTIC DATE | TOTAL ITEMS | SALES REP | ENTITY | ENTITY (CUS) |
|-------------------|-------------|------------------|-----------|--------------|
| 1/15/2002 | 10 | Eric Schmidt | eschmidt@ | |
| 1/4/2002 | 10 | Amy Boughton | boughton7 | |
| 1/28/2002 | 10 | Tony Matsuda | tonymat12 | |
| 2/13/2002 | 10 | Phillip Van Hook | vanhook@ | |

Showing 4 rows out of 4 as a statistical sample overview of the entire dataset. ⓘ

Condition.column

This property represents the column (field) that the condition applies to. This property is a `dataset.Column` object. For the top-level condition in a dataset (the single condition that child conditions are combined into), this property is always null. Child conditions are implicitly combined using the appropriate operator (AND, OR) to produce a single condition, so no column needs to be specified.

When viewing a dataset, the column (field) appears in the Criteria summary area, in each entry that represents a condition. For fields in joined records, the joined record type is added before the column (field) name.

The screenshot shows the NetSuite Dataset interface for 'Cheps' Test Dataset'. On the left, there's a sidebar with a search bar and a list of fields under 'Transaction'. The main area has two sections: 'Criteria summary' and 'Data Preview'.

Criteria summary: Shows a query structure with 'Status AND Entity (Customer): Email'. The 'Status' condition is expanded to show 'Sales Order : Billed'. The 'Entity (Customer)' condition is expanded to show 'Email contains free'. There's also an 'AND' button and a 'Reset Criteria' button.

Data Preview: Shows a table with four columns: DATE, SALES REP, ENTITY, and ENTITY (CUSTOMER): EMAIL. The data includes:

| DATE | SALES REP | ENTITY | ENTITY (CUSTOMER): EMAIL |
|-----------|-----------|------------------|-----------------------------|
| 1/15/2002 | | Eric Schmidt | eschmidt@freeversion.com |
| 1/4/2002 | | Amy Boughton | boughton751@freeversion.com |
| 1/28/2002 | | Tony Matsuda | tonymat123@freeversion.com |
| 2/13/2002 | | Phillip Van Hook | vanhook@freecheckbook.net |

At the bottom, it says 'Showing 4 rows out of 4 as a statistical sample overview of the entire dataset.'

You can also click the three-dots menu beside a condition entry and click Show Location in Field List. The column (field) for that condition is selected in the Fields list (with the appropriate joined record type selected in the Records list, if the condition applies to a column (field) on a joined record type).

Criteria summary Show details
Status • AND • Entity (Customer): Email

| | Status is Sales Order : Billed | Entity (Customer): Email contains free | Show Location in Field List |
|-----|-----------------------------------|--|-----------------------------|
| AND | | | |
| | Drop fields here to add criteria. | | |
| | New Group | | |

Data Preview [Data Preview](#)

| MY FANTASTIC DATE | TOTAL ITEMS | SALES REP | ENTITY | ENTITY (CUS) |
|-------------------|-------------|------------------|-----------|--------------|
| 1/15/2002 | 10 | Eric Schmidt | eschmidt@ | |
| 1/4/2002 | 10 | Amy Boughton | boughton7 | |
| 1/28/2002 | 10 | Tony Matsuda | tonymat12 | |
| 2/13/2002 | 10 | Phillip Van Hook | vanhook@ | |

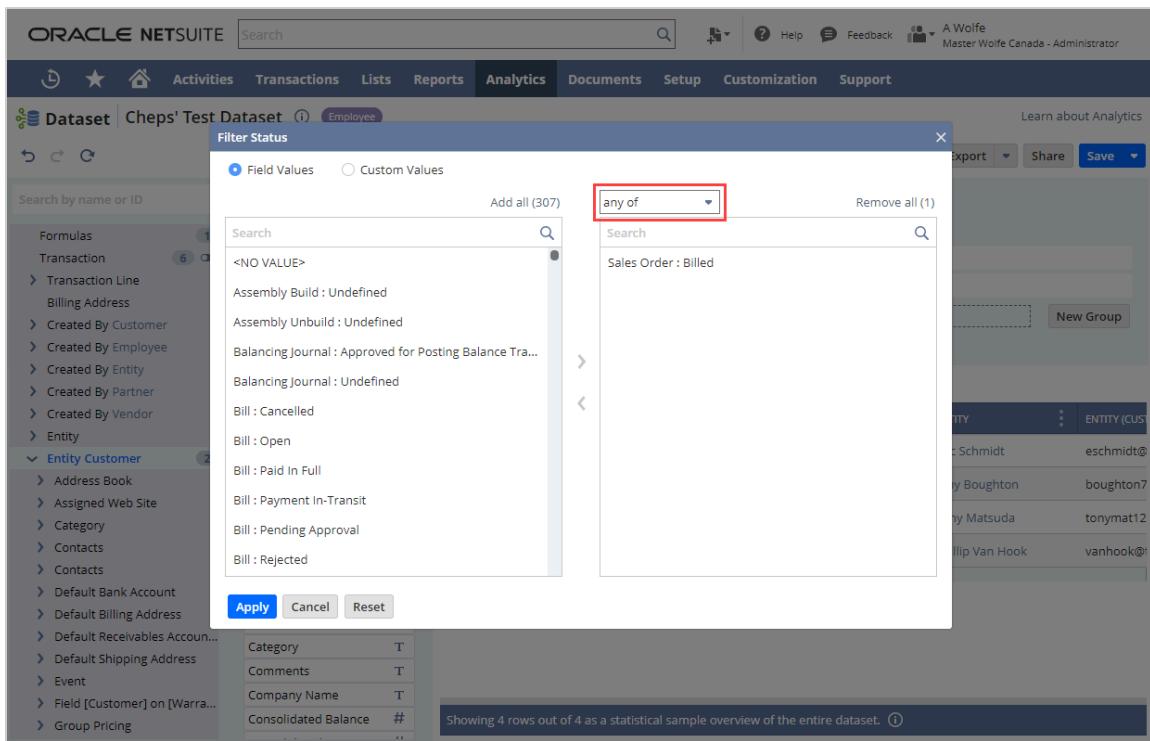
Showing 4 rows out of 4 as a statistical sample overview of the entire dataset. [View All](#)

Condition.operator

This property represents the operator used to test the condition. This property is a string. For the top-level condition in a dataset (the single condition that child conditions are combined into), this property is always either AND or OR. The top-level condition can only combine child conditions using these operators, not more advanced ones (like NOT or ANY_OF).

In the Criteria summary area, click a condition entry. The Filter window opens and shows the operator that is used for the condition, as well as the values, on the right.

The available operators are different depending on the column (field) type that's used in the condition.

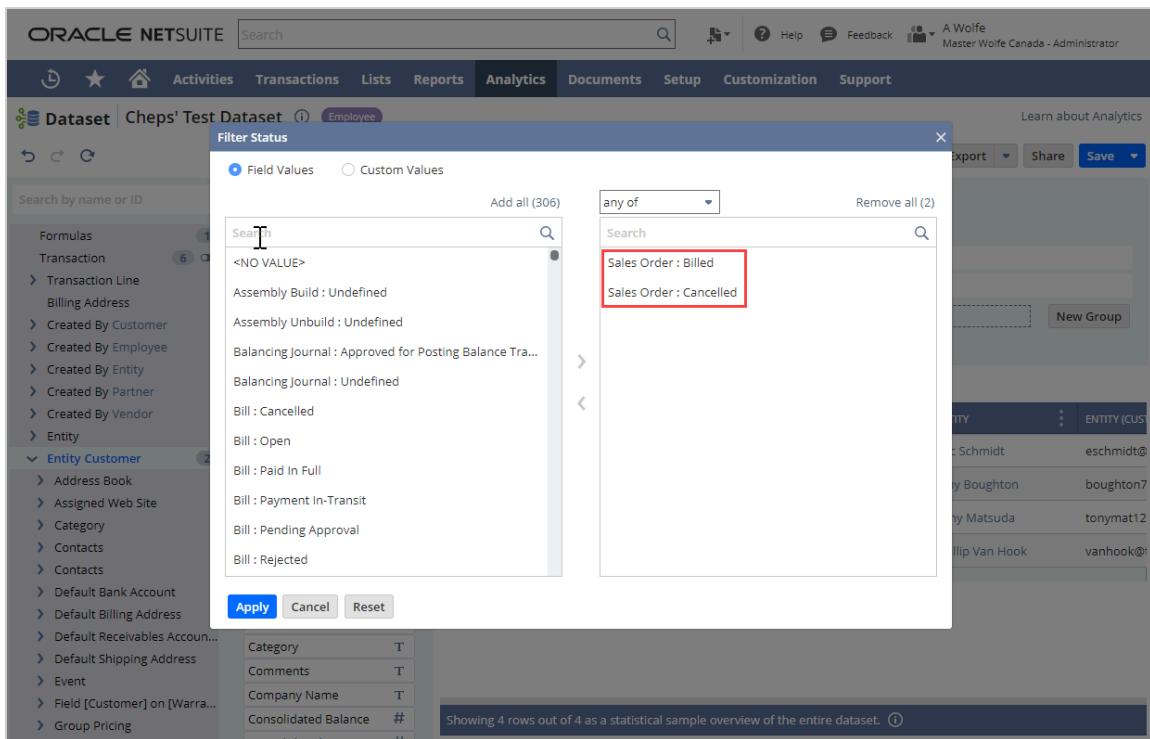


Condition.values

This property represents the set of values used to test the condition. This property is an array of a single type, but the type can vary depending on the condition being tested and the operator used (for example, string[], number[], and so on). For the top-level condition in a dataset (the single condition that child conditions are combined into), this property is always an empty array. Child conditions are implicitly combined using the appropriate operator (AND, OR) to produce a single condition, so no values need to be specified.

In the Criteria summary area, click a condition entry. The Filter window opens and shows the values that are used for the condition, as well as the operator, on the right. Available values are listed on the left.

The available values are different depending on the column (field) type that's used in the condition.



dataset.Join

Dataset joins correspond to joined fields of the dataset in the Workbook UI. Joined fields are fields (columns) that appear on a joined record type and that you can add to the Data Grid. For more information about joined fields in the Workbook UI, see the help topic [Joining Record Types in a Dataset](#).

- [dataset.Join](#)
- [Join.fieldId](#)
- [Join.join](#)
- [Join.source](#)
- [Join.target](#)

dataset.Join

This object represents information about a field on a joined record type.

When viewing a dataset, joined fields are listed when you click a joined record type in the Records list. They are highlighted and appear at the top of the Fields list.

In the Records list, you can click the toggle icon beside the root record type name to see only those record types that have already been joined to the root record type, which can make it easier to visualize multilevel joins.

Dataset | Cheps' Test Dataset | Employee

Search by name or ID

| | Entity ID | T |
|----|--------------------------|-----|
| | Alt. Email | T |
| | Phone | T |
| | Assistant | T |
| | Category | T |
| | Comments | T |
| | Company | # T |
| | Contact is Inactive | T |
| | Date Created | T |
| | Default Billing Address | T |
| | Default Shipping Address | T |
| | Email | T |
| | External ID | T |
| | Fax | T |
| | First Name | T |
| | Home Phone | T |
| | Internal ID | # |
| | Job Title | T |
| | Last Modified Date | T |
| | Last Name | T |
| | Lead Source | # T |
| 1 | Entity ID | T |
| 2 | Alt. Email | T |
| 3 | Phone | T |
| 4 | Assistant | T |
| 5 | Category | T |
| 6 | Comments | T |
| 7 | Company | # T |
| 8 | Contact is Inactive | T |
| 9 | Date Created | T |
| 10 | Default Billing Address | T |
| 11 | Default Shipping Address | T |
| 12 | Email | T |
| 13 | External ID | T |
| 14 | Fax | T |
| 15 | First Name | T |
| 16 | Home Phone | T |
| 17 | Internal ID | # |
| 18 | Job Title | T |
| 19 | Last Modified Date | T |
| 20 | Last Name | T |
| 21 | Lead Source | # T |

Criteria summary Show details

Status AND Entity (Customer): Email

- :: Status is Sales Order : Billed
- :: Entity (Customer): Email contains free

AND

Data Preview

| TASTIC DATE | ENTITY | ENTITY (CUSTOMER): EMAIL | CONTACTS: ENTITY ID |
|-------------|------------------|-----------------------------|---------------------|
| 002 | Eric Schmidt | eschmidt@freeversion.com | |
| 02 | Amy Boughton | boughton751@freeversion.com | |
| 002 | Tony Matsuda | tonymat123@freeversion.com | |
| 002 | Phillip Van Hook | vanhook@freecheckbook.net | |

Showing 4 rows out of 4 as a statistical sample overview of the entire dataset.

Dataset | Cheps' Test Dataset | Employee

Search by name or ID

| | Entity ID | T |
|----|--------------------------|-----|
| | Alt. Email | T |
| | Assist. Phone | T |
| | Assistant | T |
| | Category | T |
| | Comments | T |
| | Company | # T |
| | Contact is Inactive | T |
| | Date Created | T |
| | Default Billing Address | T |
| | Default Shipping Address | T |
| | Email | T |
| | External ID | T |
| | Fax | T |
| | First Name | T |
| | Home Phone | T |
| | Internal ID | # |
| | Job Title | T |
| | Last Modified Date | T |
| | Last Name | T |
| | Lead Source | # T |
| 1 | Entity ID | T |
| 2 | Alt. Email | T |
| 3 | Phone | T |
| 4 | Assistant | T |
| 5 | Category | T |
| 6 | Comments | T |
| 7 | Company | # T |
| 8 | Contact is Inactive | T |
| 9 | Date Created | T |
| 10 | Default Billing Address | T |
| 11 | Default Shipping Address | T |
| 12 | Email | T |
| 13 | External ID | T |
| 14 | Fax | T |
| 15 | First Name | T |
| 16 | Home Phone | T |
| 17 | Internal ID | # |
| 18 | Job Title | T |
| 19 | Last Modified Date | T |
| 20 | Last Name | T |
| 21 | Lead Source | # T |

Criteria summary Show details

Status AND Entity (Customer): Email

- :: Status is Sales Order : Billed
- :: Entity (Customer): Email contains free

AND

Data Preview

| TASTIC DATE | ENTITY | ENTITY (CUSTOMER): EMAIL | CONTACTS: ENTITY ID |
|-------------|------------------|-----------------------------|---------------------|
| 002 | Eric Schmidt | eschmidt@freeversion.com | |
| 02 | Amy Boughton | boughton751@freeversion.com | |
| 002 | Tony Matsuda | tonymat123@freeversion.com | |
| 002 | Phillip Van Hook | vanhook@freecheckbook.net | |

Showing 4 rows out of 4 as a statistical sample overview of the entire dataset.

Join.fieldId

This property represents the field on the previous record in the join trail. This property is a string. When you perform a join, you must specify the field to be used to perform the join. For example, consider a transaction record. Transaction records can be joined to customer records using the entity field on the transaction record. If you joined a transaction record to a customer record in this way, the value of this property is 'entity'. You can think of this field as representing how the current join (this dataset.Join object) relates to the previous record in the join trail.

When viewing a dataset, the full joined field ID appears in the info popup beside the field name in the Fields list. The full joined field ID indicates how the field relates to other fields and record types in the join.

This property is the first part of the full joined field ID.

Dataset | Cheps' Test Dataset | Employee

Criteria summary Show details
Status AND Entity (Customer): Email

Email

Field ID
entity/customer.email

PROPERTIES
T Text

FIELD LEVEL HELP
Enter your customer's email address.

If you allow your customers to access their accounts online, this becomes part of their access codes.

| | 1/28/2002 | Tony Matsuda | tonymat123@freeversion.com |
|--|-----------|------------------|----------------------------|
| | 2/13/2002 | Phillip Van Hook | vanhook@freecheckbook.net |

Showing 4 rows out of 4 as a statistical sample overview of the entire dataset.

Join.join

This property represents the child join of this join, if one exists. This property is a `dataset.Join` object. This property is populated only if a child join exists with the current join as its parent.

When viewing a dataset, joined fields are listed when you click a joined record type in the Records list. They are highlighted and appear at the top of the Fields list.

When displaying only used records in the Records list, you can think of the hierarchy as the join trail for the fields in the dataset. Lower-level joined records are represented by `dataset.Join` objects, which themselves can be used as the value of the `Join.join` property on higher-level joined records.

Dataset | Cheps' Test Dataset | Employee

Criteria summary Show details
Status AND Entity (Customer): Email

Entity ID

Field ID
entity_id

PROPERTIES
T Text

AND
Status is Sales Order : Billed

Reset Criteria

Data Preview

| MY FANTASTIC DATE | ENTITY | ENTITY (CUSTOMER): EMAIL | CONTACTS: ENTITY ID |
|-------------------|------------------|-----------------------------|---------------------|
| 1/15/2002 | Eric Schmidt | eschmidt@freeversion.com | |
| 1/4/2002 | Amy Boughton | boughton751@freeversion.com | |
| 1/28/2002 | Tony Matsuda | tonymat123@freeversion.com | |
| 2/13/2002 | Phillip Van Hook | vanhook@freecheckbook.net | |

Showing 4 rows out of 4 as a statistical sample overview of the entire dataset.

Join.source

This property represents the source record type of this join. This property is a string.

Join.target

This property represents the target record of this join. This property is a string. When you perform a join, you must specify the field to be used to perform the join. You may also need to specify the record type of the record you're joining to. You need to specify this type if the record you're joining to is a general record type instead of a concrete one. General record types are those that have subtypes with associated forms in NetSuite.

For example, consider a transaction record. Transaction records can be joined to customer records using the entity field on the transaction record. However, transaction records can also be joined to employee records, which share the same general record type as customer records (entity). If you joined a transaction record to a customer record in this way, the value of this property is 'customer'.

When viewing a dataset, the full joined field ID appears in the info popup beside the field name in the Fields list. The full joined field ID indicates how the field relates to other fields and record types in the join.

This property is the part of the full joined field ID immediately after the ^ character.

| Date | Customer | Email |
|-----------|------------------|----------------------------|
| 1/28/2002 | Tony Matsuda | tonymat123@freeversion.com |
| 2/13/2002 | Phillip Van Hook | vanhook@freecheckbook.net |

Workbook Objects

A workbook in the N/workbook module corresponds to a workbook in the SuiteAnalytics Workbook UI. A workbook is the UI component where you define workbook visualizations (charts, pivot tables, and table views) using the data of the underlying dataset. Workbook visualizations let you analyze your data and datasets let you define the set of data that you want to analyze. For more information about workbooks in the SuiteAnalytics Workbook UI, see the help topic [Creating a Workbook](#).

The Workbook API includes the following objects that relate to workbooks:

- [workbook.Workbook](#)

workbook.Workbook

Workbooks must contain at least a workbook visualization: a chart (represented by a Workbook.chartDefinitions object), a pivot table (represented by a Workbook.pivotDefinitions object), or a table view (represented by a Workbook.tableDefinitions object).

- [workbook.Workbook](#)
- [Workbook.chartDefinitions](#)
- [Workbook.description](#)
- [Workbook.id](#)
- [Workbook.name](#)
- [Workbook.pivotDefinitions](#)
- [Workbook.tableDefinitions](#)

workbook.Workbook

This object represents an entire workbook, including table views, charts, and pivots.

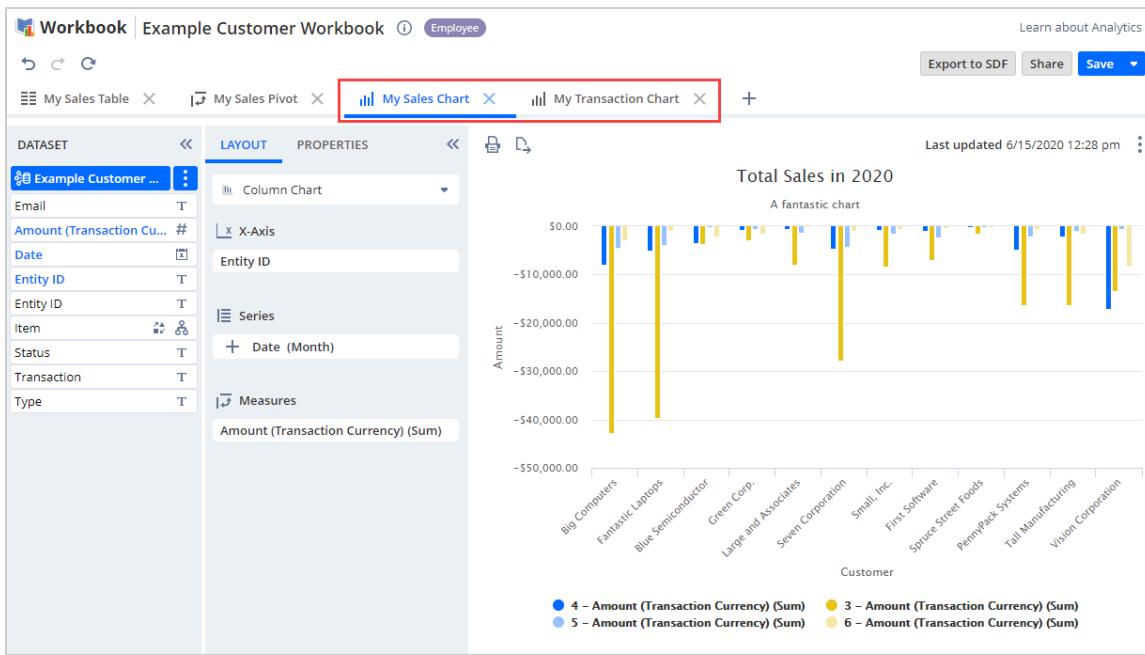
You can view a workbook by clicking Analytics, then clicking the Workbooks tab. All available workbooks are displayed, and you can click a workbook name to view that workbook.

| ENTITY ID | EMAIL | ENTITY ID | DATE | TRANSACTION | TYPE |
|---------------|--------------------------|-------------|-----------|-----------------|-------------|
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 5/13/2020 | Sales Order #14 | Sales Order |

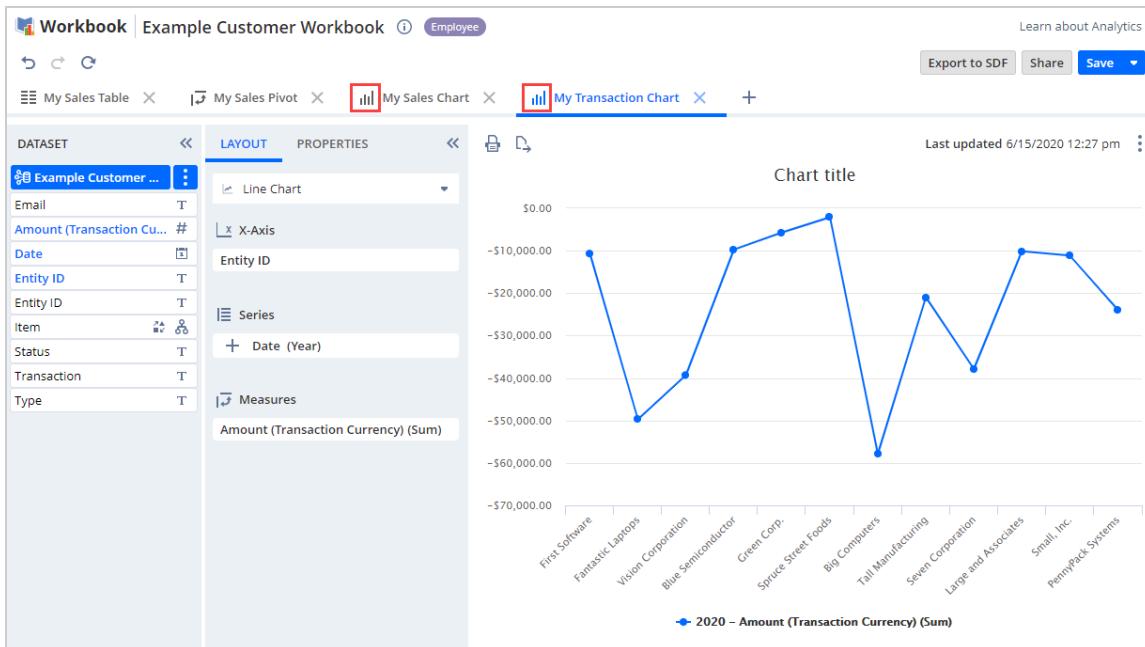
Workbook.chartDefinitions

This property represents the chart definitions included in the workbook. This property is an array of `workbook.ChartDefinition` objects. Each chart definition represents a chart in the workbook. A workbook can have multiple chart definitions.

When viewing a workbook, each chart has its own tab near the top of the workbook page.



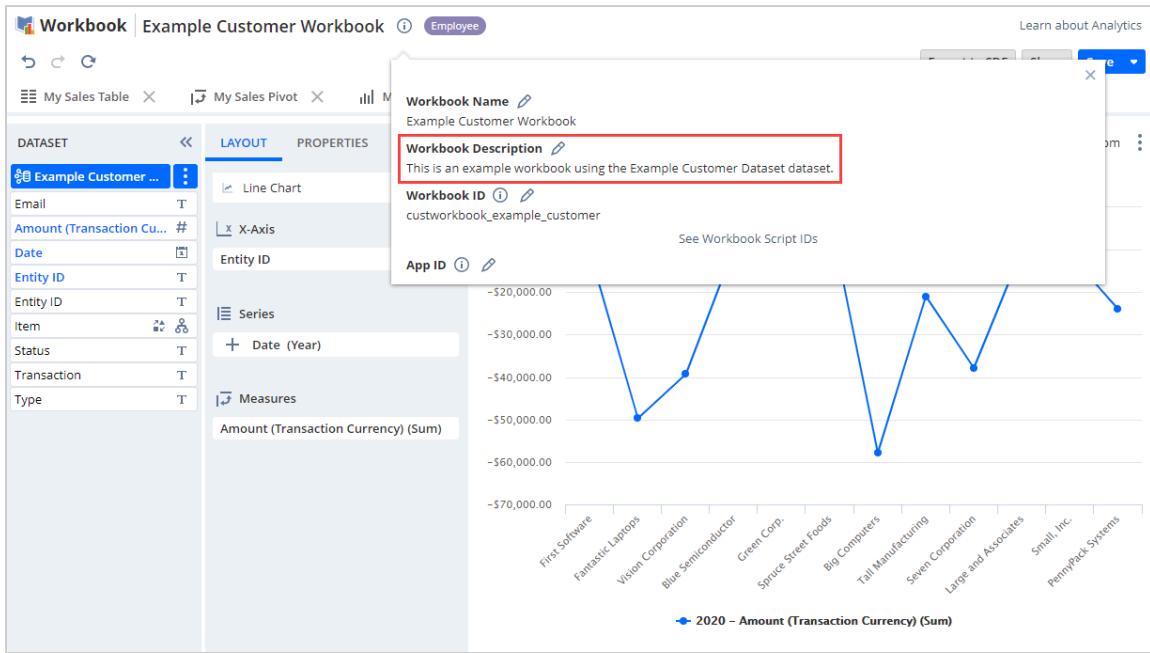
In a workbook, a tab can represent a table view, pivot, or chart. Chart tabs include a bar chart icon beside the tab name.



Workbook.description

This property represents the workbook description. This property is a string.

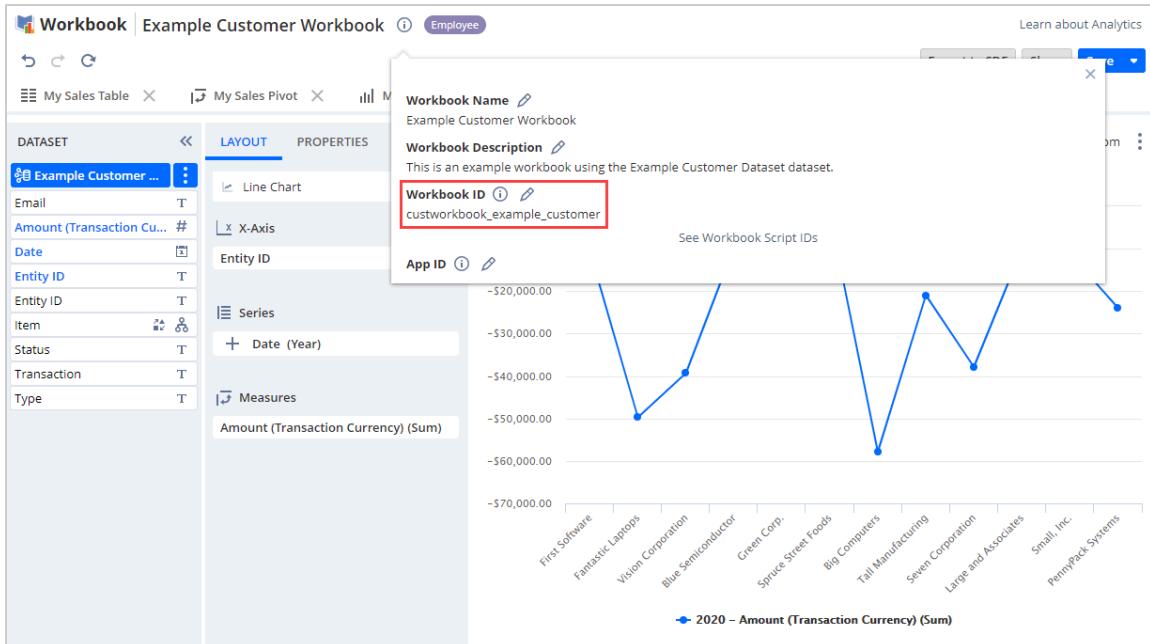
When viewing a workbook, the description appears in the info popup beside the workbook name.



Workbook.id

This property represents the workbook ID. This property is a string. This ID always starts with 'custworkbook'. This ID is used when loading the workbook using `workbook.load(options)`.

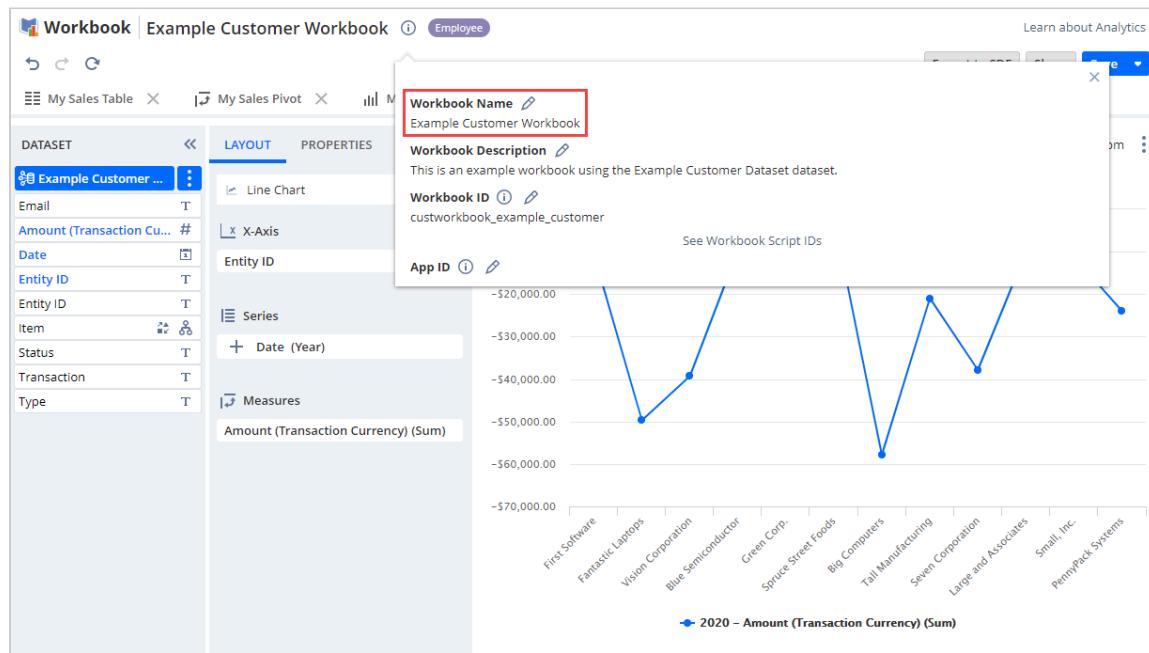
When viewing a workbook, the ID appears in the info popup beside the workbook name.



Workbook.name

This property represents the workbook name. This property is a string.

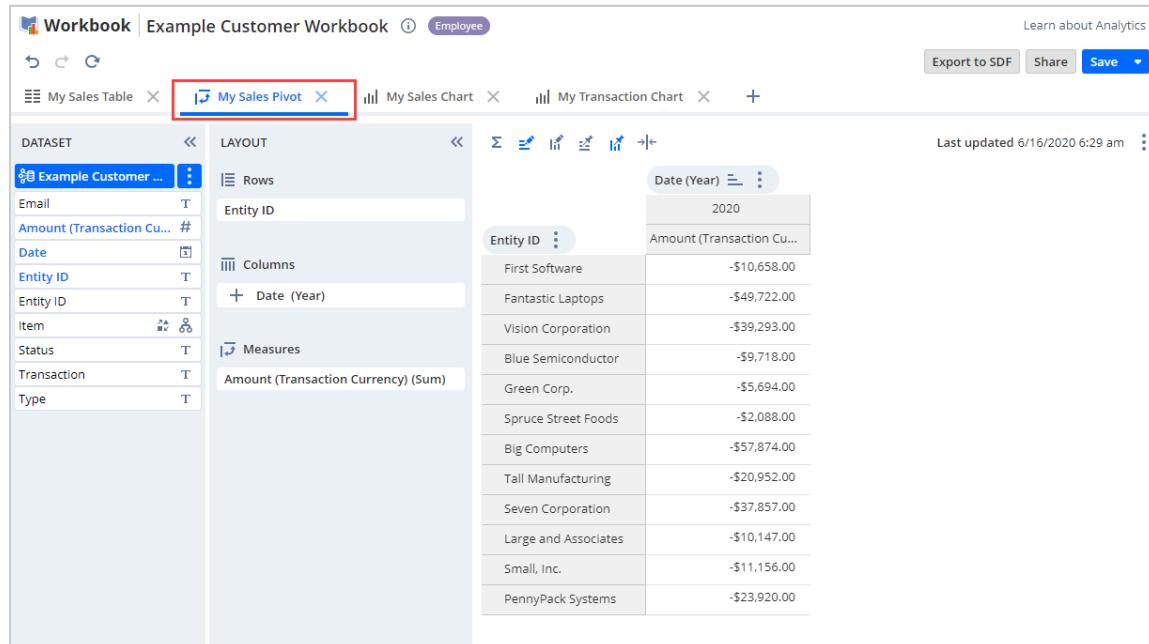
When viewing a workbook, the name appears in the info popup beside the workbook name.



Workbook.pivotDefinitions

This property represents the pivot definitions included in the workbook. This property is an array of `workbook.PivotDefinition` objects. Each pivot definition represents a pivot in the workbook. A workbook can have multiple pivot definitions.

When viewing a workbook, each pivot has its own tab near the top of the workbook page.



In a workbook, a tab can represent a table view, pivot, or chart. Pivot tabs include a measure icon beside the tab name.

Last updated 6/16/2020 6:29 am

| Date (Year) | Entity ID | Amount (Transaction Cu...) |
|-------------|----------------------|----------------------------|
| 2020 | First Software | -\$10,658.00 |
| | Fantastic Laptops | -\$49,722.00 |
| | Vision Corporation | -\$39,293.00 |
| | Blue Semiconductor | -\$9,718.00 |
| | Green Corp. | -\$5,694.00 |
| | Spruce Street Foods | -\$2,088.00 |
| | Big Computers | -\$57,874.00 |
| | Tall Manufacturing | -\$20,952.00 |
| | Seven Corporation | -\$37,857.00 |
| | Large and Associates | -\$10,147.00 |
| | Small, Inc. | -\$11,156.00 |
| | PennyPack Systems | -\$23,920.00 |

Workbook.tableDefinitions

This property represents the table definitions included in the workbook. This property is an array of `workbook.TableDefinition` objects. Each table definition represents a table view in the workbook. A workbook can have multiple table definitions.

When viewing a workbook, each table view has its own tab near the top of the workbook page.

Last updated 6/16/2020 8:18 am

| ENTITY ID | EMAIL | ENTITY ID | DATE | TRANSACTION | TYPE |
|---------------|--------------------------|-------------|-----------|-----------------|-------------|
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 5/13/2020 | Sales Order #14 | Sales Order |

Total row count: 207

In a workbook, a tab can represent a table view, pivot, or chart. Table view tabs include a grid icon beside the tab name.

The screenshot shows the SuiteAnalytics Workbook interface. At the top, there's a navigation bar with tabs for 'My Sales Table', 'My Sales Pivot', 'My Sales Chart', 'My Transaction Chart', and a '+' button. Below the navigation is a dataset sidebar with fields like 'Email', 'Amount (Transaction Cu...)', 'Date', 'Entity ID', 'Entity ID', 'Item', 'Status', 'Transaction', and 'Type'. The main area displays a table with the following data:

| ENTITY ID | EMAIL | ENTITY ID | DATE | TRANSACTION | TYPE |
|---------------|----------------------|-------------|-----------|-----------------|-------------|
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 5/13/2020 | Sales Order #14 | Sales Order |

Total row count: 207

Table Definition Objects

A table definition in the N/workbook module corresponds to a table view in the SuiteAnalytics Workbook UI. A table view is a workbook visualization that shows the data of the fields included in the underlying dataset. For more information about table views in the SuiteAnalytics Workbook UI, see the help topic [Workbook Table Views](#).

Using table definitions, you can do the following:

- View the data of the underlying dataset displayed in columns.
- Sort and filter the data in each column.

The Workbook API includes the following objects that relate to table definitions:

- [workbook.TableDefinition](#)
- [workbook.TableColumn](#)
- [workbook.TableFilter](#)

workbook.TableDefinition

Table definitions must contain table columns, an underlying dataset, and can contain filters and sorting options.

- [workbook.TableDefinition](#)
- [TableDefinition.columns](#)
- [TableDefinition.dataset](#)
- [TableDefinition.id](#)
- [TableDefinition.name](#)

workbook.TableDefinition

This object represents a table definition (table view) in a workbook, including columns and a reference to the underlying dataset.

When viewing a workbook, click a tab that represents a table view.

| ENTITY ID | EMAIL | ENTITY ID | DATE | TRANSACTION | TYPE |
|---------------|--------------------------|-------------|-----------|-----------------|-------------|
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 5/13/2020 | Sales Order #14 | Sales Order |

TableDefinition.columns

This property represents the columns (fields) that are included in the table definition. This property is an array of `workbook.TableColumn` objects.

On a table view tab, the columns (fields) appear in the Table Viewer, and the column names appear at the top of the Table Viewer.

| CUSTOMER | EMAIL | SALES REP | DATE | TRANSACTION | TYPE |
|---------------|--------------------------|-------------|-----------|-----------------|-------------|
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 5/13/2020 | Sales Order #14 | Sales Order |

On the left, the Dataset Panel lists the fields that are available in the underlying dataset. Fields that are included in the table view are highlighted.

Last updated 6/17/2020 9:28 am

Total row count: 207

| CUSTOMER | EMAIL | SALES REP | DATE | TRANSACTION | TYPE |
|---------------|----------------------|-------------|-----------|-----------------|-------------|
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 5/13/2020 | Sales Order #14 | Sales Order |

TableDefinition.dataset

This property represents the underlying dataset that the table definition is based on. This property is a `dataset.Dataset` object. A table definition can include only the columns (fields) that are included in the underlying dataset.

On a table view tab, the dataset appears in the Dataset Panel on the left.

Last updated 6/17/2020 9:39 am

Total row count: 207

| CUSTOMER | EMAIL | SALES REP | DATE | TRANSACTION | TYPE |
|---------------|----------------------|-------------|-----------|-----------------|-------------|
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 5/13/2020 | Sales Order #14 | Sales Order |

You can open the dataset directly from the Dataset Panel by clicking the three-dot icon and clicking Open Dataset. You can also save or discard pending changes to the dataset using options in the three-dot menu.

The screenshot shows the NetSuite Workbook interface with a table titled 'My Sales Table'. The table has columns: CUSTOMER, EMAIL, SALES REP, DATE, TRANSACTION, and TYPE. The 'Open Dataset' button is highlighted with a red box. The table contains 207 rows of sales data.

| CUSTOMER | EMAIL | SALES REP | DATE | TRANSACTION | TYPE |
|--------------------------|--------------------------|-------------|----------------|-----------------|-------------|
| bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order | |
| bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order | |
| bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order | |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 5/13/2020 | Sales Order #14 | Sales Order |

TableDefinition.id

This property represents the ID of the table definition. This property is a string. This ID always starts with 'custview'.

When viewing a workbook, the IDs of included table views, charts, and pivots appear when you click See Workbook Script IDs in the info popup beside the workbook name.

The ID displayed here does not include the mandatory 'custview' prefix that is added to all table definition IDs. Make sure that you use the full ID, including 'custview', when you refer to a table definition in your script.

The screenshot shows the NetSuite Workbook interface with a table titled 'My Sales Table'. The table has columns: TRANSACTION, Sales Order, Sales Order : Billed, and Video Card. The 'Workbook ID' field has a tooltip 'See Workbook Script IDs' highlighted with a red box. The table contains 207 rows of sales data.

| TRANSACTION | Sales Order | Sales Order : Billed | Video Card |
|-----------------|-------------|----------------------|-------------|
| Sales Order #1 | Sales Order | Sales Order : Billed | -\$4,600.00 |
| Sales Order #1 | Sales Order | Sales Order : Billed | -\$5,267.00 |
| Sales Order #1 | Sales Order | Sales Order : Billed | -\$6,900.00 |
| Sales Order #1 | Sales Order | Sales Order : Billed | -\$2,967.00 |
| Sales Order #13 | Sales Order | Sales Order : Billed | -\$4,000.00 |
| Sales Order #13 | Sales Order | Sales Order : Billed | -\$1,000.00 |
| Sales Order #13 | Sales Order | Sales Order : Billed | -\$2,500.00 |
| Sales Order #13 | Sales Order | Sales Order : Billed | -\$395.00 |
| Sales Order #14 | Sales Order | Sales Order : Billed | -\$3,500.00 |

| NAME | ID |
|----------------------|------------------------|
| My Sales Table | 3_15922463420114352490 |
| My Sales Pivot | 4 |
| My Sales Chart | 5 |
| My Transaction Chart | 6 |

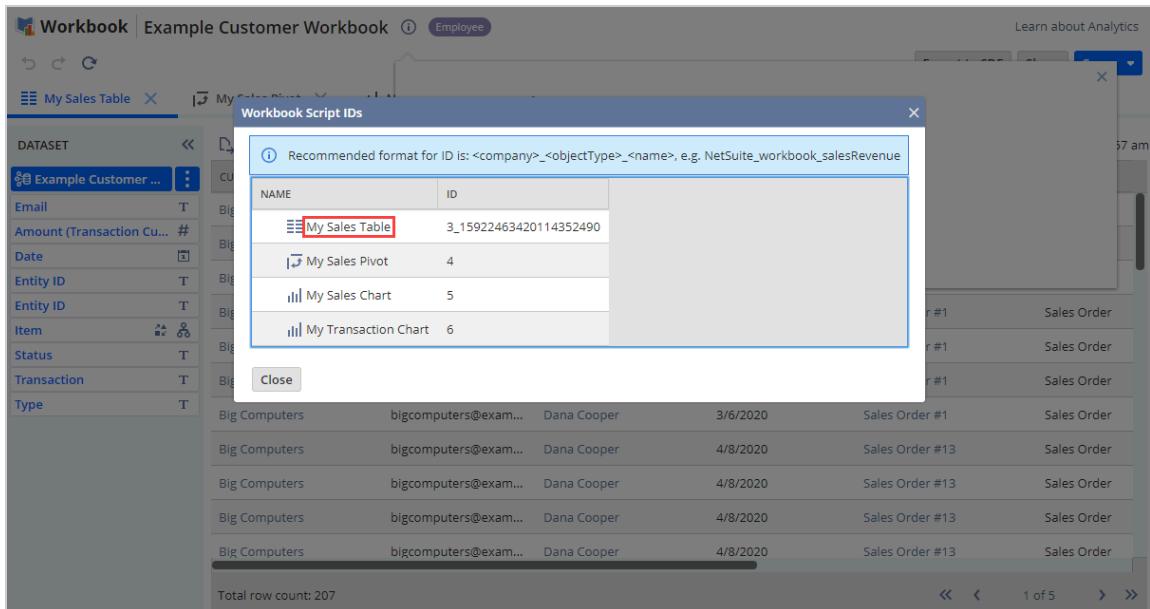
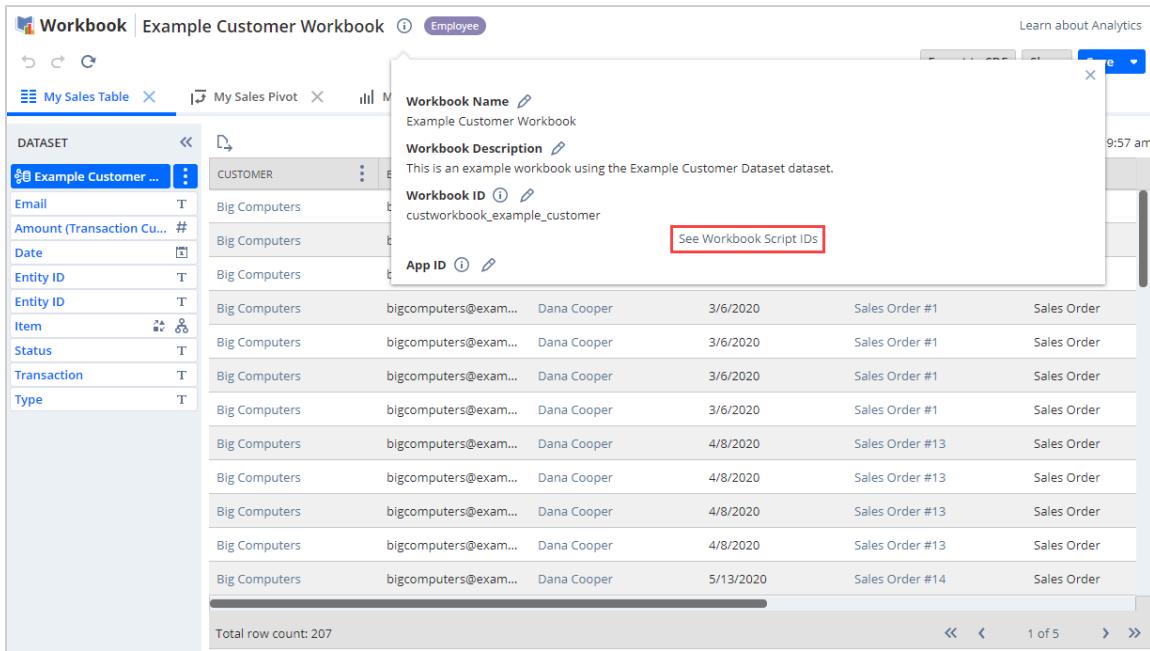
TableDefinition.name

This property represents the name of the table definition. This property is a string.

When viewing a workbook, the table definition name appears on the tab for the corresponding table view.

| CUSTOMER | EMAIL | SALES REP | DATE | TRANSACTION | TYPE |
|---------------|--------------------------|-------------|-----------|-----------------|-------------|
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@example.com | Dana Cooper | 5/13/2020 | Sales Order #14 | Sales Order |

The names of included table views, charts, and pivots also appear when you click See Workbook Script IDs in the info popup beside the workbook name.



workbook.TableColumn

Table columns correspond to the columns in table view in the Workbook UI.

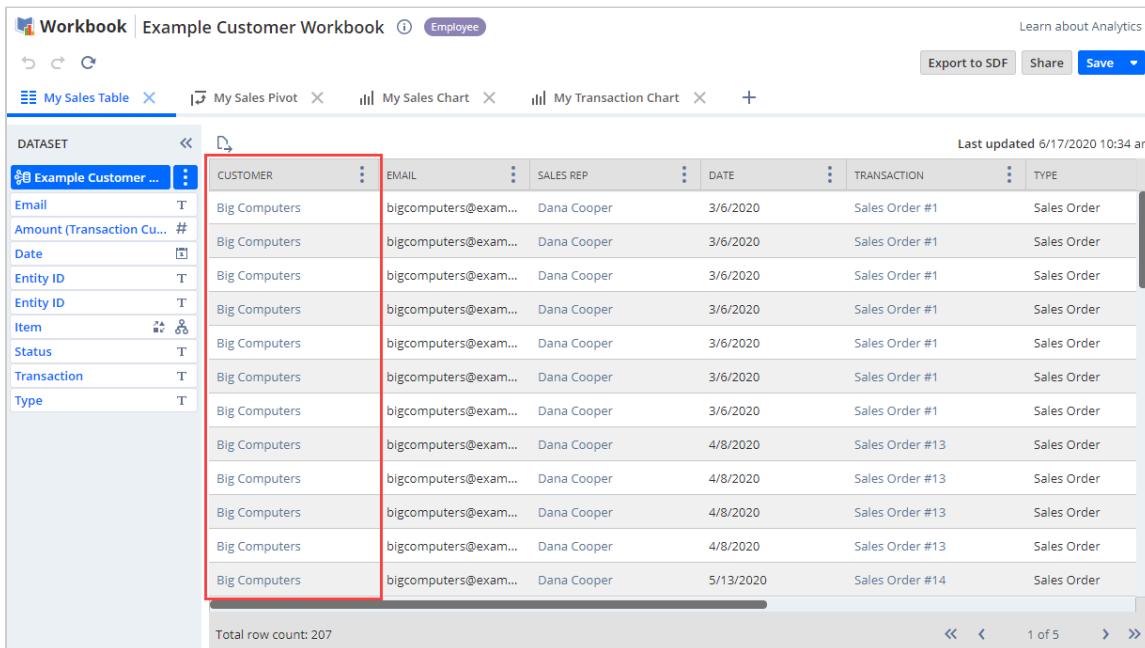
- `workbook.TableColumn`
 - `TableColumn.alias`
 - `TableColumn.datasetColumnAlias`
 - `TableColumn.datasetColumnId`
 - `TableColumn.fieldContext`
 - `TableColumn.filters`
 - `TableColumn.label`

- [TableColumn.sort](#)
- [TableColumn.width](#)

workbook.TableColumn

This object represents a single column in a table definition (table view), including filters, sorting, and underlying dataset columns.

On a table view tab, columns appears in the Table Viewer.



The screenshot shows the Table Viewer interface within a workbook. The top navigation bar includes tabs for 'My Sales Table', 'My Sales Pivot', 'My Sales Chart', and 'My Transaction Chart'. The 'Employee' tab is selected. On the left, a sidebar lists dataset columns: Email, Amount (Transaction Cu... #, Date, Entity ID, Entity ID, Item, Status, Transaction, and Type. The main area displays a table with the following data:

| CUSTOMER | EMAIL | SALES REP | DATE | TRANSACTION | TYPE |
|---------------|----------------------|-------------|-----------|-----------------|-------------|
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 3/6/2020 | Sales Order #1 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 4/8/2020 | Sales Order #13 | Sales Order |
| Big Computers | bigcomputers@exam... | Dana Cooper | 5/13/2020 | Sales Order #14 | Sales Order |

Total row count: 207

TableColumn.alias

This property represents the alias of the table column. This property is a string.

TableColumn.datasetColumnAlias

This property represents the alias of the dataset column that this table column is based on. This property is a string.

TableColumn.datasetColumnId

This property represents the ID of the dataset column that this table column is based on. This property is a number.

TableColumn.fieldContext

This property represents the field context of the table column. This property is a `workbook.FieldContext` object. The field context determines how the values in the table column are displayed in table views.

TableColumn.filters

This property represents the filters that are applied to the table column. This property is an array of `workbook.TableFilter` objects. Each object represents a single filter on a table column. A column can have multiple filters applied to it. These filters can be combined using AND or OR operators.

On a table view tab, column filters appear above the Table Viewer. This area displays all filters that are applied to table columns in the Table Viewer.

The screenshot shows the NetSuite Workbook interface with the 'My Sales Table' tab selected. In the top navigation bar, there are tabs for 'My Sales Pivot', 'My Sales Chart', and 'My Transaction Chart'. On the right side, there are buttons for 'Export to SDF', 'Share', and 'Save'. Below the tabs, a 'DATASET' sidebar lists columns: 'Email', 'Amount (Transaction Cu...)', 'Date', 'Entity ID', 'Entity ID', 'Item', 'Status', 'Transaction', and 'Type'. A red box highlights the '2 filters applied' section, which contains the query: '(Customer starts with b) AND (Transactions: Date within 3/3/2020 and 3/23/2020 OR Transactions: Date within 4/1/2020 and 4/24/2020)'. The main area is a table viewer showing data for 'Big Computers' with various transaction details. The bottom of the screen shows a message 'Total row count: 23' and navigation arrows.

In the Table Viewer, if a table column has a filter applied to it, a filter icon is displayed beside the table column label.

This screenshot is identical to the one above, but with two specific elements highlighted by red boxes: the filter icon for the 'Date' column in the dataset sidebar and the filter icon for the 'DATE' column in the table header. These icons indicate that filters have been applied to these specific columns.

TableColumn.label

This property represents the label of the table column. This property is a string.

On a table view tab, the label appears as the column heading in the Table Viewer.

The screenshot shows the NetSuite Workbook interface with the title "Example Customer Workbook". The left sidebar lists datasets: "My Sales Table" (selected), "My Sales Pivot", "My Sales Chart", "My Transaction Chart", and a plus sign icon. The main area displays a table titled "Dataset" with the following columns: CUSTOMER, MAIN EMAIL, SALES REP, DATE, TRANSACTION, and TYPE. A red box highlights the "MAIN EMAIL" column header. The table contains 13 rows of data. At the bottom, it says "Total row count: 23". The top right shows "Last updated 6/17/2020 1:54 pm", "Export to SDF", "Share", and "Save".

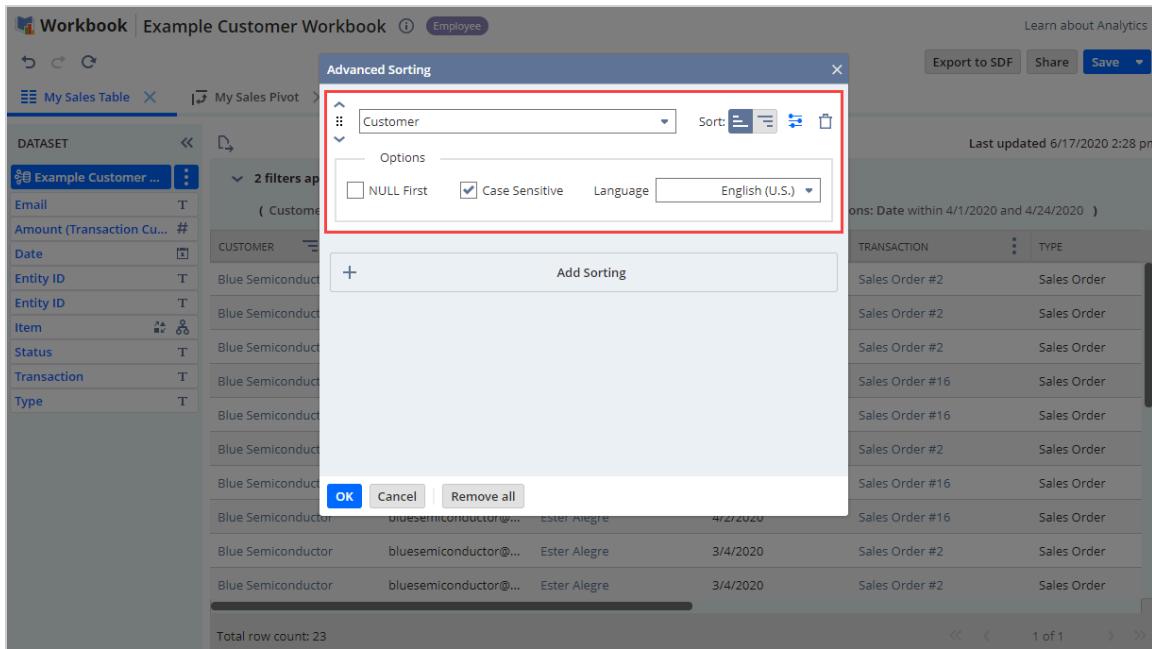
TableColumn.sort

This property represents the sorting behavior that is applied to the table column. This property is a `workbook.Sort` object. Only one sort can be applied to a table column at a time.

On a table view tab, sorts appear when clicking the three-dots menu on a table column and clicking Add Sort (if no sort is currently applied to the table column) or Edit Sort (if a sort is already applied to the table column).

In the Table Viewer, if a table column has a sort applied to it, a sort icon is displayed beside the table column label.

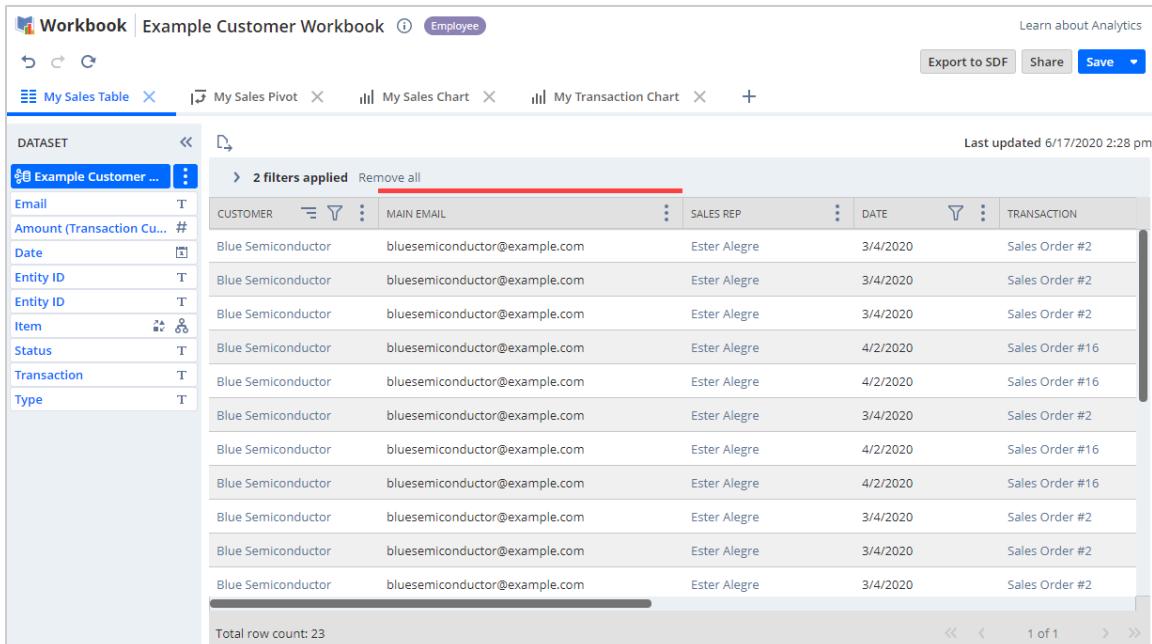
The screenshot shows the NetSuite Workbook interface with the title "Example Customer Workbook". The left sidebar lists datasets: "My Sales Table" (selected), "My Sales Pivot", "My Sales Chart", "My Transaction Chart", and a plus sign icon. The main area displays a table titled "Dataset" with the following columns: CUSTOMER, MAIN EMAIL, SALES REP, DATE, TRANSACTION, and TYPE. A red box highlights the "Edit Sort..." option in the context menu for the "MAIN EMAIL" column header. The table contains 13 rows of data. At the bottom, it says "Total row count: 23". The top right shows "Last updated 6/17/2020 2:28 pm", "Export to SDF", "Share", and "Save".



TableColumn.width

This property represents the width of the table column, in pixels, when it is displayed in the SuiteAnalytics Workbook UI. This property is a number.

On a table view tab, each table column appears with a width equal to the value of this property, in pixels.



workbook.TableFilter

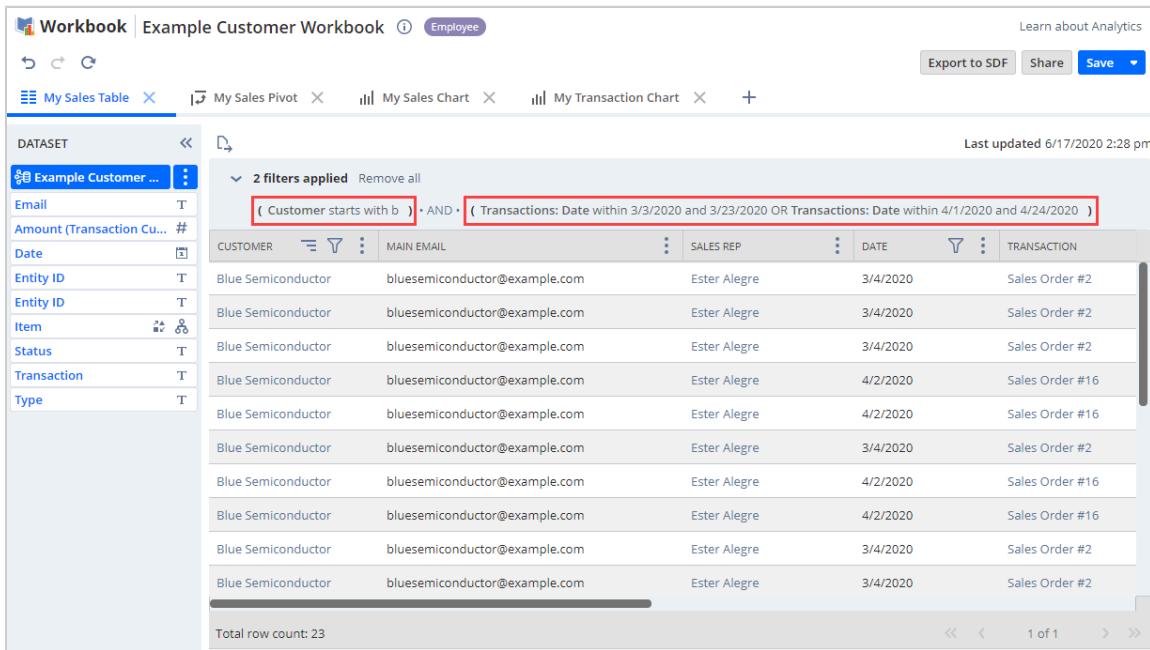
Table filters correspond to the filters that you can apply in table columns in the Workbook UI. For more information about the filtering options in the Workbook UI, see the help topic [Workbook Visualization Filters](#).

- [workbook.TableFilter](#)
- [TableFilter.operator](#)
- [TableFilter.values](#)

workbook.TableFilter

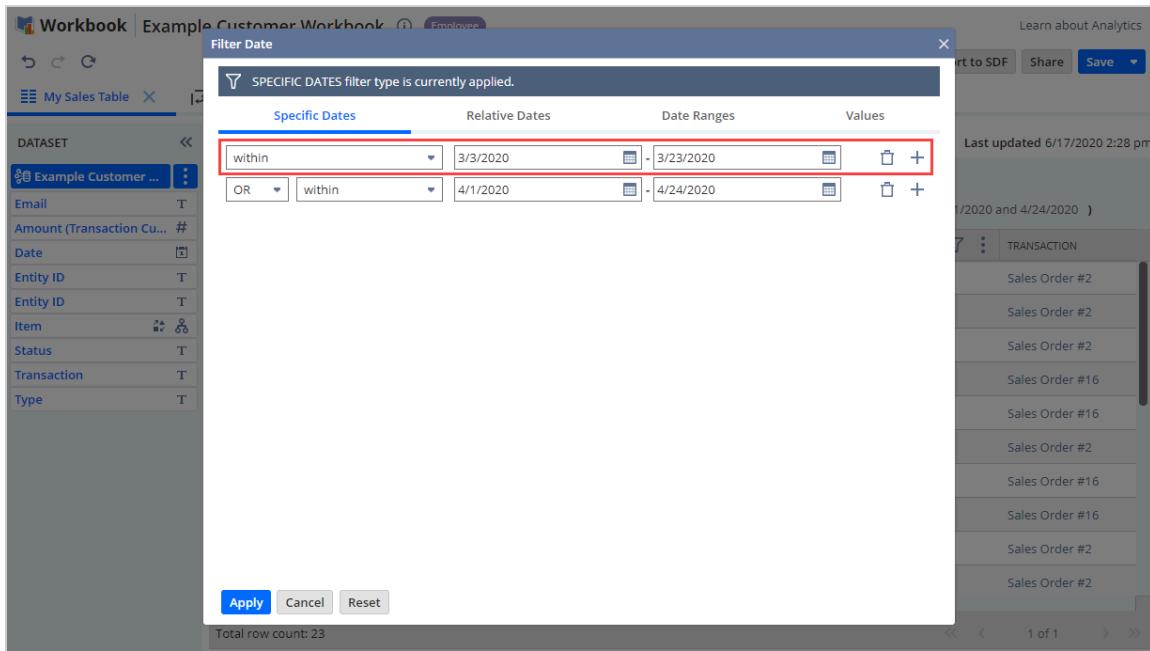
This object represents a single filter that is applied to a table column, including operator and values. Multiple filters can be applied to a single table column. These filters can be combined using AND or OR operators.

On a table view tab, column filters appear above the Table Viewer. This filters area displays all filters that are applied to table columns in the Table Viewer.



The screenshot shows the NetSuite Workbook interface with the title "Example Customer Workbook". The top navigation bar includes tabs for "Employee", "Learn about Analytics", "Export to SDF", "Share", and "Save". Below the tabs, there are several tabs: "My Sales Table" (selected), "My Sales Pivot", "My Sales Chart", "My Transaction Chart", and a "+" button. On the left, a sidebar titled "DATASET" lists columns: "Email", "Amount (Transaction Cu...)", "Date", "Entity ID", "Entity ID", "Item", "Status", "Transaction", and "Type". A dropdown menu for "Customer" is open. In the center, a table viewer displays data for "Blue Semiconductor" with columns: CUSTOMER, MAIN EMAIL, SALES REP, DATE, and TRANSACTION. The table has 23 rows. At the top of the table, it says "2 filters applied" with a "Remove all" link. Below that, the filters are listed: "(Customer starts with b) • AND • (Transactions: Date within 3/3/2020 and 3/23/2020 OR Transactions: Date within 4/1/2020 and 4/24/2020)". The bottom of the table shows "Total row count: 23" and navigation arrows.

You can view a simple filter or compound filter definition by clicking the filter entry in the filters area above the Table Viewer.

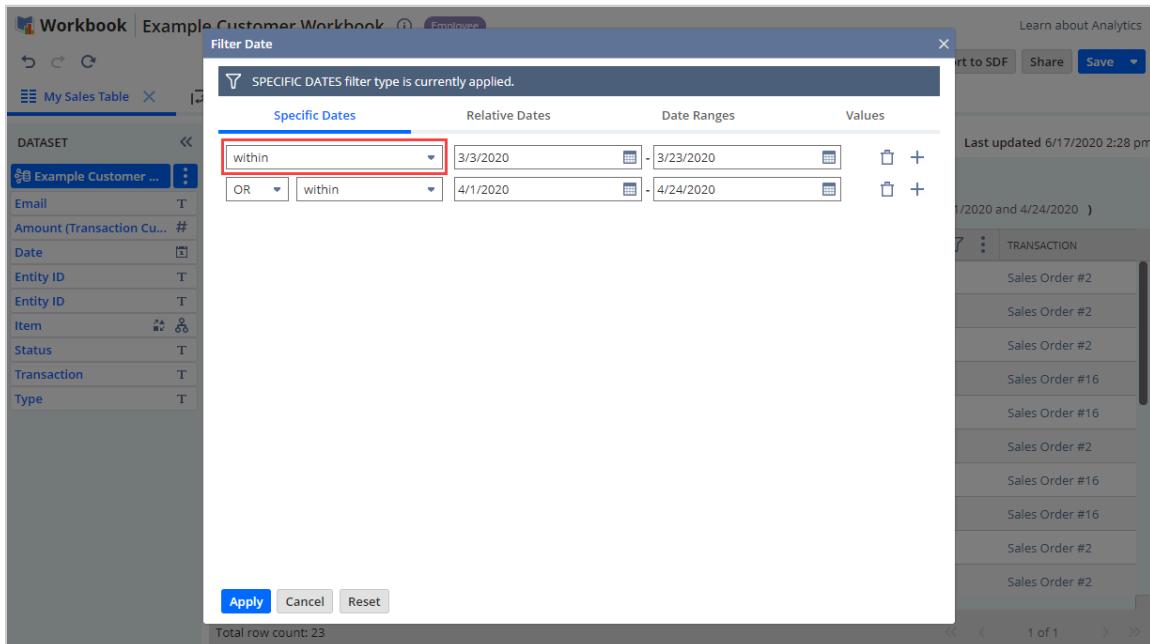


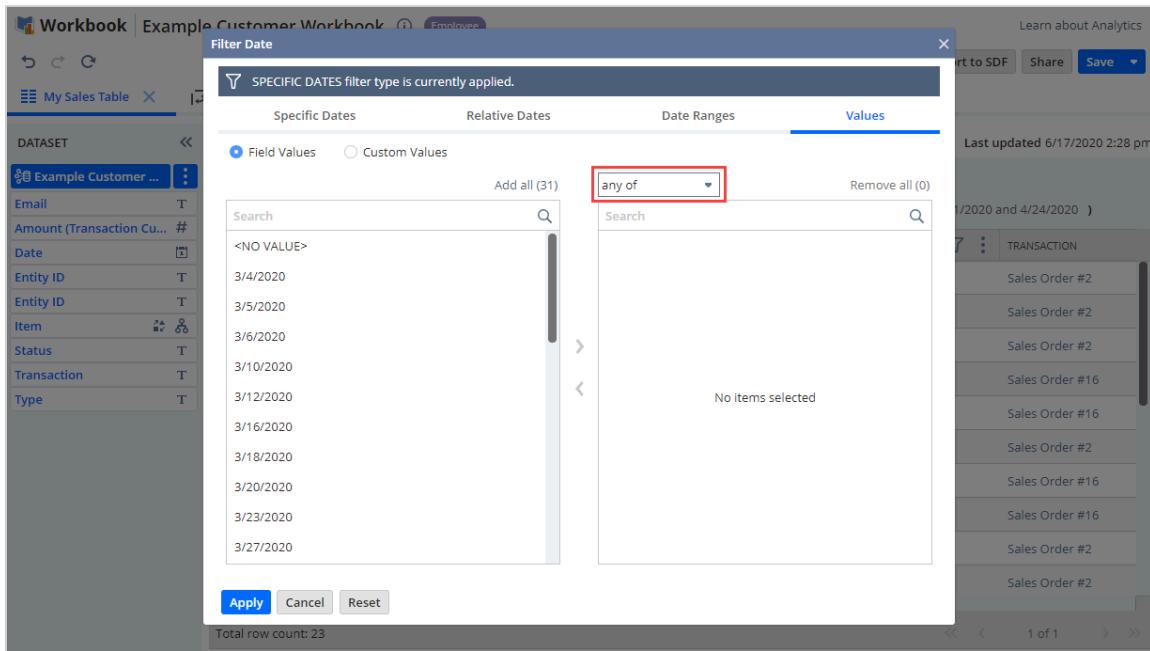
TableFilter.operator

This property represents the operator that is used in the filter. This property is a string. Available operators are documented in the `query.Operator` enum topic.

In the filters area, click a filter entry. The Filter window opens and shows the operator that is used for the condition in a dropdown menu.

The location of this dropdown menu and the available operators are different depending on the table column that's used in the filter.



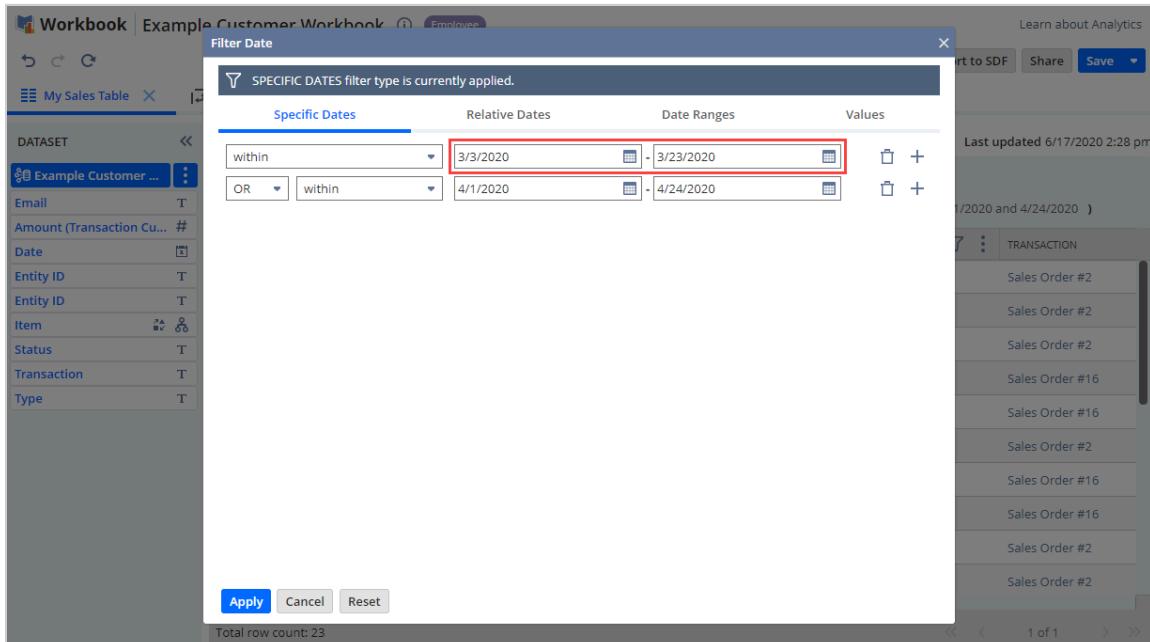


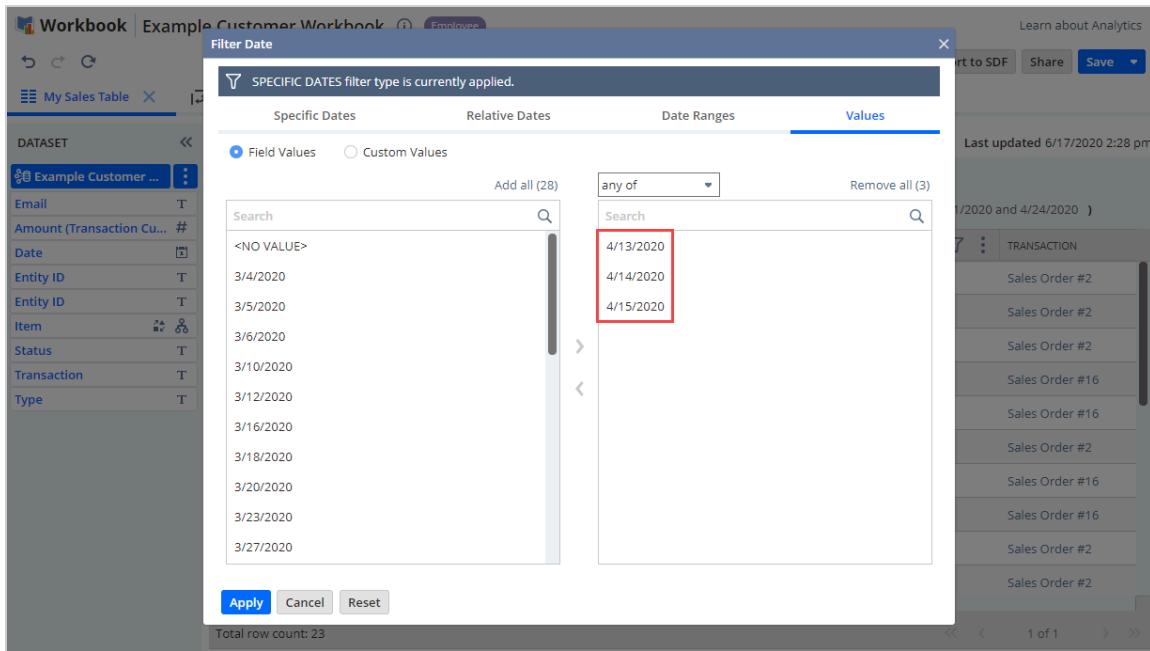
TableFilter.values

This property represents the values that are used in the filter. This property is an array of a single type, but the type can vary depending on the table column being filtered and the operator used (for example, string[], number[], and so on).

In the filters area, click a filter entry. The Filter window opens and shows the values that are used for the condition.

The location of the values and the available values are different depending on the table column that's used in the filter.





Workbook API (Beta) Limitations

Note: The content in this help topic pertains to SuiteScript 2.0.

The Workbook API is a beta feature, and some intended functionality is not yet available. The implementation of certain features may also change in future releases.

As you use the Workbook API, consider the following limitations:

- The set of available objects, properties, and methods in the N/dataset and N/workbook modules may change. New objects, properties, and methods may be added.
- Chart and pivot functionality in the N/workbook module may change. Currently, the [workbook.ChartDefinition](#) and [workbook.PivotDefinition](#) objects are available for charts and pivots. These objects and their properties may change or be replaced.
- When you use the Dataset Builder Plug-in or Workbook Builder Plug-in to create custom datasets or workbooks, you may receive an unexpected error, "Workbook does not exist" error, or "Workbook is no longer available" error when viewing the dataset or workbook in the SuiteAnalytics Workbook UI. These errors may occur if the code used to create the dataset or workbook in the plug-in is invalid or includes syntax errors. To work around this issue, see the following help topics:
 - For datasets, see the help topic [Creating a Dataset Builder Plug-in Script File](#).
 - For workbooks, see the help topic [Creating a Workbook Builder Plug-in Script File](#).

SuiteScript 2.x Script Types

SuiteScript 2.x offers several script types as described in the following table.

| Script Type | Description |
|---|--|
| SuiteScript 2.x Bundle Installation Script Type | <p>Bundle installation scripts fire triggers that execute as part of bundle installation, update, or uninstall. Trigger execution can occur either before install, after install, before update, after update, or before uninstall. These triggers automatically complete required setup, configuration, and data management tasks for the bundle.</p> <p>Bundle installation scripts execute on the server.</p> |
| SuiteScript 2.x Client Script Type | <p>Client scripts run on individual forms, can be deployed globally, and are applied to entity and transaction record types. Global client scripts enable centralized management of scripts that can be applied to an entire record type.</p> <p>Client scripts execute in the user's browser.</p> |
| SuiteScript 2.x Map/Reduce Script Type | <p>Map/reduce scripts provide a structured framework for processing a large number of records or a large amount of data. These scripts can be scheduled for submission or submitted on an on-demand basis.</p> <p>Map/reduce scripts execute on the server.</p> |
| SuiteScript 2.x Mass Update Script Type | <p>Mass update scripts allows you to programmatically perform custom mass updates to update fields that are not available through general mass updates. These scripts can run complex calculations across many records.</p> <p>Mass update scripts execute on the server.</p> |
| SuiteScript 2.x Portlet Script Type | <p>Portlet scripts create custom dashboard portlets. For example, you can use SuiteScript to create a portlet that is populated on-the-fly with company messages based on data within the system.</p> <p>Portlet scripts execute on the server and are rendered on the client's browser.</p> |
| SuiteScript 2.x RESTlet Script Type | <p>RESTlets can be used to define custom RESTful integrations to NetSuite.</p> <p>RESTlets execute on the server.</p> |
| SuiteScript 2.x Scheduled Script Type | <p>Scheduled scripts can be scheduled for submission or submitted on an on-demand basis.</p> <p>Scheduled scripts execute on the server.</p> |
| SuiteScript 2.x Suitelet Script Type | <p>Suitelets enable the creation of dynamic web content and build NetSuite-looking pages.</p> <p>Suitelets can be used to implement custom front and backends.</p> <p>Suitelets execute on the server.</p> |
| SuiteScript 2.x User Event Script Type | <p>User Event scripts are triggered when users work with records and data changes in NetSuite as they create, open, update, or save records. These scripts customize the workflow and association between your NetSuite entry forms. These scripts can also be used for doing additional processing before records are entered or for validating entries based on other data in the system.</p> <p>User event scripts execute on the server.</p> |
| SuiteScript 2.x Workflow Action Script Type | <p>Workflow action scripts allow you to create custom actions that are defined on a record in a workflow.</p> <p>Workflow action scripts execute on the server.</p> |
| SuiteScript 2.x SDF Installation Script Type | SDF installation scripts are automatically executed when a SuiteApp project is deployed. |

Best Practices

- Always thoroughly test your code before using it on your live NetSuite data.
- Type all record, field, sublist, tab, and subtab IDs in lowercase in your SuiteScript code.
- Prefix all custom script IDs and deployment IDs with an underscore (_).
- Do not hard-code any passwords in scripts. The password and password2 fields are supported for scripting.
- If the same code is used across multiple forms, ensure that you test any changes in the code for **each** form that the code is associated with.
- Include proper error handling sequences in your script wherever data may be inconsistent, not available, or invalid for certain functions. For example, if your script requires a field value to validate another, ensure that the field value is available.
- Organize your code into reusable chunks. Many functions can be used in a variety of forms. Any reusable functions should be stored in a common library file and then called into specific event functions for the required forms as needed.
- Place all custom code and markup, including third party libraries, in your own namespace.



Important: Custom code must not be used to access the NetSuite DOM. Developers must use SuiteScript APIs to access NetSuite UI components.

- Use the built in Library functions whenever possible for reading/writing Date/Currency fields and for querying XML documents
- During script development, componentize your scripts, load them individually, and then test each one — inactivating all but the one you are testing when multiple components are tied to a single user event.
- When working with script type events, your function name should correspond with the event. For example, a pageInit event can be named PageInit or formAPageInit.
- Since name values can change, ensure that you use **static** ID values in your API calls where applicable.
- Although you can use any desired naming conventions for functions within your code, it is recommended that you use custom namespaces or unique prefixes for all your function names.
- Thoroughly comment your code. This practice helps with debugging and development and assists NetSuite Customer Support in locating problems if necessary.
- You must use the `runtime.getCurrentScript()` function in the runtime module to reference script parameters. For example, use the following code to obtain the value of a script parameter named `custscript_case_field`:

```

1 define(['N/runtime'], function(runtime) {
2     function pageInit(context) {
3         var strField = runtime.getCurrentScript().getParameter('SCRIPT', 'custscript_case_field');
4         ...
5     });

```

- Make sure that your script does not take a long time to execute. A script may execute for a long time if any or all of the following occur:
 - The script performs a large number of record operations without going over the usage limit.
 - The script causes a large number of user event scripts and/or workflows to execute.
 - The script performs database searches or updates that collectively take a long time to finish

Each server script type or application has a time limit for execution. This limit is not fixed and depends on the script type or application. If a single execution of a server script or application takes longer than

the time limit for that script type or application, a SSS_TIME_LIMIT_EXCEEDED error is thrown. This error can also be thrown from a script that is executed by another script (for example, from a user event script that is executed by a scheduled script).

You can use SuiteScript Analysis to learn about when the script was installed and how it performed in the past. For more information, see the help topic [Analyzing Scripts](#).

SuiteScript 2.x Bundle Installation Script Type

Bundle installation scripts are specialized server scripts that perform processes in target accounts as part of a bundle installation, update, or uninstallation. These processes include setup, configuration, and data management tasks that would otherwise have to be completed by account administrators.

Every bundle can include a bundle installation script that is automatically run when the bundle is installed, updated, or uninstalled. Each bundle installation script can contain triggers to be executed before install, after install, before update, after update, and after uninstallation.

Bundle installation scripts have no audience because they are always executed using the Administrator role, in the context of bundle installation, update, or uninstallation. Bundle installation scripts do not have event types.

A bundle installation script can be associated with multiple bundles. Before a script can be associated with a bundle, it must have a script record and at least one deployment. A bundle creator associates a bundle installation script with a bundle by selecting one of its deployments in the Bundle Builder. The script .js file and script record are automatically included in the bundle when it is added to target accounts. Script file contents can be hidden from target accounts based on an option set for the .js file in the File Cabinet record.

Bundle installation script failures terminate bundle installations, updates, or uninstallations. Bundle installation scripts can include their own error handling in addition to errors thrown by SuiteBundler and SuiteScript. An error thrown by a bundle installation script returns an error code of **Installation Error** followed by the text defined by the script author. For information about scripting with bundle installation scripts, see [SuiteScript 2.x Bundle Installation Script Entry Points](#).

For more information on bundle installation scripts, see the help topics [Using Bundle Installation Scripts](#) and [Bundle Support Across Account Types](#).



Note: Custom modules are not supported in bundle installation scripts.

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

You can use SuiteScript Analysis to learn about when the script was installed and how it performed in the past. For more information, see the help topic [Analyzing Scripts](#).

Bundle Installation Script Sample

The script sample performs the following tasks:

- Before the bundle installation and the bundle update, ensure that the Time Off Management feature is enabled in the target NetSuite account and warn the user if the feature is not enabled.

This script sample uses SuiteScript 2.0. A newer version, SuiteScript 2.1, is also available and supports new language features that are included in the ES2019 specification. You can write bundle installation scripts using either SuiteScript 2.0 or SuiteScript 2.1.

- For help with writing scripts in SuiteScript 2.0, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).
- For more information about SuiteScript versions and SuiteScript 2.1, see [SuiteScript Versioning Guidelines](#) and [SuiteScript 2.1](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType BundleInstallationScript
4 */
5 define(['N/runtime'], function(runtime) {
6     function checkPrerequisites() {
7         if (!runtime.isFeatureInEffect({
8             feature: 'TIMEOFFMANAGEMENT'
9         })) {
10             throw 'The TIMEOFFMANAGEMENT feature must be enabled. ' +
11                  'Please enable the feature and try again.';
12         }
13     return {
14         beforeInstall: function beforeInstall(params) {
15             checkPrerequisites();
16         },
17         beforeUpdate: function beforeUpdate(params) {
18             checkPrerequisites();
19         }
20     };
21 });

```

SuiteScript 2.x Bundle Installation Script Reference

A bundle installation script's functions are executed automatically during bundle installation, update, or uninstallation, based on one or more of the following triggers:

- Before Install - Executed before a bundle is installed for the first time in a target account.
- After Install - Executed after a bundle is installed for the first time in a target account.
- Before Update - Executed before a bundle in a target account is updated.
- After Update - Executed after a bundle in a target account is updated.
- Before Uninstall - Executed before a bundle is uninstalled from a target account.

A bundle installation script file should include a function for at least one of these triggers. If you are using more than one of these, they should all be in the same script file.

The following are example uses for bundle installation script triggered functions:

- Before Install: Check the existing configuration and setup in the target account prior to bundle installation, and halt the installation with an error message if the target account does not meet minimum requirements to run the solution.
- After Install: Automate the setup and configuration of the bundled application after it has been installed in the target account, eliminating manual tasks.
- After Install or After Update: Connect to an external system to fetch some data and complete the setup of the bundled application.
- Before Update: Manage required data changes in the target account prior to executing an upgrade.

- Before Uninstall: Reset configuration settings or remove data associated with the bundle being uninstalled.

Two specialized parameters are available to functions in bundle installation scripts, to return the version of bundles, as specified on the Bundle Basics page of the Bundle Builder.

- The **toversion** parameter returns the version of the bundle that will be installed in the target account. This parameter is available to Before Install, After Install, Before Update, and After Update functions.
- The **fromversion** parameter returns the version of the bundle that is currently installed in the target account. This parameter is available to Before Update and After Update functions.

A bundle installation script file can include calls to functions in other script files, as long as these files are added as library script files on the script record. Any .js files for library script files are automatically included in the bundle when it is added to target accounts.

Bundle installation scripts can call scheduled scripts, but only in the After Install and After Update functions. Calls to scheduled scripts are not supported in the Before Install, Before Update, and Before Uninstall functions.

Bundle installation scripts are governed by a maximum of 10,000 units per execution.

You can create multiple deployments for each bundle installation script, with different parameters for each, but only one deployment can be associated with each bundle. When you associate a bundle installation script with a bundle, you select a specific script deployment.

Bundle installation scripts need to be executed with administrator privileges, so the Execute as Role field should always be set to Administrator on the script deployment record.

Bundle installation scripts can only be run in target accounts if the Status is set to Released. The Status should be set to Testing if you want to debug the script.

Any bundle installation script failure terminates bundle installation, update, or uninstallation.

Bundle installation scripts can include their own error handling, in addition to errors thrown by SuiteBundler and SuiteScript. An error thrown by a bundle installation script returns an error code of "Installation Error", followed by the text defined by the script author.

Setting Up a Bundle Installation Script

Complete the following steps to set up a bundle installation script.

Create the Bundle Installation Script File

Create a .js script file and add code. This single script file should include Before Install, After Install, Before Update, After Update, and Before Uninstall functions as necessary. It can include calls to functions in other files, but you will need to list these files as library script files on the NetSuite script record.

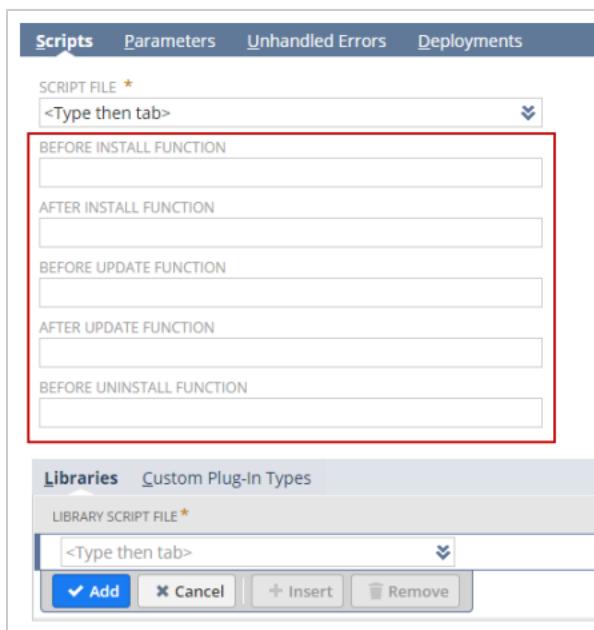
Add the Bundle Installation Script File to the File Cabinet

- Go to Documents > Files > File Cabinet, and select the folder where you want to add the file. It is recommended that you add your file to the SuiteScripts folder, but it can be added to any other folder of your choice.
- Click **Add File**, and browse to the .js file.
- In the File Cabinet folder where you added the bundle installation script file, click the **Edit** link next to file.

4. Check the **Available for SuiteBundles** box.
5. Optionally, you can check the **Hide in SuiteBundle** box.
Because this script file will be included in the bundle, by default its contents will be accessible to users of target accounts where the bundle is installed. If you do not want these users to see this file, you can set this option to hide it.
6. Click **Save**.

Create the Bundle Installation Script Record

1. Go to Setup > Customization > Scripts > New, and click **Bundle Installation**.
2. Complete fields in the script record and save.
Although you do not need to set every field on the Script record, at a minimum you must provide a **Name** for the Script record, load your SuiteScript file to the record, and specify at least one of the executing functions on the Scripts tab.



3. These functions should all be in the main script file. If these functions call functions in other script files, you need to list those files as library script files.

Define Bundle Installation Script Deployment

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

Be sure to check the **Execute as Admin** box.

If you want to debug the script, set the **Status** to **Testing**. To enable the script to be run in a target account, you must set the **Status** to **Released**.

Associate the Script with a Bundle

Note: The SuiteBundler feature must be enabled in your account for you to have access to the Bundle Builder where this task is completed.

1. Start the Bundle Builder.
 - If you are creating a new bundle, go to Customization > SuiteBuilder > Create Bundle
 - If you are editing an existing bundle, go to Customization > SuiteBundler > Search & Install Bundles > List, and select **Edit** from the Action menu for the desired bundle.
2. On the Bundle Basics page, select a bundle installation script deployment from the **Installation Script** dropdown list.
3. Proceed through the remaining Bundle Builder steps, making definitions as necessary, and click **Save**. Note the following:
 - On the Select Objects page of the Bundle Builder, you do not have to explicitly add the bundle installation script. This script record and the related .js file are included automatically in the bundle, as are any other .js files that are listed as library script files on the script record.
 - After the bundle has been saved, this script record and related file(s) are listed as Bundle Components on the Bundle Details page.

The screenshot shows the 'Components' tab of the SuiteBundler interface. It displays a table of files and scripts under the 'File Cabinet' section. The columns are 'NAME' and 'ID'. The table includes sections for 'Files' (listing various .js files like ptm_settings.js, ptm_project_task_data.js, etc.) and 'SuiteScripts' (listing a 'Bundle Installation' entry). The 'Bundle Installation' entry is for 'PTM Bundle Install (PTM Bundle Install)' with ID 'customscript_ptm_bundle_install'.

| NAME | ID |
|---|---------------------------------|
| File Cabinet | |
| Files | |
| ptm_settings.js | 2606 |
| ptm_project_task_data.js | 2770 |
| ptm_permissions.js | 2605 |
| ptm_saved_filters.js | 2609 |
| ptm_restlet_list_data.js | 2608 |
| Release.txt | 1989 |
| ptm_ss_main.js | 2610 |
| ptm_translation_ss.js | 2845 |
| SuiteScripts | |
| Bundle Installation | |
| PTM Bundle Install (PTM Bundle Install) | customscript_ptm_bundle_install |

SuiteScript 2.x Bundle Installation Script Entry Points

| Script Entry Point | |
|--------------------|--|
| afterInstall | Defines the function that is executed after a bundle is installed for the first time in a target account. |
| afterUpdate | Defines the function that is executed after a bundle in a target account is updated. |
| beforeInstall | Defines the function that is executed before a bundle is installed for the first time in a target account. |

| Script Entry Point | |
|--------------------|---|
| beforeUninstall | Defines the function that is executed before a bundle is uninstalled from a target account. |
| beforeUpdate | Defines the function that is executed before a bundle in a target account is updated. |

afterInstall

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-------------|---|
| Description | Defines the function that is executed after a bundle is installed for the first time in a target account. |
| Returns | void |
| Since | 2016.1 |

Parameters

 **Note:** The params parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|----------------|--------|--|--------|
| params.version | number | The version of the bundle that is being installed in the target account. | 2016.1 |

afterUpdate

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-------------|--|
| Description | Defines the function that is executed after a bundle in a target account is updated. |
| Returns | void |
| Since | 2016.1 |

Parameters

 **Note:** The params parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|--------------------|--------|--|--------|
| params.fromVersion | number | The version of the bundle that is currently installed in the target account. | 2016.1 |
| params.toVersion | number | The new version of the bundle that is being installed in the target account. | 2016.1 |

beforeInstall

Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines the function that is executed before a bundle is installed for the first time in a target account. Calls to scheduled scripts are not supported. |
| Returns | void |
| Since | 2016.1 |

Parameters

Note: The params parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|----------------|--------|--|--------|
| params.version | number | The version of the bundle that is being installed in the target account. | 2016.1 |

beforeUninstall

Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|--|
| Description | Defines the function that is executed before a bundle is uninstalled from a target account. Calls to scheduled scripts are not supported. |
| Returns | void |
| Since | 2016.1 |

Parameters

Note: The params parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|----------------|--------|--|--------|
| params.version | number | The version of the bundle that is being uninstalled from the target account. | 2016.1 |

beforeUpdate

Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines the function that is executed before a bundle in a target account is updated. |
|--------------------|---|

| | |
|---------|---|
| | Calls to scheduled scripts are not supported. |
| Returns | void |
| Since | 2016.1 |

Parameters

 **Note:** The `params` parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|---------------------------------|--------|--|--------|
| <code>params.fromVersion</code> | number | The version of the bundle that is currently installed in the target account. | 2016.1 |
| <code>params.toVersion</code> | number | The new version of the bundle that is being installed in the target account. | 2016.1 |

SuiteScript 2.x Client Script Type

Client scripts are scripts that are executed by predefined event triggers in the client browser. They can validate user-entered data and auto-populate fields or sublists at various form events. For details, see [SuiteScript 2.x Client Script Entry Points and API](#).

Scripts can be run on most standard records, custom record types, and custom NetSuite pages such as Suitelets. See the help topic [SuiteScript Supported Records](#) for a list of NetSuite records that support SuiteScript.

 **Important:** Client scripts only execute in edit mode. If you have a deployed client script with a `pageInit` entry point, that script does not execute when you view the form. It executes when you click **Edit**.

The following triggers can run a client script.

- Loading a form for editing
- Entering or changing a value in a field (before and after it is entered)
- Entering or changing a value in a field that sources another field
- Selecting a line item on a sublist
- Adding a line item (before and after it is entered)
- Saving a form

Record-level client scripts are executed after any existing form-based clients are run, and before any user event scripts are run.

See the help topic [Script Type Usage Unit Limits](#) for details about client script governance.

 **Tip:** You can set the order in which client scripts execute on the Scripted Records page. See the help topic [The Scripted Records Page](#).

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

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

You can use SuiteScript Analysis to learn about when the script was installed and how it performed in the past. For more information, see the help topic [Analyzing Scripts](#).

For additional information about SuiteScript 2.x client scripts, see the following:

- [SuiteScript 2.x Client Script Reference](#)
 - [SuiteScript 2.x Client Scripts on currentRecord](#)
 - [SuiteScript 2.x Client Script Role Restrictions](#)
 - [SuiteScript 2.x Remote Objects in Client Scripts](#)
 - [SuiteScript 2.x Client Script Best Practices](#)
- [SuiteScript 2.x Client Script Entry Points and API](#)
 - [fieldChanged](#)
 - [lineInit](#)
 - [pageInit](#)
 - [postSourcing](#)
 - [saveRecord](#)
 - [sublistChanged](#)
 - [validateDelete](#)
 - [validateField](#)
 - [validateInsert](#)
 - [validateLine](#)
 - [localizationContextEnter](#)
 - [localizationContextExit](#)

SuiteScript Client Script Sample

For help with writing scripts in SuiteScript 2.x, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).

The following sample shows a client script applied to a sales order form. The script performs the following tasks:

- When the form loads for editing, the pageInit event sets the customer field to a specific customer.
- When form changes are saved, the saveRecord event ensures that the customer field is set and at least one sales order item is listed.
- When editing the quantity of a sales order item, the validateField event ensures that the number is less than three.
- When selecting a sales order item, the fieldChanged event updates a memo field to indicate that the item was selected.
- When a specific sales order item is selected, the postSourcing event updates the price level for that particular item.
- When an existing partner is selected, the lineInit event changes the selected partner to a specific partner.
- When a partner is deleted, the validateDelete event updates a memo field to indicate that the partner was deleted.

- When adding a new partner or editing an existing partner, the validateInsert and validateLine events ensure that their contribution is set to 100%.

```

1  /**
2  * @NApiVersion 2.x
3  * @NScriptType ClientScript
4  */
5  define(['N/error'],
6      function(error) {
7          function pageInit(context) {
8              if (context.mode !== 'create')
9                  return;
10             var currentRecord = context.currentRecord;
11             currentRecord.setValue({
12                 fieldId: 'entity',
13                 value: 107
14             });
15         }
16         function saveRecord(context) {
17             var currentRecord = context.currentRecord;
18             if (!currentRecord.getValue({
19                 fieldId: 'entity'
20             }) || currentRecord.getLineCount({
21                 sublistId: 'item'
22             }) < 1)
23                 throw error.create({
24                     name: 'MISSING_REQ_ARG',
25                     message: 'Please enter all the necessary fields on the salesorder before saving'
26                 });
27             return true;
28         }
29         function validateField(context) {
30             var currentRecord = context.currentRecord;
31             var sublistName = context.sublistId;
32             var sublistFieldName = context.fieldId;
33             var line = context.line;
34             if (sublistName === 'item') {
35                 if (sublistFieldName === 'quantity') {
36                     if (currentRecord.getCurrentSublistValue({
37                         sublistId: sublistName,
38                         fieldId: sublistFieldName
39                     }) < 3)
40                         currentRecord.setValue({
41                             fieldId: 'otherrefnum',
42                             value: 'Quantity is less than 3'
43                         });
44                     else
45                         currentRecord.setValue({
46                             fieldId: 'otherrefnum',
47                             value: 'Quantity accepted'
48                         });
49                 }
50             }
51             return true;
52         }
53         function fieldChanged(context) {
54             var currentRecord = context.currentRecord;
55             var sublistName = context.sublistId;
56             var sublistFieldName = context.fieldId;
57             var line = context.line;
58             if (sublistName === 'item' && sublistFieldName === 'item')
59                 currentRecord.setValue({
60                     fieldId: 'memo',
61                     value: 'Item: ' + currentRecord.getCurrentSublistValue({
62                         sublistId: 'item',
63                         fieldId: 'item'
64                     }) + ' is selected'
65                 });
66         }
67         function postSourcing(context) {
68             var currentRecord = context.currentRecord;
69             var sublistName = context.sublistId;

```

```

70     var sublistFieldName = context.fieldId;
71     var line = context.line;
72     if (sublistName === 'item' && sublistFieldName === 'item')
73         if (currentRecord.getCurrentSublistValue({
74             sublistId: sublistName,
75             fieldId: sublistFieldName
76         }) === '39')
77             if (currentRecord.getCurrentSublistValue({
78                 sublistId: sublistName,
79                 fieldId: 'pricelvels'
80             }) !== '1-1')
81                 currentRecord.setCurrentSublistValue({
82                     sublistId: sublistName,
83                     fieldId: 'pricelvels',
84                     value: '1-1'
85                 });
86     }
87     function lineInit(context) {
88         var currentRecord = context.currentRecord;
89         var sublistName = context.sublistId;
90         if (sublistName === 'partners')
91             currentRecord.setCurrentSublistValue({
92                 sublistId: sublistName,
93                 fieldId: 'partner',
94                 value: '55'
95             });
96     }
97     function validateDelete(context) {
98         var currentRecord = context.currentRecord;
99         var sublistName = context.sublistId;
100        if (sublistName === 'partners')
101            if (currentRecord.getCurrentSublistValue({
102                sublistId: sublistName,
103                fieldId: 'partner'
104            }) === '55')
105                currentRecord.setValue({
106                    fieldId: 'memo',
107                    value: 'Removing partner sublist'
108                });
109        return true;
110    }
111    function validateInsert(context) {
112        var currentRecord = context.currentRecord;
113        var sublistName = context.sublistId;
114        if (sublistName === 'partners')
115            if (currentRecord.getCurrentSublistValue({
116                sublistId: sublistName,
117                fieldId: 'contribution'
118            }) !== '100.0%')
119                currentRecord.setCurrentSublistValue({
120                    sublistId: sublistName,
121                    fieldId: 'contribution',
122                    value: '100.0%'
123                });
124        return true;
125    }
126    function validateLine(context) {
127        var currentRecord = context.currentRecord;
128        var sublistName = context.sublistId;
129        if (sublistName === 'partners')
130            if (currentRecord.getCurrentSublistValue({
131                sublistId: sublistName,
132                fieldId: 'contribution'
133            }) !== '100.0%')
134                currentRecord.setCurrentSublistValue({
135                    sublistId: sublistName,
136                    fieldId: 'contribution',
137                    value: '100.0%'
138                });
139        return true;
140    }
141    function sublistChanged(context) {
142        var currentRecord = context.currentRecord;

```

```

143     var sublistName = context.sublistId;
144     var op = context.operation;
145     if (sublistName === 'item')
146       currentRecord.setValue({
147         fieldId: 'memo',
148         value: 'Total has changed to ' + currentRecord.getValue({
149           fieldId: 'total'
150         }) + ' with operation: ' + op
151       });
152   }
153   return {
154     pageInit: pageInit,
155     fieldChanged: fieldChanged,
156     postSourcing: postSourcing,
157     sublistChanged: sublistChanged,
158     lineInit: lineInit,
159     validateField: validateField,
160     validateLine: validateLine,
161     validateInsert: validateInsert,
162     validateDelete: validateDelete,
163     saveRecord: saveRecord
164   };
165 });

```

SuiteScript 2.x Client Script Reference

- [SuiteScript 2.x Client Scripts on currentRecord](#)
- [SuiteScript 2.x Client Script Role Restrictions](#)
- [SuiteScript 2.x Remote Objects in Client Scripts](#)
- [SuiteScript 2.x Client Script Best Practices](#)

SuiteScript 2.x Client Scripts on currentRecord

You use the currentRecord module to access the record that is active in the current client context. For more information about the currentRecord module, see the help topic [N/currentRecord Module](#).

The following is a sample client script. When the saveRecord client event is triggered, the script uses the currentRecord module to validate a Sales Order record. The script ensures that the transaction date is not older than one week, and that the total is valid.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType ClientScript
4 */
5 define(['N/ui/message'],
6   function(msg) {
7     function showErrorMessage(msgText) {
8       var myMsg = msg.create({
9         title: "Cannot Save Record",
10        message: msgText,
11        type: msg.Type.ERROR
12      });
13
14      myMsg.show({
15        duration: 5000
16      });
17    }
18
19    function saveRec(context) {
20      var rec = context.currentRecord;
21      var currentDate = new Date()
22      var oneWeekAgo = new Date(currentDate - 1000 * 60 * 60 * 24 * 7);
23
24      // Validate transaction date is not older than current time by one week

```

```

25     if (rec.getValue({
26         fieldId: 'trandate'
27     }) < oneWeekAgo) {
28         showErrorMessage("Cannot save sales order with trandate one week old.");
29         return false;
30     }
31
32     //validate total is greater than 0
33     if (rec.getValue({
34         fieldId: 'total'
35     }) <= 0) {
36         showErrorMessage("Cannot save sales order with negative total amount.");
37         return false;
38     }
39     return true;
40 }
41
42 return {
43     saveRecord: saveRec
44 };
45 });

```

SuiteScript 2.x Client Script Role Restrictions

Client scripts respect the role permissions specified in the user's NetSuite account. An error is thrown when running a client script to access a record with a role that does not have permission to view or edit the record.

The following client script attaches to a custom sales order form and executes on the field change client event:

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType ClientScript
4  */
5 define(['N/search'],
6     function(search) {
7         function getSalesRepEmail(context) {
8             var salesRep = context.currentRecord.getValue({
9                 fieldId: 'salesrep'
10            });
11             var salesRepEmail = search.lookupFields({
12                 type: 'employee',
13                 id: salesRep,
14                 columns: ['email']
15            });
16             alert(JSON.stringify(salesRepEmail));
17         }
18
19         return {
20             fieldChanged: getSalesRepEmail
21         }
22     });

```

If you are logged in with an admin role, you receive the alert when you load the sales order with this form. If you are logged in with a role that does not have permission to view/edit Employee records, you receive an error when you select the Sales Rep field.

The following considerations can help prevent users from receiving the error:

- Consider the types of users who may be using your custom form and running the script.
- Consider which record types that users do not have access to. If it is vital that all who run the script have access to the records in the script, you may need to redefine the permissions of the users (if your role is as an admin).
- Consider rewriting your script so that it only references record types that all users have access to.

- Consider writing the script as a user event script, and set the Execute As Admin preference on the Script Deployment page. Note that alerts are a function of client scripts only and cannot be used in user event scripts. For more information about user event scripts, see [SuiteScript 2.x User Event Script Type](#).

SuiteScript 2.x Remote Objects in Client Scripts

A client script interfaces with remote objects whenever it calls the NetSuite database to create, load, copy, or transform an object record.

The following client script loads a journal entry record and sets two line sublist lines when the saveRecord client event executes.

```

1  /**
2  * @NApiVersion 2.0
3  * @NScriptType ClientScript
4  */
5 define(['N/record'], function(record){
6     function saveRecord(context){
7         var journalEntry = record.load({
8             id:6,
9                 type: record.Type.JOURNAL_ENTRY,
10                isDynamic: true
11            });
12            journalEntry.selectNewLine({
13                sublistId: 'line'
14            });
15            journalEntry.setCurrentSublistValue({
16                sublistId: 'line',
17                    fieldId: 'account',
18                    value: 1
19            });
20            journalEntry.setCurrentSublistValue({
21                sublistId: 'line',
22                    fieldId: 'debit',
23                    value: 1
24            });
25            journalEntry.setCurrentSublistValue({
26                sublistId: 'line',
27                    fieldId: 'memo',
28                    value: "Debit 1"
29            });
30            journalEntry.commitLine({
31                sublistId: 'line'
32            });
33            journalEntry.selectNewLine({
34                sublistId: 'line'
35            });
36            journalEntry.setCurrentSublistValue({
37                sublistId: 'line',
38                    fieldId: 'account',
39                    value: 1
40            });
41            journalEntry.setCurrentSublistValue({
42                sublistId: 'line',
43                    fieldId: 'credit',
44                    value: 1
45            });
46            journalEntry.setCurrentSublistValue({
47                sublistId: 'line',
48                    fieldId: 'memo',
49                    value: "Credit 1"
50            });
51            journalEntry.commitLine({
52                sublistId: 'line'
53            });
54
55            var recordId = journalEntry.save();
56            return true;

```

```

57     }
58     return {
59       saveRecord : saveRecord
60     };
61   });

```

SuiteScript 2.x Client Script Best Practices

The following list shows best practices for both form-level and record-level client script development:

- When testing form-level client scripts, ensure that the latest scripts are being executed. Use Ctrl-Refresh to clear your browser cache.
- Use global (record-level) client scripts to get a more flexible deployment model and to port (bundle) scripts.
- [Record.setValue\(options\)](#) and [Record.setCurrentSublistValue\(options\)](#) execution is multi-threaded whenever child field values need to be sourced in. To synchronize your logic, use the postSourcing function or set the forceSyncSourcing synchronous parameter to true.
- Methods that set values accept raw data of a specific type and do not require formatting or parsing. Methods that set text accept strings but can conform to a user-specified format. For example, when setting a numerical field type, setValue() accepts any number, and setText() accepts a string in a specified format, such as "2,000.39" or "2.00,39". When setting a date field type, setValue() accepts any Date object, and setText() accepts a string in a specified format, such as "6/9/2016" or "9/6/2016".
- When you specify the modules that your script uses, make sure that the order of the modules matches the order of the parameters in your main function. For example, if your script uses the N/error, N/record, and N/file modules and you specify them in this order, make sure that your function parameters are in the same order, as follows:

```

1  require(['N/error', 'N/record', 'N/file'],
2        function(error, record, file) {
3          ...
4        });

```

- When debugging client scripts, you can insert a debugger; statement in your script, and execution will stop when this statement is reached:

```

1  function runMe() {
2    var k = 1;
3    k *= (k + 9);
4    debugger;
5    console.log(k);
6  }

```

When your script stops at the debugger; statement, you can examine your script properties and variables using the debugging tools in your browser. You can also use the debugger; statement in the SuiteScript Debugger to help you debug server scripts. For more information about the SuiteScript Debugger, see the help topic [SuiteScript Debugger](#).

- When debugging client scripts, some scripts might be minified. Minified scripts have all unnecessary characters removed, including white space characters, new line characters, and so on. Minifying scripts reduces the amount of data that needs to be processed and reduces file size, but it can also make scripts difficult to read. You can use your browser to de-minify scripts so that they're more readable. To learn how to de-minify scripts, see the documentation for your browser.
- Use execution context filtering to specify how and when a client script is executed. Execution contexts provide information about how a script is triggered to execute. For example, a script can be triggered in response to an action in the NetSuite application, or an action occurring in another context, such as a web services integration. You can use execution context filtering to ensure that your scripts are triggered only when necessary. For more information, see the help topic [Execution Contexts](#).

SuiteScript 2.x Client Script Entry Points and API

| Script Entry Point | Description |
|--------------------------|---|
| fieldChanged | Defines the function that is executed when a field is changed by a user or client call. |
| lineInit | Defines the function that is executed when an existing line is selected. |
| localizationContextEnter | Defines the function that is executed when the record enters the localization context that is specified on the script deployment record. |
| localizationContextExit | Defines the function that is executed when the record exits that context. |
| pageInit | Defines the function that is executed when the page completes loading or when the form is reset. |
| postSourcing | Defines the function that is executed when a field that sources information from another field is modified. Executes on transaction forms only. |
| saveRecord | Defines the function that is executed when a record is saved (after the submit button is pressed but before the form is submitted). |
| sublistChanged | Defines the function that is executed after a sublist has been inserted, removed, or edited. |
| validateDelete | Defines the validation function that is executed when an existing line in an edit sublist is deleted. |
| validateField | Defines the validation function that is executed when a field is changed by a user or client call. |
| validateInsert | Defines the validation function that is executed when a sublist line is inserted into an edit sublist. |
| validateLine | Defines the validation function that is executed before a line is added to an inline editor sublist or editor sublist. |

fieldChanged

 **Note:** The content in this topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | <p>Defines the function that is executed when a field is changed by a user or client call. This event may also execute directly through beforeLoad user event scripts. The following sample tasks can be performed:</p> <ul style="list-style-type: none"> ■ Provide the user with additional information based on user input. ■ Disable or enable fields based on user input. <p>For an example, see SuiteScript Client Script Sample.</p> <p> Note: This event does not execute when the field value is changed or entered in the page URL. Use the pageInit function to handle URLs that may contain updated field values. See pageInit.</p> |
| Returns | void |
| Since | Version 2015 Release 2 |

Parameters

Note: The scriptContext parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|-----------------------------|-----------------------------|--|------------------------|
| scriptContext.currentRecord | currentRecord.CurrentRecord | The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members . | Version 2015 Release 2 |
| scriptContext.sublistId | string | The sublist ID name. | Version 2015 Release 2 |
| scriptContext.fieldId | string | The field ID name. | Version 2015 Release 2 |
| scriptContext.line | string | The line number (zero-based index) if the field is in a sublist or a matrix. If the field is not a sublist or matrix, the default value is undefined. | Version 2015 Release 2 |
| scriptContext.column | string | The column number (zero-based index) if the field is in a matrix. If the field is not in a matrix, the default value is undefined. | Version 2015 Release 2 |

lineInit

Note: The content in this topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines the function that is executed when an existing line is selected. This event can behave like a pageInit event for line items in an inline editor sublist or editor sublist. For an example, see SuiteScript Client Script Sample . |
| Returns | void |
| Since | Version 2015 Release 2 |

Parameters

Note: The scriptContext parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|-----------------------------|-----------------------------|--------------------------|------------------------|
| scriptContext.currentRecord | currentRecord.CurrentRecord | The current form record. | Version 2015 Release 2 |

| Parameter | Type | Description | Since |
|-------------------------|--------|--|------------------------|
| | | For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members . | |
| scriptContext.sublistId | string | The sublist ID name. | Version 2015 Release 2 |

pageInit



Note: The content in this topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines the function that is executed after the page completes loading or when the form is reset. The following sample tasks can be performed: <ul style="list-style-type: none">■ Populate field defaults.■ Disable or enable fields.■ Change field availability or values depending on the data available for the record.■ Add flags to set initial values of fields.■ Provide alerts where the data being loaded is inconsistent or corrupt.■ Retrieve user login information and change field availability or values accordingly.■ Validate that fields required for your custom code (but not necessarily required for the form) exist. For an example, see SuiteScript Client Script Sample . |
| Returns | void |
| Since | Version 2015 Release 2 |

Parameters



Note: The scriptContext parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|-----------------------------|---|--|------------------------|
| scriptContext.currentRecord | currentRecord.CurrentRecord | The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members . | Version 2015 Release 2 |
| scriptContext.mode | string | The mode in which the record is being accessed. The mode can be set to one of the following values: <ul style="list-style-type: none">■ copy■ create■ edit | Version 2015 Release 2 |



Note: For a complete script example, see [SuiteScript Client Script Sample](#).

postSourcing



Note: The content in this topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|--|
| Description | <p>Defines the function that is executed when a field that sources information from another field is modified. Executes on transaction forms only.</p> <p>This event behaves like a fieldChanged event after all dependent field values have been set. The event waits for any cascaded field changes to complete before calling the user defined function.</p> <p>For an example, see SuiteScript Client Script Sample.</p> <div data-bbox="408 686 1380 770" style="border: 1px solid #ccc; padding: 5px;"> <p>Note: The event is not triggered by field changes for a field that does not have any cascaded fields.</p> </div> |
| Returns | void |
| Since | Version 2015 Release 2 |

Parameters



Note: The scriptContext parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|-----------------------------|---|--|------------------------|
| scriptContext.currentRecord | currentRecord.CurrentRecord | <p>The current form record.</p> <p>For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members.</p> | Version 2015 Release 2 |
| scriptContext.sublistId | string | The sublist ID name. | Version 2015 Release 2 |
| scriptContext.fieldId | string | The field ID name. | Version 2015 Release 2 |

saveRecord



Note: The content in this topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | <p>Defines the function that is executed when a record is saved (after the submit button is pressed but before the form is submitted).</p> <p>The following sample tasks can be performed:</p> <ul style="list-style-type: none"> ■ Provide alerts before committing the data. |
|--------------------|---|

| | |
|----------------|--|
| | <ul style="list-style-type: none"> ■ Enable fields that were disabled with other functions. ■ Redirect the user to a specified URL. <p>For an example, see SuiteScript Client Script Sample.</p> |
| Returns | true if the record is valid and is saved. false otherwise. |
| Since | Version 2015 Release 2 |

Parameters

Note: The scriptContext parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|-----------------------------|---|--|------------------------|
| scriptContext.currentRecord | currentRecord.CurrentRecord | The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members . | Version 2015 Release 2 |

sublistChanged

Note: The content in this topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Note: The sublistChanged function replaces the SuiteScript 1.0 function recalc.

| | |
|--------------------|--|
| Description | Defines the function that is executed after a sublist is inserted, removed, or edited. For an example, see SuiteScript Client Script Sample . |
| Returns | void |
| Since | Version 2015 Release 2 |

Parameters

Note: The scriptContext parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|-----------------------------|---|--|------------------------|
| scriptContext.currentRecord | currentRecord.CurrentRecord | The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members . | Version 2015 Release 2 |

| Parameter | Type | Description | Since |
|-------------------------|--------|----------------------|------------------------|
| scriptContext.sublistId | string | The sublist ID name. | Version 2015 Release 2 |

validateDelete



Note: The content in this topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines the validation function that is executed when an existing line in an edit sublist is deleted. For an example, see SuiteScript Client Script Sample . |
| Returns | true if the sublist line is valid and the delete is successful. false otherwise. |
| Since | Version 2015 Release 2 |

Parameters



Note: The scriptContext parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|-----------------------------|--|--|------------------------|
| scriptContext.currentRecord | currentRecord . CurrentRecord | The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members . | Version 2015 Release 2 |
| scriptContext.sublistId | string | The sublist ID name. | Version 2015 Release 2 |
| scriptContext.lineCount | number | The number of lines to be deleted. This property is defined only when validateDelete is triggered by the Clear All Lines button in the UI. | Version 2020 Release 2 |

validateField



Note: The content in this topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines the validation function that is executed when a field is changed by a user or client call. This event executes on fields added in beforeLoad user event scripts. The following sample tasks can be performed: <ul style="list-style-type: none">■ Validate field lengths. |
|--------------------|---|

| | |
|----------------|---|
| | <ul style="list-style-type: none"> ■ Restrict field entries to a predefined format. ■ Restrict submitted values to a specified range ■ Validate the submission against entries made in an associated field. <p>For an example, see SuiteScript Client Script Sample.</p> <p>Note: This event does not apply to dropdown select or check box fields.</p> |
| Returns | true if the field is valid and the change is successful. false otherwise. |
| Since | Version 2015 Release 2 |

Parameters

| |
|--|
| Note: The scriptContext parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite. |
|--|

| Parameter | Type | Description | Since |
|-----------------------------|-----------------------------|--|------------------------|
| scriptContext.currentRecord | currentRecord.CurrentRecord | The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members . | Version 2015 Release 2 |
| scriptContext.sublistId | string | The sublist ID name. | Version 2015 Release 2 |
| scriptContext.fieldId | string | The field ID name. | Version 2015 Release 2 |
| scriptContext.line | string | The line number (zero-based index) if the field is in a sublist or a matrix. If the field is not a sublist or matrix, the default value is undefined. | Version 2015 Release 2 |
| scriptContext.column | string | The column number (zero-based index) if the field is in a matrix. If the field is not in a matrix, the default value is undefined. | Version 2015 Release 2 |

validateInsert

| |
|--|
| Note: The content in this topic applies to SuiteScript 2.0 and SuiteScript 2.1. |
|--|

| | |
|--------------------|--|
| Description | Defines the validation function that is executed when a sublist line is inserted into an edit sublist. For an example, see SuiteScript Client Script Sample . |
| Returns | true if the sublist line is valid and the insertion is successful. false otherwise. |

| | |
|--------------|------------------------|
| Since | Version 2015 Release 2 |
|--------------|------------------------|

Parameters

Note: The scriptContext parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|-----------------------------|-----------------------------|--|------------------------|
| scriptContext.currentRecord | currentRecord.CurrentRecord | The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members . | Version 2015 Release 2 |
| scriptContext.sublistId | string | The sublist ID name. | Version 2015 Release 2 |

validateLine

Note: The content in this topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines the validation function that is executed before a line is added to an inline editor sublist or editor sublist. This event can behave like a saveRecord event for line items in an inline editor sublist or editor sublist. For an example, see SuiteScript Client Script Sample . |
| Returns | true if the sublist line is valid and the addition is successful. false otherwise. |
| Since | Version 2015 Release 2 |

Parameters

Note: The scriptContext parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|-----------------------------|-----------------------------|--|------------------------|
| scriptContext.currentRecord | currentRecord.CurrentRecord | The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members . | Version 2015 Release 2 |
| scriptContext.sublistId | string | The sublist ID name. | Version 2015 Release 2 |

localizationContextEnter



Note: The content in this topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|--|
| Description | Defines the function that is executed when the record enters the localization context that is specified on the script deployment record. |
| Returns | void |
| Since | Version 2020 Release 1 |

Parameters

| Parameter | Type | Description | Since |
|-----------------------------|-----------------------------|--|------------------------|
| scriptContext.currentRecord | currentRecord.CurrentRecord | The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members . | Version 2015 Release 2 |
| scriptContext.locale | string | The list of countries that represent the new localization context. | Version 2020 Release 1 |

If a script deployment is localized on the Context Filtering tab, the pageInit entry point of the script is ignored, and no other entry points of the script are called before localizationContextEnter or after localizationContextExit. It is possible that the record may never enter a localization context. In this case, no callbacks of the script are executed. For more information, see the help topic [Using the Context Filtering Tab](#).

localizationContextExit



Note: The content in this topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines the function that is executed when the record exits that context. |
| Returns | void |
| Since | Version 2020 Release 1 |

Parameters

| Parameter | Type | Description | Since |
|-----------------------------|-----------------------------|--|------------------------|
| scriptContext.currentRecord | currentRecord.CurrentRecord | The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members . | Version 2015 Release 2 |

| Parameter | Type | Description | Since |
|----------------------|--------|--|------------------------|
| scriptContext.locale | string | The list of countries that represent the new localization context. | Version 2020 Release 1 |

If a script deployment is localized on the Context Filtering tab, the pageInit entry point of the script is ignored, and no other entry points of the script are called before localizationContextEnter or after localizationContextExit. It is possible that the record may never enter a localization context. In this case, no callbacks of the script are executed. For more information, see the help topic [Using the Context Filtering Tab](#).

SuiteScript 2.x Map/Reduce Script Type

The map/reduce script type is designed for scripts that need to handle large amounts of data. It is best suited for situations where the data can be divided into small, independent parts. When the script is executed, a structured framework automatically creates enough jobs to process all of these parts. You as the user do not have to manage this process. Another advantage of map/reduce is that these jobs can work in parallel. You choose the level of parallelism when you deploy the script.

Like a scheduled script, a map/reduce script can be invoked manually or on a predefined schedule. However, map/reduce scripts offer several advantages over scheduled scripts. One advantage is that, if a map/reduce job violates certain aspects of NetSuite governance, the map/reduce framework automatically causes the job to yield and its work to be rescheduled for later, without disruption to the script. However, be aware that some aspects of map/reduce governance cannot be handled through automatic yielding. For that reason, if you use this script type, you should familiarize yourself with the [SuiteScript 2.x Map/Reduce Governance](#) guidelines.

In general, you should use map/reduce for any scenario where you want to process multiple records, and where your logic can be separated into relatively lightweight segments. In contrast, map/reduce is not as well suited to situations where you want to enact a long, complex function for each part of your data set. A complex series of steps might be one that includes the loading and saving of multiple records.

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

You can use SuiteScript Analysis to learn about when the script was installed and how it performed in the past. For more information, see the help topic [Analyzing Scripts](#).

For more information about map/reduce scripts, see the following topics:

- [Map/Reduce Use Cases](#)
- [Map/Reduce Key Concepts](#)
- [Map/Reduce Entry Points](#)
- [Map/Reduce Script Samples](#)

Map/Reduce Use Cases

Map/reduce is ideal for scenarios where you want to apply the same logic repeatedly, once for each object in a series. For example, you could use a map/reduce script to do any of the following:

- Identify a list of requisitions and transform each requisition into a purchase order.

- Search for invoices that meet certain criteria and apply a discount to each one.
- Search for customer records that appear to be duplicates, then process each apparent duplicate according to your business rules.
- Search for outstanding tasks assigned to sales reps, then send each person an email that summarizes their outstanding work.
- Identify files in the NetSuite File Cabinet, use the content of the files to create new documents, and upload the new documents to an external server.

Map/Reduce Key Concepts

Inspired by the map/reduce paradigm, the general idea behind a map/reduce script is as follows: Your script identifies some data that requires processing. This data is split into key/value pairs. Your script defines a function that the system invokes one time for each key/value pair. Optionally, your script can also use a second round of processing. Depending on how you deploy the script, the system can create multiple jobs for each round of processing and process the data in parallel.

If you are familiar with other SuiteScript 2.x script types, then you may already recognize that map/reduce scripts are significantly different from most types. Before you begin writing a map/reduce script, make sure you understand these differences. Consider the following:

- Map/reduce scripts are executed in stages
- The system supplements your logic
- The system provides robust context objects
- Multiple jobs are used to execute one script
- Map/reduce scripts permit yielding and other interruptions

Map/reduce scripts are executed in stages

With most script types, each script is executed as a single continuous process. In contrast, a map/reduce script is executed in five discrete stages. These stages occur in a specific sequence.

You can control the script's behavior in four of the five stages. That is, each of these four stages corresponds to an entry point. Your corresponding function defines the script's behavior during that stage. For example:

- For the **getInputData** stage, you write a function that returns an object that can be transformed into a list of key/value pairs. For example, if your function returns a search of NetSuite records, the system would run the search. The key/value pairs would be the results of the search: Each key would be the internal ID of a record. Each value would be a JSON representation of the record's field IDs and values.
- For the **map** stage, you can optionally write a function that the system invokes one time for each key/value pair. If appropriate, your map function can write output data, in the form of new key/value pairs. If the script also uses a reduce function, this output data is sent as input to the shuffle and then the reduce stage. Otherwise, the new key/value pairs are sent directly to the summarize stage.
- You do not write a function for the **shuffle** stage. In this stage, the system sorts through any key/value pairs that were sent to the reduce stage, if a reduce function is defined. These pairs may have been provided by the map function, if a map function is used. If a map function was not used, the shuffle stage uses data provided by the getInputData stage. The shuffle stage groups this data by key to form a new set of key/value pairs, where each key is unique and each value is an array. For example, suppose 100 key/value pairs were sent. Suppose that each key represents an employee, and each value represents a record that the employee created. If there were only two unique employees, and one employee created 90 records, while another employee created 10 records, then the shuffle stage

would provide two key/value pairs. The keys would be the IDs of the employees. One value would be an array of 90 elements, and the other would be an array of 10 elements.

- For the **reduce** stage, you write a function that is invoked one time for each key/value pair that was provided by the shuffle stage. Optionally, this function can write data as key/value pairs that are sent to the summarize stage.
- In the **summarize** stage, your function can retrieve and log statistics about the script's work. It can also take actions with data sent by the reduce stage.

Note that you may omit either the map or reduce function. You can also omit the summarize function. For more details, review [Map/Reduce Entry Points](#).

The system supplements your logic

With most script types, the functionality of the script is determined entirely by the code within your script file. Map/reduce is different. With map/reduce, the logic in your file is important, but the system also supplements your logic with standardized logic of its own. For example, the system moves data between the stages. Additionally, the system invokes your map and reduce function multiple times. For this reason, think of the logic of the map and reduce functions as being similar to the logic you would use in a loop. Each of these functions should perform a relatively small amount of work. For details about the system's behavior during and between the stages, see [SuiteScript 2.x Map/Reduce Script Stages](#).

The system provides robust context objects

For each entry point function you write, the system provides a context object. In itself, this fact is not ground-breaking – the system makes a context object available to most SuiteScript 2.x entry points. However, the objects provided to map/reduce entry point functions are especially robust. These objects contain data and properties that are critical to writing an effective script. For example, you can use these objects to access data from the previous stage and write output data that is sent to the next stage. Context objects can also contain data about errors, usage units consumed, and other statistics. For details, see [SuiteScript 2.x Map/Reduce Script Entry Points and API](#).

Multiple jobs are used to execute one script

All map/reduce scripts are powered by [SuiteCloud Processors](#), which handle work through a series of **jobs**. Each job is executed by a **processor**, which is a virtual unit of processing power. SuiteCloud Processors are also used to process scheduled scripts. However, these two script types are handled differently. For example, the system always creates only one job to handle a scheduled script. In contrast, the system creates multiple jobs to process a single map/reduce script. Specifically, the system creates at least one job to execute each stage. Additionally, multiple jobs can be created to handle the work of the map stage, and multiple jobs can be created for the reduce stage. When the system creates multiple map and reduce jobs, these jobs work independently of each other and may work in parallel across multiple processors. For this reason, the map and reduce stages are considered **parallel** stages.

In contrast, the `getInputData` and `summarize` stages are each executed by one job. In each case, that job invokes your function only one time. These stages are **serial** stages. The shuffle stage is also a serial stage.

Map/reduce scripts permit yielding and other interruptions

Since the map and reduce stages consist of multiple independent map and reduce function invocations, the work of these stages can easily be divided among multiple jobs. The structure is naturally flexible. It allows for parallel processing, and it also permits map and reduce jobs to manage some aspects of their own resource consumption.

If a job monopolizes a processor for too long, the system can naturally finish the job after the current map or reduce function has completed. In this case, the system creates a new job to continue executing

remaining key/value pairs. Based on its priority and submission timestamp, the new job either starts right after the original job has finished, or it starts later, to allow higher-priority jobs processing other scripts to execute. For more details, see [SuiteScript 2.x Map/Reduce Yielding](#).

At the same time, be aware that the system imposes some usage limits on map/reduce scripts that are not managed through yielding. For details, see [SuiteScript 2.x Map/Reduce Governance](#).

Map/Reduce Entry Points

A map/reduce script can go through a total of five stages. One stage, shuffle, does not correspond with an entry point. The other stages do. Their entry points are described in the following table.

| Entry point | Purpose of corresponding function | Required? |
|---|--|--|
| <code>getInputData(inputContext)</code> | To identify data that needs processing. The system passes this data to the next stage. | yes |
| <code>map(mapContext)</code> | To apply processing to each key/value pair provided, and optionally pass data to the next stage. | One of these two entry points is required. You can also use both entry points. |
| <code>reduce(reduceContext)</code> | To apply processing to each key/value pair provided. In this stage, each key is unique, and each value is an array of values. This function can optionally pass data to the summarize stage. | |
| <code>summarize(summaryContext)</code> | To retrieve data about the script's execution, and take any needed actions with the output of the reduce stage. | no |

For full details on the map/reduce entry points and their corresponding context objects, see [SuiteScript 2.x Map/Reduce Script Entry Points and API](#).

Map/Reduce Script Samples

See the following samples:

- [Counting Characters Example](#)
- [Processing Invoices Example](#)

These script samples use SuiteScript 2.x. A newer version, SuiteScript 2.1, is also available and supports new language features that are included in the ES2019 specification. You can write map/reduce scripts using either SuiteScript 2.0 or SuiteScript 2.1.

- For help with writing scripts in SuiteScript 2.x, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).
- For more information about SuiteScript versions and SuiteScript 2.1, see [SuiteScript Versioning Guidelines](#) and [SuiteScript 2.1](#).

Counting Characters Example

The following sample is a very simple map/reduce script. This sample does not accomplish a realistic business objective but rather is designed to demonstrate how the script type works.

This script defines a hard-coded string. The script counts the number of times that each letter of the alphabet occurs within the string. Then it creates a file that shows its results. Refer to the comments in the sample for details about how the system processes the script.

```

1  /**
2  * @NApiVersion 2.x
3  * @NScriptType MapReduceScript
4  */
5
6 define(['N/file'], function(file) {
7
8     // Define characters that should not be counted when the script performs its
9     // analysis of the text.
10    const PUNCTUATION_REGEXP = /[\u2000-\u206F\u2E00-\u2E7F\\'!"#\$\&\(\)\*\+\,\-\.\,\:/;:<>\?@\[\]\^_\`{\|\}\~]/g;
11
12    // Use the getInputData function to return two strings.
13    function getInputData() {
14        return "the quick brown fox \njumps over the lazy dog.".split('\n');
15    }
16
17    // After the getInputData function is executed, the system creates the following
18    // key/value pairs:
19    //
20    // key: 0, value: 'the quick brown fox'
21    // key: 1, value: 'jumps over the lazy dog.'
22
23    // The map function is invoked one time for each key/value pair. Each time the
24    // function is invoked, the relevant key/value pair is made available through
25    // the context.key and context.value properties.
26    function map(context) {
27
28        // Create a loop that examines each character in the string. Exclude spaces
29        // and punctuation marks.
30        for (var i = 0; context.value && i < context.value.length; i++) {
31            if (context.value[i] !== ' ' && !PUNCTUATION_REGEXP.test(context.value[i])) {
32
33                // For each character, invoke the context.write() method. This method saves
34                // a new key/value pair. For the new key, save the character currently being
35                // examined by the loop. For each value, save the number 1.
36
37                context.write({
38                    key: context.value[i],
39                    value: 1
40                });
41            }
42        }
43    }
44
45    // After the map function has been invoked for the last time, the shuffle stage
46    // begins. In this stage, the system sorts the 35 key/value pairs that were saved
47    // by the map function during its two invocations. From those pairs, the shuffle
48    // stage creates a new set of key/value pairs, where the each key is unique. In
49    // this way, the number of key/value pairs is reduced to 25. For example, the map
50    // stage saved three instances of ['e','1']. In place of those pairs, the shuffle
51    // stage creates one pair: ['e', [1,1,1]]. These pairs are made available to
52    // the reduce stage through the context.key and context.values properties.
53
54    // The reduce function is invoked one time for each of the 25 key/value pairs
55    // provided.
56    function reduce(context) {
57
58        // Use the context.write() method to save a new key/value pair, where the new key
59        // equals the key currently being processed by the function. This key is a letter
60        // in the alphabet. Make the value equal to the length of the context.values array.
61        // This number represents the number of times the letter occurred in the original
62        // string.
63
64        context.write({
65            key: context.key,
66            value: context.values.length
67        });
68    }
}

```

```

69 // The summarize stage is a serial stage, so this function is invoked only one
70 // time.
71 function summarize(context) {
72
73     // Log details about the script's execution.
74     log.audit({
75         title: 'Usage units consumed',
76         details: context.usage
77     });
78     log.audit({
79         title: 'Concurrency',
80         details: context.concurrency
81     });
82     log.audit({
83         title: 'Number of yields',
84         details: context.yields
85     });
86
87
88     // Use the context object's output iterator to gather the key/value pairs saved
89     // at the end of the reduce stage. Also, tabulate the number of key/value pairs
90     // that were saved. This number represents the total number of unique letters
91     // used in the original string.
92     var text = '';
93     var totalKeysSaved = 0;
94     context.output.iterator().each(function(key, value) {
95         text += (key + ' ' + value + '\n');
96         totalKeysSaved++;
97         return true;
98     });
99
100    // Log details about the total number of pairs saved.
101    log.audit({
102        title: 'Unique number of letters used in string',
103        details: totalKeysSaved
104    });
105
106    // Use the N/file module to create a file that stores the reduce stage output,
107    // which you gathered by using the output iterator.
108    var fileObj = file.create({
109        name: 'letter_count_result.txt',
110        fileType: file.Type.PLAINTEXT,
111        contents: text
112    });
113
114    fileObj.folder = -15;
115    var fileId = fileObj.save();
116
117    log.audit({
118        title: 'Id of new file record',
119        details: fileId
120    });
121 }
122
123 // Link each entry point to the appropriate function.
124 return {
125     getInputData: getInputData,
126     map: map,
127     reduce: reduce,
128     summarize: summarize
129 };
130 });

```

Processing Invoices Example

The following example shows a sample script that processes invoices and contains logic to handle errors. This script is designed to do the following:

- Find the customers associated with all open invoices.
- Apply a location-based discount to each invoice.

- Write each invoice to the reduce stage so it is grouped by customer.
- Initialize a new CustomerPayment for each customer applied only to the invoices specified in the reduce values.
- Create a custom record capturing the details of the records that were processed.
- Notify administrators of any exceptions via an email notification.

Prior to running this sample, you need to manually create a custom record type with id "customrecord_summary", and text fields with id "custrecord_time", "custrecord_usage", and "custrecord_yields".

Script Sample Prerequisites

1. From the NetSuite UI, select Customization > List, Records, & Fields > Record Types > New.
2. From the Custom Record Type page, enter a value for name.
3. In the ID field, enter "customrecord_summary".
4. Select Save.
5. From the Fields subtab, do the following:
 - Select New Field. Enter a label and set ID to "custrecord_time". Ensure that the Type field is set to Free-Form Text. Select Save & New.
 - Select New Field. Enter a label and set ID to "custrecord_usage". Ensure that the Type field is set to Free-Form Text. Select Save & New.
 - Select New Field. Enter a label and set ID to "custrecord_yields". Ensure that the Type field is set to Free-Form Text. Select Save.

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType MapReduceScript
4   */
5 define(['N/search', 'N/record', 'N/email', 'N/runtime', 'N/error'],
6       function(search, record, email, runtime, error)
7     {
8       function handleErrorAndSendNotification(e, stage)
9     {
10       log.error('Stage: ' + stage + ' failed', e);
11
12       var author = -5;
13       var recipients = 'notify@xxxxxx.com';
14       var subject = 'Map/Reduce script ' + runtime.getCurrentScript().id + ' failed for stage: ' + stage;
15       var body = 'An error occurred with the following information:\n' +
16           'Error code: ' + e.name + '\n' +
17           'Error msg: ' + e.message;
18
19       email.send({
20         author: author,
21         recipients: recipients,
22         subject: subject,
23         body: body
24       });
25
26
27       function handleErrorIfAny(summary)
28     {
29         var inputSummary = summary.inputSummary;
30         var mapSummary = summary.mapSummary;
31         var reduceSummary = summary.reduceSummary;
32
33         if (inputSummary.error)
34         {
35             var e = error.create({
36               name: 'INPUT_STAGE_FAILED',
37               message: inputSummary.error
38             });
39             handleErrorAndSendNotification(e, 'getInputData');

```

```

40     }
41
42     handleErrorInStage('map', mapSummary);
43     handleErrorInStage('reduce', reduceSummary);
44   }
45
46   function handleErrorInStage(stage, summary)
47   {
48     var errorMsg = [];
49     summary.errors.iterator().each(function(key, value){
50       var msg = 'Failure to accept payment from customer id: ' + key + '. Error was: ' + JSON.parse(value).message
51       + '\n';
52       errorMsg.push(msg);
53       return true;
54     });
55     if (errorMsg.length > 0)
56     {
57       var e = error.create({
58         name: 'RECORD_TRANSFORM_FAILED',
59         message: JSON.stringify(errorMsg)
60       });
61       handleErrorAndSendNotification(e, stage);
62     }
63
64   function createSummaryRecord(summary)
65   {
66     try
67     {
68       var seconds = summary.seconds;
69       var usage = summary.usage;
70       var yields = summary.yields;
71
72       var rec = record.create({
73         type: 'customrecord_summary',
74       );
75
76       rec.setValue({
77         fieldId : 'name',
78         value: 'Summary for M/R script: ' + runtime.getCurrentScript().id
79       });
80
81       rec.setValue({
82         fieldId: 'custrecord_time',
83         value: seconds
84       );
85       rec.setValue({
86         fieldId: 'custrecord_usage',
87         value: usage
88       );
89       rec.setValue({
90         fieldId: 'custrecord_yields',
91         value: yields
92       );
93
94       rec.save();
95     }
96     catch(e)
97     {
98       handleErrorAndSendNotification(e, 'summarize');
99     }
100   }
101
102   function applyLocationDiscountToInvoice(recordId)
103   {
104     var invoice = record.load({
105       type: record.Type.INVOICE,
106       id: recordId,
107       isDynamic: true
108     );
109
110     var location = invoice.getText({
111       fieldId: 'location'

```

```

112     });
113
114     var discount;
115     if (location === 'East Coast')
116         discount = 'Eight Percent';
117     else if (location === 'West Coast')
118         discount = 'Five Percent';
119     else if (location === 'United Kingdom')
120         discount = 'Nine Percent';
121     else
122         discount = '';
123
124     invoice.setText({
125         fieldId: 'discountitem',
126         text: discount,
127         ignoreFieldChange : false
128     });
129     log.debug(recordId + ' has been updated with location-based discount.');
130     invoice.save();
131 }
132
133 function getInputData()
134 {
135     return search.create({
136         type: record.Type.INVOICE,
137         filters: [[ 'status', search.Operator.IS, 'open' ]],
138         columns: ['entity'],
139         title: 'Open Invoice Search'
140     });
141 }
142
143 function map(context)
144 {
145     var searchResult = JSON.parse(context.value);
146     var invoiceId = searchResult.id;
147     var entityId = searchResult.values.entity.value;
148
149     applyLocationDiscountToInvoice(invoiceId);
150
151     context.write({
152         key: entityId,
153         value: invoiceId
154     });
155 }
156
157 function reduce(context)
158 {
159     var customerId = context.key;
160
161     var custPayment = record.transform({
162         fromType: record.Type.CUSTOMER,
163         fromId: customerId,
164         toType: record.Type.CUSTOMER_PAYMENT,
165         isDynamic: true
166     });
167
168
169     var lineCount = custPayment.getLineCount('apply');
170     for (var j = 0; j < lineCount; j++)
171     {
172         custPayment.selectLine({
173             sublistId: 'apply',
174             line: j
175         });
176         custPayment.setCurrentSublistValue({
177             sublistId: 'apply',
178             fieldId: 'apply',
179             value: true
180         });
181     }
182
183     var custPaymentId = custPayment.save();
184 }
```

```

185     context.write({
186         key: custPaymentId
187     });
188 }
189
190     function summarize(summary)
191     {
192         handleErrorIfAny(summary);
193         createSummaryRecord(summary);
194     }
195
196     return {
197         getInputData: getInputData,
198         map: map,
199         reduce: reduce,
200         summarize: summarize
201     };
202 });

```

SuiteScript 2.x Map/Reduce Script Reference

- [SuiteScript 2.x Map/Reduce Script Stages](#)
- [SuiteScript 2.x Map/Reduce Terminology](#)
- [SuiteScript 2.x Map/Reduce Yielding](#)
- [SuiteScript 2.x Map/Reduce Script Submission](#)
- [SuiteScript 2.x Map/Reduce Governance](#)
- [SuiteScript 2.x Map/Reduce Script Status Page](#)
- [SuiteScript 2.x Map/Reduce Script Error Handling](#)
- [SuiteScript 2.x Map/Reduce Script Best Practices](#)
- [Adding Logic to Handle Map/Reduce Restarts](#)

SuiteScript 2.x Map/Reduce Terminology

| Term | Definition | More Information |
|---------------------|--|--|
| buffer size | An option on the script deployment record. This choice determines the number of key/value pairs that a map or reduce job can process before information about the job's progress is saved to the database. As a general best practice, leave this value set to 1. | <ul style="list-style-type: none"> ■ Buffer Size |
| deployment instance | A task created to process a script deployment. A deployment instance is created when the deployment is submitted for processing. Only one unfinished instance of a particular script deployment can exist at any time. | <ul style="list-style-type: none"> ■ SuiteScript 2.x Map/Reduce Script Submission |
| exitOnError | A configuration option that affects the behavior of a script when an uncaught error interrupts a map or reduce function. When exitOnError is set to true, and all available retries have been used, the script exits the current stage and goes to the summarize stage. | <ul style="list-style-type: none"> ■ exitOnError |
| function invocation | An execution of a function. Some map/reduce entry point functions, such as the getInputData and summarize functions, are invoked only one time during the script's processing. Others, such as the map and reduce functions, are invoked multiple times. For example, if the getInputData stage provides 20 key/value pairs to the map stage, then | <ul style="list-style-type: none"> ■ SuiteScript 2.x Map/Reduce Script Type |

| Term | Definition | More Information |
|------------------------|---|--|
| | the map function is invoked 20 times, one time for each pair. | |
| getInputData | The first stage in the processing of a map/reduce script. During this stage, your script must return an object that can be transformed into a list of key/value pairs. You use the getInputData entry point to identify the function that executes during this stage. | ■ SuiteScript 2.x Map/Reduce Script Stages |
| governance | A system of rules governing your usage of NetSuite. Specific limits exist for every script type, including map/reduce. | ■ SuiteScript 2.x Map/Reduce Governance ■ SuiteScript Governance and Limits |
| hard limit | A type of map/reduce governance limit that, when violated, causes an interruption to the current function invocation. Make sure you follow the SuiteScript 2.x Map/Reduce Script Best Practices to avoid problems with hard limits. | ■ SuiteScript 2.x Map/Reduce Governance ■ SuiteScript 2.x Map/Reduce Script Best Practices |
| job | A unit of execution managed by SuiteCloud Processors . The processing of a map/reduce script is always handled by multiple jobs. At least one job is created to handle the processing of each stage. | ■ SuiteScript 2.x Map/Reduce Script Stages ■ SuiteCloud Processors |
| map | The second stage in the processing of a map/reduce script. A map function is optional, but your script must use either a map or reduce function (or both). When used, a map function processes data provided by the getInputData stage. The map function is invoked one time for each key/value pair that is provided. | ■ SuiteScript 2.x Map/Reduce Script Stages ■ map(mapContext) |
| map/reduce | A computing paradigm designed for the processing of large data sets. | ■ https://en.wikipedia.org/wiki/MapReduce |
| map/reduce script type | A SuiteScript 2.x script type based on the map/reduce paradigm. | ■ Map/Reduce Key Concepts |
| parallel stage | A type of stage that can be handled by multiple jobs that work simultaneously. The map and reduce stages are parallel stages. See also serial stage. | ■ SuiteScript 2.x Map/Reduce Script Stages |
| priority | A property of a job. Job priority determines the order in which the scheduler sends jobs to the processor pool. Priorities can be set on the deployment record or on the SuiteCloud Processors Priority Settings Page . | ■ Map/Reduce Script Deployment Record ■ SuiteCloud Processors Priority Levels |
| processor | In the context of SuiteCloud Processors, a virtual unit of processing power that executes a job. A processor is not an individual physical entity but rather a single processing thread. | ■ Concurrency Limit ■ SuiteCloud Processors |
| processor pool | In the context of SuiteCloud Processors, the total processors available to your NetSuite account. The number of processors available varies depending on your licensing. The number of processors available to process a particular script is determined by the value you choose for the Concurrency Limit field on the script deployment record. | ■ Concurrency Limit ■ SuiteCloud Processors |
| reduce | The fourth stage in the processing of a map/reduce script. A reduce function is optional, but your script must use either | ■ SuiteScript 2.x Map/Reduce Script Stages |

| Term | Definition | More Information |
|-----------------------|---|---|
| | a map or reduce function (or both). When used, a reduce function processes data provided by the getInputData stage or, if a map function is used, by the map stage. The reduce function is invoked one time for each unique key from the list of all key/value pairs provided. The value for each of the unique keys passed to the reduce function is an array of all values with that key. | ■ reduce(reduceContext) |
| retryCount | A configuration option that is relevant when a map or reduce function is interrupted by an uncaught error or by an application server restart. This setting determines whether the system invokes the map or reduce function again for any key/value pairs that were left in an uncertain state following the interruption. | ■ retryCount |
| script deployment | A set of configuration choices for how to execute a script. These choices are stored on a script deployment record. To execute a map/reduce script, you must create a deployment record and submit the deployment. | ■ SuiteScript 2.x Map/Reduce Script Submission |
| serial stage | A type of stage that is processed in its entirety by only one job. The getInputData, shuffle, and summarize stages are serial stages. | ■ SuiteScript 2.x Map/Reduce Script Stages |
| shuffle | The third stage in the processing of a map/reduce script. This stage is significant if your script uses a reduce function. In these cases, the shuffle stage sorts data provided by the getInputData stage or, if a map function is used, by the map stage. In either case, the shuffle stage creates a set of key/value pairs where each key is unique, and each value is an array. These pairs are passed to the reduce stage for further processing. | ■ SuiteScript 2.x Map/Reduce Script Stages |
| soft limit | A type of map/reduce governance rule that, after it is surpassed, causes a map or reduce job to automatically yield. When the job yields, its work is rescheduled. Progress toward a soft limit is checked only after a function invocation has completed. Therefore this limit never causes an interruption to the function invocation. | ■ SuiteScript 2.x Map/Reduce Governance |
| SuiteCloud Processors | The engine that processes map/reduce and scheduled scripts. | ■ SuiteCloud Processors |
| summarize | The final stage in the processing of a map/reduce script. You can use this stage to log data about the work of the script. | ■ SuiteScript 2.x Map/Reduce Script Stages ■ summarize(summaryContext) |
| task | The set of work that represents a map/reduce deployment that has been submitted for processing. Each task is executed by a minimum of five jobs, one for each stage. | ■ SuiteScript 2.x Map/Reduce Script Submission |
| yielding | A behavior that allows a map/reduce script to manage some aspects of its own resource consumption. After a map/reduce job has surpassed a soft limit, it automatically yields, and its work is rescheduled for later. Yielding never interrupts a function invocation. Note also that this process is automatic. SuiteScript 2.x does not contain an API that lets you force a script to yield. | ■ Yield After Minutes |

SuiteScript 2.x Map/Reduce Script Stages

The map/reduce script type goes through at least two of five possible stages.

The stages are processed in the following order. Note that each stage must complete before the next stage begins.

- Get Input Data – Acquires a collection of data. This stage is always processed first and is required. The input stage runs sequentially.
- Map – Parses each row of data into a key/value pair. One pair (key/value) is passed per function invocation. If this stage is skipped, the reduce stage is required. Data may be processed in parallel in this stage.
- Shuffle – Groups values based on keys. This is an automatic process that always follows completion of the map stage. There is no direct access to this stage as it is handled by the map/reduce script framework. Data is processed sequentially in this stage.
- Reduce – Evaluates the data in each group. One group (key/values) is passed per function invocation. If this stage is skipped, the map stage is required. Data is processed in parallel in this stage.
- Summarize – Summarizes the output of the previous stages. Developers can use this stage to summarize the data from the entire map/reduce process and write it to a file or send an email. This stage is optional and is not technically a part of the map/reduce process. The summarize stage runs sequentially.



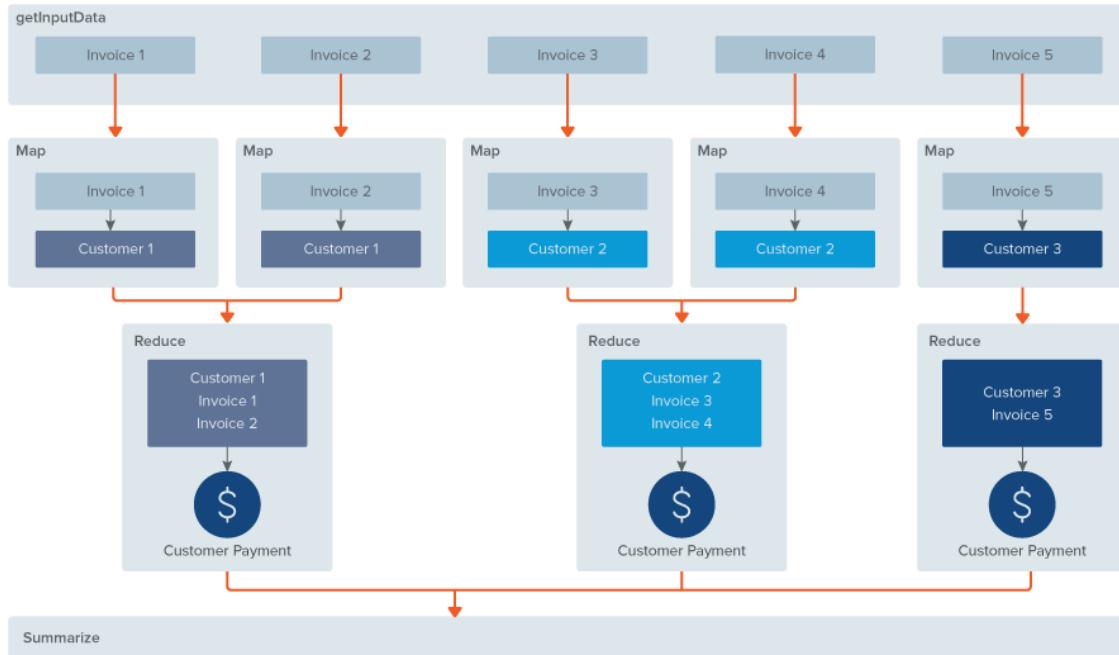
Note: It is not mandatory to use both the map stage and the reduce stage. You may skip one of those stages.

The following diagram illustrates these stages, in the context of processing a set of invoices.

In this example, the stages are used as follows:

- Get Input Data – A collection of invoices that require payment is loaded.
- Map – Each invoice is paired to the customer expected to pay it. The key/value pairs are returned, where customerID is the key and the invoice is the value. For five invoices, the map is invoked five times.
- Reduce – There are three unique groups of invoices based on customerID. For three groups, reduce is invoked three times. To create a customer payment for every group, custom logic iterates over each group using customerID as the key.
- Summarize – Custom logic fetches various metrics (for example, number of invoices paid) and sends the output as an email notification.

For a code sample similar to this example, see [Processing Invoices Example](#).



Passing Data to a Map or Reduce Stage

To prevent unintended alteration of data when it is passed between stages, key/value pairs are always serialized into strings. For map/reduce scripts, SuiteScript 2.x checks if the data passed to the next stage is a string, and uses `JSON.stringify()` to convert the key or value into a string as necessary.

Objects serialized to JSON remain in JSON format. To avoid possible errors, SuiteScript does not automatically deserialize the data. For example, an error might result from an attempt to convert structured data types (such as CSV or XML) that are not valid JSON. At your discretion, you can use `JSON.parse()` to convert the JSON string back into a native JS object.

SuiteScript 2.x Map/Reduce Yielding

A key advantage of the map/reduce script type is that it can manage some aspects of its own resource consumption. This behavior is achieved through a feature known as yielding.

To understand yielding, first be aware that all map/reduce scripts are processed by [SuiteCloud Processors](#). A processor is a virtual unit of processing power that executes a job.

With yielding, the system waits until after the completion of each map or reduce function invocation, then checks to see whether the job has exceeded certain limits. If it has, job gracefully ends its execution, making it possible for other jobs to gain access to the processor. A new job is created to take the place of the map/reduce job that ended. The new job has the same priority as the old one, but a later timestamp.

Yielding can occur after the following limits are surpassed:

- The time limit specified by the Yield After Minutes field on the script deployment record. You can set this value to any number between 3 and 60. The default is 60. For more details on this field, see [Yield After Minutes](#).
- The governance limit of 10,000 usage units per map or reduce job. For more details, see [Soft Limits on Long-Running Map and Reduce Jobs](#).

Because the system checks these limits only between function invocations, yielding never interrupts a function invocation. Additionally, be aware that a job does not yield until it has actually passed one of the relevant limits.

Yielding is unrelated to the governance limits that exist for a single invocation of a map or reduce function. Those limits are described in [Hard Limits on Function Invocations](#). Exceeding the limit for a single invocation of a map or reduce function causes the system to throw an `SSS_USAGE_LIMIT_EXCEEDED` error and ends the function invocation, even if it is not complete.

For help understanding how the map and reduce stages can each have multiple function invocations, review [Map/Reduce Key Concepts](#).



Important: Map/reduce yielding is automatic. There is no API for manually forcing a map/reduce job to yield. This behavior differs from the functionality that was available for SuiteScript 1.0 scheduled scripts.

SuiteScript 2.x Map/Reduce Script Submission

You can submit a map/reduce script for processing in any of the following ways:

- By [Scheduling a Map/Reduce Script Submission](#)
- By [Submitting an On-Demand Map/Reduce Script Deployment from the UI](#)
- By [Submitting an On-Demand Map/Reduce Script Deployment from a Script](#)

Note that you can create multiple deployments for one script record. This strategy is useful if you want to submit the same script for processing multiple times simultaneously, or within a short time. For details, see [Submitting Multiple Deployments of the Same Script](#).

Each of these processes requires you to use a map/reduce script deployment record. For details about the fields available on this record, see [Map/Reduce Script Deployment Record](#).



Important: All map/reduce script instances are executed, or processed, by SuiteCloud Processors. Before submitting a map/reduce script for processing, review [SuiteCloud Processors](#).

Map/Reduce Script Deployment Record



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

Before a map/reduce script can be executed, you must create at least one deployment record for the script.

The deployment record for a map/reduce script is similar to that of other script types. However, a map/reduce script deployment contains some additional fields. Some of these fields are specific to [SuiteCloud Processors](#), which are used to execute map/reduce scripts. Others are specific to map/reduce features. This topic describes all of the available fields.

You can access a map/reduce script deployment record in the following ways:

- To open an existing deployment record for editing, go to Customization > Scripting > Script Deployments. Locate the appropriate record, and click **Edit**.
- To start creating a new deployment record, open the appropriate script record in view mode, and click the **Deploy Script** button. For help creating a script record, see [Script Record Creation](#).

Body Fields

The following table summarizes the fields available on the map/reduce script deployment record. Note that some fields are available only when you edit or view an existing deployment record.

| Field | Description |
|---------------|--|
| Script | A link to the script record associated with the deployment. This value cannot be changed, even on a new deployment record. If you begin the process of creating a deployment and realize that you selected the wrong script record, you must start the process over. |
| Title | The user-defined name for the deployment. |
| ID | <p>A unique identifier for the deployment.</p> <p>On a new record, you can customize this identifier by entering a value in the ID field. You should customize the ID if you plan to bundle the deployment for installation into another account, because using custom IDs helps avoid naming conflicts. IDs must be lowercase and cannot use spaces. You can use an underscore as a delineator.</p> <p>If you do not enter a value, the system automatically generates one. In both cases, the system automatically adds the prefix <code>customdeploy</code> to the ID created when the record is saved.</p> <p>Although not recommended, you can change the ID on an existing deployment by clicking the Change ID button.</p> |
| Deployed | <p>A configuration option that indicates whether the deployment is active. This box must be checked if you want the script to execute. Otherwise, the system uses the following behavior:</p> <ul style="list-style-type: none"> ■ When the Deployed box is cleared, and the Status is Not Scheduled, the Save and Execute option is no longer available, and the deployment cannot be submitted programmatically. ■ When the Deployed box is cleared, and the deployment record's Status is Scheduled, any times configured on the Schedule subtab are ignored and the script deployment is not submitted. |
| Status | <p>A value that determines how and when a script deployment can be submitted for processing. The primary options are:</p> <ul style="list-style-type: none"> ■ Scheduled —The script deployment is submitted for processing at the times indicated on the Schedule subtab. ■ Not Scheduled — The script deployment is submitted for processing only when invoked manually, either through the UI or programmatically. <p>Note also that, regardless of the Status, the system submits the deployment for processing only if the Deployed box is checked.</p> <p>For more details on this choice, see Status.</p> |
| See Instances | <p>A link to the Map/Reduce Script Status Page, filtered for all instances of this deployment record, for the current day. You can change the filtering options if needed.</p> <p>For details on working with the Map/Reduce Script Status page, see SuiteScript 2.x Map/Reduce Script Status Page.</p> |
| Log Level | <p>A value that determines what type of log messages are displayed on the Execution Log of both the deployment record and associated script record. The available levels are:</p> <ul style="list-style-type: none"> ■ Debug — suitable for scripts still being tested; this level shows all debug, audit, error and emergency messages. ■ Use Audit, Error, or Emergency — suitable for scripts in production mode. Of these three, Audit is the most inclusive and Emergency the most exclusive. <p>For more details on each level, see Log Level.</p> |

| Field | Description |
|---------------------------|---|
| Execute As Role | Indicates the role used to run the script. For map/reduce deployments, this value is automatically set to Administrator and cannot be edited. |
| Priority | A measure of how urgently this script should be processed relative to other map/reduce and scheduled scripts that have been submitted. This value is assigned to each job associated with the script deployment. The priority affects when the SuiteCloud Processors scheduler sends these jobs to the processor pool. For more details, see Priority . |
| |  Important: You must understand SuiteCloud Processors before you change this setting. For details, see the help topic SuiteCloud Processors Priority Levels . |
| Concurrency Limit | Determines the number of SuiteCloud Processors that can be used to process the jobs associated with the script deployment. For more details on this field, see Concurrency Limit . |
| Submit All Stages At Once | Determines whether the system creates jobs for all of the map/reduce stages simultaneously when the script deployment is submitted for processing. In general, clear this field only if the script deployment is relatively low in priority. For more details on this field, see Submit All Stages At Once . |
| Yield After Minutes | A soft time limit on how long the script deployment's map and reduce jobs may run before yielding. You can enter a number from 3 to 60. In general, each time a map or reduce job finishes a function invocation, the system checks to see whether this time limit has been exceeded. If it has, the job yields so that other jobs can be processed. A new job is created to take on the work that was being processed by the job that yielded. For more details on this field, see Yield After Minutes . |
| Buffer Size | A value that indicates how many key/value pairs a map or reduce job can process before information about the job's progress is saved to the database. A low Buffer Size value minimizes the risk of any pairs being processed twice, which can lead to data duplication. In general, leave this value set to 1 unless you have special circumstances that dictate otherwise. For more details on this field, see Buffer Size . |

Status

The Status field determines how and when the script deployment may be submitted for processing. The default value is Not Scheduled.

Regardless of how the script deployment is submitted, it **does not necessarily execute** at the exact time scheduled, or at the exact time that it is manually invoked. There may be a short system delay, even if no scripts are before it. If there are scripts already waiting to be executed, the script may need to wait until others have completed. For details on this behavior, see the help topic [SuiteCloud Processors](#).

Scheduled

When a deployment's Status is set to **Scheduled**, the script deployment is submitted for processing according to a one-time or recurring schedule. You define this schedule by using the deployment record's [Schedule Subtab](#). With this approach, after you save the deployment, you do not have to take any other steps for the deployment to be submitted for processing.

Note also:

- If you schedule a recurring submission with an end date, or a one-time submission, the deployment record's status remains Scheduled even after the script completes its execution.
- You cannot submit an on-demand instance of the script deployment when it has a status of Scheduled.

See also [Scheduling a Map/Reduce Script Submission](#).

Not Scheduled

When a deployment's Status is set to **Not Scheduled**, the deployment is available to be submitted on an on-demand basis. If you want the deployment to be submitted for processing, you must manually submit it, either through the NetSuite UI or programmatically. You can use either of the following:

- The **Save and Execute** option on the deployment record. See also [Submitting an On-Demand Map/Reduce Script Deployment from the UI](#).
- The `task.ScheduledScriptTask` API. See also [Submitting an On-Demand Map/Reduce Script Deployment from a Script](#).

You can submit the script deployment only if there is no other unfinished instance of the same script deployment. If you want multiple instances of the script to be submitted for processing simultaneously, you must create multiple deployment records for the script record. For details, see [Submitting Multiple Deployments of the Same Script](#).

Testing

When a deployment's Status is set to **Not Scheduled**, the script deployment is submitted for processing only when invoked manually, either through the UI or programmatically. You can submit the script deployment only if there is no other unfinished instance of the same script deployment.

To be submitted programmatically, a deployment with a status of Testing must be explicitly identified by the `task.MapReduceScriptTask` object's `MapReduceScriptTask.deploymentId` property.

Log Level

The Log Level field determines what type of log messages are displayed in the Execution Log.

In general, if a script is still being tested, use the **Debug** log level. This option includes more messages than the other log levels, including messages created by `log.debug(options)`, `log.audit(options)`, `log.error(options)`, and `log.emergency(options)`.

If a script is in production, use one of the following levels:

- **Audit** — This level includes a record of events that have occurred during the processing of the script (for example, "A request was made to an external site"). This level includes log messages created by `log.audit(options)`, `log.error(options)`, and `log.emergency(options)`.
- **Error** — A log level set to Error shows only unexpected script errors, including log messages created by `log.error(options)` and `log.emergency(options)`.
- **Emergency** — Includes only the most critical messages, including log messages created by `log.emergency(options)`.

The default value is Debug.

Priority

If multiple map/reduce and scheduled script deployments are submitted for processing at the same time, some scripts may have to wait to be processed. To handle this type of situation, you can use the **Priority** field. This setting determines how quickly the script deployment instance, and each of its jobs, should be sent for processing relative to other script deployment instances that were created at the same time. The priority for the deployment is applied to each of the deployment instance's jobs.

The choices are as follows:

- **High** — Use to mark critical deployments that require more immediate processing. The scheduler sends high-priority jobs to the processor pool first.
- **Standard** — This is the default setting. It is considered to be a medium priority level. The scheduler sends medium-priority jobs to the processor pool if there are no high-priority jobs waiting.
- **Low** — Use to mark deployments that can tolerate a longer wait time. The scheduler sends low-priority jobs to the processor pool if there are no high- or standard-priority jobs waiting.



Important: You must understand [SuiteCloud Processors](#) before you change this setting. See the help topic [SuiteCloud Processors Priority Levels](#).

Concurrency Limit

The map/reduce script type permits parallel processing. With parallel processing, multiple SuiteCloud Processors can work simultaneously to execute a single script deployment instance. You can control the number of processors used for each script instance by using the Concurrency Limit field on the script deployment record.



Note: This setting affects the map and reduce stages only. These stages are the only ones that permit parallel processing.

For example, if you specify a concurrency limit of 5, the system creates five map jobs and five reduce jobs. If you do not specify a limit, the maximum number of processors available to your account is used. The default value is 2. For more information, see the help topic [SuiteCloud Processors Processor Allotment Per Account](#).

If you use a SuiteCloud Development Framework (SDF) project to modify the concurrency limit in a script deployment record, SDF adjusts the concurrency limit automatically if you specify a value that exceeds the limit available in the target account. For example, if you specify a concurrency limit of 10 and deploy your SDF project to an account with a limit of 5, the Concurrency Limit field on the script deployment record in the account is set to 5. The XML representation of the map/reduce script remains at the original value of 10. For more information, see the help topic [Setting a Concurrency Limit on your Map/Reduce Script Deployment](#).



Note: The Concurrency Limit field was introduced in 2017.2, as part of the SuiteCloud Processors feature. If you are editing a deployment record that was created prior to 2017.2, be aware that when your account was upgraded, the Concurrency Value field was initially set to a value that corresponds to the number of queues that had been saved for the Queues field.

Submit All Stages At Once

Every map/reduce script deployment instance is processed by multiple jobs. At least one job is created for each stage used by the script. Every map/reduce script must use either four or five stages: getInputData, shuffle, summarize, and either map or reduce (or both). However, the jobs for the various stages are not necessarily submitted at the same time. This behavior is controlled by the **Submit All Stages at Once** option.

Map/reduce stages must occur in a specific sequence. When the Submit All Stages at Once option is disabled, the system waits to submit the jobs for each stage until after the prerequisite job completes.

In contrast, when the Submit All Stages at Once option is enabled, the system submits jobs for all stages simultaneously. This behavior increases the likelihood that all jobs associated with the script deployment instance finish, without gaps, before another script begins executing. However, be aware that this option **does not guarantee** that no gaps occur. For example, because a map/reduce job can yield, a long-running job may be forced to end, and a job associated with another script may begin executing in its place. For these reasons, you should not rely on this option if you need to enforce a strict execution sequence among scripts. The only way to enforce a strict sequence is to have one script schedule another during the summarize stage. A script can be scheduled programmatically by using the `task.create(options)` method. For more details, see [Submitting an On-Demand Map/Reduce Script Deployment from a Script](#).

The Submit All Stages at Once option is enabled by default. In general, Oracle recommends that you leave this option enabled.

Yield After Minutes

The Yield After Minutes field helps you prevent any processor from being monopolized by a long-running map or reduce job.

Here's how this setting works: During the map and reduce stages, after each function invocation, the system checks to see how long the map or reduce job has been running. If the amount of elapsed time has surpassed the number of minutes identified in the Yield After Minutes field, the job gracefully ends its execution, and a new job is created to take its place. The new job has the same priority as the old one, but a later timestamp. This behavior is known as yielding.

By default, Yield After Minutes is set to 60, but you can enter any number from 3 to 60.

The system never interrupts a function invocation for this limit. Also, the system never ends a map or reduce job before the limit has been reached, but only after it has been passed. For that reason, the degree to which the limit is surpassed varies depending on the duration of your function invocation. For example, if the Yield After Minutes limit is 3 minutes, but your function takes 15 minutes to complete, then in practice the job yields after 15 minutes, not 3 minutes.

Yielding is also affected by a governance limit. This limit is 10,000 usage units for each map and reduce job. This limit works in the same way as the Yield After Minutes limit: The system waits until after each function invocation ends to determine whether the usage-unit limit has been surpassed. If it has, the job yields, even if the Yield After Minutes limit has not been exceeded.

See also [SuiteScript 2.x Map/Reduce Yielding](#) and [Soft Limits on Long-Running Map and Reduce Jobs](#).

Buffer Size

The purpose of the **Buffer Size** field is to control how frequently a map or reduce job saves data about its progress. In general, Oracle recommends that you leave this field set to 1.

To understand this setting, remember how map and reduce jobs work: First, the job flags some number of key/value pairs that require processing. Then it processes the pairs that it flagged. Then it saves data about the work it did. This data includes information about which key/value pairs were processed, how many usage points were consumed, and so on. The process repeats either until the job yields or until no key/value pairs remain.

The Buffer Size field controls how many pairs are flagged at one time. So if you leave this field set to its default of 1, the job flags one pair, processes it, and saves the data. Then it repeats the cycle.

You can set Buffer Size to any of the following values: 1, 2, 4, 8, 16, 32, or 64. However, the disadvantage of choosing a higher number is that, if the job is interrupted by an application server restart, there is a greater likelihood that one or more key/value pairs will be processed twice. On the other hand, setting this field to a higher number can be more efficient in certain cases.

Use the following guidance:

- In general, leave this value set to 1, particularly if the script is processing records.
- You can choose a higher buffer size value if the script is performing fast, algorithmic operations, or if other special circumstances dictate that you deviate from the default setting.

Schedule Subtab

The following table summarizes the fields on the Schedule subtab. These settings are honored only if the deployment record's **Status** is set to Scheduled and the Deployed box is checked.

| Field | Description |
|--------------|---|
| Single Event | The map/reduce script deployment is submitted only once. Use this option if you want to schedule a future one-time submission. |
| Daily Event | The map/reduce script deployment is submitted every x number of days. If you schedule the submission to recur every x minutes or hours, the schedule starts over on the next scheduled day. |

| Field | Description |
|---------------|---|
| | For example, your deployment is set to submit daily, starting at 3:00 am and recurring every five hours. A scheduled script instance is submitted at 3:00 am, 8:00 am, 1:00 pm, 6:00 pm, and 11:00 pm. At midnight, the schedule starts over and the next submission is at 3:00 am. |
| Weekly Event | The map/reduce script deployment is submitted at least one time per scheduled week. If you schedule the submission to recur every x minutes or hours, the schedule starts over on the next scheduled day. For example, your deployment is set to submit on Tuesday and Wednesday, starting at 3:00 am and recurring every five hours. The deployment is submitted on Tuesday at 3:00 am, 8:00 am, 1:00 pm, 6:00 pm, and 11:00 pm. On Wednesday, the schedule starts over and the next submission is at 3:00 am. |
| Monthly Event | The map/reduce script deployment is submitted at least one time per month. |
| Yearly Event | The scheduled script deployment is submitted at least one time per year. |
| Start Date | The first submission occurs on this date. This field is mandatory if a one-time or recurring schedule is set. |
| Start Time | If a value is selected, the first submission occurs at this time. |
| Repeat | If a value is selected, the first submission occurs on the date and time selected. A new script deployment instance is then created and submitted every x minutes or hours until the end of the start date. If applicable, the schedule starts over on the next scheduled day. For example, your deployment is set to submit on Tuesday and Wednesday, starting at 3:00 am and recurring every five hours. The deployment is submitted on Tuesday at 3:00 am, 8:00 am, 1:00 pm, 6:00 pm, and 11:00 pm. On Wednesday, the schedule starts over and the next submission is at 3:00 am. |
| End By | If a value is entered, the last submission ends by this date. If you schedule the submission to recur every x minutes or hours, a new script deployment instance is created and submitted every x minutes or hours until the end date. |
| No End Date | The schedule does not have a set end date. |

Scheduling a Map/Reduce Script Submission



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

Map/reduce script deployments can be submitted for processing on a scheduled basis. For example, you could configure the following schedules:

- A one-time submission, at a predefined time.
- Repeated submission, on a daily, weekly, monthly, or yearly basis.

To set a scheduled submission, the Status field on the deployment record must be set to **Scheduled**. Additionally, you must configure one or more upcoming times on the record's Schedule subtab.

Deployment times can be scheduled with a frequency of every 15 minutes. For example, you could configure a script to run at 2:00 pm, 2:15 pm, 2:30 pm, and so on.

The times you set on the Schedule subtab are the times the script deployment is submitted for processing. However, **the times you set on the Schedule subtab are not necessarily the times the script will execute**. Script deployment does not mean the script will actually execute precisely at 2:00 pm, 2:15 pm, 2:30 pm, and so on. There may be a short system delay before the script is actually executed, even if no scripts are before it. If there are scripts already waiting to be executed, the script may wait to be executed until other scripts have completed. For more details about how map/reduce scripts are processed, see the help topic [SuiteCloud Processors](#).



Note: The times displayed on the Schedule subtab use the session time zone, meaning that the times are displayed using the time zone of the user who is currently logged in. This time zone can be different from the time zone that is set for the company or the time zone of the map/reduce script owner.

To schedule a one-time or recurring map/reduce script submission:

1. Create your map/reduce script entry point script. This process includes uploading a JavaScript file and creating a script record based on that file. To review a map/reduce script example, see [SuiteScript 2.x Map/Reduce Script Type](#). If this script is your first, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).
2. Open the appropriate script record in view mode. Click the **Deploy Script** button.
3. When the script deployment record loads, check the **Deployed** box, if it is not already checked.
4. Set the **Status** field to **Scheduled**.
5. Set the remaining body fields. For help understanding the fields, see [Map/Reduce Script Deployment Record](#).
6. On the **Schedule Subtab**, set the deployment options.

For example, if you wanted to **submit the script hourly**, you would configure the subtab as follows:

- Deployed = checked
- Daily Event = [radio button enabled]
- Repeat every 1 day
- Start Date = [today's date]
- **Start Time = 12:00 am**
- Repeat = every hour
- End By = [blank]
- No End Date = checked
- Status = Scheduled
- Log Level = Error

If the **Start Time** is set to any other time than 12:00 am (for example, it is set to 2:00 pm), the script will start at 2:00 pm, but then finish its hourly execution at 12:00 am. It will not resume until the next day at 2:00 pm.

7. Click **Save**.



Note: In some cases, you may want to submit a map/reduce script for processing multiple times simultaneously, or within a short time frame. However, the system does not permit a script deployment to be submitted if a previous instance of the deployment has already been submitted and is not yet finished. The solution in this case is to create multiple deployments for the script. In other words, repeat Steps 2 through 7 of the procedure above as needed.

Submitting an On-Demand Map/Reduce Script Deployment from the UI



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

Map/reduce scripts can be submitted for processing on an on-demand basis from the NetSuite UI. To submit a script in this way, use the **Save and Execute** command on the Script Deployment page. The **Status** field on the deployment must be set to **Not Scheduled**.

You can also submit an on-demand deployment programmatically. For details, see [Submitting an On-Demand Map/Reduce Script Deployment from a Script](#).

Note that after you submit an on-demand deployment of a script, the script **does not necessarily execute right away**. After a script is submitted for processing, there may be a short system delay before the script is actually executed, even if no scripts are before it. If there are scripts already waiting to be executed, the script may wait to be executed until other scripts have completed. For more details about how map/reduce scripts are processed, see the help topic [SuiteCloud Processors](#).

To submit an on-demand map/reduce script for processing from the UI:

1. Create your map/reduce script entry point script. This process includes uploading a JavaScript file and creating a script record based on that file. To review a map/reduce script example, see [SuiteScript 2.x Map/Reduce Script Type](#). If this script is your first, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).
2. Open the appropriate script record in view mode. Click the **Deploy Script** button.
3. When the script deployment record loads, check the **Deployed** box, if it is not already checked.
4. Set the Status field to **Not Scheduled**.
5. Set the remaining body fields. For help understanding the fields, see [Map/Reduce Script Deployment Record](#).
6. Select **Save and Execute** from the **Save** dropdown list.



Note: In some cases, you may want to submit a map/reduce script for processing multiple times simultaneously, or within a short time frame. However, the system does not permit a script deployment to be submitted if a previous instance of the deployment has already been submitted and is not yet finished. The solution in this case is to create multiple deployments for the script. In other words, repeat Steps 2 through 6 of the procedure above as needed.

Submitting an On-Demand Map/Reduce Script Deployment from a Script



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

Map/reduce scripts can be submitted for processing on an on-demand basis from another server script type. You can submit a deployment this way by using the [task.ScheduledScriptTask](#) API. For the call to be successful, the **Status** field on the script deployment record must be set to **Not Scheduled**.

You can also submit an on-demand deployment from the UI. For details, see [Submitting an On-Demand Map/Reduce Script Deployment from the UI](#).

Note that after you submit an on-demand deployment of a script, the script **does not necessarily execute right away**. After a script is submitted for processing, there may be a short system delay before the script is actually executed, even if no scripts are before it. If there are scripts already waiting to be executed, the script may wait to be executed until other scripts have completed. For more details about how map/reduce scripts are processed, see the help topic [SuiteCloud Processors](#).

To submit an on-demand map/reduce script instance from a script:

1. Create your map/reduce script entry point script. This process includes uploading a JavaScript file and creating a script record based on that file. To review a map/reduce script example, see [SuiteScript 2.x Map/Reduce Script Type](#). If this script is your first, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).

2. Open the appropriate script record in view mode. Click the **Deploy Script** button.
3. When the script deployment record loads, check the **Deployed** box, if it is not already checked.
4. Set the Status field to **Not Scheduled**.
5. Set the remaining body fields. For help understanding the fields, see [Map/Reduce Script Deployment Record](#).
6. Click **Save**.
7. In the server script where you want to submit the map/reduce script, call `task.create(options)` to return a `task.MapReduceScriptTask` object:

```
1 | var scriptTask = task.create({taskType: task.TaskType.MAP_REDUCE});
```

8. Set the `MapReduceScriptTask.scriptId` and `MapReduceScriptTask.deploymentId` properties:

```
1 | scriptTask.scriptId = 'customscript1';
2 | scriptTask.deploymentId = 'customdeploy1';
```

Note that the system does not require you to set a value for the `deploymentId` property. If you omit this value, the system searches for and uses any available deployment record for the script record you identified with the `scriptId` property. For a deployment to be considered available, three conditions must be met: The deployment record must have a status of Not Scheduled, its Deployed option must be set to true, and no unfinished instances of the deployment can exist.

9. Call `MapReduceScriptTask.submit()` to submit the script for processing. For example:

```
1 | scriptTask.submit();
```



Note: In some cases, you may want to submit a map/reduce script for processing multiple times simultaneously, or within a short time frame. However, the system does not permit a script deployment to be submitted if a previous instance of the deployment has already been submitted and is not yet finished. The solution in this case is to create multiple deployments for the script. In other words, repeat Steps 2 through 9 of the procedure above as needed. If you are using this approach, consider omitting the `deploymentId` property when you create the map/reduce script task object. When you omit the `deploymentId`, the system searches for and uses whichever deployment record is available.

Submitting Multiple Deployments of the Same Script



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

The system does not permit you to submit a single script deployment for processing multiple times simultaneously. That is, if one instance of the deployment has been submitted and not yet finished, you cannot submit the same deployment again. You must wait for the unfinished instance to complete.

If you need to submit multiple instances of a single map/reduce script for processing at the same time, or within a very short time frame, the correct approach is to create multiple deployments of the script and submit each deployment as needed.

There are multiple advantages to this technique. For example, this approach lets you process two or more instances of the script in parallel. Additionally, if you need a map/reduce script to be processed multiple times in the same general time frame, you can submit both instances at the same time without having to worry about the second submission failing with an error. The alternative would be to monitor the progress of the first script deployment instance and then be careful to submit the second deployment only after completion of the first.

Note that if a deployment has already been submitted but not yet finished, and you open the deployment record for editing in the UI, the deployment record's Save and Execute option is not available. If you try to submit the deployment programmatically in this case, the system throws the MAP_REDUCE_ALREADY_RUNNING error.

For help creating or submitting a deployment, see the following topics:

- [Scheduling a Map/Reduce Script Submission](#)
- [Submitting an On-Demand Map/Reduce Script Deployment from the UI](#)
- [Submitting an On-Demand Map/Reduce Script Deployment from a Script](#)

SuiteScript 2.x Map/Reduce Governance

As with all script types, NetSuite imposes usage limits on map/reduce scripts. Governance rules for map/reduce scripts fall into two main categories:

- Certain limits, if violated, cause an interruption to the current function invocation. These limits are known as **hard limits**.
- Other limits are managed automatically by the system. These limits never interrupt a function invocation. Rather, after the completion of a function invocation, these limits can cause a job to yield and its work to be rescheduled for later. These limits are known as **soft limits**.

See the following sections for more details:

- [Hard Limits on Total Persisted Data](#)
- [Hard Limits on Function Invocations](#)
- [Soft Limits on Long-Running Map and Reduce Jobs](#)

Be aware that the system does not impose any limit on the full duration of a map/reduce script deployment instance. It also is not possible for a user to impose such a limit. Rather, the system's limits regulate isolated components of the deployment, such as the duration of a single function invocation.



Note: One way that NetSuite measures a script's activity is through usage units. For more information about usage units, see the help topic [SuiteScript Governance and Limits](#).

Hard Limits on Total Persisted Data

The total persisted data used by a map/reduce script cannot exceed 200MB at any one time. If your script exceeds this limit at any point during its processing, the system throws a STORAGE_SIZE_EXCEEDED error. Additionally, the script ends its current function invocation, exits the current stage, and goes immediately to the summarize stage. (This error does not occur in the summarize stage, because the total persisted data cannot be increased during that stage.)

Note the following about how persisted data is measured:

- The system takes into account any search results retrieved and returned by the input function. Note that a large number of columns in a result set can significantly increase the amount of data used.
- During the map and reduce stages, the total size is a measure of the keys and values yet to be processed. After a key or value is processed, it does not count toward the limit.

Hard Limits on Function Invocations

The following table describes the limits applied to a map/reduce script's function invocations. If your script exceeds any of these limits, the system throws an SSS_USAGE_LIMIT_EXCEEDED error. The way the system

responds to this error varies depending on the stage and the configuration of your script, as shown in the following table.

| Stage | Usage limit per function invocation | Response to SSS_USAGE_LIMIT_EXCEEDED error |
|----------------|-------------------------------------|--|
| Get Input Data | 10,000 units | The script ends the function invocation and exits the stage. It proceeds directly to the summarize stage. |
| Map | 1,000 units | The response includes two parts. Note that you can configure the second part: |
| Reduce | 5,000 units | <ol style="list-style-type: none"> 1. The function invocation ends, even if its work on the current key/value pair is incomplete. 2. The default behavior is that the job continues working and moves on to other key/value pairs. However, you can configure the script to invoke the function again for the same key/value pair. For details, see Configuration Options for Handling Map/Reduce Interruptions. |
| Summarize | 10,000 units | The script stops executing. |

Note that if you are using the map/reduce script type as intended, your script should not approach these limits, particularly for map and reduce function invocations. In general, each invocation of a map or reduce function should do a relatively small portion of work. For more details, see [SuiteScript 2.x Map/Reduce Script Best Practices](#).

Soft Limits on Long-Running Map and Reduce Jobs

In addition to the limits described in [Hard Limits on Function Invocations](#), the system includes a soft limit of 10,000 usage units on each map and reduce job.

To understand how this limit works, first be aware that all map/reduce scripts are processed by [SuiteCloud Processors](#). A processor is a virtual unit of processing power that executes a job.

The 10,000-unit soft limit is a mechanism designed to prevent any processor from being monopolized by a long-running map or reduce job. During the map and reduce stages, after each function invocation, the system checks the total number of units that have been used by the job. If the total usage has surpassed 10,000 units, the job gracefully ends its execution and a new job is created to take its place. The new job has the same priority as the old one, but a later timestamp. This behavior is known as yielding.

Yielding is also affected by the script deployment record's Yield After Minutes field. This time limit works in the same way as the 10,000-unit limit: The system waits until after each function invocation ends to determine whether the time limit has been exceeded. If it has, the job yields, even if the 10,000-unit limit has not been exceeded. By default, Yield After Minutes is set to 60, but you can enter any number from 3 to 60. For more details, see [Yield After Minutes](#). See also [SuiteScript 2.x Map/Reduce Yielding](#).



Important: Yielding is unrelated to the limits that exist for a single invocation of a map function and a single invocation of a reduce function. Those limits are described in [Hard Limits on Function Invocations](#). Exceeding the limits for a single invocation of a map or reduce function causes the system to throw an SSS_USAGE_LIMIT_EXCEEDED error and ends the function invocation, even if it is not complete.

SuiteScript 2.x Map/Reduce Script Status Page

- [Viewing Map/Reduce Script Status](#)
- [Viewing Details of Script Instances](#)

- [Programmatically Retrieving Script Instance Details](#)

Viewing Map/Reduce Script Status

You can monitor map/reduce script execution via the map/reduce script status page in the UI. With the script status page, you can see whether a map/reduce script deployment is pending, in progress, or unable to complete.

If all tasks are pending, you can cancel a script deployment from this page.

From NetSuite, go to Customization > Scripting > Map/Reduce Script Status.

To help you understand and optimize the performance of script entry points used, you can drill down for more details about map stages, processing utilization, and timing. You can use this information to gain insight about the time required to complete a stage or process a task.

Additionally, from the script status page, you can view the deployment record and consider changing the concurrency limit.

Viewing Details of Script Instances

For scheduled scripts and map/reduce scripts, a script instance is a scheduled script task or map/reduce script task that is submitted for processing. This script instance can also be called a task.

To view the details of a map/reduce script instance, from the Map/Reduce Script Status page, click **View Details**. Each row contains information about an individual map/reduce job that belongs to the script instance. A job always belongs to one of the stages of a map/reduce script, and there is at least one job per stage. There is exactly one job for each of the getInputData, shuffle, and summarize stages. The number of jobs for the map and reduce stages depends on concurrency limits and the number of yields.

For more information, see [SuiteScript 2.x Map/Reduce Script Stages](#) and [SuiteScript 2.x Map/Reduce Yielding](#).

Programmatically Retrieving Script Instance Details

To get the script status via the [N/search Module](#), create and load a search using `scheduledscriptinstance` as the type argument.

To get the status via the [N/task Module](#), see the help topic `task.MapReduceScriptTask`.

SuiteScript 2.x Map/Reduce Script Testing and Troubleshooting

You have several options to test and debug your map/reduce script:

- Make a separate unit test suite as you develop and modify your map/reduce script. See [Map/Reduce Script Unit Testing](#).
- Check the execution logs. See the help topic [N/log Module](#).
- Monitor the script status. See [SuiteScript 2.x Map/Reduce Script Status Page](#) and [Programmatically Retrieving Script Instance Details](#).
- Detect any server restarts that interrupted map/reduce script execution. See `inputContext.isRestarted`, `mapContext.isRestarted`, `reduceContext.isRestarted`, and `summaryContext.isRestarted`.

You cannot use the SuiteScript Debugger for deployed debugging of a map/reduce script type. However, you may wish to test any dependencies on other types of scripts. Remember that to test existing scripts, the Script Deployment Status must be set to Testing and the currently logged in user must be listed as

the script record owner. For information about the SuiteScript Debugger, see the help topic [SuiteScript Debugger](#).

Map/Reduce Script Unit Testing

 **Note:** You cannot use the SuiteScript Debugger for deployed debugging of a map/reduce script type.

To test a map/reduce script on demand, NetSuite recommends splitting the script into entry point level sections. Use the sections to form unit tests. Each section should function as an entry point script that can be executed without external dependencies. If passing in modules, make sure that you use the require function and absolute paths.

To test map or reduce stages, you will need to create a mock context that seeds values and provides the dependent objects and parameters. Then, check the states, behavior, inputs, and outputs of map/reduce functions using assertion statements. Note that the getInputData stage does not take parameters, so it will not require mock context and can be tested more conventionally.

To test a summarize stage, if it contains logic operating on a final set of data that has no return, use assertions and logs to gather information.

To assist development of your unit test, download the SuiteScript 2.x API files, specifically those representing the `mapContext`, `reduceContext`, and `summaryContext` objects. These file can act as a schema for the properties and methods you want to test or mock. To access the files, do the following:

1. From NetSuite, select Documents > Files > File Cabinet.
2. Select SuiteScript 2.0 API to download the zip folder.
3. Extract the `mapReduceContext` and `mapReduceSummary.js` files.

For an example of unit testing for SuiteScript 2.x scripts, see the help topic [SuiteScript 2.x Unit Testing Project Sample](#).

SuiteScript 2.x Map/Reduce Script Error Handling

A map/reduce script can be interrupted by either of the following:

- An application-server disruption, which can occur because of a NetSuite update, NetSuite maintenance, or an unexpected failure of the execution environment
- An uncaught error

Interruptions can leave portions of a script's work incomplete. However, the system includes measures for dealing with this problem.

After an **application-server disruption**, the system automatically restarts the script, regardless of which stage was in progress when the failure occurred. For example, if the failure occurred during the map or reduce stage, the map or reduce job restarts. In this case, the default behavior is that the restarted job invokes the map or reduce function again for all key/value pairs that were previously flagged for processing but not yet marked complete. This behavior can be modified by using the `retryCount` option.

After an **uncaught error** is thrown, the behavior varies depending on the stage. If the error is thrown during the `getInputData` stage, the script goes immediately to the summarize stage, and the `getInputData` stage is not restarted. If an uncaught error is thrown during the map or reduce stage, the default behavior is that the current function invocation ends, even if incomplete, and the map or reduce job moves on to other key/value pairs that require processing. However, the script can be configured so that the map or reduce function is invoked again for the pair that was being processed when the error occurred. You manage the system's behavior in this case by using the `retryCount` and `exitOnError` options.

To fully understand the system response to interruptions, see [System Response After a Map/Reduce Interruption](#).

In general, your script should include logic that checks to see whether a restart has occurred. If the function has been restarted, the script should take any actions needed to avoid unwanted duplicate processing. For more information, see [Adding Logic to Handle Map/Reduce Restarts](#).

For additional explanation of how each stage responds to a restart, see [Execution of Restarted Map/Reduce Stages](#).

System Response After a Map/Reduce Interruption



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

A map/reduce script can be interrupted at any time. For example, a disruption to the application server immediately stops the script's execution. Additionally, an uncaught error, although it does not cause the script to stop executing, stops the current function invocation, even if it is not complete.

For more details, review the following sections:

- [System Response After an Application-Server Disruption](#)
- [System Response After an Uncaught Error](#)



Important: Regardless of how your script is configured, you should make sure that it includes logic that checks to see whether a restart has occurred. If the function has been restarted, the script should take any actions needed to avoid unwanted duplicate processing. For details, see [Adding Logic to Handle Map/Reduce Restarts](#).

System Response After an Application-Server Disruption

An application disruption can occur because of a NetSuite update, NetSuite maintenance, or an unexpected failure of the execution environment. When the application server is disrupted in this way, the script stops executing. After the application server restarts, the script also restarts, resuming the same stage that it was in process when the script was interrupted.

When an application server restart interrupts the map or reduce stage, the system writes the SSS_APP_SERVER_RESTART error code to the relevant iterators. This error code is shown alongside the codes recorded for any uncaught errors that were thrown.

For more details, see the following table.

| Stage where interruption occurred | Script behavior | SSS_APP_SERVER_RESTART error code written to |
|-----------------------------------|--|--|
| Get Input Data | When the disruption occurs, the script stops executing. After the application server restarts, the system restarts the function. | n/a |
| Map | When the disruption occurs, the script stops executing. Any data that was saved during the previous invocation by using the context.write() method is discarded. Afterward, the response is as follows: | <ul style="list-style-type: none"> ■ <code>mapContext.errors</code> — Contains the error codes recorded during previous attempts to process the current key/value pair. |

| Stage where interruption occurred | Script behavior | SSS_APP_SERVER_RESTART error code written to |
|-----------------------------------|---|---|
| Reduce | <ol style="list-style-type: none"> 1. The system evaluates the <code>retryCount</code> config setting. If <code>retryCount</code> is set to a value greater than 0 or if the <code>retryCount</code> setting is not used, the script tries to process the same set of key/value pairs it was processing when the application server became unavailable. This data includes all pairs that were flagged for processing but not marked complete. However, if <code>retryCount</code> is set to 0, the script moves on to Step 2 without attempting further processing for these key/value pairs. 2. The job moves on to other key/value pairs that require processing and were not previously flagged as in progress. | <ul style="list-style-type: none"> ■ <code>mapSummary.errors</code> — Contains all error codes recorded during the map stage. ■ <code>reduceContext.errors</code> — Contains the error codes recorded during previous attempts to process the current key/value pair. ■ <code>reduceSummary.errors</code> — Contains all error codes recorded during the reduce stage. |
| Summarize | <p>When the disruption occurs, the entire script stops executing.</p> <p>After the application server restarts, the system restarts the function.</p> | n/a |

System Response After an Uncaught Error

An error that is not caught in a try-catch block does not necessarily end the execution of a map/reduce script, but the error can disrupt the script's work. Some aspects of this behavior are configurable. For details, see the following table.

| Stage where error occurred | Script behavior | Errors written to |
|----------------------------|--|---|
| Get Input Data | The script ends the function invocation and exits the stage. It proceeds directly to the summarize stage. This behavior cannot be configured. | <code>inputSummary.error</code> |
| Map | When the error occurs, the function invocation ends, even if its work is not complete. Any data that was saved during the invocation by using the <code>context.write()</code> method is discarded. Afterward, the system responds as follows: | <ul style="list-style-type: none"> ■ <code>mapContext.errors</code> — Contains the error codes recorded during previous attempts to process the current key/value pair. ■ <code>mapSummary.errors</code> — Contains all error codes recorded during the map stage. |
| Reduce | <ol style="list-style-type: none"> 1. The system evaluates the <code>retryCount</code> setting. If <code>retryCount</code> is set to a value greater than 0, and the maximum number of retries has not yet been used, the script tries to process the same key/value pair again. The rest of the steps in this process are not used. 2. The system evaluates the <code>exitOnError</code> setting. If <code>exitOnError</code> is set to true, the script exits the current stage and proceeds directly to the summarize stage. The rest of the steps in this process are not used. 3. The job continues by moving on to other key/value pairs that require processing. | <ul style="list-style-type: none"> ■ <code>reduceContext.errors</code> — Contains the error codes recorded during previous attempts to process the current key/value pair. ■ <code>reduceSummary.errors</code> — Contains all error codes recorded during the reduce stage. |

| Stage where error occurred | Script behavior | Errors written to |
|----------------------------|--|-------------------|
| | It does not do any further work on the pair it was processing when the error occurred. | |
| Summarize | The script stops executing. This behavior cannot be configured. | n/a |

i Note: This table describes the behavior for the majority of errors, but a few errors result in different behavior. For example, the SSS_USAGE_LIMIT_EXCEEDED error causes a script to exit the map or reduce stage immediately. For details, see [Hard Limits on Total Persisted Data](#).

Configuration Options for Handling Map/Reduce Interruptions

i Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

A map/reduce script can be interrupted at any time. For example, a disruption to the application server immediately stops the script's execution. Additionally, an uncaught error, although it does not cause the script to stop executing, stops the current function invocation, even if it is not complete.

The system includes two configuration options that let you fine-tune the overall response of your script to interruptions. For details, see the following sections:

- [retryCount](#)
- [exitOnError](#)
- [Adding a Configuration Option to a Script](#)

retryCount

The retryCount option affects the map and reduce stages only. This option lets you configure your script to restart the map or reduce function for any key/pairs that were left in an uncertain state following an interruption, including application server restarts and uncaught errors.

The effect of this setting varies slightly depending on whether the script was disrupted by application server restart or an uncaught error. For example:

- When an **application server restart** occurs, the script cannot identify the exact key/value pair that was being processed when the interruption occurred. However, the script can identify the pairs that had been flagged for processing but were not yet marked as complete. In the event of a retry, the script retries processing for all of these pairs.
- When an **uncaught error** occurs, the script can identify the exact key/value pair that was being processed when the interruption occurred. When the retryCount option is being used, the script invokes the map or reduce function again for that specific key/value pair.

Valid values are 0 to 3. The number you choose for the retryCount setting is the number of retries permitted for **each** key/value pair that was left in an uncertain state after an interruption. For example, suppose you have retryCount set to 2, and an error interrupts a map job. In this case, the script would restart the function for the key/value pair that was being processed when the error was thrown. If the second function invocation for that pair was also interrupted by an error, the script would invoke the function for that key/value pair another time. However, if an error was thrown during the third attempt, the script would not retry processing again, because the retryCount setting permits only two retries. Later, when the job starts processing a different key/value pair, two retries are again available.

The retryCount setting is optional. You can set a value for it in the return statement of your map/reduce script, as described in [Adding a Configuration Option to a Script](#).

If you do not add the `retryCount` option to your script, the behavior varies depending on whether the script was disrupted by an uncaught error or by an application server restart. For example:

- When `retryCount` is not used and an application server restart interrupts the script, the system **restarts** processing for all key/value pairs that were left in an uncertain state.
- When `retryCount` is not used and an error interrupts the map or reduce function, the system **does not restart** processing for the key/value pair that was left in an uncertain state.

See also [System Response After an Uncaught Error](#) and [System Response After an Application-Server Disruption](#).

 **Note:** In the case of an uncaught error, the system's overall response is also affected by the value of the `exitOnError` option. However, the script evaluates and reacts to the `retryCount` setting first.

exitOnError

The `exitOnError` option is used when an uncaught error occurs in the map or reduce stage. It is evaluated after the `retryCount` option.

When `exitOnError` is set to true, the script exits the stage after both of the following occur:

- An error is thrown and not caught.
- All retries permitted by the `retryCount` option have been exhausted.

This setting has no impact on the system's behavior after an application server restart. It is used only when an uncaught error is thrown.

The `exitOnError` setting is optional. You can set a value for it by adding it in the return statement of your map/reduce script, as described in [Adding a Configuration Option to a Script](#). Possible values are:

- **false** — The script continues processing in the current stage. (This is also the behavior used when the option is not added to the script.)
- **true** — The script exits the stage and goes immediately to the summarize stage.

See also [System Response After an Uncaught Error](#).

Adding a Configuration Option to a Script

If you want to use the `retryCount` or `exitOnError` option, add them to the script's return statement in a config block. For example:

```

1 // Add additional code.
2 ...
3
4     return {
5
6         config:{
7             retryCount: 3,
8             exitOnError: true
9         },
10
11        getInputData: myGetInputData,
12        map: myMap,
13        reduce: myReduce,
14        summarize: mySummarize
15
16    };
17
18 ...
19 // Add additional code.

```

Logging Errors



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

With any map/reduce script, you should include logic that checks for errors that may have occurred during the getInputData, map, and reduce stages. You can access data about errors using the context objects made available to each entry point. Use the properties shown in the following table.

| Stage where error occurred | Property that contains data |
|----------------------------|---|
| Get Input Data | <code>inputSummary.error</code> |
| Map | <ul style="list-style-type: none"> ■ <code>mapContext.errors</code> — Contains the error codes recorded during previous attempts to process the current key/value pair. ■ <code>mapSummary.errors</code> — Contains all error codes recorded during the map stage. |
| Reduce | <ul style="list-style-type: none"> ■ <code>reduceContext.errors</code> — Contains the error codes recorded during previous attempts to process the current key/value pair. ■ <code>reduceSummary.errors</code> — Contains all error codes recorded during the reduce stage. |

Syntax

The following snippets shows how you can capture data about errors in various stages.

Map Stage

You can use this snippet in a map function. The snippet logs data only if an error was thrown during a previous invocation of the map function for the same key/value pair currently being processed.

```

1 // Create a log entry showing each full serialized error thrown
2 // during previous attempts to process the current key/value pair.
3
4 mapContext.errors.iterator().each(function (key, error, executionNo){
5
6     log.error({
7         title: 'Map error for key: ' + key + ', execution no ' + executionNo,
8         details: error
9     });
10
11     return true;
12 });

```

Reduce Stage

You can use this snippet in a reduce function. The snippet logs data only if an error was thrown during a previous invocation of the reduce function for the same key/value pair currently being processed.

```

1 // Create a log entry showing each full serialized error thrown
2 // during previous attempts to process the current key/value pair.
3
4 reduceContext.errors.iterator().each(function (key, error, executionNo){
5     log.error({
6         title: 'Reduce error for key: ' + key + ', execution no ' + executionNo,
7         details: error
8     });
9     return true;
10 });

```

Summarize Stage

You can use these snippets in a summarize function. They log data about errors thrown during previous stages.

```

1 // If an error was thrown during the input stage, log the error.
2
3 if (summary.inputSummary.error)
4 {
5     log.error({
6         title: 'Input Error',
7         details: summary.inputSummary.error
8     });
9
10
11 // For each error thrown during the map stage, log the error, the corresponding key,
12 // and the execution number. The execution number indicates whether the error was
13 // thrown during the the first attempt to process the key, or during a
14 // subsequent attempt.
15
16 summary.mapSummary.errors.iterator().each(function (key, error, executionNo){
17     log.error({
18         title: 'Map error for key: ' + key + ', execution no. ' + executionNo,
19         details: error
20     });
21     return true;
22 });
23
24
25 // For each error thrown during the reduce stage, log the error, the corresponding
26 // key, and the execution number. The execution number indicates whether the error was
27 // thrown during the the first attempt to process the key, or during a
28 // subsequent attempt.
29
30 summary.reduceSummary.errors.iterator().each(function (key, error, executionNo){
31     log.error({
32         title: 'Reduce error for key: ' + key + ', execution no. ' + executionNo,
33         details: error
34     });
35     return true;
36 });

```

Execution of Restarted Map/Reduce Stages



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

A map/reduce script can involve many jobs. The input, shuffle, and summarize stages are each processed with a single job. However, multiple jobs can participate in the map and reduce stages. Within a map stage or a reduce stage, jobs can run in parallel. Any of the jobs can be forcefully terminated at any moment. The impact of this event depends on the status of the job (what it was doing), and in which stage it was running.

For details, see the following topics:

- [Termination of getInput Stage](#)
- [Termination of Shuffle Stage](#)
- [Termination of Parallel Stages](#)
- [Termination of Summarize Stage](#)

Termination of getInput Stage

The work of a serial stage (getInput, shuffle, and summarize stages) is done in a single job. If the getInput stage job is forcefully terminated, it is later restarted. The getInput portion of the script can find out

whether it is the restarted execution by examining the `isRestarted` attribute of the context argument (`inputContext.isRestarted`). The script is being restarted if and only if (`context.isRestarted === true`).

Note that the input for the next stage is computed from the return value of the `getInput` script. Next stage input is written after the `getInput` stage finishes. Therefore, even the restarted `getInput` script is expected to return the same data. The map/reduce framework helps to ensure that no data is written twice.

However, if the `getInput` script is changing some additional data (for example, creating NetSuite records), it should contain code to handle duplicated processing. The script needs idempotent operations to ensure that these records are not created twice, if this is undesired.

Termination of Shuffle Stage

The shuffle stage does not contain any custom code, so if the shuffle stage job is forcefully terminated, it is later restarted and all the work is completely redone. There is no impact other than that the stage takes longer to finish.

Termination of Parallel Stages

Map and reduce stages can execute jobs in parallel, so they are considered parallel stages. An application restart will affect parallel stages in the same way. The following example covers impact of restart during the map stage. Note that termination of a reduce stage will behave very similarly.

The purpose of a map stage is to execute a map function on each key/value pair supplied by the previous stage (`getInput`). Multiple jobs participate in the map stage. Map jobs will claim key/value pairs (or a specific number of key/value pairs) for which the map function was not executed yet. The job sets a flag for these key/value pairs so that no other job can execute the map function on them. Then, the job sequentially executes the map function on the key/value pairs it flagged. The map stage is finished when the map function is executed on all key/value pairs.

The number of jobs that can participate on the map stage is unlimited. Only the maximum concurrency is limited. Initially, the number of map jobs is equal to the selected concurrency in the corresponding map/reduce script deployment. However, to prevent a single map/reduce task from monopolizing all computational resources in the account, each map job can yield itself to allow other jobs to execute. The yield creates an additional map job and the number of yields is unlimited.



Note: This is a different type of yield compared to yield in a SuiteScript 1.0 scheduled script. In SuiteScript 1.0, the yield happens in the middle of a script execution. In a map job, the yield can happen only between two map function executions, and not in the middle of one.

If a map job is forcefully terminated, it is later restarted. First, the job executes the map function on all key/value pairs that it took and did not mark finished before termination. It is the only map job that can execute the map function on those pairs. They cannot be taken by other map job. After those key/value pairs are processed, the map job continues normally (takes other unfinished key/value pairs and executes the map function on them).

In some cases, the map function can be re-executed on multiple key/value pairs. The number of pairs that a map function can re-execute will depend on the buffer size selected on the deployment page. The buffer size determines the number of key/value pairs originally taken in a batch. The job marks the batch as finished only when the map function is executed on all of them. Therefore, if the map job is forcefully terminated in the middle of the batch, the entire batch will be processed from the beginning when the map job is restarted.

Note that the map/reduce framework deletes all key/value pairs written from a partially-executed batch, so that they are not written out twice. Therefore, the map function does not need to check whether or not `mapContext.write(options)` for a particular key/value has already been executed. However, if the

map function is changing some additional data, it must also be designed to use idempotent operations. For example, if a map function created NetSuite records, the script should perform additional checks to ensure that these records are not created twice, if this is undesired.

To check if a map function execution is a part of a restarted batch, the script must examine the `isRestarted` attribute in the context argument (`mapContext.isRestarted`). The map function is in the restarted batch if and only if (`context.isRestarted === true`).

Be aware that a restarted value of true is only an indication that some part of the script might have already been executed. Even if `context.isRestarted === true`, a map function could run on a particular key/value for the first time. For example, the map job was forcefully terminated after the map job took the key/value pair for processing, but before it executed the map function on it. This is more likely to occur if a high buffer value is set on the map/reduce deployment. For more information about the buffer, see [Minimizing the Risk of Data Duplication](#).

Termination of Summarize Stage

If the summarize stage job is forcefully terminated, it is later restarted. The summarize portion of the script can find out whether it is the restarted execution by examining the `isRestarted` attribute of the `summary` argument (`summaryContext.isRestarted`).

The script is being restarted if and only if (`summary.isRestarted === true`).

Adding Logic to Handle Map/Reduce Restarts



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

Occasionally, a script failure may occur due to an application server restart. This could be due to a NetSuite update, NetSuite maintenance, or an unexpected failure of the execution environment.

Restarts can terminate an application forcefully at any moment. Therefore, robust scripts must account for restarts and be able to recover from an unexpected interruption.

In a map/reduce script, each restarted piece of the script will automatically delete any internal map/reduce data that this piece created (for example, the key/value pairs that drive the execution of a entire mapping task). However, you must develop your own code to handle any parts of the script that modify additional data (for example, creation of NetSuite records like sales orders), which is *never* automatically deleted.

See the following topics to learn more about how restarts and map/reduce script execution:

- [Design of a Robust Map/Reduce Script Example](#)
- [A Problematic Map/Reduce Script Example](#)
- [A Robust Map/Reduce Script Example](#)
- [Execution of Restarted Map/Reduce Stages](#)



Note: A map or reduce function can also be restarted if interrupted by an uncaught error. For the script to restart in this situation, you must use the `retryCount` option. For additional details, see [System Response After an Uncaught Error](#).

Design of a Robust Map/Reduce Script Example

The following script is designed to detect restarts at particular stages in processing, and to hold logic to run in the event of a restart.

Consider this example as a basic template, where the comment `// I might do something differently` denotes implementation of a special function for each stage, to ensure that the script can repeat itself with the same result. Or, to run a recovery task, such as removing duplicate records.

The script includes a check on the `isRestarted` property for each entry point function. If the value of `isRestarted` is true, the example script shows a placeholder for invoking a function. This is a meant as a placeholder where implementation of logic for protection against restarts and data errors could be inserted.

For more information about an interrupted map/reduce stage, see the help topic [Execution of Restarted Map/Reduce Stages](#).

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType MapReduceScript
4   */
5  define([], function(){
6      return {
7          getInputData: function (context)
8          {
9              if (context.isRestarted)
10              {
11                  // I might do something differently
12              }
13              .
14              .
15              .
16              return inputForNextStage;
17          },
18          map: function (context)
19          {
20              if (context.isRestarted)
21              {
22                  // I might do something differently
23              }
24              .
25              .
26              .
27          },
28          reduce: function (context)
29          {
30              if (context.isRestarted)
31              {
32                  // I might do something differently
33              }
34              .
35              .
36              .
37          },
38          summarize: function (summary)
39          {
40              if (summary.isRestarted)
41              {
42                  // I might do something differently
43              }
44              .
45              .
46              .
47          }
48      });
49 });

```

A Problematic Map/Reduce Script Example

The purpose of this script is to perform a search and process the results. However, it is not adequately prepared for an unexpected restart. The script still needs logic to help prevent an unrecoverable state and prevent creation of erroneous or duplicate data during re-execution.

In Example 4, if the script is forcefully interrupted during the map stage, some sales orders might be updated twice when the map function is re-executed. See [A Robust Map/Reduce Script Example](#) for an improved example.

Note that the other stages in this script do not require improvement for handling a restart. If the get input stage is re-executed, the map/reduce framework ensures that each result of the search is passed to the map stage only one time. In this script, the getInput stage does not change any additional data, so no special restart logic is needed to ensure correct updates of getInput data. Likewise, the reduce and summarize stages do not change any additional data. They process only internal map/reduce data.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType MapReduceScript
4 */
5 define(['N/search', 'N/record'],
6   function(search, record){
7     return {
8       getInputData: function (context)
9     {
10       var filter1 = search.createFilter({
11         name: 'mainline',
12         operator: search.Operator.IS,
13         values: true
14       });
15       var column1 = search.createColumn({name: 'recordtype'});
16       var srch = search.create({
17         type: search.Type.SALES_ORDER,
18         filters: [filter1],
19         columns: [column1]
20       });
21       return srch;
22     },
23     map: function (context)
24     {
25       var soEntry = JSON.parse(context.value);
26       var so = record.load({
27         type: soEntry.values.recordtype,
28         id: context.key
29       });
30       // UPDATE so FIELDS
31       so.save();
32
33       context.write({
34         key: soEntry.values.recordtype,
35         value: context.key
36       });
37
38     },
39     reduce: function (context)
40     {
41
42       context.write({
43         key: context.key,
44         value: context.values.length
45     });
46
47     },
48     summarize: function (summary)
49     {
50       var totalRecordsUpdated = 0;
51       summary.output.iterator().each(function (key, value)
52     {
53       log.audit({
54         title: key + ' records updated',
55         details: value
56       });
57       totalRecordsUpdated += parseInt(value);
58       return true;
59     });
60     log.audit({
61       title: 'Total records updated',

```

```

62     details: totalRecordsUpdated
63   });
64 }
65 });
66 });

```

A Robust Map/Reduce Script Example

Comparing Example 5 to Example 4, a filter was added to the search in the getInput stage. The purpose is to filter out already processed sales orders. The filter makes it possible to re-execute the whole map/reduce task repeatedly, because when the whole task is re-executed, the additional filter ensures that only unprocessed sales orders will be returned from the input stage and not all sales orders.

There are also substantial improvements to the map function. In Example 5, if the ((context.isRestarted === false)) condition is met, the script knows it is the first execution of the map function for the current key/value pair. It won't need to perform any additional checks and can go directly to the sales order record update.

During the sales order update, an operation sets the custbody_processed_flag flag. The script performs a check on this flag only as necessary. If (context.isRestarted === true), then the script looks up the appropriate processed flag value, and executes the sales order update only if it wasn't already updated.

Although the script includes more checks and lookups than example 4, the processing demand is light. To perform the check, the script uses a lookup method that doesn't load the full record. If the processed flag value is true, then the record is not loaded again.

In the map function, note that the context.write(...) statement is not in the if-statement body. It is because when a map function for a particular key/value pair is restarted, all these writes done in the previous execution of the map function are deleted. So there is no need to check which writes have or haven't been done.

The reduce function is not changed from Example 4. This reduce stage handles only the map/reduce internal data, and so the map/reduce framework ensures that even when the reduce function is restarted for a particular key/value pair, only the writes from its last execution for the key/value pair are passed to the next stage.

The summarize function also didn't require improvement. However, it is a good practice to log any restarts. For example, to account for when the "Total records updated" entry appears twice in the execution log for a single map/reduce task execution.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType MapReduceScript
4 */
5 define(['N/search', 'N/record'],
6   function(search, record){
7     return {
8       getInputData: function (context)
9     {
10       var filter1 = search.createFilter({
11         name: 'mainline',
12         operator: search.Operator.IS,
13         values: true
14       });
15       var filter2 = search.createFilter({
16         name: 'custbody_processed_flag',
17         operator: search.Operator.IS,
18         values: false
19       });
20       var column1 = search.createColumn({name: 'recordtype'});
21       var srch = search.create({
22         type: search.Type.SALES_ORDER,
23         filters: [filter1, filter2],
24         columns: [column1]

```

```

25     });
26     return srch;
27   },
28   map: function (context)
29   {
30     var soEntry = JSON.parse(context.value);
31     var alreadyProcessed = false;
32     if (context.isRestarted)
33     {
34       var lookupResult = search.lookupFields({
35         type: soEntry.values.recordtype,
36         id: context.key,
37         columns: ['custbody_processed_flag']
38       });
39       alreadyProcessed = lookupResult.custbody_processed_flag;
40     }
41     if (!alreadyProcessed)
42     {
43       var so = record.load({
44         type: soEntry.values.recordtype,
45         id: context.key
46       });
47       // UPDATE so FIELDS
48       so.setValue({
49         fieldId: 'custbody_processed_flag',
50         value: true
51       });
52       so.save();
53     }
54
55     context.write({
56       key: soEntry.values.recordtype,
57       value: context.key
58     });
59   },
60   reduce: function (context)
61   {
62     context.write({
63       key: context.key,
64       value: context.values.length
65     });
66   },
67   summarize: function (summary)
68   {
69     if (summary.isRestarted)
70     {
71       log.audit({details: 'Summary stage is being restarted!'});
72     }
73     var totalRecordsUpdated = 0;
74     summary.output.iterator().each(function (key, value)
75     {
76       log.audit({
77         title: key + ' records updated',
78         details: value
79       });
80       totalRecordsUpdated += parseInt(value);
81       return true;
82     });
83     log.audit({
84       title: 'Total records updated',
85       details: totalRecordsUpdated
86     });
87   }
88 }
89 });
90 });
91 });

```

SuiteScript 2.x Map/Reduce Script Best Practices

- Writing Efficient Map and Reduce Functions

- [Passing Search Data to the getInputData Stage](#)
- [Minimizing the Risk of Data Duplication](#)

Writing Efficient Map and Reduce Functions

As described in [Hard Limits on Function Invocations](#), NetSuite imposes governance limits on single invocations of map, reduce, getInputData, and summarize functions:

- **map**: 1,000 usage units (the same as mass update scripts)
- **reduce**: 5,000 usage units
- **getInputData**: 10,000 usage units
- **summarize**: 10,000 usage units

If you are concerned about potential issues with these limits, review your script to make sure that your map and reduce functions are relatively lightweight. Your map and reduce functions should not include a long or complex series of actions. For example, consider a situation in which your map or reduce function loads and saves multiple records all at once. This approach might cause an issue with the limits described above. If your getInputData function returns a list of record IDs, a better approach might be to use the map function to load each record, update fields on the record, and save it.

If you have a script that performs a significantly more complex series of operations within a single function (such as loading and saving multiple records, or transforming multiple records), consider using a different script type, such as a scheduled script.

Passing Search Data to the getInputData Stage

In the getInputData stage, your script must return an object that can be transformed into a list of key/value pairs. A common approach is to use a search. If you decide to use this technique, note that the recommended approach is to have your function return either a [search.Search](#) object or an object reference to a saved search. By contrast, if you execute a search within the getInputData function and return the results (for example, as an array), there is a greater risk that the search will time out.

Instead, Oracle recommends using one of the following approaches:

- Return a search object. That is, return an object created using [search.create\(options\)](#) or [search.load\(options\)](#).
- Return a search object reference. That is, return an [inputContext.ObjectRef](#) object that references a saved search.

In both cases, the time limit available to the search is more generous than it would be for a search executed within the function.

The following snippet shows how to return a search object:

```

1 | function getInputData()
2 | {
3 |   return search.create({
4 |     type: record.Type.INVOICE,
5 |     filters: [['status', search.Operator.IS, 'open']],
6 |     columns: ['entity'],
7 |     title: 'Open Invoice Search'
8 |   });
9 | }

```

And the following snippet shows how to return a search object reference:

```

1 | ...

```

```

2 | function getInputData {
3 | {
4 |     // Reference a saved search with internal ID 1234.
5 |
6 |     return {
7 |         type: 'search',
8 |         id: 1234
9 |     };
10 |
11 | ...

```

For information about additional ways to return data, see [getInputData\(inputContext\)](#).

Minimizing the Risk of Data Duplication

A map/reduce script can be interrupted at any time by an application server disruption. Afterward, the script is restarted.

Depending on how the script is configured, when a map or reduce job starts again, it may attempt to retry processing for the same key/value pairs it had flagged for processing when the interruption occurred. Similarly, if an uncaught error disrupts the job, the system may retry processing for the pair that was being processed when the error occurred.



Note: For an overview of the system's behavior following an interruption, see [System Response After a Map/Reduce Interruption](#).

When a job is restarted, there is an inherent risk of data duplication. To minimize this risk, use the following guidance.

Add Logic for Handling Restarts

Every map/reduce script should be written in such a way that each entry point function checks to see whether the function has been previously invoked. To do this, use the context.isRestarted property, which exists for every map/reduce entry point. If the function has been restarted, the script should provide any logic needed to avoid duplicate processing. For examples, see [Adding Logic to Handle Map/Reduce Restarts](#).

Leave Buffer Size Set to 1

When you deploy a script, the deployment record includes a field called Buffer Size. The default value of this field is 1. In general, Oracle recommends that you leave this value set to the default.

The purpose of the Buffer Size field is to control how many key/value pairs are flagged for processing at one time, and how frequently a map or reduce job saves data about its progress. Setting this field to a higher value may have a small performance advantage. However, the disadvantage is that, if the job is interrupted by an application server restart, there is a greater likelihood of one or more key/value pairs being processed twice. For that reason, Oracle recommends leaving this value set to 1, particularly if the script is processing records.

For more details on this field, see [Buffer Size](#).

SuiteScript 2.x Map/Reduce Script Entry Points and API

This topic includes two parts:

- [Map/Reduce Script Entry Points](#) — A summary of the four entry points available to a map/reduce script.

- Map/Reduce Script API — A summary of properties available through the map/reduce entry points.

Map/Reduce Script Entry Points

The map/reduce script type includes four entry points. These entry points let you control both the script's behavior and the data flow within the map/reduce stages. For an overview of the stages, see [SuiteScript 2.x Map/Reduce Script Stages](#).

| Entry Point | Context Object | Required/Optional | Description |
|---|-----------------------------|---|--|
| <code>getInputData(inputContext)</code> | <code>inputContext</code> | Required | Marks the beginning of the map/reduce script execution. Invokes the input stage. This function is invoked one time in the execution of the script. |
| <code>map(mapContext)</code> | <code>mapContext</code> | Optional, but if this entry point is skipped, the <code>reduce(reduceContext)</code> entry point is required. | Invokes the map stage. If this entry point is used, the map function is invoked one time for each key/value pair provided by the <code>getInputData(inputContext)</code> function. |
| <code>reduce(reduceContext)</code> | <code>reduceContext</code> | Optional, but if this entry point is skipped, the <code>map(mapContext)</code> entry point is required. | Invokes the reduce stage. If this entry point is used, the reduce function is invoked one time for each key and list of values provided. Data is provided either by the map stage or, if the map stage is not used, by the getInpuData stage. |
| <code>summarize(summaryContext)</code> | <code>summaryContext</code> | Optional | Invokes the summarize stage. If the summarize entry point is used, the summarize function is invoked one time in the execution of the script. |

Map/Reduce Script API

The following tables describe properties that are available through the map/reduce entry points.

inputContext Object Members

The following members are called on `inputContext`.

| Member Type | Name | Return Type / Value Type | Description |
|-------------|---------------------------------------|--------------------------|--|
| Property | <code>inputContext.isRestarted</code> | Boolean | Indicates whether the current invocation of the <code>getInputData(inputContext)</code> function represents a restart. |
| Object | <code>inputContext.ObjectRef</code> | object | An object that contains the input data. |

The following members are called on `inputContext.ObjectRef`.

| Member Type | Name | Return Type / Value Type | Description |
|-------------|---------------------------|--------------------------|---|
| Property | <code>ObjectRef.id</code> | string number | The internal ID or script ID of the object. For example, this value could be a saved search ID. |

| Member Type | Name | Return Type / Value Type | Description |
|-------------|----------------|--------------------------|--------------------|
| | ObjectRef.type | string | The object's type. |

mapContext Object Members

The following members are called on [mapContext](#).

| Member Type | Name | Return Type / Value Type | Description |
|-------------|---------------------------|--------------------------|---|
| Property | mapContext.isRestarted | Boolean | Indicates whether the current invocation of the map(mapContext) function represents a restart. If the value of isRestarted is true, then the function was invoked previously, but unsuccessfully, for the current key/value pair. |
| | mapContext.executionNo | property | Indicates whether the current invocation of the map(mapContext) function represents the first or a subsequent attempt to process the current key/value pair. |
| | mapContext.errors | iterator | Holds serialized errors that were thrown during previous attempts to execute the map(mapContext) function on the current key/value pair. |
| | mapContext.key | string | The key to be processed during the current invocation of the map(mapContext) function. |
| | mapContext.value | string | The value to be processed during the current invocation of the map(mapContext) function. |
| Method | mapContext.write(options) | void | Writes the map(mapContext) output as key/value pairs. This data is passed to the reduce stage, if the reduce entry point is used, or to the summarize stage. |

reduceContext Object Members

The following members are called on [reduceContext](#).

| Member Type | Name | Return Type / Value Type | Description |
|-------------|---------------------------|--------------------------|---|
| Property | reduceContext.isRestarted | Boolean | Indicates whether the current invocation of the reduce(reduceContext) function represents a restart. If the value of isRestarted is true, then the function was invoked previously, but unsuccessfully, for the current key/value pair. |
| | reduceContext.executionNo | number | Indicates whether the current invocation of the reduce(reduceContext) function represents the first or a subsequent attempt to process the current key/value pair. |
| | reduceContext.errors | iterator | Holds serialized errors that were thrown during previous attempts to execute the |

| Member Type | Name | Return Type / Value Type | Description |
|-------------|------------------------------|--------------------------|---|
| | | | reduce(reduceContext) function on the current key and its associated values. |
| | reduceContext.key | string | The input key to process during the reduce stage. |
| | reduceContext.values | string[] | The key to be processed during the current invocation of the reduce(reduceContext) function. |
| Method | reduceContext.write(options) | void | Writes the reduce(reduceContext) function as key/value pairs. This data is passed to the summarize stage. |

summaryContext Object Members

The following members are called on the [summaryContext](#).

| Member Type | Name | Value Type | Description |
|-------------|------------------------------|---------------------|---|
| Property | summaryContext.isRestarted | boolean (read-only) | Indicates whether the current invocation of the summarize(summaryContext) function represents a restart. If the value of isRestarted is true, then the function was invoked previously, but unsuccessfully. |
| | summaryContext.concurrency | number | The maximum concurrency number when running the map/reduce script. |
| | summaryContext.dateCreated | Date | The time and day when the script began running. |
| | summaryContext.seconds | number | The total number of seconds that elapsed during the processing of the script. |
| | summaryContext.usage | number | The total number of usage units consumed during the processing of the script. |
| | summaryContext.yields | number | The total number of yields that occurred during the processing of the script. |
| | summaryContext.inputSummary | object | Object that contains data about the input stage. |
| | summaryContext.mapSummary | object | Object that contains data about the map stage. |
| | summaryContext.reduceSummary | object | Object that contains data about the reduce stage. |
| | summaryContext.output | iterator | Iterator that contains the keys and values saved as the output of the reduce stage. |

inputSummary Object members

The following members are called on [summaryContext.inputSummary](#).

| Member Type | Name | Value Type | Description |
|-------------|--------------------------|------------|--|
| Property | inputSummary.dateCreated | Date | The time and day when the getInputData(inputContext) function began running. |

| Member Type | Name | Value Type | Description |
|-------------|----------------------|------------|---|
| | inputSummary.error | string | Holds serialized errors thrown from the <code>getInputData(inputContext)</code> function. |
| | inputSummary.seconds | number | The total number of seconds that elapsed during execution of the <code>getInputData(inputContext)</code> function. This tally does not include idle time. |
| | inputSummary.usage | number | The total number of usage units consumed by processing of the <code>getInputData(inputContext)</code> function. |

mapSummary Members

The following members are called on `summaryContext.mapSummary`.

| Member Type | Name | Value Type | Description |
|-------------|------------------------|------------|--|
| Property | mapSummary.concurrency | number | Maximum concurrency number when running <code>map(mapContext)</code> . |
| | mapSummary.dateCreated | Date | The time and day when the first invocation of <code>map(mapContext)</code> function began. |
| | mapSummary.errors | iterator | Holds serialized errors thrown during the map stage. |
| | mapSummary.keys | iterator | Holds the keys passed to the map stage by the <code>getInputData</code> stage. |
| | mapSummary.seconds | number | The total number of seconds that elapsed during the map stage. |
| | mapSummary.usage | number | The total number of usage units consumed during the map stage. |
| | mapSummary.yields | number | The total number of yields that occurred during the map stage. |

reduceSummary Members

The following members are called on `summaryContext.reduceSummary`.

| Member Type | Name | Value Type | Description |
|-------------|---------------------------|------------|--|
| Property | reduceSummary.concurrency | number | Maximum concurrency number when running <code>reduce(reduceContext)</code> . |
| | reduceSummary.dateCreated | Date | The time and day when the first invocation of the <code>reduce(reduceContext)</code> function began. |
| | reduceSummary.errors | iterator | Holds serialized errors thrown during the reduce stage. |
| | reduceSummary.keys | iterator | Holds the keys passed to the reduce stage. |
| | reduceSummary.seconds | number | The total number of seconds that elapsed during the reduce stage. |
| | reduceSummary.usage | number | The total number of usage units consumed during the reduce stage. |

| Member Type | Name | Value Type | Description |
|-------------|----------------------|------------|---|
| | reduceSummary.yields | number | The total number of yields that occurred during the reduce stage. |

getInputData(inputContext)

Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|--|
| Description | <p>Marks the beginning of the script's execution. The purpose of the input stage is to generate the input data.</p> <p>Executes when the <code>getInputData</code> entry point is triggered. This entry point is required.</p> <p>For information about the context object provided to this entry point, see inputContext.</p> <p>Note: When returning a <code>inputContext.ObjectRef</code>, the supported types are search and file.</p> <p>Note: When <code>getInputData()</code> returns a data structure with a non-string value, before the value is stored, it is converted to a JSON string with <code>JSON.stringify()</code>.</p> <p>Also see Passing Search Data to the getInputData Stage.</p> |
| Returns | <p>Array Object <code>search.Search</code> <code>inputContext.ObjectRef</code> <code>file.File</code> Object <code>query.Query</code> Object</p> <p>Examples:</p> <ul style="list-style-type: none"> ■ Array <pre>1 [{a : 'b'},{c : 'd'}]</pre> ■ Object <pre>1 { 2 a: {...}, 3 b: {...} 4 }</pre> ■ <code>search.Search</code> Object <pre>1 search.load({ 2 id: 1234 3 })</pre> ■ <code>search.Search</code> Object Reference <pre>1 { 2 type: 'search', 3 id: 1234 4 }</pre> ■ <code>file.File</code> Object <pre>1 file.load({ 2 id: 1234 3 })</pre> ■ <code>file.File</code> Object Reference <pre>1 {</pre> |

| | |
|-------|---|
| | <pre> 2 type: 'file', 3 path: '/SuiteScripts/data/names.txt' 4 }</pre> <p>Important: When returning a file.File Object or Object Reference, consider the following:</p> <ul style="list-style-type: none"> □ When using an Object Reference to a file, you must use an absolute path or bundle virtual path. Relative paths are not permitted. □ The output file cannot include blank characters. <ul style="list-style-type: none"> ■ query.Query Object <pre> 1 query.load({ 2 id: 'custworkbook237' 3 })</pre> <ul style="list-style-type: none"> ■ query.Query Object Reference <pre> 1 { 2 type: 'query', 3 id: 'custworkbook237' 4 }</pre> |
| Since | 2015.2 |

Parameters

| Parameter | Type | Required / Optional | Description |
|------------------------------|--------|---------------------|---|
| inputContext | Object | Required | <p>Object that contains:</p> <ul style="list-style-type: none"> ■ An indication of whether the current invocation of this method represents a restart ■ An Object that represents the input data <p>For a description of each property in this object, see inputContext Object Members.</p> |

Errors

When an error is thrown in this function, the job proceeds to the [summarize\(summaryContext\)](#) function. The serialized error is encapsulated in the [inputSummary.error](#) property.

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 // Add additional code
2 ...
3 function getInputData {
4 {
5
6   // Reference a saved search that returns a list of NetSuite records that
7   // require processing - for example, sales orders that are pending fulfillment.
8
9   return {
10     type: 'search',
```

```

11     id: 1234
12   };
13 }
14 ...
15 // Add additional code

```

inputContext



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|---|
| Object Description | <p>This object is passed to the getInputData(inputContext) entry point function.</p> <p>This object includes the following properties:</p> <ul style="list-style-type: none"> ■ inputContext.isRestarted ■ inputContext.ObjectRef <ul style="list-style-type: none"> □ ObjectRef.id □ ObjectRef.type |
| Since | 2016.1 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 // Add additional code
2 ...
3 function getInputData {
4 {
5
6 // Reference a saved search that returns a list of NetSuite records that
7 // require processing - for example, sales orders that are pending fulfillment.
8
9 return {
10   type: 'search',
11   id: 1234
12 };
13 }
14 ...
15 // Add additional code

```

inputContext.isRestarted



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | <p>Indicates whether the current invocation of the getInputData(inputContext) function is the first.</p> <p>Typically, the getInput function is invoked one time only. However, if the function is interrupted by an application server restart, the system restarts the getInputData function.</p> <p>If the value of this property is true, the getInputData(inputContext) has been restarted.</p> <p>You can use this property to help make your script more robust.</p> |
| Type | boolean (read-only) |
| Since | 2016.1 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 function getInputData(context)
3 {
4     if (context.isRestarted)
5     {
6         log.debug('GET_INPUT isRestarted', 'YES');
7     }
8     else
9     {
10        log.debug('GET_INPUT isRestarted', 'NO');
11    }
12
13 var extractSearch = search.load({ id: 'customsearch35' });
14 return extractSearch;
15 }
16 ...

```

inputContext.ObjectRef

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|---|
| Object Description | References the object that contains the input data. For example, a reference to a saved search. You can use getInputData(inputContext) to return this object. |
| |  Note: The only supported object types are search and file. |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 {
3     type: 'search',
4     id: 1234 //search internal id
5 }
6 ...

```

ObjectRef.id

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | The internal ID or script ID of the object. For example, the saved search ID. |
| Type | string number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 {
3   type: 'search',
4   id: 1234 //search internal id
5 }
6 ...

```

ObjectRef.type



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--------------------|
| Property Description | The object's type. |
| Type | string |
| Values | 'search' |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 {
3   type: 'search',
4   id: 1234 //search internal id
5 }
6 ...

```

map(mapContext)



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | <p>Executes when the map entry point is triggered.</p> <p>The logic in your map function is applied to each key/value pair that is provided by the getInputData stage. One key/value pair is processed per function invocation, then the function is invoked again for the next pair.</p> <p>The output of the map stage is another set of key/value pairs. During the shuffle stage that always follows, these pairs are automatically grouped by key.</p> <p>For information about the context object provided to this entry point, see mapContext.</p> |
| Returns | Void |
| Since | 2015.2 |

Parameters

| Parameter | Type | Required / Optional | Description |
|----------------------------|--------|---------------------|---|
| mapContext | Object | Required | <p>Object that contains:</p> <ul style="list-style-type: none"> ■ The key/value pairs to process during the map stage. |

| Parameter | Type | Required / Optional | Description |
|-----------|------|---------------------|--|
| | | | <ul style="list-style-type: none"> ■ Logic that lets you save data to pass to the reduce stage. ■ Other properties you can use within the map function. <p>For a description of each property in this object, see mapContext Object Members.</p> |

Errors

When an error is thrown, the behavior of the job varies depending on the setting of the [retryCount](#) configuration option.

If the function has been restarted for a key/value pair that it previously attempted to process, any errors logged during prior attempts can be accessed through [mapContext.errors](#).

In the summary stage, you can review all map stage errors by using [mapSummary.errors](#).

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 // Add additional code
2 ...
3 function map(context)
4 {
5     for (var i = 0; context.value && i < context.value.length; i++)
6         if (context.value[i] !== ' ' && !PUNCTUATION_REGEXP.test(context.value[i]))
7             {
8                 context.write({
9                     key: context.value[i],
10                    value: 1
11                });
12            }
13    }
14 ...
15 ...
16 // Add additional code

```

mapContext



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|---|
| Object Description | <p>Object that contains:</p> <ul style="list-style-type: none"> ■ The key/value pairs to process during the map stage. ■ Logic that lets you save data to pass to the reduce stage. ■ Other properties you can use within the map function. <p>This object includes the following properties and methods:</p> <ul style="list-style-type: none"> ■ mapContext.isRestarted ■ mapContext.executionNo ■ mapContext.errors ■ mapContext.key ■ mapContext.value ■ mapContext.write(options) |
|---------------------------|---|

| | |
|-------|--------|
| Since | 2015.2 |
|-------|--------|

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 function map(context)
3 {
4     for (var i = 0; context.value && i < context.value.length; i++)
5         if (context.value[i] !== ' ' && !PUNCTUATION_REGEXP.test(context.value[i]))
6             {
7                 context.write({
8                     key: context.value[i],
9                     value: 1
10                });
11            }
12        }
13 ...

```

mapContext.isRestarted

| | |
|--|--|
| i Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1. | |
| Property Description | <p>Indicates whether the <code>map(mapContext)</code> function has been invoked previously for the current key/value pair.</p> <p>For an overview of events that can cause a restart, see System Response After a Map/Reduce Interruption.</p> <p>When the map function is restarted for a key/value pair, data previously written by the incomplete function is deleted. However, some of the function's logic might have been executed before the function invocation was interrupted. For that reason, if the <code>mapContext.isRestarted</code> value is true, your script should take the necessary actions to avoid duplicate processing. For examples, see Adding Logic to Handle Map/Reduce Restarts.</p> <p>Related properties include <code>mapContext.executionNo</code> and <code>mapContext.errors</code>.</p> |
| Type | Boolean |
| Since | 2016.1 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 // Add additional code.
2 ...
3 function map(context) {
4     if (context.isRestarted)
5     {
6         // Add logic designed to assess how much processing was completed for this key/value pair and react accordingly.
7     }
8     else
9     {
10        // Let full processing continue for the key/value pair.
11    }
12 ...
13 // Add additional code.

```

mapContext.executionNo



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Indicates whether the current invocation of the map(mapContext) function is the first or a subsequent invocation for the current key/value pair. For an overview of events that can cause a restart, see System Response After a Map/Reduce Interruption . When the map function is restarted for a key/value pair, data previously written by the incomplete function is deleted. However, some of the function's logic might have been executed before the function invocation was interrupted. For that reason, you may want to use the <code>mapContext.executionNo</code> property to provide logic designed to avoid duplicate processing. For examples of how to write a robust map/reduce script, see Adding Logic to Handle Map/Reduce Restarts . Related properties include mapContext.isRestarted and mapContext.errors . |
| Type | number |
| Since | 2018.1 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 // Add additional code
2 ...
3
4 if (context.executionNo === 1){
5     // Permit full processing of the key/value pair.
6 }
7
8 else if (context.executionNo === 2){
9     // Take steps to check whether any processing was previously completed.
10 }
11
12 else {
13     // Take other steps that might be necessary in the case of more than
14     // one previous attempt.
15 }
16 ...
17 // Add additional code

```

mapContext.errors



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | Holds serialized errors that were thrown during previous attempts to execute the map function on the current key/value pair. This iterator may also hold the SSS_APP_SERVER_RESTART error code, which is recorded if the function is interrupted by an application server restart. For an overview of events that can cause the map function to be invoked more than once for a key/value pair, see System Response After a Map/Reduce Interruption . |
| Type | iterator |

| | |
|--------------|--------|
| Since | 2018.1 |
|--------------|--------|

Members

| Member | Type | Required/Optional | Description |
|---|----------|-------------------|-----------------------------------|
| iterator().each(parameters) | function | required | Executes one time for each error. |

parameters

| Member | Type | Required/Optional | Description |
|--|----------|-------------------|--|
| iteratorFunction(key, error, executionNo) See also functionParameters . | function | required | Provides logic to be executed during each iteration. |

functionParameters

| Parameter | Type | Required/Optional | Description |
|-------------|--------|-------------------|--|
| key | string | optional | Represents the key/value pair that the map function was attempting to process when the error occurred. |
| error | string | optional | A serialization of the error thrown. |
| executionNo | number | optional | Indicates whether the error occurred during the first or a subsequent attempt to process the key/value pair. |

Syntax

The following snippets shows three ways you could use this iterator.

This code is not a functional example. For a complete script sample, see [Map/Reduce Script Samples](#).

```

1 //Add additional code
2 ...
3
4 // Create a log entry showing each full serialized error, and the corresponding key.
5
6 context.errors.iterator().each(function (key, error, executionNo){
7     log.error({
8         title: 'Map error for key: ' + key + ', execution no ' + executionNo,
9         details: error
10    });
11    return true;
12 });
13
14
15 // Log only the name and description of each error thrown.
16
17 context.errors.iterator().each(function (key, error, executionNo){
18     var errorObject = JSON.parse\(error\);
19     log.error({
20         title: 'Reduce error for key ' + key + ', execution no. ' + executionNo,
21         details: errorObject.name + ': ' + errorObject.message
22    });
23    return true;
24 });
25
26
27 // Calculate and log the number of errors encountered.
28
29 var errorCount = 0;
```

```

30 | context.errors.iterator().each(function() {
31 |     errorCount++;
32 |     return true;
33 | });
34 |
35 | log.audit ({
36 |     title: 'Errors for map key: ' + context.key,
37 |     details: 'Total number of errors: ' + errorCount
38 | });

```

mapContext.key



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | The key to be processed during the map stage. <ul style="list-style-type: none"> ■ If the input type is an array, the key is the index of the element. ■ If the input type is an object, the key is the key in the object. ■ If the input type is a result set, the key is the internal ID of the result. If the search result has no internal ID, the key is the index of the search result. |
| Type | string |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 // Add additional code
2 ...
3
4     context.write({
5         key: context.value[i],
6         value: 1
7     });
8 ...
9 // Add additional code

```

mapContext.value



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | The value to be processed during the map stage. <ul style="list-style-type: none"> ■ If the input type is an array, the mapContext.value is the value in the element. ■ If the input type is an object, the mapContext.value is the value in the object. ■ If the input type is a result set, the mapContext.value is a search.Result object converted to a JSON string by using JSON.stringify(). |
| Type | string |

| | |
|--------------|--------|
| Since | 2015.2 |
|--------------|--------|

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 // Add additional code.
2 ...
3
4 // Assume that the search result is a list of phone call records. This snippet parses the results
5 // and uses the values of the title and the message fields from each record.
6
7 var searchResult = JSON.parse(context.value);
8
9 var title = searchResult.values.title;
10 var message = searchResult.values.message;
11
12 ...
13 // Add additional code.

```

`mapContext.write(options)`

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|---|
| Method Description | Writes the key/value pairs to be passed to the shuffle and then the reduce stage. If your script includes both a map and reduce function, you must use this method in order for the reduce function to be invoked. |
| Returns | Void |
| Since | 2015.2 |

Parameters

 **Note:** The options parameter is a JavaScript object.

| Parameter | Type | Required / Optional | Description |
|---------------|--|---------------------|--------------------|
| options.key | String or object. However, note that if you provide an object, the system calls <code>JSON.stringify()</code> on your input. | required | The key to write |
| options.value | | required | The value to write |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 function map(context)
3 {
4     for (var i = 0; context.value && i < context.value.length; i++)
5         if (context.value[i] !== ' ' && !PUNCTUATION_REGEXP.test(context.value[i]))
6             {
7                 context.write({
8                     key: context.value[i],
9                     value: 1

```

```

10      });
11  }
12 }
13 ...

```

reduce(reduceContext)



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Executes when the reduce entry point is triggered. |
| | The logic in your reduce function is applied to each key, and its corresponding list of value. Only one key, with its corresponding values, is processed per function invocation. The function is invoked again for the next key and corresponding set of values. |
| | Data is provided to the reduce stage by one of the following: <ul style="list-style-type: none"> ■ The getInputData stage — if your script has no map function. ■ The shuffle stage — if your script uses a map function. The shuffle stage follows the map stage. Its purpose is to sort data from the map stage by key. |
| | For information about the context object provided to this entry point, see reduceContext . |
| Returns | Void |
| Since | 2015.2 |

Parameters

| Parameter | Type | Required / Optional | Description |
|-------------------------------|--------|---------------------|---|
| reduceContext | Object | Required | Object that contains: <ul style="list-style-type: none"> ■ The data to process during the reduce stage. ■ Logic that lets you save data to pass to the summarize stage. ■ Other properties you can use within the reduce function. For a description of each property in this object, see reduceContext Object Members . |

Errors

When an error is thrown, the behavior of the job varies depending on the setting of the [retryCount](#) configuration option.

If the function has been restarted for a key/value pair that it previously attempted to process, any errors logged during prior attempts can be accessed through [reduceContext.errors](#).

In the summary stage, you can review all reduce stage errors by using [reduceSummary.errors](#).

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...

```

```

2 | function reduce(context)
3 | {
4 |   context.write({
5 |     key: context.key ,
6 |     value: context.values.length
7 |   });
8 | }
9 | ...

```

reduceContext



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|--|
| Object Description | <p>Contains the key/value pairs to process during the reduce stage. This object includes the following properties and methods:</p> <ul style="list-style-type: none"> ▪ reduceContext.isRestarted ▪ reduceContext.executionNo ▪ reduceContext.errors ▪ reduceContext.key ▪ reduceContext.values ▪ reduceContext.write(options) |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 | ...
2 | function reduce(context)
3 | {
4 |   context.write({
5 |     key: context.key ,
6 |     value: context.values.length
7 |   });
8 | }
9 | ...

```

reduceContext.isRestarted



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | <p>Indicates whether the reduce(reduceContext) function has been invoked previously for the current key and values.</p> <p>For an overview of events that can cause a restart, see System Response After a Map/Reduce Interruption.</p> <p>When the reduce function is restarted for a key/value pair, data previously written by the incomplete function is deleted. However, some of the function's logic might have been executed before the function invocation was interrupted. For that reason, if the <code>reduceContext.isRestarted</code> property is true, your script should take the necessary actions to avoid duplicate processing. For examples, see Adding Logic to Handle Map/Reduce Restarts.</p> <p>Related properties include reduceContext.executionNo and reduceContext.errors.</p> |
|-----------------------------|--|

| | |
|-------|---------|
| Type | Boolean |
| Since | 2016.1 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 // Add additional code.
2 ...
3 function reduce (context) {
4     if (context.isRestarted)
5     {
6         // Add logic designed to assess how much processing was completed for this key
7         // and react accordingly.
8     }
9     else
10    {
11        // Let full processing continue for the current key and its values.
12    }
13 ...
14 // Add additional code.
```

reduceContext.executionNo



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Indicates whether the current invocation of the reduce(reduceContext) function is the first, second, third, or fourth for the current key and its values. For an overview of events that can cause a restart, see System Response After a Map/Reduce Interruption . When the reduce function is restarted for a key, data previously written by the incomplete function is deleted. However, some of the function's logic might have been executed before the function invocation was interrupted. For that reason, you may want to use the <code>reduceContext.executionNo</code> property to provide logic designed to avoid duplicate processing. For examples of how to write a robust map/reduce script, see Adding Logic to Handle Map/Reduce Restarts . Related properties include <code>reduceContext.isRestarted</code> and <code>reduceContext.errors</code> . |
| Type | number |
| Since | 2018.1 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 // Add additional code
2 ...
3
4 if (context.executionNo === 1){
5     // Permit full processing of the key/value pair.
6 }
7
8 else if (context.executionNo === 2){
9     // Take steps to check whether any processing was previously completed.
```

```

10  }
11
12 else {
13     // Take other steps that might be necessary in the case of more than
14     // one previous attempt.
15 }
16 ...
17 //Add additional code

```

reduceContext.errors



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Holds serialized errors that were thrown during previous attempts to execute the reduce function on the current key and its values. This iterator may also hold the SSS_APP_SERVER_RESTART error code, which is recorded if the function is interrupted by an application server restart. For an overview of events that can cause the map function to be invoked more than once for a key/value pair, see System Response After a Map/Reduce Interruption . |
| Type | iterator |
| Since | 2018.1 |

Members

| Member | Type | Required/Optional | Description |
|---|----------|-------------------|-----------------------------------|
| iterator().each(Parameters) | function | required | Executes one time for each error. |

parameters

| Member | Type | Required/Optional | Description |
|--|----------|-------------------|--|
| iteratorFunction(key, error, executionNo) See also functionParameters . | function | required | Provides logic to be executed during each iteration. |

functionParameters

| Parameter | Type | Required/Optional | Description |
|-------------|--------|-------------------|---|
| key | string | optional | Represents the key/value pair that the reduce function was attempting to process when the error occurred. |
| error | string | optional | A serialization of the error thrown. |
| executionNo | number | optional | Indicates whether the error occurred during the first or a subsequent attempt to process the key. |

Syntax

The following snippets shows three ways you could use this iterator.

This code is not a functional example. For a complete script sample, see [Map/Reduce Script Samples](#).

```

1 //Add additional code
2 ...
3
4 // Create a log entry showing each full serialized error, and the corresponding key.
5
6 context.errors.iterator().each(function (key, error, executionNo){
7     log.error({
8         title: 'Reduce error for key: ' + key + ', execution no ' + executionNo,
9         details: error
10    });
11    return true;
12 });
13
14
15 // Log only the name and description of each error thrown.
16
17 context.errors.iterator().each(function (key, error, executionNo){
18     var errorObject = JSON.parse(error);
19     log.error({
20         title: 'Reduce error for key ' + key + ', execution no. ' + executionNo,
21         details: errorObject.name + ': ' + errorObject.message
22    });
23    return true;
24 });
25
26
27 // Calculate and log the number of errors encountered.
28
29 var errorCount = 0;
30 context.errors.iterator().each(function() {
31     errorCount++;
32     return true;
33 });
34
35 log.audit ({
36     title: 'Errors for reduce key: ' + context.key,
37     details: 'Total number of errors: ' + errorCount
38 });

```

reduceContext.key



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | <p>When the map/reduce process includes a map stage, the reduce keys are the keys written by mapContext.write(options).</p> <p>When the map stage is skipped, the reduce keys are provided by the getInputData stage:</p> <ul style="list-style-type: none"> ■ If the input type is an array, the key is the index of the element. ■ If the input type is an object, the key is the key in the object. ■ If the input type is a result set, the key is the internal ID of the result. <div data-bbox="518 1584 1380 1647" style="background-color: #e0f2ff; padding: 10px;"> <p>Note: Each key cannot exceed 4000 bytes.</p> </div> |
| Type | string |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 function reduce(context)
3 {
4     context.write({
5         key: context.key ,
6         value: context.values.length
7     });
8 }
9 ...

```

reduceContext.values



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | This array holds all values associated with a unique key that was passed to the reduce stage for processing. These values are listed in lexicographical order . When the map/reduce process includes a map stage, the key/value pairs passed to the reduce stage are derived from the values written by mapContext.write(options) . When the map stage is skipped, the values are determined by the getInputData stage: <ul style="list-style-type: none"> ■ If the input type is an array, it is the value in the element. ■ If the input type is an object, it is the value in the object. ■ If the input type is a result set, the value is a search.Result object converted to a JSON string with <code>JSON.stringify()</code>. |
| Type | string[] |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 function reduce(context)
3 {
4     context.write({
5         key: context.key ,
6         value: context.values.length
7     });
8 }
9 ...

```

reduceContext.write(options)



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|-------------------------|
| Method Description | Writes key/value pairs. |
| Returns | Void |
| Since | 2015.2 |

Parameters

i Note: The options parameter is a JavaScript object.

| Parameter | Type | Required / Optional | Description |
|---------------|---|---------------------|--------------------|
| options.key | String or object. However, note that if you provide an object, the system calls JSON.stringify() on your input. | required | The key to write. |
| options.value | | required | The value to write |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 function reduce(context)
3 {
4     context.write({
5         key: context.key,
6         value: context.values.length
7     });
8 }
9 ...

```

summarize(summaryContext)

i Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|--|
| Description | Executes when the summarize entry point is triggered. When you add custom logic to this entry point function, that logic is applied to the result set. For information about the context object provided to this entry point, see summaryContext . |
| Returns | Void |
| Since | 2015.2 |

Parameters

| Parameter | Type | Required / Optional | Description |
|----------------|--------|---------------------|--|
| summaryContext | Object | Required | Holds statistics regarding the execution of a map/reduce script. For a description of each property in this object, see summaryContext Object Members . |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 function summarize(summary)

```

```

3 | {
4 |     var type = summary.toString();
5 |     log.audit(type + ' Usage Consumed', summary.usage);
6 |     log.audit(type + ' Concurrency Number ', summary.concurrency);
7 |     log.audit(type + ' Number of Yields', summary.yields);
8 |     var contents = '';
9 |     summary.output.iterator().each(function (key, value)
10 |     {
11 |         contents += (key + ' ' + value + '\n');
12 |         return true;
13 |     });
14 | ...

```

summaryContext

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|---|
| Object Description | <p>Holds statistics regarding execution of a map/reduce script.</p> <p>This object includes the following properties:</p> <ul style="list-style-type: none"> ■ summaryContext.isRestarted ■ summaryContext.concurrency ■ summaryContext.dateCreated ■ summaryContext.seconds ■ summaryContext.usage ■ summaryContext.yields ■ summaryContext.inputSummary <ul style="list-style-type: none"> □ inputSummary.dateCreated □ inputSummary.seconds □ inputSummary.usage □ inputSummary.error ■ summaryContext.mapSummary <ul style="list-style-type: none"> □ mapSummary.concurrency □ mapSummary.dateCreated □ mapSummary.keys □ mapSummary.seconds □ mapSummary.usage □ mapSummary.yields □ mapSummary.errors ■ summaryContext.reduceSummary <ul style="list-style-type: none"> □ reduceSummary.concurrency □ reduceSummary.dateCreated □ reduceSummary.keys □ reduceSummary.seconds □ reduceSummary.usage □ reduceSummary.yields □ reduceSummary.errors ■ summaryContext.output |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 function summarize(summary)
3 {
4     var type = summary.toString();
5     log.debug(type + ' Usage Consumed', summary.usage);
6     log.debug(type + ' Concurrency Number ', summary.concurrency);
7     log.debug(type + ' Number of Yields', summary.yields);
8     var contents = '';
9     summary.output.iterator().each(function (key, value)
10    {
11        contents += (key + ' ' + value + '\n');
12        return true;
13    });
14 ...

```

summaryContext.isRestarted

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Indicates whether the <code>summarize(summaryContext)</code> function was invoked again. To reduce negative impact to map/reduce processing if the Java virtual machine (JVM) restarts, NetSuite automatically restarts the current summary function. Summary data previously written by the incomplete function is deleted. If the value is true, the current process invoked by <code>summarize(summaryContext)</code> was restarted. You can use this information to help you write a more robust map/reduce script that is designed to continue interrupted work as necessary. |
| Type | boolean (read-only) |
| Since | 2016.1 |

Syntax

```

1 ...
2 function summarize(summary) {
3     if (summary.isRestarted)
4     {
5         log.debug('SUMMARY isRestarted', 'YES');
6     }
7     else
8     {
9         log.debug('SUMMARY isRestarted', 'NO');
10    }
11    log.debug('summarize', JSON.stringify(summary));
12 ...

```

summaryContext.concurrency

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | The maximum concurrency number when executing parallel tasks for the map/reduce script. |
|-----------------------------|---|

| | |
|-------|--|
| | Note: This number may be less than the number allocated on the script deployment. For example, tasks that remained in the pending state are not reflected in this number. |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 | ...
2 | log.audit(type + ' Concurrency Number ', summary.concurrency);
3 | ...

```

summaryContext.dateCreated

| |
|---|
| Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1. |
|---|

| | |
|-----------------------------|--|
| Property Description | The time and day when the map/reduce script began running. |
| Type | Date |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 | ...
2 | log.audit(type + ' Creation Date', summary.dateCreated);
3 | ...

```

summaryContext.seconds

| |
|---|
| Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1. |
|---|

| | |
|-----------------------------|---|
| Property Description | Total seconds elapsed when running the map/reduce script. |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 | ...
2 | log.audit(type + ' Total seconds elapsed', summary.seconds);

```

3 | ...

summaryContext.usage



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Total number of usage units consumed when running the map/reduce script. |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```
1 | ...
2 | log.audit(type + ' Usage Consumed', summary.usage);
3 | ...
```

summaryContext.yields



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Total number of yields when running the map/reduce script. |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```
1 | ...
2 | log.audit(type + ' Number of Yields', summary.yields);
3 | ...
```

summaryContext.inputSummary



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|---|
| Object Description | Holds statistics regarding the input stage. |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 logMetrics(summary.inputSummary);
3 log.error('Input Error', summary.inputSummary.error);
4 ...
5 ...

```

inputSummary.dateCreated



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | The time and day when getInputData(inputContext) began running. |
| Type | Date |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 log.audit(' Creation Date', summary.inputSummary.dateCreated);
3 ...

```

inputSummary.seconds



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | Total seconds elapsed when running getInputData(inputContext) (does not include idle time). |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 log.audit(' Time Elapsed', summary.inputSummary.seconds);
3 ...

```

inputSummary.usage



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Total number of usage units consumed when running getInputData(inputContext) . |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 log.audit(' Usage', summary.inputSummary.usage);
3 ...

```

inputSummary.error

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | If applicable, holds a serialized error that is thrown from getInputData(inputContext) . |
| Type | string |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 logMetrics(summary.inputSummary);
3 log.error('Input Error', summary.inputSummary.error);
4 ...
5 ...

```

summaryContext.mapSummary

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|--|
| Object Description | Holds statistics regarding the map stage |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 logMetrics(summary.mapSummary);
3 var mapKeys = [];
4 summary.mapSummary.keys.iterator().each(function (key)
5 {
6     mapKeys.push(key);
7     return true;
8 });
9 log.audit('MAP keys processed', mapKeys);
10 summary.mapSummary.errors.iterator().each(function (key, error)
11 {
12     log.error('Map Error for key: ' + key, error);
13     return true;

```

```

14 |     });
15 | ...

```

mapSummary.concurrency



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | The maximum concurrency number for executing parallel tasks during the map stage. |
| | <p>Note: This number may be less than the concurrency number allocated on the script deployment. For example, tasks that remained in the pending state are not reflected in this number.</p> |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 | ...
2 | log.audit(' Concurrency', summary.mapSummary.concurrency);
3 | ...

```

mapSummary.dateCreated



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | The time and day when map(mapContext) began running. |
| Type | Date |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 | ...
2 | log.audit(' Creation Date', summary.mapSummary.dateCreated);
3 | ...

```

mapSummary.keys



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Holds the keys that were passed to the map stage by the getInputData stage. Note that keys are sorted in the following way: |
|-----------------------------|--|

| | |
|-------|--|
| | <ul style="list-style-type: none"> ■ If all values are numeric, values are listed in numeric order. ■ If one or more values are strings, values are listed in lexicographical order. |
| Type | iterator |
| Since | 2015.2 |

Members

| Member | Type | Required/Optional | Description |
|---|----------|-------------------|---------------------------------|
| iterator().each(parameters) | function | required | Executes one time for each key. |

parameters

| Member | Type | Required/Optional | Description |
|--|----------|-------------------|--|
| iteratorFunction(key, executionCount, completionState) | function | required | Provides logic to be executed during each iteration. |

See also [functionParameters](#).

functionParameters

| Parameter | Type | Required/Optional | Description |
|-----------------|--------|-------------------|--|
| key | string | optional | Represents a key that was passed to the map stage. |
| executionCount | number | optional | The number of times the map function was invoked for the key. |
| completionState | string | optional | A setting that indicates whether the map function was executed successfully for the key. Possible values are 'COMPLETE', 'FAILED', and 'PENDING'. The system uses 'PENDING' if the script exited the stage before trying to process the key. This behavior can occur when exitOnError is set to true. |

Syntax

The following snippets show three ways you could use this iterator.

This code is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 //Add additional code.
2 ...
3
4 // Log all keys from the map stage.
5
6 var mapKeys = [];
7 summary.mapSummary.keys.iterator().each(function (key){
8     mapKeys.push(key);
9     return true;
10 });
11
12 log.debug({
13     title: 'Map stage keys',
14     details: mapKeys
15 });
16

```

```

17 // Create a log entry showing whether the map function was executed successfully for each key.
18
19 summary.mapSummary.keys.iterator().each(function (key, executionCount, completionState){
20
21 log.debug({
22   title: 'Map key ' + key,
23   details: 'Outcome for map key ' + key + ': ' + completionState + ' //Number of attempts used: ' + executionCount
24 });
25
26
27 return true;
28
29 });
30
31
32 // Create a log entry showing the total number of keys for which the map function executed successfully.
33
34 var mapKeysProcessed = 0;
35 summary.mapSummary.keys.iterator().each(function (key, executionCount, completionState){
36
37 if (completionState === 'COMPLETE'){
38   mapKeysProcessed++;
39 }
40
41 return true;
42
43 });
44
45 log.debug({
46   title: 'Map key statistics',
47   details: 'Total number of map keys processed successfully: ' + mapKeysProcessed
48 });

```

mapSummary.seconds

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | Total seconds elapsed when running <code>map(mapContext)</code> . |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 log.audit(' Time Elapsed', summary.mapSummary.seconds);
3 ...

```

mapSummary.usage

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Total number of usage units consumed when running <code>map(mapContext)</code> . |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 | ...
2 | log.audit(' Usage', summary.mapSummary.usage);
3 | ...

```

mapSummary.yields



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | Total number of times yields when running map(mapContext) . |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 | ...
2 | log.audit(' Total Yields', summary.mapSummary.yields);
3 | ...

```

mapSummary.errors



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | Holds all serialized errors thrown from the map(mapContext) function. May also hold the SSS_APP_SERVER_RESTART error code, which is recorded in the event of an application server restart. |
| Type | iterator |
| Since | 2015.2 |

Members

| Member | Type | Required/Optional | Description |
|---|----------|-------------------|-----------------------------------|
| iterator().each(parameters) | function | required | Executes one time for each error. |

parameters

| Member | Type | Required/Optional | Description |
|--|----------|-------------------|--|
| iteratorFunction(key, error, executionNo) See also functionParameters . | function | required | Provides logic to be executed during each iteration. |

functionParameters

| Parameter | Type | Required/Optional | Description |
|-------------|--------|-------------------|---|
| key | string | optional | Represents a key that was passed to the map stage for processing. |
| error | string | optional | The error thrown during processing of the corresponding key. |
| executionNo | number | optional | Indicates whether the error occurred during the first, second, third, or fourth attempt to process the key. |

Syntax

The following snippets show three ways you could use this iterator.

This code is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 // Add additional code
2 ...
3 // Create a log entry showing each full serialized error
4 summary.mapSummary.errors.iterator().each(
5   function (key, error, executionNo) {
6     log.error({
7       title: 'Map error for key: ' + key + ", execution no. " + executionNo,
8       details: error
9     });
10    return true;
11  }
12);
13
14 // Log only the name and description of each error thrown
15 summary.mapSummary.errors.iterator().each(
16   function (key, error, executionNo) {
17     var errorObject = JSON.parse(error);
18     log.error({
19       title: Map error for key: ' + key + ", execution no. " + executionNo,
20       details: errorObject.name + ': ' + errorObject.message
21     });
22    return true;
23  }
24);
25
26 // Calculate and log the total number of errors encountered during the map stage
27 var errorCodeCount = 0;
28 summary.mapSummary.errors.iterator().each(
29   function() {
30     errorCodeCount++;
31     return true;
32   }
33 );
34
35 log.audit({
36   title: 'Map stage errors',
37   details: 'Total number of errors: ' + errorCodeCount
38 });
39 ...
40 // Add additional code

```

summaryContext.reduceSummary



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|--|
| Object Description | Holds statistics regarding the reduce stage. |
|---------------------------|--|

| | |
|--------------|--------|
| Since | 2015.2 |
|--------------|--------|

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 summary.reduceSummary.errors.iterator().each(function (key, error)
3 {
4     log.error('Reduce Error for key: ' + key, error);
5     return true;
6 });
7 ...

```

reduceSummary.concurrency

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | The maximum concurrency number for executing parallel tasks during the reduce stage. |
| |  Note: This number may be less than the concurrency number allocated on the script deployment. For example, tasks that remained in the pending state are not reflected in this number. |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 log.audit(' Concurrency', summary.reduceSummary.concurrency);
3 ...

```

reduceSummary.dateCreated

 **Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | The time and day when reduce(reduceContext) began running. |
| Type | Date |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 log.audit(' Creation Date', summary.reduceSummary.dateCreated);

```

3 | ...

reduceSummary.keys**Note:** The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | Holds the keys that were passed to the reduce stage, either by the map stage, if it was used, or by the getInputData stage. These keys are listed in lexicographical order . |
| Type | iterator |
| Since | 2015.2 |

Members

| Member | Type | Required/Optional | Description |
|---|----------|-------------------|---------------------------------|
| iterator().each(parameters) | function | required | Executes one time for each key. |

parameters

| Member | Type | Required/Optional | Description |
|--|----------|-------------------|--|
| iteratorFunction(key, executionCount, completionState) | function | required | Provides logic to be executed during each iteration. |

See also [functionParameters](#).

functionParameters

| Parameter | Type | Required/Optional | Description |
|-----------------|--------|-------------------|--|
| key | string | optional | Represents a key that was passed to the reduce stage. |
| executionCount | number | optional | The number of times the reduce function was invoked for the key. |
| completionState | string | optional | A setting that indicates whether the map function was executed successfully for the key. Possible values are 'COMPLETE', 'FAILED', and 'PENDING'. The system uses 'PENDING' if the script exited the stage before trying to process the key. This behavior can occur when exitOnError is set to true. |

Syntax

The following snippets show a variety of ways you could use the reduceSummary.keys iterator. The code in the section does not comprise not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 //Add additional code.
2 ...
3
4 // Create a single log entry listing all of the keys that were passed to the reduce stage.
5

```

```

6  var reduceKeys = [];
7
8 summary.reduceSummary.keys.iterator().each(function (key){
9     reduceKeys.push(key);
10    return true;
11 });
12
13 log.debug({
14     title: 'Reduce stage keys',
15     details: reduceKeys
16 });
17
18
19 // Create a log entry for each key. The entry shows whether the reduce function was executed successfully for that key.
20
21 summary.reduceSummary.keys.iterator().each(function (key, executionCount, completionState){
22
23 log.debug({
24     title: 'Reduce key ' + key,
25     details: 'Outcome for reduce key ' + key + ':' + completionState + ' // Number of attempts used: ' + executionCount
26 });
27
28 return true;
29 });
30
31
32 // creates a single log entry showing the total number of keys for which the reduce function was invoked successfully.
33
34 var reduceKeysProcessed = 0;
35 summary.reduceSummary.keys.iterator().each(function (key, executionCount, completionState){
36
37 if (completionState === 'COMPLETE'){
38     reduceKeysProcessed++;
39 }
40
41 return true;
42 });
43
44 });
45
46 log.debug({
47     title: 'Reduce key statistics',
48     details: 'Total number of reduce keys processed successfully: ' + reduceKeysProcessed
49 });
50
51 ...
52 //Add additional code.

```

reduceSummary.seconds



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|---|
| Property Description | Total seconds elapsed when running <code>reduce(reduceContext)</code> . |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 log.audit(' Time Elapsed', summary.reduceSummary.seconds);
3 ...

```

reduceSummary.usage

Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Total number of usage units consumed when running <code>reduce(reduceContext)</code> . |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 log.audit(' Usage', summary.mapSummary.usage);
3 ...

```

reduceSummary.yields

Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Total number of times yields when running <code>reduce(reduceContext)</code> . |
| Type | number |
| Since | 2015.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [Map/Reduce Script Samples](#).

```

1 ...
2 log.audit(' Total Yields', summary.reduceSummary.yields);
3 ...

```

reduceSummary.errors

Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Holds all serialized errors thrown from the <code>reduce(reduceContext)</code> function. May also hold the SSS_APP_SERVER_RESTART error code, which is recorded in the event of an application server restart. |
| Type | iterator |
| Since | 2015.2 |

Members

| Member | Type | Required/Optional | Description |
|--|----------|-------------------|-----------------------------------|
| <code>iterator().each(parameters)</code> | function | required | Executes one time for each error. |

parameters

| Member | Type | Required/Optional | Description |
|---|----------|-------------------|--|
| iteratorFunction(key, error, executionNo) | function | required | Provides logic to be executed during each iteration. |
| See also functionParameters . | | | |

functionParameters

| Parameter | Type | Required/Optional | Description |
|-------------|--------|-------------------|---|
| key | string | optional | Represents a key that was passed to the reduce stage for processing. |
| error | string | optional | A serialization of the error thrown. |
| executionNo | number | optional | Indicates whether the error occurred during the first, second, third, or fourth attempt to process the key. |

Syntax

The following snippets shows three ways you could use this iterator.

This code is not a functional example. For a complete script sample, see [Map/Reduce Script Samples](#).

```

1 //Add additional code
2 ...
3
4 // Create a log entry showing each full serialized error.
5
6 summary.reduceSummary.errors.iterator().each(function (key, error, executionNo){
7     log.error({
8         title: 'Reduce error for key: ' + key + ', execution no. ' + executionNo,
9         details: error
10    });
11    return true;
12 });
13
14
15 // Log only the name and description of each error thrown.
16
17 summary.reduceSummary.errors.iterator().each(function (key, error){
18     var errorObject = JSON.parse(error);
19     log.error({
20         title: 'Reduce error for key: ' + key + ', execution no. ' + executionNo,
21         details: errorObject.name + ': ' + errorObject.message
22    });
23    return true;
24 });
25
26
27 // Calculate and log the total number of errors encountered during the reduce stage.
28
29 var errorCount = 0;
30 summary.reduceSummary.errors.iterator().each(function() {
31     errorCount++;
32     return true;
33 });
34
35 log.audit ({
36     title: 'Reduce stage errors',
37     details: 'Total number of errors: ' + errorCount
38 });

```

summaryContext.output



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | Iterator that provides keys and values that are saved as output during the reduce stage. Note that keys are listed in lexicographical order . |
| Type | iterator |
| Since | 2015.2 |

Members

| Member | Type | Required/Optional | Description |
|---|----------|-------------------|---|
| iterator().each(parameters) | function | required | Executes one time for each keyvalue pair. |

parameters

| Member | Type | Required/Optional | Description |
|--|----------|-------------------|--|
| iteratorFunction(key, value). See the help topic functionParameters . | function | required | Provides logic to be executed during each iteration. |

functionParameters

| Parameter | Type | Required/Optional | Description |
|-----------|--------|-------------------|--|
| key | string | optional | The key saved at the end of the reduce stage by using the reduceContext.write(options) method. |
| value | string | optional | The value saved at the end of the reduce stage by using the reduceContext.write(options) method. |

Syntax

```

1 // Add additional code
2 ...
3
4 // Create a variable to track how many key/value pairs were written.
5 var totalRecordsUpdated = 0;
6
7
8 // If the number of key/value pairs is expected to be manageable, log
9 // each one.
10
11 summary.output.iterator().each(function (key, value){
12     log.debug({
13         title: 'summary.output.iterator',
14         details: 'key: ' + key + ' / value: ' + value
15     });
16     totalRecordsUpdated++;
17     return true;
18 });
19
20 // Create a log entry showing the number of
21
22 log.debug({

```

```

23 |     title: 'Total records updated',
24 |     details: totalRecordsUpdated
25 |   });
26 |   ...
27 | // Add additional code.

```

SuiteScript 2.x Mass Update Script Type

Mass update scripts allow you to programmatically perform custom updates to fields that are not available through general mass updates. Mass update scripts can run complex calculations, as defined in your script, across records. Mass update scripts are started on demand, and you cannot prioritize them or schedule them to run at specific times. If you want your script to run at a specific time, consider using a scheduled script or map/reduce script instead of a mass update script. For more information, see [SuiteScript 2.x Scheduled Script Type](#) and [SuiteScript 2.x Map/Reduce Script Type](#).

Mass update scripts are executed on the server when you click the Perform Update button on the Mass Update Preview Results page. You cannot invoke a mass update script from another script type. You also cannot programmatically check the status of a mass update script or determine when the script started or ended.



Important: The following items should be considered when running a mass update script:

- Updates made to records during a custom mass update can trigger user event scripts if there are user event scripts associated with the records being updated.
- Mass update script deployments and mass updates can both be assigned an audience. It is the script owner's responsibility to ensure the two audiences are in sync. If the two audiences do not match, the mass update script will not run when users click the Perform Update button on the Mass Update page.
- When users run custom mass updates, they must have the appropriate permission (Edit/Full) for the record type(s) they are updating.
- Users must also have SuiteScript enabled in their accounts. (Administrators can go to Setup > Company > Setup Tasks > Enable Features (Administrator). In the SuiteCloud tab, click the Server SuiteScript check box and the Client SuiteScript check box.)

For information about scripting with mass update scripts, see [SuiteScript 2.x Mass Update Script Entry Points](#).

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

You can use SuiteScript Analysis to learn about when the script was installed and how it performed in the past. For more information, see the help topic [Analyzing Scripts](#).

Mass Update Script Sample

The following code updates the probability field of all existing records to 61%.



Note: From the script deployment record, ensure that the **Applies To** field is set to **Opportunity**.

For help with writing scripts in SuiteScript 2.x, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType MassUpdateScript
4   */
5 define(['N/record'],
6       (record) => {
7         function each(params) {
8           // Set the probability to 61%
9           let recOpportunity = record.load({
10             type: params.type,
11             id: params.id
12           });
13           recOpportunity.setValue('probability', 61);
14           recOpportunity.save();
15         }
16         return {
17           each: each
18         };
19     });

```

SuiteScript 2.x Mass Update Script Entry Points

| Script Entry Point | |
|--------------------|---|
| each | <p>Defines the mass update script trigger point.</p> <p>Iterates through all applicable records so that you can apply logic to each record. See each for parameters and syntax.</p> |

each



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|--|
| Description | Iterates through each applicable record. |
| Returns | void |
| Since | Version 2016 Release 1 |

Parameters



Note: The `params` parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|--------------------------|--------|---|------------------------|
| <code>params.id</code> | number | The ID of the record being processed by the mass update. | Version 2016 Release 1 |
| <code>params.type</code> | string | The record type of the record being processed by the mass update. | Version 2016 Release 1 |

Syntax



Important: The following code snippet shows the syntax for this entry point. It is not a functional example. For a full script sample, see [Mass Update Script Sample](#).

```

1 // Add additional code
2 ...
3 function each(params) {
4     // Set the probability to 61%
5     let recOpportunity = record.load({
6         type: params.type,
7         id: params.id
8     });
9 ...
10 // Add additional code

```

SuiteScript 2.x Portlet Script Type

Portlet scripts are run on the server and are rendered in the NetSuite dashboard.

The following portlet script types are supported:

- **Simple Form** — A data entry form with up to one submit button embedded into a portlet. This type supports the Portlet module that can refresh and resize the portlet, as well as the use of record-level client script to implement validation. See the help topic [N/portlet Module](#).
- **Inline HTML** — An HTML-based portlet that is used to display free-form HTML. (images, Flash, custom HTML)
- **Links and Indents**— A portlet that consists of rows of formatted content.
- **Simple List**— A standard list of user-defined column headers and rows.

The following image shows a form portlet and a links portlet displayed on the NetSuite dashboard.



You can designate that a portlet script should be used for a SuiteApp portlet. A SuiteApp portlet is a specialized type of custom portlet. It provides direct access from users' dashboards to a SuiteApp installed in their account. This type of script is called a Dashboard SuiteApp portlet script. If a Dashboard SuiteApp portlet is included in a SuiteApp, a user can add a SuiteApp portlet to their dashboard from the Personalize Dashboard menu. For instructions, see the help topic [SuiteApp Portlets](#). This type of portlet is supported for installed SuiteApps that include a dashboard component. A SuiteApp portlet script not only provides content for SuiteApp portlets. It also enables you to include your choice of graphics as branding for the icons that are shown for SuiteApp portlets in the Personalize Dashboard window.

To view content produced by a portlet script that is not intended for a SuiteApp portlet, a user must add a custom portlet to their dashboard and select the script in the portlet setup. For more information, see the help topics [Custom Portlets](#) and [Adding a Portlet to a Dashboard](#).

For steps to create and deploy a portlet script, see [Creating and Deploying a Portlet Script](#).

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

You can use SuiteScript Analysis to learn about when the script was installed and how it performed in the past. For more information, see the help topic [Analyzing Scripts](#).

See the following for more information about the Portlet Script type:

- [SuiteScript 2.x Portlet Script Reference](#)
 - [Creating and Deploying a Portlet Script](#)
 - [Guidelines for Creating a Dashboard SuiteApp Icon](#)
- [SuiteScript 2.x Portlet Script Entry Points and API](#)
- [render\(params\)](#)
- [Portlet Object](#)
 - [Portlet.addColumn\(options\)](#)
 - [Portlet.addEditColumn\(options\)](#)
 - [Portlet.addField\(options\)](#)
 - [Portlet.addLine\(options\)](#)
 - [Portlet.addRow\(options\)](#)
 - [Portlet.addRows\(options\)](#)
 - [Portlet.setSubmitButton\(options\)](#)
 - [Portlet.clientScriptFileDialog](#)
 - [Portlet.clientScriptModulePath](#)
 - [Portlet.html](#)
 - [Portlet.title](#)

SuiteScript 2.x Portlet Script Samples

- [Simple Form Portlet Script Sample](#)
- [Inline HTML Portlet Script Sample](#)
- [Links and Indents Portlet Script Sample](#)
- [Simple List Portlet Script Sample](#)

Simple Form Portlet Script Sample

For help with writing scripts in SuiteScript 2.x, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Portlet

```

```

4  /*
5
6 // This sample creates a portlet that includes a simple form with a text field and a submit button
7 define([], function() {
8     function render(params) {
9         var portlet = params.portlet;
10        portlet.title = 'Simple Form Portlet';
11        var fld = portlet.addField({
12            id: 'text',
13            type: 'text',
14            label: 'Text'
15        });
16        fld.updateLayoutType({
17            layoutType: 'normal'
18        });
19        fld.updateBreakType({
20            breakType: 'startcol'
21        });
22        portlet.setSubmitButton({
23            url: 'http://httpbin.org/post',
24            label: 'Submit',
25            target: '_top'
26        });
27    }
28
29    return {
30        render: render
31    };
32 });

```

Inline HTML Portlet Script Sample

```

1 /**
2 * @NApiVersion 2.x
3 * @NScriptType Portlet
4 */
5
6 // This sample creates a portlet that displays simple HTML
7 define([], function() {
8     function render(params) {
9         params.portlet.title = 'My Portlet';
10        var content = '<td><span><b>Hello!!!</b></span></td>';
11        params.portlet.html = content;
12    }
13
14    return {
15        render: render
16    };
17 });

```

Links and Indents Portlet Script Sample

```

1 /**
2 * @NApiVersion 2.x
3 * @NScriptType Portlet
4 */
5
6 // This sample creates a portlet with two links
7 define([], function() {
8     function render(params) {
9         var portlet = params.portlet;
10        portlet.title = 'Search Engines';
11        portlet.addLine({
12            text: 'NetSuite',
13            url: 'http://www.netsuite.com/'
14        });
15        portlet.addLine({
16            text: 'Oracle',

```

```

17     url: 'http://www.oracle.com/'
18   });
19 }
20
21 return {
22   render: render
23 };
24 });

```

Simple List Portlet Script Sample

⚠ Important: This sample references a custom entity field with the ID custentity_multiselect. Before attempting to use this sample, you must either create a field with that ID or edit the sample so that it references an existing custom entity field in your account.

```

1 /**
2 * @NApiVersion 2.x
3 * @NScriptType Portlet
4 */
5
6 // This sample creates a portlet with a simple list
7 define(['N/search'], function(search) {
8   function render(params) {
9     var isDetail = (params.column === 2);
10    var portlet = params.portlet;
11    portlet.title = isDetail ? "My Detailed List" : "My List";
12    portlet.addColumn({
13      id: 'internalid',
14      type: 'text',
15      label: 'Number',
16      align: 'LEFT'
17    });
18    portlet.addColumn({
19      id: 'entity',
20      type: 'text',
21      label: 'ID',
22      align: 'LEFT'
23    });
24    if (isDetail) {
25      portlet.addColumn({
26        id: 'email',
27        type: 'text',
28        label: 'E-mail',
29        align: 'LEFT'
30      });
31      portlet.addColumn({
32        id: 'custentity_multiselect',
33        type: 'text',
34        label: 'Multiselect',
35        align: 'LEFT'
36      });
37    }
38    var filter = search.createFilter({
39      name: 'email',
40      operator: search.Operator.ISNOTEMPTY
41    });
42    var customerSearch = search.create({
43      type: 'customer',
44      filters: filter,
45      columns: ['internalid', 'entity', 'email', 'custentity_multiselect']
46    });
47    var count = isDetail ? 15 : 5;
48    customerSearch.run().each(function(result) {
49      portlet.addRow(result.getAllValues());
50      return --count > 0;
51    });
52  }
53
54  return {

```

```

55     render: render
56   };
57 });

```

SuiteScript 2.x Portlet Script Reference

- [Creating and Deploying a Portlet Script](#)
- [Guidelines for Creating a Dashboard SuiteApp Icon](#)

Creating and Deploying a Portlet Script

i Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following is a basic process flow for creating and deploying a portlet script, including a Dashboard SuiteApp portlet script:

1. Create a portlet script and upload the file to your File Cabinet.
For more information about this process, see [SuiteScript 2.x Script Creation Process](#).
2. Create a script record for your portlet script, and ensure that the **Portlet Type** field is set to the correct portlet type used in your script.
For more information about this process, see [Creating a Script Record](#).
3. Create a script deployment record for your portlet script.

If you are creating a script to be used for a SuiteApp portlet, enable the **Dashboard SuiteApp** option.

When this option is enabled, you can upload an image to the File Cabinet. This image represents the SuiteApp icon shown for the portlet in the Personalize Dashboard window. Note that only SVG images are supported for the icon. SVG is a vector format, so it assures perfect image scalability. For more information about icon guidelines, see the help topic [Guidelines for Creating a Dashboard SuiteApp Icon](#).

The following image indicates where the Dashboard fields are located:

The screenshot shows the 'Script Deployment' dialog box. At the top are three buttons: 'Save' (blue), 'Cancel', and 'Reset'. Below these are sections for 'SCRIPT' and 'TITLE *'. The 'TITLE' field contains 'Financial Report Systems Form Portlet'. Under 'ID', there is an empty input field. A checked checkbox labeled 'DEPLOYED' is followed by another checkbox labeled 'DASHBOARD SUITEAPP', which is also checked and has a red border around it. Below this is a section for 'ICON' with a dropdown menu showing 'NetSuite Logo'.

For more information about the script deployment process, see [Methods of Deploying a Script](#).

Guidelines for Creating a Dashboard SuiteApp Icon

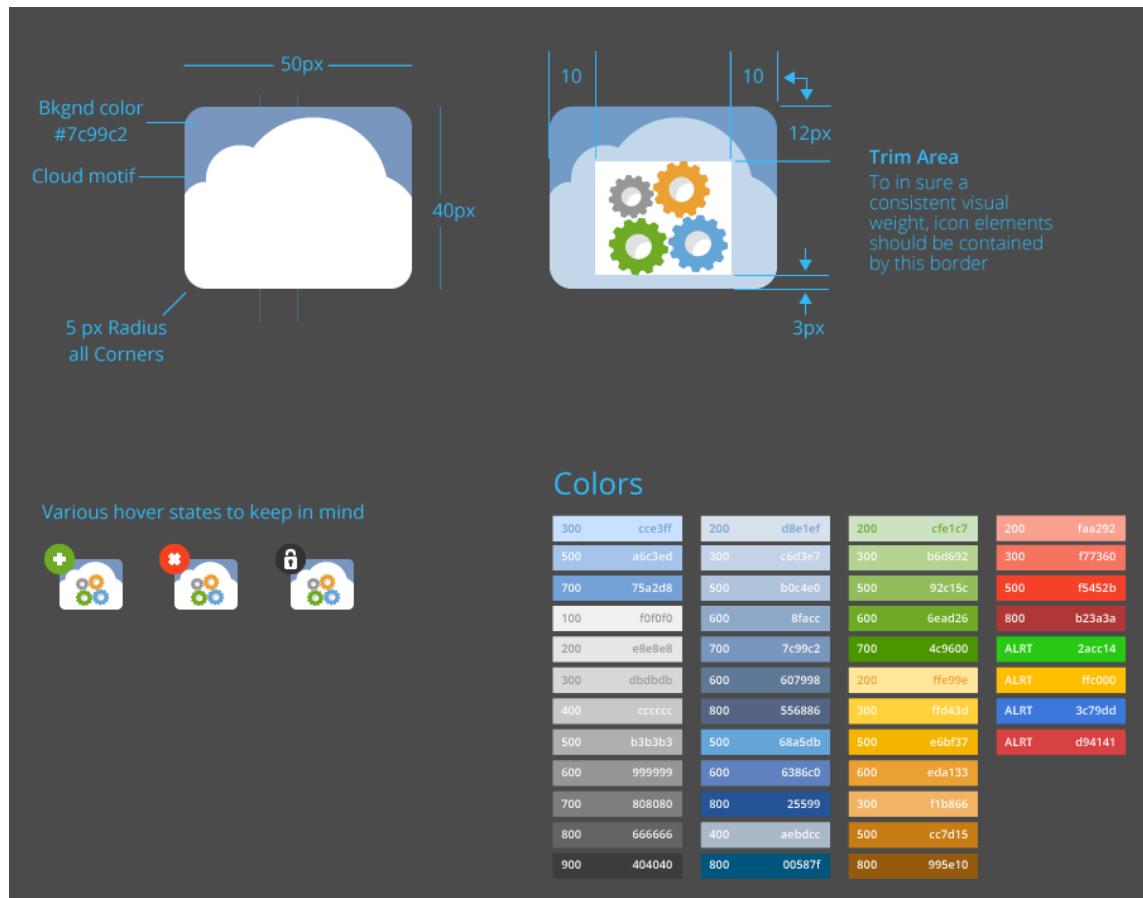


Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Dashboard SuiteApp icons require certain visual characteristics to align them with other icons in the dashboard. The following guidelines preserve the visual characteristics of Dashboard SuiteApp icons:

- Dashboard SuiteApp icons use a unique background with a cloud motif. This background ensures a common size and shape for all Dashboard SuiteApp icons and provides a cloud motif that serves as an additional modifier.
- NetSuite recommends using a restricted color palette. This palette is a major contributor to the character of the icons used in NetSuite. Restrict your colors to this palette whenever possible. If additional colors are required, add them as special exceptions. See the image below for more information about the recommended color palette.
- Graphical elements should be more geometrical than illustrative. Avoid using complex and irregular shapes. Try to reduce elements in your composition to their most primitive geometry.
- Dashboard SuiteApp icons use a simulated light source to increase detail and definition in icon elements. This light source is evidenced by a cast shadow that proceeds down and to the right of elements in the composition at a 45 degree angle. Cast shadows are rendered with a black fill set to 10% opacity.
- Your icon image must be saved as an SVG file.

The following image shows the guidelines in use:



SuiteScript 2.x Portlet Script Entry Points and API

- [render\(params\)](#)
- [Portlet Object](#)
 - [Portlet.addColumn\(options\)](#)
 - [Portlet.addEditColumn\(options\)](#)
 - [Portlet.addField\(options\)](#)
 - [Portlet.addLine\(options\)](#)
 - [Portlet.addRow\(options\)](#)
 - [Portlet.addRows\(options\)](#)
 - [Portlet.setSubmitButton\(options\)](#)
 - [Portlet.clientScriptFileDialog](#)
 - [Portlet.clientScriptModulePath](#)
 - [Portlet.html](#)
 - [Portlet.title](#)

render(params)

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines the portlet script trigger point. |
| Returns | <code>void</code> |
| Since | Version 2015 Release 2 |

Parameters

 **Note:** The `params` parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite. The values for `params` are read-only.

| Parameter | Type | Description | Since |
|-----------------------------|---------------------------------|--|------------------------|
| <code>params.portlet</code> | Portlet Object | <p>The Portlet object used for rendering.</p> <p>The availability of Portlet members depends on the type of Portlet that is passed in. For more information, see the following:</p> <ul style="list-style-type: none"> ■ Simple Form Portlet Object Members ■ Inline HTML Portlet Object Members ■ Links and Indents Portlet Object Members ■ Simple List Portlet Object Members | Version 2015 Release 2 |
| <code>params.column</code> | <code>string</code> (read-only) | <p>The column index for the portlet on the dashboard.</p> <p>The index is a string representation of one of the following numeric values:</p> <ol style="list-style-type: none"> 1. left column | Version 2015 Release 2 |

| Parameter | Type | Description | Since |
|---------------|--------------------|--|---------------------------|
| | | 2. center column 3. right column | |
| params.entity | string (read-only) | The customer ID for the selected customer. | Version 2015 Release 2 |

Portlet Object



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Portlet objects are used to encapsulate scriptable dashboard portlets. They are automatically passed to the render(params) entry point by NetSuite. For more information, see [render\(params\)](#).

Simple Form Portlet Object Members

These members are only available to Form Portlet objects.

| Member Type | Name | Return Type / Value Type | Description |
|-------------|----------------------------------|--------------------------|--|
| Method | Portlet.addField(options) | serverWidget.Field | Adds a field to the form. |
| | Portlet.setSubmitButton(options) | serverWidget.Button | Adds a submit button to the form. |
| Property | Portlet.clientScriptFileDialog | number or string | The script file ID to be used in the portlet. Can be in either numerical or string format. |
| | Portlet.clientScriptModulePath | string | The script path to be used in the portlet. |
| | Portlet.title | string | The title of the portlet. |

Inline HTML Portlet Object Members

These members are only available to HTML Portlet objects.

| Member Type | Name | Return Type / Value Type | Description |
|-------------|---------------|--------------------------|--|
| Property | Portlet.html | string | The complete HTML contents of the portlet. |
| | Portlet.title | string | The title of the portlet. |

Links and Indents Portlet Object Members

These members are only available to Links Portlet objects.

| Member Type | Name | Return Type / Value Type | Description |
|-------------|--------------------------|--------------------------|-----------------------------|
| Method | Portlet.addLine(options) | Object | Adds a line to the portlet. |
| Property | Portlet.title | string | The title of the portlet. |

Simple List Portlet Object Members

These members are only available to List Portlet objects.

| Member Type | Name | Return Type / Value Type | Description |
|-------------|--------------------------------|--------------------------|--|
| Method | Portlet.addColumn(options) | serverWidget.ListColumn | Adds a column to the portlet. |
| | Portlet.addEditColumn(options) | serverWidget.ListColumn | Adds an Edit or Edit/View column to the portlet. |
| | Portlet.addRow(options) | Object | Adds a row to the portlet. |
| | Portlet.addRows(options) | Object | Adds multiple rows to the portlet. |
| Property | Portlet.title | string | The title of the portlet. |

Portlet.addColumn(options)

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|------------------------------------|
| Method Description | Adds a list column to the portlet. |
| Returns | serverWidget.ListColumn |
| Entry Point | render(params) |
| Since | 2016.2 |

Parameters

 **Note:** The options parameter is a JavaScript object.

| Parameter | Type | Required / Optional | Description | Since |
|---------------|--------|---------------------|--|--------|
| options.id | string | required | The internal ID of this column. The internal ID must be in lowercase, contain no spaces. | 2016.2 |
| options.label | string | required | The label of this column. | 2016.2 |
| options.type | string | required | The field type for this column. For more information about possible values, see the help topic serverWidget.FieldType . | 2016.2 |
| options.align | string | optional | The layout justification for this column. For more information about possible values, see the help topic serverWidget.LayoutJustification . | 2016.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [SuiteScript 2.x Portlet Script Type](#).

```

1 ...
2 var newColumn = params.portlet.addColumn({
3   id: 'column1',
4   label: 'Text',
5   type: serverWidget.FieldType.TEXT,
6   align: serverWidget.LayoutJustification.RIGHT
7 });
8 ...

```

Portlet.addColumn(options)

i Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|--|
| Method Description | Adds an Edit or Edit/View column to the portlet. |
| Returns | serverWidget.ListColumn |
| Entry Point | render(params) |
| Since | 2016.2 |

Parameters

i Note: The options parameter is a JavaScript object.

| Parameter | Type | Required / Optional | Description | Since |
|---|---------|---------------------|--|--------|
| options.column | string | required | The internal ID of the column to the left of which the Edit/View column is added. | 2016.2 |
| options.showHrefCol | string | optional | If set, it must contain a name of a column. The value of the column determines whether the View/Edit link is clickable for a given data row (T=clickable, F=non-clickable). | 2016.2 |
| options.showView | boolean | optional | If true, then an Edit/View column is added. Otherwise, only an Edit column is added. The default setting is false. | 2016.2 |
| options.link | string | optional | The Edit/View base link. (For example: /app/common/entity/employee.nl) The complete link is formed like this: <link>?<linkParamName>=<row data from linkParam>. (For example: /app/common/entity/employee.nl?id=123) | 2019.2 |
| options.linkParam | string | optional | The internal ID of the field in the row data where to take the parameter from. The default value is the value set in the options.column parameter. | 2019.2 |
| ✓ Tip: In most cases, the value to use here is internalid | | | | |
| options.linkParamName | string | optional | The name of the parameter. The default value is id. | 2019.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [SuiteScript 2.x Portlet Script Type](#).

```

1 ...
2 var portlet = params.portlet;
3 portlet.title = 'My Detailed List';
4 portlet.addColumn({
5     id: 'internalid',
6     type: serverWidget.FieldType.TEXT,
7     label: 'Number',
8     align: serverWidget.LayoutJustification.LEFT
9 });
10 portlet.addColumn({
11     id: 'entityid',
12     type: serverWidget.FieldType.TEXT,
13     label: 'Name',
14     align: serverWidget.LayoutJustification.LEFT
15 });
16 portlet.addEditColumn({
17     column: 'entityid',
18     showView: true,
19     showHrefCol: true,
20     link: '/app/common/entity/custjob.nl',
21     linkParam: 'internalid',
22     linkParamName: 'id',
23 });
24 ...

```

Portlet.addField(options)

i Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|------------------------------------|
| Method Description | Adds a field to the form. |
| Returns | serverWidget.Field |
| Entry Point | render(params) |
| Since | 2016.2 |

Parameters

i Note: The options parameter is a JavaScript object.

| Parameter | Type | Required / Optional | Description | Since |
|---------------|--------|---------------------|--|--------|
| options.id | string | required | The internal ID of the field. The internal ID must be in lowercase, contain no spaces, and include the prefix custpage if you are adding the field to an existing page. For example, if you add a field that appears as Purchase Details , the field internal ID should be something similar to custpage_purchasedetails or custpage_purchase_details. | 2016.2 |
| options.label | string | required | The label for this field. | 2016.2 |
| options.type | string | required | The field type for the field. | 2016.2 |

| Parameter | Type | Required / Optional | Description | Since |
|----------------|--------|---------------------|---|--------|
| | | | For more information about possible values, see the help topic serverWidget.FieldType . | |
| options.source | string | optional | <p>The internal ID or script ID of the source list for this field if it is a select (List/Record) or multi-select field.</p> <p>Note: For radio fields only, the source parameter must contain the internal ID for the field.</p> <p>Important: After you create a select or multi-select field that is sourced from a record or list, you cannot add additional values with <code>Field.addSelectOption(options)</code>. The select values are determined by the source record or list.</p> | 2016.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [SuiteScript 2.x Portlet Script Type](#).

```

1 ...
2 var newField = params.portlet.addField({
3   id: 'textfield',
4   type: serverWidget.FieldType.TEXT,
5   label: 'text'
6 });
7 ...

```

Portlet.addLine(options)

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|-----------------------------|
| Method Description | Adds a line to the portlet. |
| Returns | Object |
| Entry Point | <code>render(params)</code> |
| Since | 2016.2 |

Parameters

Note: The options parameter is a JavaScript object.

| Parameter | Type | Required / Optional | Description | Since |
|--------------|--------|---------------------|------------------------|--------|
| options.text | string | required | The text for the line. | 2016.2 |
| options.url | string | optional | The URL link. | 2016.2 |

| Parameter | Type | Required / Optional | Description | Since |
|---------------|--------|---------------------|--|--------|
| options.align | number | optional | The justification for the line. This value indicates the number of spaces to indent. The align value must be between 0 and 5. | 2016.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [SuiteScript 2.x Portlet Script Type](#).

```

1 ...
2     params.portlet.addLine({
3         text: 'NetSuite',
4         url: 'http://www.netsuite.com',
5         align: 4
6     );
7 ...

```

Portlet.addRow(options)

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|----------------------------|
| Method Description | Adds a row to the portlet. |
| Returns | Object |
| Entry Point | render(params) |
| Since | 2016.2 |

Parameters

Note: The options parameter is a JavaScript object.

| Parameter | Type | Required / Optional | Description | Since |
|-------------|------------------------|---------------------|---|--------|
| options.row | search.Result object | required | A row that consists of either a search.Result, or name/value pairs. Each pair should contain the value for the corresponding Column object in the list. | 2016.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [SuiteScript 2.x Portlet Script Type](#).

```

1 ...
2     params.portlet.addRow({
3         row: {
4             columnid1: 'value1',
5             columnid2: 'value2'
6         }
7     );

```

```

7 | });
8 | ...

```

Portlet.addRows(options)



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|------------------------------------|
| Method Description | Adds multiple rows to the portlet. |
| Returns | Object |
| Entry Point | render(params) |
| Since | 2016.2 |

Parameters



Note: The options parameter is a JavaScript object.

| Parameter | Type | Required / Optional | Description | Since |
|--------------|--|---------------------|---|--------|
| options.rows | search.Result[] object[] | required | An array of rows that consist of either a search.Result array, or an array of name/value pairs. Each pair should contain the value for the corresponding Column object in the list. | 2016.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [SuiteScript 2.x Portlet Script Type](#).

```

1 | ...
2 | params.portlet.addRows({
3 |   ROWS:
4 |   [
5 |     [
6 |       {
7 |         columnid1: 'value1',
8 |         columnid2: 'value2'
9 |       },
10 |       {
11 |         columnid1: 'value2',
12 |         columnid2: 'value3'
13 |       }
14 |     ]
15 |   }
16 | );
17 | ...

```

Portlet.setSubmitButton(options)



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|---------------------------|-------------------------------------|
| Method Description | Adds a submit button to the form. |
| Returns | serverWidget.Button |
| Entry Point | render(params) |

| | |
|--------------|--------|
| Since | 2016.2 |
|--------------|--------|

Parameters

| |
|--|
| Note: The options parameter is a JavaScript object. |
|--|

| Parameter | Type | Required / Optional | Description | Since |
|----------------|--------|---------------------|---|--------|
| options.url | string | required | The URL that the form posts data to. | 2016.2 |
| options.label | string | optional | The button label. | 2016.2 |
| options.target | string | optional | <p>The target attribute of the form element, if it is different from the portlet's own embedded iframe. Supported values include standard HTML target attributes such as _top, _parent, and _blank. It can also be set to frame names and the NetSuite-specific identifier _hidden.</p> <p>Setting this value to _hidden allows submission to a backend that returns results to a hidden child iframe within the portlet's embedded iframe.</p> | 2016.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [SuiteScript 2.x Portlet Script Type](#).

```

1 ...
2 params.portlet.setSubmitButton({
3   url: 'http://httpbin.org/post',
4   label: 'Submit',
5   target: '_top'
6 });
7 ...

```

Portlet.clientScriptFileDialog

| |
|---|
| Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1. |
|---|

| | |
|-----------------------------|---|
| Property Description | The script file ID to be used in the portlet. |
| Type | number or string |
| Entry Point | <code>render(params)</code> |
| Since | 2016.2 |

Errors

| Error Code | Thrown If |
|-------------------------|--|
| PROPERTY_VALUE_CONFLICT | You attempted to set this value when the <code>Portlet.clientScriptModulePath</code> property value has already been specified. For more information, see Portlet.clientScriptFileDialog . |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [SuiteScript 2.x Portlet Script Type](#).

```
1 | ...
2 | params.portlet.clientScriptModulePath = 32;
3 | ...
```

Portlet.clientScriptModulePath

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | The script path to be used in the portlet. |
| Type | string |
| Entry Point | render(params) |
| Since | 2016.2 |

Errors

| Error Code | Thrown If |
|-------------------------|---|
| PROPERTY_VALUE_CONFLICT | You attempted to set this value when the Portlet.clientScriptModulePath property value has already been specified. For more information, see Portlet.clientScriptModulePath . |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [SuiteScript 2.x Portlet Script Type](#).

```
1 | ...
2 | params.portlet.clientScriptModulePath = '/SuiteScripts/clientScript.js';
3 | ...
```

Portlet.html

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|--|
| Property Description | The complete HTML contents of the portlet. |
| Type | string |
| Entry Point | render(params) |
| Since | 2016.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [SuiteScript 2.x Portlet Script Type](#).

```
1 | ...
```

```

2 | params.portlet.html = htmlcontents;
3 | ...

```

Portlet.title



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-----------------------------|-----------------------------|
| Property Description | The title of the portlet. |
| Type | string |
| Entry Point | <code>render(params)</code> |
| Since | 2016.2 |

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [SuiteScript 2.x Portlet Script Type](#).

```

1 | ...
2 | params.portlet.title = 'My Portlet';
3 | ...

```

SuiteScript 2.x RESTlet Script Type

A RESTlet is a SuiteScript that you make available for other applications to call, either from an external application or from within NetSuite. When an application or another script calls a RESTlet, the RESTlet script executes and, in some cases, returns a value to the calling application.

RESTlets can be useful when you want to bring data from another system into NetSuite, or if you want to extract data from NetSuite. RESTlets can also be used, in combination with other scripts, to customize the behavior of a page within NetSuite.

For more details, see the following sections:

- [SuiteScript 2.x Getting Started with RESTlets](#)
- [RESTlet Authentication](#)
- [SuiteScript 2.x RESTlet Reference](#)
- [SuiteScript 2.x RESTlet Script and Request Samples](#)
- [SuiteScript 2.x RESTlet Script Entry Points](#)

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

You can use SuiteScript Analysis to learn about when the script was installed and how it performed in the past. For more information, see the help topic [Analyzing Scripts](#).

SuiteScript 2.x Getting Started with RESTlets

For help getting started with RESTlets, see the following topics:

- RESTlet Key Concepts
- Deploying a RESTlet
- Identifying a RESTlet in a Call
- Selecting an HTTP Method for Calling a RESTlet
- Creating a Content-Type Header

RESTlet Key Concepts



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

A RESTlet is a SuiteScript that executes when called by an external application or by another SuiteScript. Depending on how the RESTlet is written and called, it may also return data to the calling application.

As with other script types, RESTlets have broad potential applications. A RESTlet can perform any function that can be implemented by using SuiteScript. But at a high level, potential uses of RESTlets generally include the following:

- Retrieving, adding, or manipulating data within NetSuite, from an external source. In this sense, RESTlets can be seen as an alternative to NetSuite's SOAP-based web services.
- Customizing the behavior of pages and features within NetSuite. In this sense, RESTlets can be seen as an alternative to other script types, such as Suitelets. The advantage of using a RESTlet compared with a Suitelet is that the RESTlet can return data, in plain text or JSON, to the client script.

To use a RESTlet, you follow the same guidelines as you would with an entry point script deployed at the record level. For example, you must create a script record and a deployment record based on the RESTlet script file. These processes are described further in [Deploying a RESTlet](#).

When you save a script deployment record for a RESTlet, the system automatically generates a URL that can be used to call the RESTlet. Because a RESTlet executes only when it is called, this information is critical for using the RESTlet. For more details, see [Identifying a RESTlet in a Call](#).

When you are ready to call a RESTlet that you have deployed, you can use one of four supported HTTP methods: delete, get, post, or put. Depending on which method you use, you may be required to embed input for the RESTlet in the URL, or you may be required to submit arguments in a request body. Additionally, for the call to be successful, your RESTlet script must contain an entry point that corresponds with the method you use to make the call. For details on supported HTTP methods and formatting your request, see [Selecting an HTTP Method for Calling a RESTlet](#).

One advantage of RESTlets over other script types is that NetSuite requires authentication for RESTlets. If a RESTlet call originates from a client that does not have an existing session in the NetSuite account where the RESTlet is deployed, NetSuite requires the call to include an authorization header. For details about adding an authentication header to a RESTlet, see the help topic [RESTlet Authentication](#).

For most RESTlet calls, you must also include a content-type header, which tells NetSuite how your request body will be formatted and how NetSuite should format its response. For details, see [Creating a Content-Type Header](#).

Deploying a RESTlet



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Before you can use a RESTlet, you must follow the same guidelines as you would with most entry point scripts. At a high level, you must do the following:

- [Make Sure the Script Is Formatted Properly](#)
- [Create a Script and Script Deployment Record](#)

Make Sure the Script Is Formatted Properly

All of the following must be true:

- The script must have the correct structure for SuiteScript 2.x. For example, the script must have an interface that includes at least one entry point appropriate for the RESTlet script type. The script must also contain a corresponding entry point function. For details on how to structure a SuiteScript 2.x script, see [Correct Structure for Entry Point Scripts](#). Note that the entry points you use will determine how the RESTlet can be called. For details on the RESTlet entry points, see [SuiteScript 2.x RESTlet Script Entry Points](#).
- The script must use the required JSDoc tags. The @NScriptType must be RESTlet (or Restlet; these values are not case-sensitive). For further details on the required JSDoc tags, see [Required JSDoc Tags for Entry Point Scripts](#).

Create a Script and Script Deployment Record

Before you can use a RESTlet, you must upload it to your File Cabinet. Then you use the file to create a script record and a script deployment record.

For general help creating script records and script deployment records, see [SuiteScript 2.x Record-Level Scripts](#).

When creating these records for a RESTlet, be aware of the following:

- NetSuite recommends that you enter meaningful data in the script record's ID field and the script deployment record's ID field. When you save the records, the system creates IDs that include the text you entered. One possible use of these IDs is to identify the RESTlet when calling it from another SuiteScript. For this reason, it may be helpful to have created meaningful ID strings.
- Unlike some other script types, you do not deploy a RESTlet for any particular record type. Each RESTlet is available independently of any particular record type or record instance.
- The script deployment record includes a field called Status, which has possible values of Released and Testing. Before you can call the RESTlet from an external source, the Status field must be set to Released.
- When you save a script deployment record for a RESTlet, the system automatically generates a partial and full URL that you can use to call the RESTlet. However, if you are calling the RESTlet from within an integration and you want to use the full URL, you must include logic that dynamically discovers the RESTlet domain. See the following image for an example of internal and external RESTlet URLs. For more information, see [Identifying a RESTlet in a Call](#).

| Script Deployment | |
|--|---|
| Edit | Back |
| Actions ▾ | |
| SCRIPT | STATUS |
| Advanced Promotions Client Translator | Testing |
| TITLE | LOG LEVEL |
| Advanced Promotions Client Translator 2 | Debug |
| ID | URL |
| customdeploy2 | /app/site/hosting/restlet.nl?script=58&deploy=2 |
| <input checked="" type="checkbox"/> DEPLOYED | EXTERNAL URL |
| | https://123456.restlets.api.netsuite.com/app/site/hosting/restlet.nl?script=58&deploy=2 |

Identifying a RESTlet in a Call

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Before you can access a RESTlet, you must know how to identify it in your call. In general, you use values from the script deployment record. In some cases, you may also have to use the ID value from the script record.

For details, see the following sections:

- [Internal Versus External Calls](#)
- [Dynamically Generating a Full URL](#)

Internal Versus External Calls

The values you use to identify a RESTlet vary depending on the source of the call. For details, see the following table. Each number in the table refers to a callout in the screenshot above.

| Source of the Call | Identify the RESTlet with: |
|--|---|
| An external client | A full URL, similar to the one shown on the script deployment record, in the External URL field (4). For details, see Dynamically Generating a Full URL . |
| A client SuiteScript with an active session in the same NetSuite account where the RESTlet is deployed | Either of the following: <ul style="list-style-type: none"> ▪ The partial URL shown on the script deployment record, in the URL field (3). For an example, see Example of Client Script that Calls a RESTlet. ▪ A URL generated by using the N/url module. To generate the URL this way, you need the script ID, which is viewable on the script record in the ID field (1), in combination with the deployment ID, which is viewable on the script deployment record, in its ID field (2). For an example, see Example of Suitelet that Calls a RESTlet. |
| A server SuiteScript in the same NetSuite account where the RESTlet is deployed | A full URL, similar to the one shown on the script deployment record, in the External URL field (4). This URL must be dynamically generated, in one of the following ways: <ul style="list-style-type: none"> ▪ By using the REST roles service. For details, see the help topic The REST Roles Service. ▪ By using the N/url module. To generate the URL this way, you need the script ID, which is viewable on the script record in the ID field (1), in combination with the deployment ID, which is viewable on the script deployment record, in its ID field (2). For an example, see Example of Suitelet that Calls a RESTlet. |

| Source of the Call | Identify the RESTlet with: |
|---|--|
| A server SuiteScript in a different NetSuite account from where the RESTlet is deployed | A full URL, similar to the one shown on the script deployment record, in the External URL field (4). This URL must be dynamically generated by using the REST roles service. For details, see Dynamically Generating a Full URL . |
| Note: If you are calling a RESTlet by using the delete or get method, you must extend the URL to include any input data that is required by the RESTlet's logic. For details, see Selecting an HTTP Method for Calling a RESTlet . | |

Dynamically Generating a Full URL

When you save a script deployment record for a RESTlet, the system automatically generates a full URL that can be used to call the RESTlet. This value is shown in the **External URL** field.

However, in general, you should not hard-code this URL in a script, or in any other part of your integration. Instead, you should create logic that dynamically generates the portion of the URL that represents the RESTlet domain.

Use the following approaches:

- For calling a RESTlet from an external source, use NetSuite's roles service. For details on using this service, see the help topic [The REST Roles Service](#).
- If you are calling a RESTlet from within NetSuite, you can use the N/url module. With this approach, you provide the ID values from the script record and script deployment record. For an example, see [Example of Suitelet that Calls a RESTlet](#).

| |
|--|
| Note: As of 2017.2, account-specific domains are supported for RESTlets, and you can access your RESTlet domain at the following URL, <accountID>.restlets.api.netsuite.com. The data center-specific domains supported before 2017.2 will continue to be supported. For more information, see the help topic URLs for Account-Specific Domains . |
|--|

Selecting an HTTP Method for Calling a RESTlet

| |
|---|
| Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1. |
|---|

When you call a RESTlet, you can use one of four supported HTTP methods: delete, get, post, or put.

As of NetSuite 2020.2, you can use the N/https module to communicate between SuiteScript scripts, RESTlets, and SuiteTalk REST endpoints without having to reauthenticate. You can use the [https.requestRestlet\(options\)](#) and [https.requestSuiteTalkRest\(options\)](#) methods to send HTTPS requests to a RESTlet or SuiteTalk REST endpoints.

For more details on using these methods with RESTlets, see the following sections:

- [A Call's Method Must Match an Entry Point](#)
- [HTTP Method Functionality](#)
- [Input Data Is Handled Differently by Different Methods](#)
- [Additional HTTPS Methods for Secure Communication Between SuiteScripts and REST Endpoints](#)

A Call's Method Must Match an Entry Point

For each supported HTTP method there is a corresponding supported entry point. For a call to be successful, the method used in the call must match an entry point defined in the RESTlet's interface.

The snippets shown in the following screenshot include a successful pairing of a RESTlet and a call to that RESTlet: In the first snippet, the Suitelet calls the RESTlet by using the get method. Because the RESTlet's interface includes a get entry point, as shown in the second snippet, the call would be successful.

```
// Here, the Suitelet calls a RESTlet
var response = https.get ({url: url, headers: headers});

// In this function, the RESTlet retrieves
// a standard NetSuite record.

function _get(context) {
    doValidation([context.recordtype, context.id], ['recordtype', 'id'], 'GET');
    return record.load({
        type: context.recordtype,
        id: context.id
    });
}

return {
    get : _get,
};

}
```

HTTP Method Functionality

The following HTTP methods are supported: delete, get, post, and put. These methods are defined at the following link: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>.

However, be aware that the way a RESTlet behaves is defined in the RESTlet script and may not necessarily correlate with the intended behavior of the HTTP method being used. Although it is not recommended, your RESTlet could for example define an entry point get function that creates data, rather than retrieving it. Similarly, whether or not a method requires data varies depending on how you write the function. You could write a post entry point function that does not require input.

Input Data Is Handled Differently by Different Methods

The way you pass input parameters to a RESTlet varies depending on the HTTP method you use, as described in the following table.

| Method | Placement of required input |
|--------|--|
| delete | Arguments must be embedded in the URL used to make the call. |
| get | |
| post | Arguments must be included in a request body written in JSON (JavaScript Object Notation) or plain text. |
| put | |

Additional HTTPS Methods for Secure Communication Between SuiteScripts and REST Endpoints

In NetSuite 2020.2, interoperability between SuiteScript scripts, RESTlets, and SuiteTalk REST endpoints is improved by allowing authentication for an already established session to be passed using an HTTPS

call. Previously, when an HTTPS call was used from one NetSuite script type to another, users had to reauthenticate in the second script even though they were still logged in to NetSuite.

Now there are two new N/https methods that don't require reauthentication:

- [https.requestRestlet\(options\)](#) – This method sends an HTTPS request to a RESTlet and returns the response. Authentication headers are automatically added. The RESTlet endpoint identified by its script ID is passed as a parameter. The RESTlet will run with the same privileges as the calling script.
- [https.requestSuiteTalkRest\(options\)](#) – This method sends an HTTPS request to a SuiteTalk REST endpoint and returns the response. Authentication headers are automatically added. The REST endpoint identified by its URL is passed as a parameter.

This new solution replaces the possible need to use Suitelets with Available without login and Execute as Role of Administrator to communicate between a SuiteScript script and a RESTlet or a REST API.



Note: For examples of requests, see [SuiteScript 2.x RESTlet Script and Request Samples](#).

Creating a Content-Type Header



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Depending on the design of a RESTlet, calls to that RESTlet may require a request body. Additionally, sometimes data is returned by a RESTlet, and you may want to use that data.

In the first situation, you must tell NetSuite how your request body will be formatted. In the second, you might want to specify the format of the data that is returned. Note that both formats must be the same. You control this choice by adding a content-type header to your request.

For details on the supported values for this header, see the following table.

| Value | Notes |
|------------------|---|
| application/json | JSON is the appropriate choice for most RESTlets that require a request body, because it lets you map values to fields. |
| application/xml | XML is supported only for the get method. |
| text/plain | Because plain text does not let you map values to fields, this choice should be used only for RESTlets that require limited and simple input. |

Structuring the Header

Format your content-type header as Content-Type: application/json.

Note that content-type header values are case-sensitive.

For an example of a shell script that generates a content-type header, see [Example of Shell Script that Calls a RESTlet](#).

Error Handling

If you omit a content-type header, the request fails with an HTTP error code reading 206: Partial Content.

SuiteScript 2.x RESTlet Reference

See the following topics for more details about working with RESTlets:

- [SuiteScript 2.x RESTlet Error Handling](#)
- [SuiteScript 2.x RESTlet Governance](#)
- [SuiteScript 2.x RESTlet Security](#)
- [Debugging a RESTlet](#)

SuiteScript 2.x RESTlet Error Handling

For details about error handling for RESTlets, refer to the following sections:

- [Supported HTTP Status Codes for RESTlets](#)
- [SuiteScript Errors Returned by RESTlets](#)

Supported HTTP Status Codes for RESTlets

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

For details about the HTTP status codes supported for RESTlets, see:

- [HTTP Success Code](#)
- [HTTP Error Codes](#)

HTTP Success Code

NetSuite supports one HTTP success code for RESTlets: **200 OK**. This code indicates that the request was executed successfully. Note that this code does not necessarily mean that your request worked as you intended. In some cases, a SuiteScript error might occur and be successfully handled by the RESTlet script. In these cases, an HTTP code of 200 is used, and details of the error are described in the response body.

HTTP Error Codes

NetSuite supports the following HTTP error codes for RESTlets.

| Code | Description |
|------------------------------|---|
| 206 Partial Content | You receive this error if you attempt to use a content-type that is unsupported for the HTTP method you are using. |
| 302 Moved Temporarily | This error can be displayed if your request went to the wrong NetSuite data center. Review your integration and make sure that it generates the NetSuite RESTlet domain dynamically. For details, see Dynamically Generating a Full URL . |
| 400 Bad Request | The RESTlet request failed with a user error. Any errors encountered at run time that are unhandled return a 400 error. (If the user code catches the error, a status code of 200 is returned.) |
| 401 Unauthorized | There is no valid NetSuite login session for the RESTlet call. |
| 403 Forbidden | The RESTlet request was sent to an invalid domain. |

| Code | Description |
|-----------------------------------|---|
| 404 Not Found | A RESTlet script is not defined in the RESTlet request. |
| 405 Method Not Allowed | The request method used is not valid. |
| 415 Unsupported Media Type | An unsupported content type was specified. |
| 500 Internal Server Error | This error occurs for non-user errors that cannot be recovered by resubmitting the same request. If this type of error occurs, contact Customer Support to file a case. |
| 503 Service Unavailable | The NetSuite database is offline, or a database connection is not available. |

SuiteScript Errors Returned by RESTlets

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

In some cases, the response to a RESTlet is a SuiteScript error. For details, see the following sections:

- [SuiteScript Error Codes Used by RESTlets](#)
- [SuiteScript Error Message Formatting for RESTlets](#)

SuiteScript Error Codes Used by RESTlets

The following table describes some of the supported SuiteScript errors that can occur when using RESTlets.

| Code | Description | Notes |
|-----------------------------------|--|--|
| INVALID_LOGIN_ATTEMPT | Invalid login attempt. | This error indicates a problem in an OAuth header. It can be returned when the nonce, consumer key, token, or signature in the OAuth header is invalid. |
| INVALID_LOGIN_CREDENTIALS | You have entered an invalid email address or account number. Please try again. | This error indicates a problem in an NLAuth header. |
| INVALID_REQUEST | The request could not be understood by the server due to malformed syntax. | This error is returned because of malformed syntax in an OAuth header. For example, this error can occur when the signature method, version, or timestamp parameter is rejected. |
| INVALID_RETURN_DATA_FORMAT | You should return {1}. | This error is used if the response data does not match the expected format, as specified by the content-type header. |
| SSS_INVALID_SCRIPTLET_ID | That Suitelet is invalid, disabled, or no longer exists. | If you receive this error, make sure that the URL points to the correct RESTlet script deployment ID. |
| UNEXPECTED_ERROR | An unexpected error occurred. Error ID: {1} | |
| TWO_FA_REQD | Two-Factor Authentication required. | This error indicated that two-factor authentication is required for the role, but it is missing. |

SuiteScript Error Message Formatting for RESTlets

The following examples illustrate SuiteScript errors formatting for each supported content-type.

JSON

```

1  {
2    "error":
3    {
4      "code": "SSS_INVALID_SCRIPTLET_ID",
5      "message": "That Suitelet is invalid, disabled, or no longer exists."
6    }
7 }
```

XML

```

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

Text

```

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

SuiteScript 2.x RESTlet Governance

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

There is a guarantee of 10MB per string used as RESTlet input or output. In ideal circumstances, you may receive a larger string in an output, however, the 10MB size is the guarantee.

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



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

SuiteScript 2.x RESTlet Security

The URLs for accessing RESTlets are protected by TLS encryption. Only requests sent using TLS encryption are granted access. For more information, see the help topic [Supported TLS Protocol and Cipher Suites](#).

Debugging a RESTlet

You can use the NetSuite Debugger to debug RESTlet code in the same manner that you debug other types of SuiteScript code, as described in [SuiteScript Debugger](#).

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

Note: In addition to debugging RESTlet script code, it is recommended that you test the HTTP request to be sent to the RESTlet. Free tools are available for this purpose, such as Send HTTP Tool (<https://www.softpedia.com/get/Internet/Servers/Server-Tools/WIN-HTTP-Sender.shtml>).

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

To debug a deployed RESTlet:

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

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

5. Select the RESTlet script that you want to debug in the Script Debugger popup.
After you click the **Select** option button, the RESTlet's cookies display in a banner.
6. Copy the cookies and paste them into a text file so that you have them available.
7. Click the **Select and Close** button in the Script Debugger popup.
The main Script Debugger page displays a message that it is waiting for user action.
8. Set the cookies in your client application to the values you copied in step 6, and send the RESTlet request.
The main Script Debugger page displays the script execution as pending at the NetSuite function `restletwrapper(request)`.
9. With the script loaded into the Debugger, you can now step through each line to inspect local variables and object properties. You can also add watches, evaluate expressions, and set break points. See the help topic [Script Debugger Interface](#) for information on stepping into/out of functions, adding watches, setting and removing break points, and evaluating expressions.

SuiteScript 2.x RESTlet Script and Request Samples

For sample RESTlet scripts and requests, see the following sections:

- [Hello World RESTlet Example](#)
- [Example of RESTlet that Can Retrieve, Delete or Create](#)
- [Example of RESTlet that Adds Multiple Records](#)

- Example of RESTlet that Manipulates Scheduled Script
- Example of Client Script that Calls a RESTlet
- Example of Suitelet that Calls a RESTlet
- Example of Shell Script that Calls a RESTlet

For help with writing other types of scripts in SuiteScript 2.x, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).

Hello World RESTlet Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Using RESTlets can be more complicated than using other script types because, whereas other script types that can be deployed and then will execute as needed, a RESTlet must be deployed and then called. To call a RESTlet successfully, you must correctly identify your NetSuite account's RESTlet domain, use an HTTP method that matches an entry point in your script, and use two required headers: Content-Type and Authorization. (For details on these headers, see [Creating a Content-Type Header](#) and [RESTlet Authentication](#).)

When getting started, you may want to test your RESTlet setup with a simple script such as the following.

RESTlet

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType restlet
4 */
5 define([], function() {
6   return {
7     get : function() {
8       return "Hello World!"
9     }
10  }
11 });

```

Sample Get Call and Response

You call this RESTlet by using the get method. Because the RESTlet takes no arguments, you would not need to extend the URL with additional values. For testing purposes, you could use the value that appears in the External URL field of the script deployment record. (However, in an integration, remember that you must dynamically discover the RESTlet domain.)

For example, you could call this RESTlet by using a URL like the following — one that does not include embedded parameters:

```
1 | https://<accountID>.restlets.api.netsuite.com/app/site/hosting/restlet.nl?script=482&deploy=1
```

After you add the two required headers, the generated call would look like the following.

```

1 | GET /app/site/hosting/restlet.nl?script=482&deploy=1 HTTP/1.1
2 | HOST: <accountID>.restlets.api.netsuite.com
3 | authorization: NLAuth nlauth_account=12345, nlauth_email=john@smith.com, nlauth_signature=Welcome123
4 | content-type: application/json
5 | cookie: ...

```

The RESTlet would return the following response:

```
1 | Hello World!
```

Example of RESTlet that Can Retrieve, Delete or Create



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following RESTlet includes logic for all supported entry points: get, delete, post, and put.

RESTlet

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Restlet
4 */
5 define(['N/record', 'N/error'],
6     function(record, error) {
7         function doValidation(args, argNames, methodName) {
8             for (var i = 0; i < args.length; i++)
9                 if (!args[i] && args[i] !== 0)
10                     throw error.create({
11                         name: 'MISSING_REQ_ARG',
12                         message: 'Missing a required argument: [' + argNames[i] + '] for method: ' + methodName
13                     });
14     }
15     // Get a standard NetSuite record
16     function _get(context) {
17         doValidation([context.recordtype, context.id], ['recordtype', 'id'], 'GET');
18         return JSON.stringify(record.load({
19             type: context.recordtype,
20             id: context.id
21         }));
22     }
23     // Delete a standard NetSuite record
24     function _delete(context) {
25         doValidation([context.recordtype, context.id], ['recordtype', 'id'], 'DELETE');
26         record.delete({
27             type: context.recordtype,
28             id: context.id
29         });
30         return String(context.id);
31     }
32     // Create a NetSuite record from request params
33     function post(context) {
34         doValidation([context.recordtype], ['recordtype'], 'POST');
35         var rec = record.create({
36             type: context.recordtype
37         });
38         for (var fldName in context)
39             if (context.hasOwnProperty(fldName))
40                 if (fldName !== 'recordtype')
41                     rec.setValue(fldName, context[fldName]);
42         var recordId = rec.save();
43         return String(recordId);
44     }
45     // Upsert a NetSuite record from request param
46     function put(context) {
47         doValidation([context.recordtype, context.id], ['recordtype', 'id'], 'PUT');
48         var rec = record.load({
49             type: context.recordtype,
50             id: context.id
51         });
52         for (var fldName in context)
53             if (context.hasOwnProperty(fldName))
```

```

54     if (fldName !== 'recordtype' && fldName !== 'id') {
55         rec.setValue(fldName, context[fldName]);
56     }
57     rec.save();
58     return JSON.stringify(rec);
59 }
60 return {
61     get: _get,
62     delete: _delete,
63     post: post,
64     put: put
65 };
66 });

```

Sample Get Call and Response

To retrieve a record by using this RESTlet, you would use the get method. To identify the record you want to retrieve, you would add values to the URL you use to call the RESTlet. These values would identify the record type and internal ID of the record instance you want. These parameters are defined in the RESTlet's get function as:

- recordtype
- id

You add a value for each parameter by using an ampersand, the name of the parameter, an equals sign, and the parameter's value, as follows:

```
1 | &[name of parameter]=[value]
```

For example, to retrieve a phone call record with the internal ID of 9363, you would use URL like the following:

```
1 | https://<accountID>.restlets.api.netsuite.com/app/site/hosting/restlet.nl?script=474&deploy=1&recordtype=phonecall&id=9363
```

Using the get method in conjunction with this URL would produce the following request:

```

1 | GET /app/site/hosting/restlet.nl?script=474&deploy=1&recordtype=phonecall&id=9363 HTTP/1.1
2 | HOST: <accountID>.restlets.api.netsuite.com
3 | authorization: NLAuth nlauth_account=12345, nlauth_email=john@smith.com, nlauth_signature=Welcome123
4 | content-type: application/json
5 | cookie: ...

```

In response, the system would return data like the following:

```

1 | {
2 |     "id": "9363"
3 |     "type": "phonecall"
4 |     "isDynamic": false
5 |     "fields": {
6 |         "wfinstances": ""
7 |         "nlloc": "0"
8 |         "nlsub": "1"
9 |         "createddate": "5/18/2016 1:01 am"
10 |        "timezone": "America/Los_Angeles"
11 |        "accesslevel": "F"
12 |        "_eml_nkey_": "143659438"
13 |        "starttime": ""
14 |        "startdate": "5/18/2016"
15 |        "title": "Project kickoff"
16 |        "type": "call"
17 |        ...
18 |    }

```

```

19 "sublists": {
20   "contact": {
21     "currentline": {
22       "company": ""
23       "contact": ""
24       ...
25     }
26   }
27 }
28 }
29 }
```

Sample Post Call and Response

To use this RESTlet to add a record, you would use the post method. Your arguments must be included in a request body, and the request body would have to be written in JSON rather than plain text. For example:

```
1 | {"recordtype": "phonecall", "type": "phonecall", "title": "Project Kickoff"}
```

You would send your post method to a URL that has not been extended. For testing purposes, you could use the value that appears in the External URL field of the script deployment record. (However, in an integration, remember that you must dynamically discover the RESTlet domain.)

For example, you could call this RESTlet by using a URL like the following — one that does not include embedded parameters:

```
1 | https://<accountID>.restlets.api.netsuite.com/app/site/hosting/restlet.nl?script=474&deploy=1
```

Making a post call using the request body above, plus the appropriate headers, would produce the following request:

```

1 | POST /app/site/hosting/restlet.nl?script=474&deploy=1 HTTP/1.1
2 | HOST: <accountID>.restlets.api.netsuite.com
3 | authorization: NLAUTH nlauth_account=12345, nlauth_email=john@smith.com, nlauth_signature=Welcome123
4 | content-type: application/json
5 | cookie: ...
```

In response, the system returns the internal ID of the newly created record. For example:

```
1 | 9564
```

Example of RESTlet that Adds Multiple Records



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following RESTlet example can create several contact records in one call.

RESTlet

```

1 | /**
2 |  * @NApiVersion 2.x
3 |  * @NScriptType restlet
4 |  */
5 | define([ 'N/contact' ], function(record) {
```

```

6   return {
7     post : function(restletBody) {
8       var restletData = restletBody.data;
9       for ( var contact in restletData) {
10         var objRecord = record.create({
11           type : record.Type.CONTACT,
12           isDynamic : true
13         });
14         var contactData = restletData[contact];
15         for ( var key in contactData) {
16           if (contactData.hasOwnProperty(key)) {
17             objRecord.setValue({
18               fieldId : key,
19               value : contactData[key]
20             });
21           }
22         }
23         var recordId = objRecord.save({
24           enableSourcing : false,
25           ignoreMandatoryFields : false
26         });
27         log.debug(recordId);
28       }
29     }
30   });
31 });

```

Sample Post Call

To create records with this RESTlet, you would use the post method. You would use a request body to send your values for the new contact records.

```

1 { "data": [{"firstname": "John", "middlename": "Robert", "lastname": "Smith", "subsidiary": "1"}, {"firstname": "Anne", "middle
name": "Doe", "lastname": "Smith", "subsidiary": "1"}]}

```

Example of RESTlet that Manipulates Scheduled Script

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following RESTlet passes a value to a scheduled script.

Note: The scriptId and deploymentId in this sample are placeholders. Before using this script, replace the IDs with valid values from your NetSuite account.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType restlet
4  */
5 define([ 'N/task' ], function(task) {
6   return {
7     get : function() {
8       var mrTask = task.create({
9         taskType : task.TaskType.SCHEDULED_SCRIPT
10      });
11      mrTask.scriptId = 488;
12      mrTask.deploymentId = 'customdeploy_scheduledscript';
13      mrTask.params = {
14        custscriptcustom_data : 'data'
15      };
16      mrTask.submit();
17      return "Scheduled script updated";
18    }
19  }
}

```

20 | }));

Example of Client Script that Calls a RESTlet



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example shows how a client script can call a RESTlet. Because the client script is expected to have an active session, it uses a partial URL to call the RESTlet. That is, the URL does not have to include the RESTlet domain. You can find this partial URL in the **URL** field of the script deployment record for the RESTlet.



Note: The partial URL in this sample is a placeholder. Before using this script, replace this string with a valid value from your NetSuite account.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType ClientScript
4 */
5 define(['N/https'], function(https) {
6     return {
7         pageInit : function() {
8             var dataFromRestlet = https
9                 .get('/app/site/hosting/restlet.nl?script=200&deploy=1');
10            console.log(dataFromRestlet.body);
11        }
12    }
13 });

```

Example of Suitelet that Calls a RESTlet



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example shows how a Suitelet can call a RESTlet. Because the Suitelet is deployed in the same NetSuite account as the RESTlet, the script can use the N/url module to resolve the URL for the RESTlet. With this approach, you need the script ID and the script deployment ID associated with the RESTlet.



Note: ThescriptId, deploymentId, and authorization data in this sample are placeholders. Before using this script, replace these values with valid data from your NetSuite account.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define(['N/https', 'N/url'], function(https, urlMod) {
6     return {
7         onRequest : function(options) {
8             var url = urlMod.resolveScript({
9                 scriptId: 'customscript196',
10                deploymentId: 'customdeploy1',
11                returnExternalUrl: true,
12                params: {parametername: 'value'}
13            });
14
15             var headers = {'Authorization': 'NLAUTH nauth_account=12345, nauth_email=john@smith.com, nauth_signature=Wellcome123, nauth_role=3'};
16
17             var response = https.get({url: url, headers: headers});

```

```

18     options.response.write(response.body);
19 }
20 });
21 });
22 });

```

Example of Shell Script that Calls a RESTlet

i Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following shell script shows you how to send a request using NLAuth authentication and a content-type value of application/json. For information about constructing an NLAuth header, see the help topic [Using User Credentials for RESTlet Authentication](#).

```

1 #!/bin/sh
2
3 #Put your RESTlet url here.
4 my_url="https://<accountID>.restlets.api.netsuite.com/app/site/hosting/restlet.nl?script=126&deploy=1"
5
6 #Put the body of your request in a file named data.json.
7 DATA_FILE="@data.json"
8
9 CONTENT_FLAG="Content-Type: application/json"
10
11 #Update the following line with valid values from your NetSuite account.
12 AUTH_STRING="Authorization: NLAuth nlauth_account=123456, nlauth_email=jsmith%40ABC.com, nlauth_signature=xxxx"
13
14 #Capture the response from your RESTlet.
15 /usr/bin/curl -H "${CONTENT_FLAG}" -H "${AUTH_STRING}" -d "${DATA_FILE}" $my_url > /tmp/restlet_response.txt

```

SuiteScript 2.x RESTlet Script Entry Points

| Script Entry Point | |
|--------------------|---|
| get | All RESTlet entry points return the HTTP response body. |
| delete | |
| put | |
| post | |

delete

i Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|--|
| Description | Defines the function that is executed when a DELETE request is sent to a RESTlet. An HTTP response body is returned. |
| Returns | string Object <ul style="list-style-type: none"> ■ Returns a string when request Content-Type is 'text/plain'. ■ Returns an Object when request Content-Type is 'application/json' or 'application/xml'. |
| Since | Version 2015 Release 2 |

Parameters

Note: The `requestParams` parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Required / Optional | Description | Since |
|----------------------------|--------|---------------------|--|------------------------|
| <code>requestParams</code> | Object | required | The parameters from the HTTP request URL. For all content types, parameters are passed as a JavaScript Object. | Version 2015 Release 2 |

get

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines the function that is executed when a GET request is sent to a RESTlet. An HTTP response body is returned. |
| Returns | <p>string Object</p> <ul style="list-style-type: none"> ■ Returns a string when request Content-Type is 'text/plain'. ■ Returns an Object when request Content-Type is 'application/json' or 'application/xml'. |
| Since | Version 2015 Release 2 |

Parameters

Note: The `requestParams` parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Description | Since |
|----------------------------|--------|--|------------------------|
| <code>requestParams</code> | Object | The parameters from the HTTP request URL. For all content types, parameters are passed as a JavaScript Object. | Version 2015 Release 2 |

post

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines the function that is executed when a POST request is sent to a RESTlet. An HTTP response body is returned. |
| Returns | <p>string Object</p> <ul style="list-style-type: none"> ■ Returns a string when request Content-Type is 'text/plain'. ■ Returns an Object when request Content-Type is 'application/json' or 'application/xml'. |
| Since | Version 2015 Release 2 |

Parameters

Note: The `requestBody` parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Required / Optional | Description | Since |
|--------------------------|-----------------|---------------------|---|------------------------|
| <code>requestBody</code> | string Object | required | <p>The HTTP request body.</p> <ul style="list-style-type: none"> ■ Pass the request body as a string when the request Content-Type is 'text/plain' ■ Pass the request body as a JavaScript Object when the request Content-Type is 'application/json' or 'application/xml'. | Version 2015 Release 2 |

put

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines the function that is executed when a PUT request is sent to a RESTlet. An HTTP response body is returned. |
| Returns | <p>string Object</p> <ul style="list-style-type: none"> ■ Returns a string when request Content-Type is 'text/plain'. ■ Returns an Object when request Content-Type is 'application/json' or 'application/xml'. |
| Since | Version 2015 Release 2 |

Parameters

Note: The `requestBody` parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.

| Parameter | Type | Required / Optional | Description | Since |
|--------------------------|-----------------|---------------------|---|------------------------|
| <code>requestBody</code> | string Object | required | <p>The HTTP request body.</p> <ul style="list-style-type: none"> ■ Pass the request body as a string when the request Content-Type is 'text/plain' ■ Pass the request body as a JavaScript Object when the request Content-Type is 'application/json' or 'application/xml'. | Version 2015 Release 2 |

SuiteScript 2.x Scheduled Script Type

Scheduled scripts are server scripts that are executed (processed) with [SuiteCloud Processors](#). You can deploy scheduled scripts so they are submitted for processing at a future time, or at future times on a recurring basis. You can also submit scheduled scripts on demand from the deployment record or from another script with the [task.ScheduledScriptTask](#) API.

For additional information about SuiteScript 2.x scheduled scripts, see the following:

- [SuiteScript 2.x Scheduled Script Reference](#)
 - [SuiteScript 2.x Scheduled Script Submission](#)
 - [Scheduled Script Deployment Record](#)
 - [Scheduled Script Deployments that Continue to Use Queues](#)
 - [Scheduling a One Time or Recurring Scheduled Script Submission](#)
 - [Submitting an On Demand Scheduled Script Instance from the UI](#)
 - [Submitting an On Demand Scheduled Script Instance from a Script](#)
 - [SuiteScript 2.x Scheduled Script Execution](#)
 - [SuiteScript 2.x Scheduled Script Debugging](#)
 - [SuiteScript 2.x Scheduled Script Status Page](#)
 - [SuiteScript 2.x Scheduled Script Handling of Server Restarts](#)
- [SuiteScript 2.x Script Entry Points and API](#)
 - [execute](#)
 - [context.InvocationType](#)

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

You can use SuiteScript Analysis to learn about when the script was installed and how it performed in the past. For more information, see the help topic [Analyzing Scripts](#).

Scheduled Script Use Cases

Use this script type for basic scheduled or on demand tasks. Your SuiteScript 2.x scheduled script should not process a large amount of data or a large number of records. It should not be used for operations that are long running.

For example, use this script type if:

- You need to log basic information on a recurring schedule
- You need to schedule the future execution of a maintenance script
- You need to create and then purge temporary records
- You need to asynchronously execute logic within another server script



Note: If you previously used SuiteScript 1.0 scheduled scripts, many of the use cases for those scripts now apply to the SuiteScript 2.x [SuiteScript 2.x Map/Reduce Script Type](#).

Scheduled Script Governance

Each scheduled script instance can use a maximum of 10,000 usage units. For additional information on governance and usage units, see the help topic [SuiteScript Governance and Limits](#).

With SuiteScript 2.x scheduled scripts, you cannot set recovery points and you do not have the ability to yield. There is no SuiteScript 2.x equivalent to the SuiteScript 1.0 `nlapiYieldScript()` and

[nlapiSetRecoveryPoint\(\)](#) APIs. If you need to process a large amount of data or a large number of records, use the [SuiteScript 2.x Map/Reduce Script Type](#) instead. The map/reduce script type has built in yielding and can be submitted for processing in the same ways as scheduled scripts.

Scheduled Script Entry Points

| Script Entry Point | |
|-------------------------|---|
| execute | Defines the scheduled script trigger point. |

Scheduled Script API

| API | |
|--|---|
| context.InvocationType | Enumeration that holds the string values for scheduled script execution contexts. |

Scheduled Script Sample

This script sample finds and fulfills sales orders created on the current day.

Before you submit this script:

1. Create a sales order type of saved search. You can use [search.create\(options\)](#) and the [search.Type](#) enum to set up a saved search with the correct search id and search type.
2. Create a script parameter on the script record **Parameters** subtab. The sample accepts a search id from a script parameter that it assumes was created with the script record.
 - Set the id to `custscript_searchid`.
 - Set Type to Free-Form Text.
 - Assign the saved search id to the parameter. This is done on the deployment record **Parameters** subtab,

Note that this script sample uses the `log.debug()` method but does not load the N/log module. A `log` object is loaded by default for all script types, and you do not need to load the N/log module explicitly. For more information, see the help topic [log Object](#).

This script sample uses SuiteScript 2.0. A newer version, SuiteScript 2.1, is also available and supports new language features that are included in the ES2019 specification. You can write scheduled scripts using either SuiteScript 2.0 or SuiteScript 2.1.

- For help with writing scripts in SuiteScript 2.x, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).
- For more information about SuiteScript versions and SuiteScript 2.1, see [SuiteScript Versioning Guidelines](#) and [SuiteScript 2.1](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType ScheduledScript
4 */
5 define(['N/search', 'N/record', 'N/email', 'N/runtime'],
6   function(search, record, email, runtime) {
7     function execute(context) {
8       if (context.type !== context.InvocationType.ON_DEMAND)
9         return;
10      var searchId = runtime.getCurrentScript().getParameter("custscript_searchid");
11      try {
12        search.load({
13          id: searchId

```

```

14     }).run().each(function(result) {
15         log.debug({
16             details: 'transforming so : ' + result.id + ' to item fulfillment'
17         });
18         var fulfillmentRecord = record.transform({
19             fromType: record.Type.SALES_ORDER,
20             fromId: result.id,
21             toType: record.Type.ITEM_FULFILLMENT,
22             isDynamic: false
23         });
24         var lineCount = fulfillmentRecord.getLineCount('item');
25         for (var i = 0; i < lineCount; i++) {
26             fulfillmentRecord.setSublistValue('item', 'location', i, 1);
27         }
28         var fulfillmentId = fulfillmentRecord.save();
29         var so = record.load({
30             type: record.Type.SALES_ORDER,
31             id: result.id
32         });
33         so.setValue('memo', fulfillmentId);
34         so.save();
35         return true;
36     });
37 } catch (e) {
38     var subject = 'Fatal Error: Unable to transform salesorder to item fulfillment!';
39     var authorId = -5;
40     var recipientEmail = 'notify@xxxxxx.com';
41     email.send({
42         author: authorId,
43         recipients: recipientEmail,
44         subject: subject,
45         body: 'Fatal error occurred in script: ' + runtime.getCurrentScript().id + '\n\n' + JSON.stringify(e)
46     });
47 }
48 return {
49     execute: execute
50 };
51 );
52 );

```

SuiteScript 2.x Scheduled Script Reference

- [SuiteScript 2.x Scheduled Script Submission](#)
- [SuiteScript 2.x Scheduled Script Execution](#)
- [SuiteScript 2.x Scheduled Script Debugging](#)
- [SuiteScript 2.x Scheduled Script Status Page](#)
- [SuiteScript 2.x Scheduled Script Handling of Server Restarts](#)

SuiteScript 2.x Scheduled Script Submission

All Scheduled script instances are executed (processed) by [SuiteCloud Processors](#). You can submit a scheduled script instance for processing in one of three ways:

- By [Scheduling a One Time or Recurring Scheduled Script Submission](#) from the script deployment record UI
- By [Submitting an On Demand Scheduled Script Instance from the UI](#) with the **Save and Execute** option
- By [Submitting an On Demand Scheduled Script Instance from a Script](#) with the [task.ScheduledScriptTask API](#)

Each of these options requires you to create a [Scheduled Script Deployment Record](#).



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

Scheduled Script Deployment Record



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Before you can submit a scheduled script instance to [SuiteCloud Processors](#), you must first create at least one deployment record for the script.

Body Fields

| Field | Description |
|----------|--|
| Script | A link to the script record associated with the deployment <div style="border: 1px solid #ccc; padding: 5px;"> i Note: This value cannot be changed, even on a new deployment record. If you begin the process of creating a deployment and realize that you have selected the wrong script record, you must start the process over. </div> |
| Title | The user-defined name for the deployment |
| ID | A unique identifier for the deployment: <ul style="list-style-type: none"> ▪ If this field is left blank when the deployment is created, the system generates the ID. ▪ You should enter a custom ID if you plan to bundle the script and deployment for installation into another account. This reduces the risk of naming conflicts. Custom IDs must be all lower case and cannot contain spaces. You can use "_" as a delineator. <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> i Note: For both custom and system-defined IDs, the system appends the string "customdeploy" to the ID. </div> <p>Although not recommended, you can edit this value on an existing deployment. To do this, click Edit on the deployment record. Then click Change ID.</p> |
| Deployed | Indicates whether the deployment is active. This box must be enabled if you want your script to execute. If you disable this option: <ul style="list-style-type: none"> ▪ The script does not execute, regardless of the status. This applies even if a submission schedule is configured on the Schedule subtab. ▪ The Save and Execute option no longer displays on the Save dropdown when the deployment record is in edit mode. ▪ The script cannot be submitted programmatically. For more information, see Submitting an On Demand Scheduled Script Instance from a Script. |
| Status | This value determines how and when a script can be submitted to SuiteCloud Processors for processing. Possible options are: <ul style="list-style-type: none"> ▪ Testing: You can test the script in the SuiteScript Debugger as a deployed script. For more information on debugging deployed scripts, see the help topics Debugging Deployed SuiteScript 1.0 and SuiteScript 2.0 Server Scripts and Debugging Deployed SuiteScript 2.1 Server Scripts. Deployed debugging is restricted to the script owner. |

| Field | Description |
|-----------------------|--|
| | <ul style="list-style-type: none"> ■ Scheduled: You can schedule a single or recurring automatic submission of the script on the Schedule subtab. You cannot submit an on demand instance of the script when it has this status. <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p> Note: If you schedule a recurring submission of the script, the script's deployment status on the SuiteScript 2.x Scheduled Script Status Page remains as Scheduled, even after the script completes its execution.</p> </div> <ul style="list-style-type: none"> ■ Not Scheduled: The script is not currently scheduled for automatic submission. You can submit an on demand instance of the script with either: <ul style="list-style-type: none"> □ The Save and Execute button on the deployment record □ The <code>task.ScheduledScriptTask</code> API <p>You can submit an on demand instance of the script only if there is no other unfinished instance of the same script.</p> |
| See Instances | A link to the SuiteScript 2.x Scheduled Script Status Page , filtered for all deployment instances of the script associated with this particular deployment. |
| Log Level | <p>This value determines the information logged for this deployment. Entries are displayed on the Execution Log subtab. Possible options are:</p> <ul style="list-style-type: none"> ■ Debug: Generally set when a script is in testing mode. A log level set to Debug shows all Audit, Error, and Emergency information in the script log. ■ Audit: Generally set for scripts running in production mode. A log level set to Audit provides a record of events that have occurred during the processing of the script (for example, "A request was made to an external site"). ■ Error: Generally used for scripts running in production mode. A log level set to Error shows only unexpected script errors in the script log. ■ Emergency: Generally used for scripts running in production mode. A log level set to Emergency shows only the most critical errors in the script log. |
| Execute as Role | For scheduled script deployments, this value is automatically set to Administrator and cannot be edited. |
| Priority | <p>This value impacts when the SuiteCloud Processors scheduler sends the scheduled script job to the processor pool. By default, this field is set to Standard. For additional information, see the help topic SuiteCloud Processors Priority Levels</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p> Note: Each SuiteScript 2.x scheduled script instance is handled by a single job within SuiteCloud Processors.</p> </div> <div style="background-color: #ffffcc; border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p> Important: You must understand SuiteCloud Processors before you change this setting.</p> </div> <p>Possible options are:</p> |
| | <ul style="list-style-type: none"> ■ High: Use to mark critical jobs that require more immediate processing. The scheduler sends these jobs to the processor pool first. ■ Standard: This is the default setting. It is considered to be a medium priority level. The scheduler sends these jobs to the processor pool if there are no high priority jobs waiting. ■ Low: Use to mark jobs that can tolerate a longer wait time. The scheduler sends these jobs to the processor pool if there are no high or standard priority jobs waiting. |
| Queue (Deprecated) | This field is deprecated with the introduction of SuiteCloud Processors . After you click Remove Queue , it no longer displays on the deployment record. |

Schedule Subtab

| Field | Description |
|---------------|--|
| Single Event | The scheduled script task is submitted only once. Use this option if you want to schedule a future one time submission. |
| Daily Event | <p>The scheduled script task is submitted every x number of days. If you schedule the submission to recur every x minutes or hours, the schedule will start over on the next scheduled day.</p> <p>For example, your deployment is set to submit daily, starting at 3:00 am and recurring every five hours. A scheduled script instance is submitted at 3:00 am, 8:00 am, 1:00 pm, 6:00 pm, and 11:00 pm. At midnight, the schedule starts over and the next submission is at 3:00 am.</p> |
| Weekly Event | <p>The scheduled script instance is submitted at least once per scheduled week. If you schedule the submission to recur every x minutes or hours, the schedule will start over on the next scheduled day.</p> <p>For example, your deployment is set to submit on Tuesday and Wednesday, starting at 3:00 am and recurring every five hours. A scheduled script instance is submitted on Tuesday at 3:00 am, 8:00 am, 1:00 pm, 6:00 pm, and 11:00 pm. On Wednesday, the schedule starts over and the next submission is at 3:00 am.</p> |
| Monthly Event | The scheduled script instance is submitted at least once per scheduled month. |
| Yearly Event | The scheduled script instance is submitted at least once per year. |
| Start Date | The first submission occurs on this date. This field is mandatory if a one time or recurring schedule is set. |
| Start Time | If a value is selected, the first submission occurs at this time. |
| Repeat | <p>If a value is selected, the first submission occurs on the date and time selected. A new script instance is then created and submitted every x minutes or hours until the end of the start date. If applicable, the schedule starts over on the next scheduled day.</p> <p>For example, your deployment is set to submit on Tuesday and Wednesday, starting at 3:00 am and recurring every five hours. A scheduled script instance is submitted on Tuesday at 3:00 am, 8:00 am, 1:00 pm, 6:00 pm, and 11:00 pm. On Wednesday, the schedule starts over and the next submission is at 3:00 am.</p> |
| End By | If a value is entered, the last submission occurs on this date. If you schedule the submission to recur every x minutes or hours, a new script instance is created and submitted every x minutes or hours until the end of the end date . |
| No End Date | The schedule does not have a set end date. |

Scheduled Script Deployments that Continue to Use Queues



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

For all scheduled script deployments created **prior** to the introduction of [SuiteCloud Processors](#), the **Queue** field remains by default. This applies to accounts with and without SuiteCloud Plus. You have control over whether to stop using queues. The deployment record includes a **Remove Queue** option. After you select this option, the deployment no longer uses a queue **and cannot revert back to using a queue**.

The **Queue** field remains to accommodate deployments that rely on the FIFO order of processing imposed by an individual queue. However, all jobs that use queues are actually processed by the same processor pool that handles the jobs that do not use queues. All jobs compete with each other using the same common processing algorithm.



Note: For deployments that continue to use queues, all jobs assigned to the same queue should have the same priority. In most cases, you can keep the default (standard) priority of these jobs. However, in some cases, you may want to change these jobs to a higher or lower priority. One scenario is if you want to ensure that a specific queue always has a processor available. In that case, designate the jobs assigned to that queue as high priority. Alternatively, if you have a group of lower priority jobs, you can designate them as low priority and assign them to the same queue. That will ensure that only one is processed at a time.



Important: If your existing scheduled script deployments rely on implicit dependencies imposed by queues, you should update and test these scripts before you remove queues. Your scripts may be impacted if they rely on the sequence of FIFO (first in, first out).

One possible solution is to programmatically submit scripts in a certain order. To do this, use `task.ScheduledScriptTask` within the first script to submit the second script. This will ensure that the jobs are submitted to the processor pool in the correct order.

Scheduling a One Time or Recurring Scheduled Script Submission



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Scheduled script instances can be submitted for processing once at a pre-defined time in the future, or repeatedly on a regular daily, weekly, monthly, or yearly basis. To set a submission scheduled from the deployment record, the Status field must be set to **Scheduled**.

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

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

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

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



Important: Before you submit a scheduled script instance for processing, you must understand how [SuiteCloud Processors](#) works.



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

To schedule a one time or recurring scheduled script submission:

1. First, create your scheduled script entry point script. This includes your JavaScript file and its associated script record. To review a scheduled script example, see [Scheduled Script Sample](#). If this is your first script, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).
2. On the associated script record, select the **Deploy Script** button.
3. When the script deployment record loads, click the **Deployed** check box if it is not already checked.

4. Review the scheduled script deployment field descriptions at [Scheduled Script Deployment Record](#).
5. Set the Status field to **Scheduled**.
6. Set the remaining body fields.
7. On the [Schedule Subtab](#), set all deployment options.

If you want the schedule to **submit hourly on a 24 hour basis**, use the following sample values as a guide:

- Deployed = checked
- Daily Event = [radio button enabled]
- Repeat every 1 day
- Start Date = [today's date]
- **Start Time = 12:00 am**
- Repeat = every hour
- End By = [blank]
- No End Date = checked
- Status = Scheduled
- Log Level = Error
- Execute as Role = Set to **Administrator**

If the **Start Time** is set to any other time than 12:00 am (for example, it is set to 2:00 pm), the script will start at 2:00 pm, but then finish its hourly execution at 12:00 am. It will not resume until the next day at 2:00 pm.

8. Click **Save**.

Submitting an On Demand Scheduled Script Instance from the UI



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

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

The **Testing** status is primarily used to debug and test a script. If you wish to step through the script with the [SuiteScript Debugger](#), you must use deployed debugging. For more information on debugging deployed scripts, see the help topics [Debugging Deployed SuiteScript 1.0](#) and [SuiteScript 2.0 Server Scripts](#) and [Debugging Deployed SuiteScript 2.1 Server Scripts](#).



Important: Before you submit a scheduled script instance for processing, you must understand how [SuiteCloud Processors](#) works.



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

To submit an on demand scheduled script instance from the UI:

1. First, create your scheduled script entry point script. This includes your JavaScript file and its associated script record. To review a scheduled script example, see [Scheduled Script Sample](#). If this

is your first script, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).

2. On the associated script record, select the **Deploy Script** button.
3. When the script deployment record loads, click the **Deployed** check box if it is not already checked.
4. Review the scheduled script deployment field descriptions at [Scheduled Script Deployment Record](#).
5. Set the Status field to **Not Scheduled** or **Testing**.
6. Set the remaining body fields.
7. On the [Schedule Subtab](#), set all deployment options.
8. Click **Save and Execute** from the **Save** dropdown.

Submitting an On Demand Scheduled Script Instance from a Script



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

You can programmatically submit a scheduled script instance to [SuiteCloud Processors](#) using the [task.ScheduledScriptTask](#) API.

To schedule a one time or recurring scheduled script submission:

1. First, create your scheduled script entry point script. This includes your JavaScript file and its associated script record. To review a scheduled script example, see [Scheduled Script Sample](#). If this is your first script, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).
2. On the associated script record, select the **Deploy Script** button.
3. When the script deployment record loads, click the **Deployed** check box if it is not already checked.
4. Review the scheduled script deployment field descriptions at [Scheduled Script Deployment Record](#).
5. Set the Status field to **Not Scheduled**.
6. Set the remaining body fields.
7. Click **Save**.
8. In the server script where you want to submit the scheduled script instance, call `task.create(options)` to return a [task.ScheduledScriptTask](#) object:

```
1 | var scriptTask = task.create({taskType: task.TaskType.SCHEDULED_SCRIPT});
```

9. Set the [ScheduledScriptTask.scriptId](#) and [ScheduledScriptTask.deploymentId](#) properties:

```
1 | scriptTask.scriptId = 1234;
2 | scriptTask.deploymentId = 'customdeploy1';
```

10. Call [ScheduledScriptTask.submit\(\)](#) to submit the scheduled script instance to [SuiteCloud Processors](#).

SuiteScript 2.x Scheduled Script Execution

All scheduled script instances are executed (processed) by [SuiteCloud Processors](#). For additional information on how scheduled scripts are submitted, see [SuiteScript 2.x Scheduled Script Submission](#). Each submitted scheduled script instance is handled by one job. A scheduler sends the jobs to a processor pool in a particular order. This order is determined by the [SuiteCloud Processors Priority Levels](#) and the order of submission. Jobs with a higher priority are sent before jobs with a lower priority. Jobs with the same priority go to the processor pool in the order of submission.

SuiteCloud Processors includes advanced settings that can also impact the order in which jobs are sent to the processor pool. For additional information, see the help topic [SuiteCloud Processors Priority Elevation and Processor Reservation \(Advanced Settings\)](#).

SuiteScript 2.x Scheduled Script Debugging

The deployment record **Testing** status is primarily used to debug and test a script. If you wish to step through the script with the [SuiteScript Debugger](#), you must use deployed debugging. For more information on debugging deployed scripts, see the help topics [Debugging Deployed SuiteScript 1.0](#) and [SuiteScript 2.0 Server Scripts](#) and [Debugging Deployed SuiteScript 2.1 Server Scripts](#).

SuiteScript 2.x Scheduled Script Status Page

The Scheduled Script Status page shows the current and past runtime statuses of all submitted scheduled script instances in your account.

| Scheduled Script Status | | | | | | | | | | | |
|--|-------------|--|----------------------|----------|------------|------------|----------|-------|------------|--------|--|
| <input type="button" value="Refresh"/> <input type="button" value="FILTERS"/> | | | | | | | | | | | |
| SCRIPT | API VERSION | DEPLOYMENT ID | DATE CREATED | STATUS | START | END | PRIORITY | QUEUE | % COMPLETE | CANCEL | |
| TOTAL: 1 | | | | | | | | | | | |
| Migrate PT SAFT Account Grouping SS | 2.0 | customdeploy_migrate_pt_acct_grouping_ss | 15.4.2020 7:57:43 am | Complete | 7:57:44 am | 7:57:45 am | Standard | | 100.0% | | |

There are different ways you can access the Scheduled Script Status page. How you access the page determines the view that the page opens with.

- Go to Customization > Scripting > Scheduled Script Status. The columns shown on the Scheduled Script Status page are:

| SCRIPT | API VERSION | DEPLOYMENT ID | DATE CREATED | STATUS | START | END | PRIORITY | QUEUE | % COMPLETE | CANCEL |
|-------------------------------------|-------------|--|----------------------|----------|------------|------------|----------|-------|------------|--------|
| Migrate PT SAFT Account Grouping SS | 2.0 | customdeploy_migrate_pt_acct_grouping_ss | 15.4.2020 7:57:43 am | Complete | 7:57:44 am | 7:57:45 am | Standard | | 100.0% | |

- Click the Status Page link on a Script Deployment record for a single scheduled script. The columns shown on the Scheduled Script Status page are:

| API VERSION | DEPLOYMENT ID | DATE CREATED | STATUS | START | END | PRIORITY | QUEUE | % COMPLETE | CANCEL |
|-------------------------------------|---------------|----------------------|----------|------------|------------|----------|-------|------------|--------|
| Migrate PT SAFT Account Grouping SS | 2.0 | 15.4.2020 7:57:43 am | Complete | 7:57:44 am | 7:57:45 am | Standard | | 100.0% | |

You will notice that the same information is shown on each form of the Scheduled Script Status page but in a different order.

You can also filter the scheduled scripts shown on the page using the Date, From, To, Script, Deployment Id, Queue, Status, and API Version filters.

The following table describes each field on the page.

| Field | Description |
|---------------|---|
| API Version | This field shows the API Version specified in the script using the API Version JSDoc tag for the current version of the script. This means that if you submit a scheduled script using API Version 2.0 and then submit it a second time using API Version 2.1, the version that is shown in the API Version field will be 2.1 for all listings of the script. |
| Date Created | This field shows the date the script was created. |
| Deployment ID | This field shows the deployment id for the deployment record for the scheduled script. |

| Field | Description |
|----------|--|
| Start | This field shows the time the scheduled script execution started. |
| End | This field shows the time the scheduled script execution ended. |
| Priority | This field shows the priority for the scheduled script. For more information on priorities, see the help topic SuiteCloud Processors Priority Levels . |
| Queue | This field shows the queue for the scheduled script. The Queue field only pertains to scheduled script deployments that occurred before the introduction of SuiteCloud Processors. For more information, see Scheduled Script Deployments that Continue to Use Queues . |
| Status | <p>This field shows the execution status for the scheduled script. This status will be one of the following:</p> <ul style="list-style-type: none"> ■ Cancelled. This status indicates that due to a NetSuite server error, the script was cancelled before or during script execution. ■ Complete. This status indicates the script completed normally. ■ Deferred. This status indicates that the script is eligible for processing, but has not been processed due to processing constraints. For example, deferred status occurs when one job must wait for another job to finish. ■ Failed. This status indicates that the script has been processed, but failed to complete normally. If your script has failed, examine it for possible errors. ■ Pending. This status indicates that the script has been submitted and is waiting to be processed. ■ Processing. This status indicates the script is running. ■ Retry. This status indicates that the script entered the processing state, but failed to complete normally. In this case, the script is eligible to be retried. The script will be retried automatically — you do not need to create a new deployment. Review the script execution log to help you determine why the script initially failed (for example, a timeout problem occurred). |

SuiteScript 2.x Scheduled Script Handling of Server Restarts

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

In SuiteScript 2.x, in the event of an unexpected system failure, the script is restarted from the beginning. The NetSuite system can automatically detect when a scheduled script is forcefully terminated, and restart the script as soon as the resources required to run the script are available. Note that in SuiteScript 2.x, yields and recovery points are not manually scripted. Therefore, handling a restart situation is simpler.

The following sample scripts demonstrate how restarts impact scheduled script execution:

- [Simple Scheduled Script](#)
- [Example: A Problematic Scheduled Script](#)
- [Example: A Robust Scheduled Script](#)

Simple Scheduled Script

If there is a forceful termination in any part of the script, the script is always restarted from the beginning.

The script can detect a restarted execution by examining the type attribute of the context argument. The script is being restarted if the value of this argument is (`context.type === "aborted"`). For more information about the context argument, see [execute context.InvocationType](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType ScheduledScript
4 */
5 define([], function(){
6     return {
7         execute: function (context)
8         {
9             if (context.type === 'aborted')
10            {
11                // I might do something differently
12            }
13            .
14            .
15            .
16        }
17    }
18 });

```

Example: A Problematic Scheduled Script

A very common pattern seen in scheduled scripts is to perform a search and then processing the results. Consider the following script:

The purpose of this example script is to update each sales order in the system. The filter1 filter ensures that the search returns exactly one entry per sales order. However, if the script is forcefully interrupted during its processing and then restarted, some sales orders might be updated twice.

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

This script does not use idempotent operations, and a large number of sales orders could be updated twice (for example, doubling a price on a sales order from a repeated operation). To improve a script like this, see [Example: A Robust Scheduled Script](#) and [A Robust Map/Reduce Script Example](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType ScheduledScript
4 */
5 define(['N/search', 'N/record'],
6     function(search, record){
7         return {
8             execute: function (context)
9             {
10                 var filter1 = search.createFilter({
11                     name: 'mainline',
12                     operator: search.Operator.IS,
13                     values: true
14                 });
15                 var srch = search.create({
16                     type: search.Type.SALES_ORDER,
17                     filters: [filter1],
18                     columns: []
19                 });
20
21                 var pagedResults = srch.runPaged();
22
23                 pagedResults.pageRanges.forEach(function(pageRange){
24                     var currentPage = pagedResults.fetch({index: pageRange.index});
25                     currentPage.data.forEach(function(result){
26                         var so = record.load({
27                             type: record.Type.SALES_ORDER,
28                             id: result.id
29                         });
30                     //UPDATE so FIELDS>

```

```

31         so.save()
32     });
33   });
34 }
35 });
36 });

```

Example: A Robust Scheduled Script

The following sample code uses `custbody_processed_flag` on the sales order. It is a custom boolean field that must be previously created in the UI. When the sales order is processed, the field is set to true. The search then contains an additional filter that excludes flagged sales orders. When the script is restarted, only the sales order which have not been updated are processed.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType ScheduledScript
4 */
5 define(['N/search', 'N/record'],
6   function(search, record){
7     return {
8       execute: function (context)
9     {
10       var filter1 = search.createFilter({
11         name: 'mainline',
12         operator: search.Operator.IS,
13         values: true
14       });
15       var filter2 = search.createFilter({
16         name: 'custbody_processed_flag',
17         operator: search.Operator.IS,
18         values: false
19       });
20       var srch = search.create({
21         type: search.Type.SALES_ORDER,
22         filters: [filter1, filter2],
23         columns: []
24     });
25
26       var pagedResults = srch.runPaged();
27
28       pagedResults.pageRanges.forEach(function(pageRange){
29         var currentPage = pagedResults.fetch({index: pageRange.index});
30         currentPage.data.forEach(function(result){
31           var so = record.load({
32             type: record.Type.SALES_ORDER,
33             id: result.id
34           });
35           // UPDATE so FIELDS
36           so.setValue({
37             fieldID: 'custbody_processed_flag',
38             value: true
39           });
40           so.save()
41         });
42       });
43     }
44   });
45 });

```

SuiteScript 2.x Script Entry Points and API

Entry Points:

- `execute`

API:

- [context.InvocationType](#)

execute



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|--|
| Description | Defines the Scheduled script trigger point |
| Returns | void |
| Since | 2015.2 |

Parameters



Note: The scriptContext parameter is a JavaScript object. NetSuite automatically passes this object to the script entry point.

| Parameter | Type | Required / Optional | Description | Since |
|--------------------|--------|---------------------|--|--------|
| scriptContext.type | string | required | The script execution context. Values are reflected in the context.InvocationType enum. | 2015.2 |

context.InvocationType



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|-------------------------|---|
| Enum Description | Enumeration that holds the string values for scheduled script execution contexts. |
| Module | SuiteScript 2.x Scheduled Script Type |
| Since | 2015.2 |

Values

| | |
|----------------|---|
| SCHEDULED | The normal execution according to the deployment options specified in the UI. |
| ON_DEMAND | The script is executed via a call from a script (using ScheduledScriptTask.submit()). |
| | <div style="border: 1px solid #ccc; padding: 5px;"> i Note: The scheduled script must have a status of Not Scheduled on the Script Deployment page. </div> |
| USER_INTERFACE | The script is executed via the UI (the Save & Execute button has been clicked). |
| ABORTED | The script re-executed automatically following an aborted execution (system went down during execution). |
| SKIPPED | The script is executed automatically following downtime during which the script should have been executed. |

SuiteScript 2.x Suitelet Script Type

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

There are two types of Suitelets:

1. **Suitelets** use UI objects to create custom pages that look like NetSuite pages. SuiteScript UI objects encapsulate the elements for building NetSuite-looking portlets, forms, fields, sublists, tabs, lists, and columns.
2. **Backend Suitelets** do not use any UI objects and execute backend logic, which can then be parsed by other parts of a custom application. Backend Suitelets are best used for the following purposes:
 - Providing a service for backend logic to other SuiteScripts.
 - Offloading server logic from client scripts to a backend Suitelet shipped without source code to protect sensitive intellectual property.



Important: RESTlets can be used as an alternative to backend Suitelets.



Note: Suitelets are not intended for use in systems integration use cases.



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



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

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

You can use SuiteScript Analysis to learn about when the script was installed and how it performed in the past. For more information, see the help topic [Analyzing Scripts](#).

See the following for more information about the Suitelet script type:

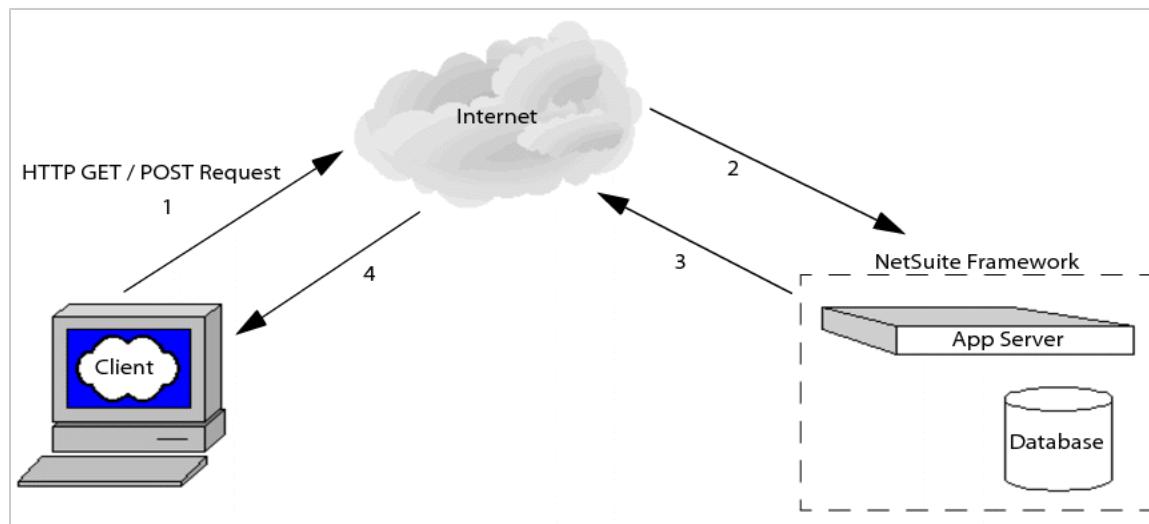
- [SuiteScript 2.x Suitelet Script Reference](#)
 - [SuiteScript 2.x How Suitelet Scripts are Executed](#)
 - [SuiteScript 2.x Reserved Parameter Names in Suitelet URLs](#)
 - [SuiteScript 2.x Suitelet Script Best Practices](#)
 - [SuiteScript 2.x Suitelet Script Deployment Page](#)
 - [SuiteScript 2.x Embedding HTML in Suitelets](#)
- [SuiteScript 2.x Suitelet Script Entry Points and API](#)
 - [onRequest\(params\)](#)
- [SuiteScript 2.x Suitelet Script Type Code Samples](#)

SuiteScript 2.x Suitelet Script Reference

- [SuiteScript 2.x How Suitelet Scripts are Executed](#)
- [SuiteScript 2.x Reserved Parameter Names in Suitelet URLs](#)
- [SuiteScript 2.x Suitelet Script Best Practices](#)
- [SuiteScript 2.x Suitelet Script Deployment Page](#)
- [SuiteScript 2.x Embedding HTML in Suitelets](#)

SuiteScript 2.x How Suitelet Scripts are Executed

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



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



Important: You can only redirect to external URLs from Suitelets that are accessed externally (in other words, the Suitelet has been designated as "Available Without Login" and is accessed from its external URL).

4. The data renders in the user's browser.

SuiteScript 2.x Reserved Parameter Names in Suitelet URLs

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

The following table contains a list of reserved parameter names:

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

If any of your parameters are named after any of the reserved parameter names, your Suitelet may throw an error saying, "There are no records of this type." To avoid naming conflicts, NetSuite recommends that all custom URL parameters are prefixed with **custom**. For example, use `custom_id` instead of `id`.

SuiteScript 2.x Suitelet Script Best Practices

The following list shows best practices for both form-level and record-level client script development:

- Suitelets are ideal for generating NetSuite pages (forms, lists), returning data (XML, text), and redirecting requests.
- Limit the number of UI objects on a page (< 100 rows for sublists, < 100 options for on demand select fields, < 200 rows for lists).
- Experiment with inline HTML fields embedded on the form before going the full custom HTML page route.
- Deploy Suitelets as "Available without Login" only if absolutely necessary (no user context, login performance overhead). (See the help topic [Setting Available Without Login](#).)
- Append "ifrmcntnr=T" to the external URL when embedding in iFrame especially if you are using Firefox. (For more about NetSuite and iFrame, see the help topic [Embedding an Online Form in your Website Page](#).)
- When building custom UI outside of the standard NetSuite UI (such as building a custom mobile page using Suitelet), use the User Credentials APIs to help users manage their credentials within the custom UI. For more information, see User Credentials APIs.
- For access or redirection from another script to a Suitelet, the best practice is to use `url.resolveDomain(options)` to discover the URL instead of hard- coding the URL.
- When calling a Suitelet using its external URL, properly escape the parameter values in order to avoid cross-site scripting injections, for example, by converting the appropriate characters to HTML entities.

SuiteScript 2.x Suitelet Script Deployment Page

Before a suitelet script can be executed, you must create at least one deployment record for the script.

The deployment record for a suitelet script is similar to that of other script types. However, a suitelet script deployment contains some additional fields. This topic describes all of the available fields.

You can access a suitelet script deployment record in the following ways:

- To open an existing deployment record for editing, go to Customization > Scripting > Script Deployments (Administrator). Locate the appropriate record, and click the corresponding **Edit** link.
- To start creating a new deployment record, open the appropriate script record in view mode, and click the **Deploy Script** button. For help creating a script record, see [Script Record Creation](#).

Suitelet Script Deployment Page Body Fields

The following table summarizes the body fields available on the Suitelet script deployment record. Note that some fields are available only when you edit or view an existing deployment record.

| Field | Description |
|------------|--|
| Script | A link to the script record associated with the deployment. This value cannot be changed, even on a new deployment record. If you begin the process of creating a deployment and realize that you selected the wrong script record, you must start the process over. |
| Title | The user-defined name for the deployment. |
| ID | <p>A unique identifier for the deployment.</p> <p>On a new record, you can customize this identifier by entering a value in the ID field. You should customize the ID if you plan to bundle the deployment for installation into another account, because using custom IDs helps avoid naming conflicts. IDs must be lowercase and cannot use spaces. You can use an underscore as a delineator.</p> <p>If you do not enter a value, the system automatically generates one. In both cases, the system automatically adds the prefix <code>customdeploy</code> to the ID created when the record is saved.</p> <p>Although not recommended, you can change the ID on an existing deployment by clicking the Change ID button.</p> |
| Deployed | A configuration option that indicates whether the deployment is active. This box must be checked if you want the script to execute. |
| Status | <p>The status can be set to Testing or Released.</p> <ul style="list-style-type: none"> ■ Testing — The script will execute for the script owner and specified audience. ■ Released — The script will run in the accounts of all specified audience members. <p>See Errors Related to the Available Without Login URL to learn more about the relevance of the Released status when deploying internally and externally available Suitelets.</p> |
| Event Type | <p>Use the Event Type list to specify a script execution context at the time of script deployment.</p> <p>After an event type is specified, the deployed script executes only on that event, regardless of the event types specified in the actual script file.</p> <div style="border: 1px solid #f0e68c; padding: 10px; margin-top: 10px;"> <p> Important: Event types specified in the UI take precedence over the types specified in the script file. For example, if the create event type is specified in the script, selecting delete from the Event Type list restricts the script from running on any event other than delete. If the Event Type field is left blank, your script will execute only on the event type(s) specified in the script file.</p> </div> |
| Log Level | A value that determines what type of log messages are displayed on the Execution Log of both the deployment record and associated script record. The available levels are: |

| | |
|-------------------------|--|
| | <ul style="list-style-type: none"> ■ Debug — suitable for scripts still being tested; this level shows all debug, audit, error and emergency messages. ■ Audit, Error, or Emergency — suitable for scripts in production mode. Of these three, Audit is the most inclusive and Emergency the most exclusive. |
| Execute As Role | Indicates the role used to run the script. |
| Available Without Login | Indicates if users without an active NetSuite session can access the Suitelet . See Setting Available Without Login . |

Setting Available Without Login

When you select Available Without Login and then save the Script Deployment record, an External URL appears on the Script Deployment page. Use this URL for Suitelets you want to make available to users who do not have an active NetSuite session.

Only a subset of the SuiteScript API is supported in externally available Suitelets (Suitelets set to Available Without Login on the Script Deployment page). Note that if you want to use all available SuiteScript APIs in a Suitelet, your Suitelet will require a valid NetSuite session. (A valid session means that users have authenticated to NetSuite by providing their email address and password.)

The Website feature must be enabled for Clients Scripts to work in externally available Suitelets

 **Note:** For NetSuite 2019.1, two fields display: **External URL (Deprecated)** uses a forms.netsuite.com domain that will no longer be supported as of NetSuite 2020.1. The **External URL** field displays the account-specific domain, <accountID>.extforms.netsuite.com, which is supported for 2019.1 and future releases. For more information about NetSuite domains, see the help topic [Understanding NetSuite URLs](#).

The following are a few uses cases that address when you might want to make a Suitelet externally available:

- Hosting one-off online forms, such as capturing partner conference registrations.
- Inbound partner communication, such as listening for payment notification responses from PayPal or Google checkout; or for generating the unsubscribe from email campaigns page, which requires access to account information but should not require a login or hosted website.
- For Facebook, Google, and Yahoo mashups in which the Suitelet lives in those websites but needs to communicate to NetSuite via POST requests.

For access or redirection from another script to a Suitelet, the best practice is to use `url.resolveDomain(options)` to discover the URL instead of hard- coding the URL.

 **Note:** Suitelets are not intended for use in systems integration use cases.

 **Important:** Because there are no login requirements for Suitelet that are available without login, be aware that the data contained within the Suitelet will be less secure.

Errors Related to the Available Without Login URL

You will use either the internal URL or the external URL as the launching point for a Suitelet.

Some factors that determine whether a Suitelet will successfully deploy are:

- Dependencies between the type of URL you are referencing (internal or external)
- The Suitelet deployment status (Testing or Released)
- Whether the Select All checkbox has been selected on the Audience subtab of the Script Deployment Page.

The following table summarizes these dependencies:

| Suitelet URL Type | Deployment Status | Select All check boxes | Result |
|-------------------|-------------------|------------------------|---|
| internal | Testing | not checked | Suitelet deploys successfully |
| internal | Testing | checked | Suitelet deploys successfully |
| internal | Released | not checked | Error message: You do not have privileges to view this page. |
| internal | Released | checked | Suitelet deploys successfully |
| external | Testing | not checked | Error message: You are not allowed to navigate directly to this page. |
| external | Testing | checked | Error message: You are not allowed to navigate directly to this page. |
| external | Released | checked | Suitelet deploys successfully |
| external | Released | not checked | Error message: You do not have privileges to view this page. |

Suitelet Script Deployment Page Audience Subtab

Use the Suitelet Script Deployment page's Audience subtab to specify the roles that can access your Suitelet.

The following table summarizes the Audience subtab fields.

| Field | Description |
|-------------|---|
| Roles | Select the Roles that can access your Suitelet. To give access to all Roles, check the All Roles box. If you do not select any Roles, the script will not run. |
| Groups | Select the Groups that can access your Suitelet. If you do not select any Groups, all Groups have access. |
| Partners | Select the Partners that can access your Suitelet. To give access to all Partners, check the All Partners box. |
| Departments | Select the Departments that can access your Suitelet. If you do not select any Departments, all Departments have access. |
| Employees | Select the Employees that can access your Suitelet. To give access to all Employees, check the All Employees box. |

Suitelet Script Deployment Page Links Subtab

Use the Suitelet Script Deployment page's Links subtab to create links in NetSuite Centers to your Suitelet. For example, create a link to a Suitelet from the Support Section in the Classic Center.

The following table summarizes the Links subtab fields.

| Field | Description |
|--------|--|
| Center | The NetSuite Center's from which users can access your Suitelet. |

| | |
|---------------|--|
| Section | The NetSuite section of your Suitelet. Users access your Suitelet from this section. For example, Support or Documents. |
| Category | The name of the Suitelet. Users see this label in the section menu, after the category name. For example, Support > Suitelet Category Name > Your Suitelet Name. |
| Label | The name of the Suitelet. Users see this label in the section menu, after the category name. For example, Support > Suitelet Category Name > Your Suitelet Name. |
| Translation | If you have the multi-language feature enabled for your NetSuite account, you can add translations for your Suitelet label in the Translation column. |
| Insert Before | Specify the section category you would like to insert your Suitelet before. |

SuiteScript 2.x Embedding HTML in Suitelets

The following examples illustrate how to embed HTML within Suitelet code to add custom elements to a form. Embedding HTML in Suitelets is useful for adding components that are not available through the SuiteScript N/ui/serverWidget module.

- [Embedding Inline HTML in a Field](#)
- [Embedding HTML from a Linked HTML Page in a Suitelet](#)

Embedding Inline HTML in a Field

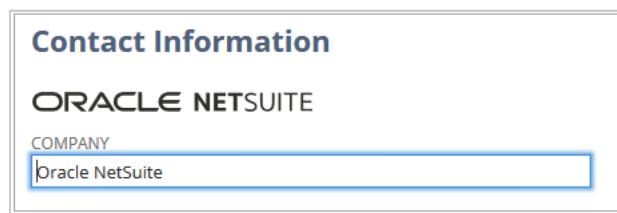
In a Suitelet, you can use [Form.addField\(options\)](#) to add inline HTML as a field on a form. Specify the type as INLINEHTML and use the [Field.defaultValue](#) property to set the HTML value. For more information, see the help topic [N/ui/serverWidget Module](#).

```

1 var htmlImage = form.addField({
2   id: 'custpage_htmlfield',
3   type: serverWidget.FieldType.INLINEHTML,
4   label: 'HTML Image'
5 });
6 htmlImage.defaultValue = "<img src='https://<accountID>.app.netsuite.com/images/logos/netsuite-oracle.svg' alt='Oracle Netsuite
  logo'>";

```

The following screenshot shows a form with an INLINEHTML field containing an image field.



Embedding HTML from a Linked HTML Page in a Suitelet

In a Suitelet, you can use [https.get\(options\)](#) from the N/https module to embed HTML from a linked HTML page in the form created by the Suitelet. The Suitelet manages the data submitted by users to the HTML page. When you use a linked HTML page, you manage HTML code through strings. As a result, it can be harder to manipulate data than it is with components created with N/serverWidget module methods. To pass data through the string containing the HTML code, your Suitelet code must change values within the string. This string becomes increasing complex as the number of values increases. For more information, see the help topic [N/https Module](#).

The following example creates a simple volunteer sign-up sheet. It consists of two parts: the Suitelet code and the HTML. This Suitelet code uses the [https.get\(options\)](#) method from the N/https module to access

the content from the HTML page. It uses methods from the N/record module, N/email module, and N/search module to collect, process, and respond to user-submitted data.

```

1  /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4  */
5
6 define(['N/https', 'N/record', 'N/email', 'N/search'],
7     function callbackFunction(https, record, email, search) {
8         function getFunction(context) {
9             var contentRequest = https.get({
10                 url: "https://LinkToFormPage.html" // see the next snippet
11             });
12             var contentDocument = contentRequest.body;
13             var sponsorid = context.request.parameters.sponsorid;
14
15             if (sponsorid && sponsorid != "" && sponsorid != null) {
16                 contentDocument = contentDocument.replace("{${sponsorid}}", sponsorid);
17                 log.debug("Setting Sponsor", sponsorid)
18             }
19
20             var projectid = context.request.parameters.projectid;
21
22             if (projectid && projectid != "" && projectid != null) {
23                 contentDocument = contentDocument.replace("{${projectid}}", projectid);
24                 log.debug("Setting Project", projectid);
25             }
26
27             context.response.write(contentDocument);
28         }
29
30         function postFunction(context) {
31             var params = context.request.parameters;
32             var emailString = "First Name: {{firstname}}\nLast Name: {{lastname}}\nEmail: {{email}}\nFacebook URL: {{custentity_fb_url}}"
33             var contactRecord = record.create({
34                 type: "contact",
35                 isDynamic: true
36             });
37
38             for (param in params) {
39                 if (param === "company") {
40                     if (params[param] !== "{${sponsorid}}") {
41                         contactRecord.setValue({
42                             fieldId: param,
43                             value: params[param]
44                         });
45                         var lkpfld = search.lookupFields({
46                             type: "vendor",
47                             id: params["company"],
48                             columns: ["entityid"]
49                         });
50                         emailString += "\nSponsor: " + lkpfld.entityid;
51                     }
52                 else {
53                     contactRecord.setValue({
54                         fieldId: "custentity_sv_shn_isindi",
55                         value: true
56                     })
57                 }
58             }
59             else {
60                 if (param !== "project") {
61                     contactRecord.setValue({
62                         fieldId: param,
63                         value: params[param]
64                     });
65                     var replacer = "(" + param + ")";
66                     emailString = emailString.replace(replacer, params[param]);
67                 }
68             }
69         }
70     }
71 }
```

```

69     }
70
71     var contactId = contactRecord.save({
72         ignoreMandatoryFields: true,
73         enableSourcing: true
74     });
75
76     log.debug("Record ID", contactId);
77
78     if (params["project"] && params["project"] !== "" && params["project"] != null && params["project"] != "{projectid}") {
79         var lkpfld = search.lookupFields({
80             type: "job",
81             id: params["project"],
82             columns: ["companyname"]
83         });
84
85         emailString += "\nProject Name: " + lkpfld.companyname;
86
87         var participationRec = record.create({
88             type: "customrecord_project_participants",
89             isDynamic: true
90         });
91
92         participationRec.setValue({
93             fieldId: "custrecord_participants_volunteer",
94             value: contactId
95         });
96
97         participationRec.setValue({
98             fieldId: "custrecord_participants_project",
99             value: params["project"]
100        });
101
102         var participationId = participationRec.save({
103             enableSourcing: true,
104             ignoreMandatoryFields: true
105         });
106     }
107
108     log.debug("Email String", emailString);
109
110     email.send({
111         author: -5,
112         recipients: 256,
113         subject: "New Volunteer Signed Up",
114         body: "A new volunteer has joined:\n\n" + emailString
115     });
116
117     email.send({
118         author: -5,
119         recipients: params["email"],
120         subject: "Thank you!",
121         body: "Thank you for volunteering:\n\n" + emailString
122     });
123
124     var contentRequest = https.get({
125         url: "https://LinkToFormCompletePage.html"
126     });
127
128
129     var contentDocument = contentRequest.body;
130
131     context.response.write(contentDocument);
132
133 }
134 function onRequestFxn(context) {
135     if (context.request.method === "GET") {
136         getFunction(context)
137     }
138     else {
139         postFunction(context)
140     }
141

```

```

142     }
143     return {
144       onRequest: onRequestFxn
145     });
146   });

```

The following HTML, contained in the LinkToFormPage.html file (see above snippet), creates the form that the Suitelet links to.

```

1 <form method="post" class="form-horizontal" action="https://LinkToSuitelet.js">
2   <table>
3     <tbody>
4       <tr>
5         <td>First Name</td>
6         <td class="col-md-8"><input class="form-control" id="firstname" placeholder="First Name" name="firstname" required="" type="text"></td>
7       </tr>
8       <tr>
9         <td>Last Name</td>
10        <td class="col-md-8"><input class="form-control" id="lastname" placeholder="Last Name" name="lastname" required="" type="text"></td>
11      </tr>
12      <tr>
13        <td>Email</td>
14        <td class="col-md-8"><input class="form-control" id="email" placeholder="email" name="email" required="" type="email"></td>
15      </tr>
16      <tr>
17        <td>Facebook URL</td>
18        <td class="col-md-8"><input class="form-control" id="custentity_fb_url" placeholder="Facebook" name="custentity_fb_url" required="" type="text"></td>
19      </tr>
20      <tr>
21        <td><input name="company" value="{{sponsorid}}></td>
22        <td><input name="project" value="{{projectid}}></td>
23      </tr>
24    </tbody></table>
25    <br>
26    <button type="submit" class="btn btn-inverse">Sign Up as a Volunteer</button>
27  </form>

```

The following screenshot illustrates the form that is created by the linked HTML page.

The screenshot shows a simple HTML form with four text input fields and one button. The fields are labeled 'First Name', 'Last Name', 'Email', and 'Facebook URL'. Each label is followed by a text input box. Below the form is a large blue button with the text 'Sign Up as a Volunteer'.

SuiteScript 2.x Suitelet Script Entry Points and API

- [onRequest\(params\)](#)

onRequest(params)

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|--|
| Description | Defines the Suitelet script trigger point. |
| Returns | void |
| Since | Version 2015 Release 2 |

Parameters

 **Note:** The params parameter is a JavaScript object.

| Parameter | Type | Required / Optional | Description | Since |
|-----------------|-------------------------------------|---------------------|------------------------|------------------------|
| params.request | http.ServerRequest | required | The incoming request. | Version 2015 Release 2 |
| params.response | http.ServerResponse | required | The Suitelet response. | Version 2015 Release 2 |

SuiteScript 2.x Suitelet Script Type Code Samples

This topic contains examples of different types of Suitelets: Suitelets used to create custom pages (forms, list or assistants), and backend Suitelets.

For help with writing scripts in SuiteScript 2.x, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).

- [Basic Samples](#)
 - [Write Your First Suitelet](#)
 - [Return a Simple XML Document](#)
 - [Add a Suitelet to a Tab](#)
- [Custom Forms](#)
 - [Write and Send Email](#)
 - [Include an Inline Editor Sublist](#)
 - [Create a Survey](#)
 - [Create a Form Containing Several Field Types, Reset and Submit Buttons, Tabs, and a Sublist](#)
 - [Create and Use Secret Keys](#)
 - [Generate Credential Field](#)
 - [Add a Secret Key Field](#)
 - [Add a Field that Displays Running Total to a Sublist](#)
 - [Parse Strings and Display Result](#)
- [Custom Lists](#)
 - [Create a Custom List](#)
- [Custom Assistants](#)

- Sample Custom Assistant Script
- Suitelets with Embedded HTML
- Backend Suitelets
 - Find Plugin Implementations
 - Generate a SuiteSignOn Token
 - Load an XML File and Obtain Child Element Values
 - Parse an XML File and Log Element Values
 - Redirect to a New Sales Order
 - Render a PDF
 - Render Search Results into a PDF File
 - Retrieve the Name of the City Based on a ZIP Code
 - Write User and Session Information to the Response

Basic Samples

- Write Your First Suitelet
- Return a Simple XML Document
- Add a Suitelet to a Tab

Write Your First Suitelet

i Note: This script sample uses the `define` function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the `require` function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

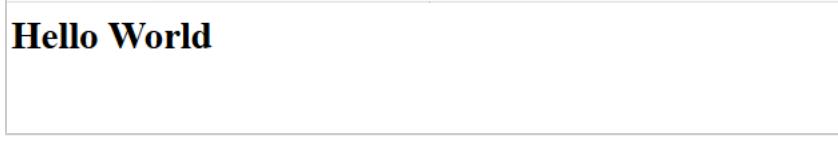
The following sample creates a Suitelet that shows an empty page with a Hello World header.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4  */
5 define([], function() {
6   function onRequest(context) {
7     var html = '<html><body><h1>Hello World</h1></body></html>';
8     context.response.write(html);
9     context.response.setHeader({
10       name: 'Custom-Header-Demo',
11       value: 'Demo'
12     });
13   }
14
15   return {
16     onRequest: onRequest
17   };
18 });

```

The following screenshot shows the output of this script.



Hello World

Return a Simple XML Document



Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see SuiteScript 2.0 Global Objects.

The following sample creates a Suitelet that returns a simple XML document.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define([], function() {
6     function onRequest(context) {
7         var xml = '<?xml version="1.0" encoding="utf-8"?>' +
8             '<message>'+'Hello World'+'</message>';
9         context.response.write(xml);
10        context.response.setHeader({
11            name: 'Custom-Header-Demo',
12            value: 'Demo'
13        });
14    }
15
16    return {
17        onRequest: onRequest
18    };
19 });

```

The following screenshot shows the output of this script.

Add a Suitelet to a Tab



Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript Debugger](#).

The following sample shows how to add a link to a Suitelet to a tab in a user event script.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType UserEventScript
4 */
5 define(['N/record', 'N/runtime', 'N/ui/serverWidget', 'N/url'], function(record, runtime, serverWidget, url) {
6     function beforeLoad(scriptContext) {
7         var currentUserID = runtime.getCurrentUser().id;
8         /* If the record is edited or viewed, a tab called Sample Tab is added.
9          * Note that the script execution context is set to userinterface.
10         * This ensures that this script is ONLY invoked from a user event
11         * occurring through the UI.
12         */
13         if ((runtime.executionContext === runtime.ContextType.USER_INTERFACE) && (scriptContext.type === scriptContext.UserEventType.EDIT || scriptContext.type === scriptContext.UserEventType.VIEW)) {
14             //Creates the new tab Sample Tab on the form
15             var SampleTab = scriptContext.form.addTab({

```

```

16     id: 'custpage_sample_tab',
17     label: 'Sample Tab'
18   });
19
20   //On Sample Tab, create a field of type inlinehtml
21   var createNewReqLink = scriptContext.form.addField({
22     id: 'custpage_new_req_link',
23     type: 'inlinehtml',
24     label: '',
25     container: 'custpage_sample_tab'
26   );
27
28   //Define the parameters of the Suitelet that will be executed
29   //Replace the scriptId and the deploymentId with the values of your Suitelet
30   var linkURL = url.resolveScript({
31     scriptId: 'customscriptsuiteletsample_yourfirstsamp',
32     deploymentId: 'customdeploysuiteletsample_yourfirstsamp'
33   }) + '&recordid=' + scriptContext.newRecord.id;
34
35   //Create a link to launch the Suitelet.
36   createNewReqLink.defaultValue = '<B>Click <A href=' + linkURL +'>here</A> to create a new document signature
request record.</B>';
37
38   //Add a sublist to Sample Tab
39   var signatureRequestSublist = scriptContext.form.addSublist({
40     id: 'custpage_sig_req_sublist',
41     type: 'list',
42     label: 'Document Signature Requests',
43     tab: 'custpage_sample_tab'
44   );
45   signatureRequestSublist.addField({
46     id: 'custpage_req_name',
47     type: 'text',
48     label: 'Name'
49   );
50   signatureRequestSublist.addField({
51     id: 'custpage_req_status',
52     type: 'text',
53     label: 'Status'
54   );
55   signatureRequestSublist.addField({
56     id: 'custpage_req_created',
57     type: 'date',
58     label: 'Date Created'
59   );
60 }
61
62 return {
63   beforeLoad: beforeLoad,
64 };
65 });
66 });

```

Custom Forms

- Write and Send Email
- Include an Inline Editor Sublist
- Create a Survey
- Create a Form Containing Several Field Types, Reset and Submit Buttons, Tabs, and a Sublist
- Create and Use Secret Keys
- Generate Credential Field
- Add a Secret Key Field
- Add a Field that Displays Running Total to a Sublist
- Parse Strings and Display Result

Write and Send Email

i Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript Debugger](#).

The following sample creates a Suitelet form that lets you write and send an email.

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType Suitelet
4   */
5 /**
6  * A simple Suitelet for building an email form and sending out an email
7  * from the current user to the recipient email address specified on the form.
8  */
9 define(['N/email', 'N/runtime', 'N/ui/serverWidget'], function(email, runtime, serverWidget) {
10     function onRequest(context) {
11         if (context.request.method === 'GET') {
12             var form = serverWidget.createForm({
13                 title: 'Email Form'
14             });
15
16             var subject = form.addField({
17                 id: 'subject',
18                 type: serverWidget.FieldType.TEXT,
19                 label: 'Subject'});
20             subject.layoutType = serverWidget.FieldLayoutType.NORMAL;
21             subject.updateBreakType = serverWidget.FieldBreakType.STARTCOL;
22             subject.isMandatory = true;
23
24             var recipient = form.addField({
25                 id: 'recipient',
26                 type: serverWidget.FieldType.EMAIL,
27                 label: 'Recipient email'
28             });
29             recipient.isMandatory = true;
30
31             var message = form.addField({
32                 id: 'message',
33                 type: serverWidget.FieldType.TEXTAREA,
34                 label: 'Message'
35             });
36             message.updateDisplaySize({
37                 height: 10,
38                 width : 60
39             });
40
41             form.addSubmitButton({
42                 label: 'Send Email'
43             });
44
45             context.response.writePage(form);
46         }
47     else
48     {
49         var request = context.request;
50         var currentuser = runtime.getCurrentUser().id;
51         var subject = request.parameters.subject;
52         var recipient = request.parameters.recipient;
53         var message = request.parameters.message;
54         email.send({
55             author: currentuser,
56             recipients: recipient,
57             subject: subject,
58             body: message
59         });
60     }
61 }

```

```

62 |
63     return {
64         onRequest: onRequest
65     };
66 });

```

Include an Inline Editor Sublist

Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

Important: This sample uses SuiteScript 2.1. For more information, see [SuiteScript 2.1](#).

The following sample creates a Suitelet that generates a sample form with a submit button, fields, and an inline editor sublist.

```

1 /**
2  * @NApiVersion 2.1
3  * @NScriptType Suitelet
4 */
5 define(['N/ui/serverWidget'], (serverWidget) => {
6     const onRequest = (scriptContext) => {
7         if (scriptContext.request.method === 'GET') {
8             let form = serverWidget.createForm({
9                 title: 'Simple Form'
10            });
11
12            let field = form.addField({
13                id: 'textfield',
14                type: serverWidget.FieldType.TEXT,
15                label: 'Text'
16            });
17            field.layoutType = serverWidget.FieldLayoutType.NORMAL;
18            field.updateBreakType({
19                breakType: serverWidget.FieldBreakType.STARTCOL
20            });
21
22            form.addField({
23                id: 'datefield',
24                type: serverWidget.FieldType.DATE,
25                label: 'Date'
26            });
27            form.addField({
28                id: 'currencyfield',
29                type: serverWidget.FieldType.CURRENCY,
30                label: 'Currency'
31            });
32
33            let select = form.addField({
34                id: 'selectfield',
35                type: serverWidget.FieldType.SELECT,
36                label: 'Select'
37            });
38            select.addSelectOption({
39                value: 'a',
40                text: 'Albert'
41            );
42            select.addSelectOption({
43                value: 'b',
44                text: 'Baron'
45            );
46
47            let sublist = form.addSublist({
48                id: 'sublist',
49                type: serverWidget.SublistType.INLINEEDITOR,
50                label: 'Inline Editor Sublist'
51            });
52        }
53    }
54 }
55 
```

```

51    });
52    sublist.addField({
53      id: 'sublist1',
54      type: serverWidget.FieldType.DATE,
55      label: 'Date'
56    });
57    sublist.addField({
58      id: 'sublist2',
59      type: serverWidget.FieldType.TEXT,
60      label: 'Text'
61    });
62
63    form.addSubmitButton({
64      label: 'Submit Button'
65    });
66
67    scriptContext.response.writePage(form);
68 } else {
69   const delimiter = /\u0001/;
70   const textField = scriptContext.request.parameters.textfield;
71   const dateField = scriptContext.request.parameters.datefield;
72   const currencyField = scriptContext.request.parameters.currencyfield;
73   const selectField = scriptContext.request.parameters.selectfield;
74   const sublistData = scriptContext.request.parameters.sublistdata.split(delimiter);
75   const sublistField1 = sublistData[0];
76   const sublistField2 = sublistData[1];
77
78   scriptContext.response.write(`You have entered: ${textField} ${dateField} ${currencyField} ${selectField} ${sublist
Field1} ${sublistField2}`);
79 }
80
81 return {onRequest}
82 });
83

```

Create a Survey

i Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

The following sample creates a Suitelet that generates a customer survey form with inline HTML fields, radio fields, and a submit button.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define(['N/ui/serverWidget'], function(serverWidget) {
6   function onRequest(context) {
7     var form = serverWidget.createForm({
8       title: 'Thank you for your interest in Wolfe Electronics',
9       hideNavBar: true
10    });
11
12    var htmlHeader = form.addField({
13      id: 'custpage_header',
14      type: serverWidget.FieldType.INLINEHTML,
15      label: ''
16    }).updateLayoutType({
17      layoutType: serverWidget.FieldLayoutType.OUTSIDEABOVE
18    }).updateBreakType({
19      breakType: serverWidget.FieldBreakType.STARTROW
20    }).defaultValue = '<p style=\`font-size:20px\`>We pride ourselves on providing the best' +
21      ' services and customer satisfaction. Please take a moment to fill out our survey.</p><br><br>';
22
23    var htmlInstruct = form.addField({
24      id: 'custpage_p1',

```

```

25     type: serverWidget.FieldType.INLINEHTML,
26     label: ''
27   }).updateLayoutType({
28     layoutType: serverWidget.FieldLayoutType.OUTSIDEABOVE
29   }).updateBreakType({
30     breakType: serverWidget.FieldBreakType.STARTROW
31   }).defaultValue = '<p style=\''font-size:14px\'>When answering questions on a scale of 1 to 10, +  

32   ' 1 = Greatly Unsatisfied and 10 = Greatly Satisfied.</p><br><br>';
33
34   var productRating = form.addField({
35     id: 'custpage_lblproductrating',
36     type: serverWidget.FieldType.INLINEHTML,
37     label: ''
38   }).updateLayoutType({
39     layoutType: serverWidget.FieldLayoutType.NORMAL
40   }).updateBreakType({
41     breakType: serverWidget.FieldBreakType.STARTROW
42   }).defaultValue = '<p style=\''font-size:14px\'>How would you rate your satisfaction with our products?</p>';
43
44   form.addField({
45     id: 'custpage_rdoproducing',
46     type: serverWidget.FieldType.RADIO,
47     label: '1',
48     source: 'p1'
49   }).updateLayoutType({
50     layoutType: serverWidget.FieldLayoutType.STARTROW
51   });
52   form.addField({
53     id: 'custpage_rdoproducing',
54     type: serverWidget.FieldType.RADIO,
55     label: '2',
56     source: 'p2'
57   }).updateLayoutType({
58     layoutType: serverWidget.FieldLayoutType.MIDROW
59   });
60   form.addField({
61     id: 'custpage_rdoproducing',
62     type: serverWidget.FieldType.RADIO,
63     label: '3',
64     source: 'p3'
65   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
66   form.addField({
67     id: 'custpage_rdoproducing',
68     type: serverWidget.FieldType.RADIO,
69     label: '4',
70     source: 'p4'
71   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
72   form.addField({
73     id: 'custpage_rdoproducing',
74     type: serverWidget.FieldType.RADIO,
75     label: '5',
76     source: 'p5'
77   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
78   form.addField({
79     id: 'custpage_rdoproducing',
80     type: serverWidget.FieldType.RADIO,
81     label: '6',
82     source: 'p6'
83   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
84   form.addField({
85     id: 'custpage_rdoproducing',
86     type: serverWidget.FieldType.RADIO,
87     label: '7',
88     source: 'p7'
89   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
90   form.addField({
91     id: 'custpage_rdoproducing',
92     type: serverWidget.FieldType.RADIO,
93     label: '8',
94     source: 'p8'
95   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
96   form.addField({
97     id: 'custpage_rdoproducing',

```

```

98     type: serverWidget.FieldType.RADIO,
99     label: '9',
100    source: 'p9'
101   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
102   form.addField({
103     id: 'custpage_rdoprodproductrating',
104     type: serverWidget.FieldType.RADIO,
105     label: '10',
106     source: 'p10'
107   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.ENDROW});

108
109   var serviceRating = form.addField({
110     id: 'custpage_lblservicerating',
111     type: serverWidget.FieldType.INLINEHTML,
112     label: ''
113   }).updateLayoutType({
114     layoutType: serverWidget.FieldLayoutType.NORMAL
115   }).updateBreakType({
116     breakType: serverWidget.FieldBreakType.STARTROW
117   }).defaultValue = '<p style=\'font-size:14px\'>How would you rate your satisfaction with our services?</p>';
118
119   form.addField({
120     id: 'custpage_rdoservicerating',
121     type: serverWidget.FieldType.RADIO,
122     label: '1',
123     source: 'p1'
124   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.STARTROW});
125   form.addField({
126     id: 'custpage_rdoservicerating',
127     type: serverWidget.FieldType.RADIO,
128     label: '2',
129     source: 'p2'
130   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
131   form.addField({
132     id: 'custpage_rdoservicerating',
133     type: serverWidget.FieldType.RADIO,
134     label: '3',
135     source: 'p3'
136   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
137   form.addField({
138     id: 'custpage_rdoservicerating',
139     type: serverWidget.FieldType.RADIO,
140     label: '4',
141     source: 'p4'
142   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
143   form.addField({
144     id: 'custpage_rdoservicerating',
145     type: serverWidget.FieldType.RADIO,
146     label: '5',
147     source: 'p5'
148   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
149   form.addField({
150     id: 'custpage_rdoservicerating',
151     type: serverWidget.FieldType.RADIO,
152     label: '6',
153     source: 'p6'
154   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
155   form.addField({
156     id: 'custpage_rdoservicerating',
157     type: serverWidget.FieldType.RADIO,
158     label: '7',
159     source: 'p7'
160   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
161   form.addField({
162     id: 'custpage_rdoservicerating',
163     type: serverWidget.FieldType.RADIO,
164     label: '8',
165     source: 'p8'
166   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
167   form.addField({
168     id: 'custpage_rdoservicerating',
169     type: serverWidget.FieldType.RADIO,
170     label: '9',
171   })

```

```

171     source: 'p9'
172   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.MIDROW});
173   form.addField({
174     id: 'custpage_rdoservicerating',
175     type: serverWidget.FieldType.RADIO,
176     label: '10',
177     source: 'p10'
178   }).updateLayoutType({layoutType: serverWidget.FieldLayoutType.ENDROW});

179   form.addSubmitButton({
180     label: 'Submit'
181   });

182   context.response.writePage(form);
183 }

184 return {
185   onRequest: onRequest
186 };
187 });
188 });
189 });
190 });

```

Create a Form Containing Several Field Types, Reset and Submit Buttons, Tabs, and a Sublist

i Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript Debugger](#).

The following code creates a Suitelet that generates a custom form containing several field types, tabs, a sublist, and a submit button. For steps to create this script, see [Sample Custom Form Script](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define(['N/ui/serverWidget'], function(serverWidget) {
6   function onRequest(context) {
7     if (context.request.method === 'GET') {
8
9       //Section One - Forms - See 'Steps for Creating a Custom Form' in topic 'Sample Custom Form Script'
10      var form = serverWidget.createForm({
11        title: 'Customer Information'
12      });
13
14      var usergroup = form.addFieldGroup({
15        id: 'usergroup',
16        label: 'User Information'
17      });
18      usergroup.isSingleColumn = true;
19
20      var companygroup = form.addFieldGroup({
21        id: 'companygroup',
22        label: 'Company Information'
23      });
24
25      var select = form.addField({
26        id: 'titlefield',
27        type: serverWidget.FieldType.SELECT,
28        label: 'Title',
29        container: 'usergroup'
30      });
31      select.addSelectOption({
32        value: 'Mr.',
33        text: 'Mr.'
34      );
35      select.addSelectOption({
36        value: 'MS.',

```

```

37     text: 'Ms.'
38 });
39 select.addSelectOption({
40   value: 'Dr.',
41   text: 'Dr.'
42 });
43
44 var fname = form.addField({
45   id: 'fnamefield',
46   type: serverWidget.FieldType.TEXT,
47   label: 'First Name',
48   container: 'usergroup'
49 });
50 fname.isMandatory = true;
51
52 var lname = form.addField({
53   id: 'lnamefield',
54   type: serverWidget.FieldType.TEXT,
55   label: 'Last Name',
56   container: 'usergroup'
57 });
58 lname.isMandatory = true;
59
60 form.addField({
61   id: 'emailfield',
62   type: serverWidget.FieldType.EMAIL,
63   label: 'Email',
64   container: 'usergroup'
65 });
66
67 var companyname = form.addField({
68   id: 'companyfield',
69   type: serverWidget.FieldType.TEXT,
70   label: 'Company',
71   container: 'companygroup'
72 });
73 companyname.defaultValue = 'Company Name';
74
75 form.addField({
76   id: 'phonefield',
77   type: serverWidget.FieldType.PHONE,
78   label: 'Phone Number',
79   container: 'companygroup'
80 });
81
82 form.addField({
83   id: 'urlfield',
84   type: serverWidget.FieldType.URL,
85   label: 'Website',
86   container: 'companygroup'
87 });
88
89 form.addSubmitButton({
90   label: 'Submit'
91 });
92
93 // Section Two - Tabs - See 'Steps for Adding a Tab to a Form' in topic 'Sample Custom Form Script'
94 var tab1 = form.addTab({
95   id: 'tab1id',
96   label: 'Payment'
97 });
98 tab1.helpText = 'Help Text Goes Here';
99
100 var tab2 = form.addTab({
101   id: 'tab2id',
102   label: 'Inventory'
103 });
104
105 form.addSubtab({
106   id: 'subtab1id',
107   label: 'Payment Information',
108   tab: 'tab1id'
109 });

```

```

110
111     form.addSubtab({
112         id: 'subtab2id',
113         label: 'Transaction Record',
114         tab: 'tab1id'
115     });
116
117     // Subtab One Fields
118     var ccselect = form.addField({
119         id: 'cctypefield',
120         type: serverWidget.FieldType.SELECT,
121         label: 'Credit Card',
122         container: 'subtab1id'
123     });
124     ccselect.addSelectOption({
125         value: 'PayCard0',
126         text: 'Payment Card 0'
127     });
128     ccselect.addSelectOption({
129         value: 'PayCard1',
130         text: 'Payment Card 1'
131     });
132     ccselect.addSelectOption({
133         value: 'PayCard2',
134         text: 'Payment Card 2'
135     });
136
137     var expmonth = form.addField({
138         id: 'expmonth',
139         type: serverWidget.FieldType.SELECT,
140         label: 'Expiry Date:',
141         container: 'subtab1id'
142     });
143     expmonth.updateLayoutType({
144         layoutType: serverWidget.FieldLayoutType.STARTROW
145     });
146     expmonth.addSelectOption({
147         value: '01',
148         text: 'Jan'
149     });
150     expmonth.addSelectOption({
151         value: '02',
152         text: 'Feb'
153     });
154     expmonth.addSelectOption({
155         value: '03',
156         text: 'Mar'
157     });
158     expmonth.addSelectOption({
159         value: '04',
160         text: 'Apr'
161     });
162     expmonth.addSelectOption({
163         value: '05',
164         text: 'May'
165     });
166     expmonth.addSelectOption({
167         value: '06',
168         text: 'Jun'
169     });
170     expmonth.addSelectOption({
171         value: '07',
172         text: 'Jul'
173     });
174     expmonth.addSelectOption({
175         value: '08',
176         text: 'Aug'
177     });
178     expmonth.addSelectOption({
179         value: '09',
180         text: 'Sep'
181     });
182     expmonth.addSelectOption({

```

```

183     value: '10',
184     text: 'Oct'
185   });
186   expmonth.addSelectOption({
187     value: '11',
188     text: 'Nov'
189   });
190   expmonth.addSelectOption({
191     value: '12',
192     text: 'Dec'
193   });
194
195   var expyear = form.addField({
196     id: 'expyear',
197     type: serverWidget.FieldType.SELECT,
198     label: 'Expiry Year',
199     container: 'subtabId'
200   });
201   expyear.updateLayoutType({
202     layoutType: serverWidget.FieldLayoutType.ENDROW
203   });
204   expyear.addSelectOption({
205     value: '2020',
206     text: '2020'
207   });
208   expyear.addSelectOption({
209     value: '2019',
210     text: '2019'
211   });
212   expyear.addSelectOption({
213     value: '2018',
214     text: '2018'
215   });
216
217   var credfield = form.addCredentialField({
218     id: 'credfield',
219     label: ' Credit Card Number',
220     restrictToDomains: 'www.mysite.com',
221     restrictToScriptIds: 'customscript_my_script',
222     restrictToCurrentUser: false,
223     container: 'subtabId'
224   });
225   credfield.maxLength = 32;
226
227 // Subtab two Fields
228 form.addField({
229   id: 'transactionfield',
230   type: serverWidget.FieldType.LABEL,
231   label: 'Transaction History - Coming Soon',
232   container: 'subtab2Id'
233 });
234
235 // Tab Two Fields
236 form.addField({
237   id: 'inventoryfield',
238   type: serverWidget.FieldType.LABEL,
239   label: 'Inventory - Coming Soon',
240   container: 'tab2Id'
241 });
242
243 //Section Three - Sublist - See 'Steps for Adding a Sublist to a Form' in topic 'Sample Custom Form Script'
244 var sublist = form.addSublist({
245   id: 'sublistid',
246   type: serverWidget.SublistType.INLINEEDITOR,
247   label: 'Inline Sublist',
248   tab: 'tab2Id'
249 });
250 sublist.addButton({
251   id: 'buttonId',
252   label: 'Print',
253   functionName: '' //Add the function triggered on button click
254 });
255

```

```

256 //Sublist Fields
257 sublist.addField({
258   id: 'datefieldid',
259   type: serverWidget.FieldType.DATE,
260   label: 'Date'
261 });
262
263 sublist.addField({
264   id: 'productfieldid',
265   type: serverWidget.FieldType.TEXT,
266   label: 'Product'
267 });
268
269 sublist.addField({
270   id: 'qtyfieldid',
271   type: serverWidget.FieldType.INTEGER,
272   label: 'Quantity'
273 });
274
275 sublist.addField({
276   id: 'upfieldid',
277   type: serverWidget.FieldType.CURRENCY,
278   label: 'Unit Cost'
279 });
280
281 context.response.writePage(form);
282 } else {
283   //Section Four - Output - Used in all sections
284   var delimiter = /\u0001/;
285   var titleField = context.request.parameters.titlefield;
286   var fnameField = context.request.parameters.fnamefield;
287   var lnameField = context.request.parameters.lnamefield;
288   var emailField = context.request.parameters.emailfield;
289   var companyField = context.request.parameters.companyfield;
290   var phoneField = context.request.parameters.phonefield;
291   var urlField = context.request.parameters.urlfield;
292   var ccField = context.request.parameters.cctypefield;
293   var ccNumber = context.request.parameters.credfield;
294   var expMonth = context.request.parameters.expmonth;
295   var expYear = context.request.parameters.expyear;
296
297   context.response.write('You have entered:'
298     + '<br/> Name: ' + titleField + ' ' + fnameField + ' ' + lnameField
299     + '<br/> Email: ' + emailField
300     + '<br/> Company: ' + companyField
301     + '<br/> Phone: ' + phoneField + ' Website: ' + urlField
302     + '<br/> Credit Card: ' + ccField
303     + '<br/> Number: ' + ccNumber
304     + '<br/> Expiry Date: ' + expMonth + '/' + expYear);
305 }
306
307 return {
308   onRequest: onRequest
309 };
310 });

```

Create and Use Secret Keys



Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

The following sample creates a simple Suitelet that requests user credentials, creates a secret key, and encodes a sample string.



Note: The default maximum length for a secret key field is 32 characters. If needed, use the `Field.maxLength` property to change this value.

```

1  /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4  */
5 define(['N/ui/serverWidget', 'N/runtime', 'N/crypto', 'N/encode'], (ui, runtime, crypto, encode) => {
6     function onRequest(option) {
7         if (option.request.method === 'GET') {
8             let form = ui.createForm({
9                 title: 'My Credential Form'
10            });
11            let skField = form.addSecretKeyField({
12                id: 'mycredential',
13                label: 'Credential',
14                restrictToScriptIds: [runtime.getCurrentScript().id],
15                restrictToCurrentUser: false
16            })
17            skField.maxLength = 200;
18
19            form.addSubmitButton();
20
21            option.response.writePage(form);
22        } else {
23            let form = ui.createForm({
24                title: 'My Credential Form'
25            });
26
27            const inputString = "YWJjZGVmZwo=";
28            let myGuid = option.request.parameters.mycredential;
29
30            // Create the key
31            let sKey = crypto.createSecretKey({
32                guid: myGuid,
33                encoding: encode.Encoding.UTF_8
34            });
35
36            try {
37                let hmacSha512 = crypto.createHmac({
38                    algorithm: 'SHA512',
39                    key: sKey
40                });
41                hmacSha512.update({
42                    input: inputString,
43                    inputEncoding: encode.Encoding.BASE_64
44                });
45                let digestSha512 = hmacSha512.digest({
46                    outputEncoding: encode.Encoding.HEX
47                });
48            } catch (e) {
49                log.error({
50                    title: 'Failed to hash input',
51                    details: e
52                });
53            }
54
55            form.addField({
56                id: 'result',
57                label: 'Your digested hash value',
58                type: 'textarea'
59            }).defaultValue = digestSha512;
60
61            option.response.writePage(form);
62        }
63    }
64    return {
65        onRequest: onRequest
66    };
67});

```

Generate Credential Field



Note: This script sample uses the `define` function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the `require` function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

The following sample shows how to use a Suitelet to create a form field that generates a GUID. For more information about credential fields, see the help topic [Form.addCredentialField\(options\)](#).



Note: The default maximum length for a credential field is 32 characters. If needed, use the `Field.maxLength` property to change this value.

The values for `restrictToDomains`, `restrictToScriptIds`, and `baseUrl` in this sample are placeholders. You must replace them with valid values from your NetSuite account.

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType Suitelet
4   */
5
6 // This script creates a form with a credential field.
7 define(['N/ui/serverWidget', 'N/https', 'N/url'], (ui, https, url) => {
8     function onRequest(context) {
9         if (context.request.method === 'GET') {
10             let form = ui.createForm({
11                 title: 'Password Form'
12             });
13
14             let credField = form.addCredentialField({
15                 id: 'password',
16                 label: 'Password',
17                 restrictToDomains: ['<accountID>.app.netsuite.com'],
18                 restrictToCurrentUser: false,
19                 restrictToScriptIds: 'customscript_my_script'
20             });
21
22             credField.maxLength = 32;
23
24             form.addSubmitButton();
25
26             context.response.writePage({
27                 pageObject: form
28             });
29         }
30     else {
31         // Request to an existing Suitelet with credentials
32         let passwordGuid = context.request.parameters.password;
33
34         // Replace SCRIPTID and DEPLOYMENTID with the internal ID of the suitelet script and deployment in your account
35         let baseUrl = url.resolveScript({
36             scriptid: SCRIPTID,
37             deploymentId: DEPLOYMENTID,
38             returnExternalURL: true
39         });
40
41         let authUrl = baseUrl + '&pwd={' + passwordGuid + '}';
42
43         let secureStringUrl = https.createSecureString({
44             input: authUrl
45         });
46         let secureStringPWD = https.createSecureString({
47             input: '{' + passwordGuid + '}'
48         });
49         let headers = ({
50             'pwd': secureStringPWD

```

```

51     });
52
53     let response = https.post({
54       credentials: [passwordGuid],
55       url: secureStringUrl,
56       body: {authorization: ' + secureStringPWD + ', data:'anything can be here'},
57       headers: headers
58     });
59   }
60   return {
61     onRequest: onRequest
62   };
63 });
64 });

```

Add a Secret Key Field

i Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

The following sample shows how to add a secret key field.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define(['N/ui/serverWidget', 'N/file', 'N/keyControl', 'N/runtime'], function(ui, file, keyControl, runtime) {
6   function onRequest(context) {
7     var request = context.request;
8     var response = context.response;
9
10    if (request.method === 'GET') {
11      var form = ui.createForm({
12        title: 'Enter Password'
13      });
14
15      var credField = form.addSecretKeyField({
16        id: 'custfield_password',
17        label: 'Password',
18        restrictToScriptIds: [runtime.getCurrentScript().id],
19        restrictToCurrentUser: true //Depends on use case
20      });
21      credField.maxLength = 64;
22
23      form.addSubmitButton();
24      response.writePage(form);
25    } else {
26      // Read the request parameter matching the field ID we specified in the form
27      var passwordToken = request.parameters.custfield_password;
28
29      var pem = file.load({
30        id: 422
31      );
32
33      var key = keyControl.createKey();
34      key.file = pem;
35      key.name = 'Test';
36      key.password = passwordToken;
37      key.save();
38    }
39  }
40  return {
41    onRequest: onRequest
42  };
43 });

```

Add a Field that Displays Running Total to a Sublist

The following sample shows how to add a field to the sublist that calculates and displays a running total for it.

Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType Suitelet
4   */
5  define(['N/ui/serverWidget', 'N/record'], function(serverWidget, record){
6      return {
7          onRequest: function(params) {
8              var form = serverWidget.createForm({
9                  title: 'Simple Form'
10             });
11             var sublistObj2 = form.addSublist({
12                 id: 'mylist',
13                 type: serverWidget.SublistType.INLINEEDITOR,
14                 label: 'List'
15             });
16             sublistObj2.addField({
17                 id: 'description',
18                 type: serverWidget.FieldType.TEXT,
19                 label: 'Description'
20             });
21             sublistObj2.addField({
22                 id: 'amount',
23                 type: serverWidget.FieldType.CURRENCY,
24                 label: 'Amount'
25             });
26             sublistObj2.updateTotallingFieldId({
27                 id: 'amount'
28             });
29             sublistObj2.setSublistValue({
30                 id: 'description',
31                 line: 0,
32                 value: 'foo'
33             });
34             sublistObj2.setSublistValue({
35                 id: 'amount',
36                 line: 0,
37                 value: '10'
38             });
39             sublistObj2.setSublistValue({
40                 id: 'description',
41                 line: 1,
42                 value: 'bar'
43             });
44             sublistObj2.setSublistValue({
45                 id: 'amount',
46                 line: 1,
47                 value: '15'
48             });
49             form.addSublist({
50                 id: 'dummy',
51                 type: serverWidget.SublistType.STATICLIST,
52                 label: 'Dummy'
53             });
54             params.response.writePage(form);
55         }
56     });
57 });

```

Parse Strings and Display Result



Note: This sample script uses the `require` function so that you can copy it into the SuiteScript Debugger and test it. You must use the `define` function in an entry point script (the script you attach to a script record and deploy). For more information, see [SuiteScript 2.x Script Basics](#) and [SuiteScript 2.x Script Types](#).

The following sample parses a string (formatted according to the user's preferences) to a raw Date object, and then parses it back to the formatted string. This sample uses `format.parse(options)` and `format.format(options)`.

This sample assumes the Date Format set in the preferences is MM/DD/YYYY. You may need to change the value for the date used in this script to match the preferences set in your account.

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType Suitelet
4   */
5 define(['N/ui/serverWidget', 'N/format'], function(serverWidget, format) {
6     function parseAndFormatDateString() {
7       // Assuming Date format is MM/DD/YYYY
8       var initialFormattedDateString = "07/28/2015";
9       var parsedDateStringAsRawDateObject = format.parse({
10         value: initialFormattedDateString,
11         type: format.Type.DATE
12       });
13       var formattedDateString = format.format({
14         value: parsedDateStringAsRawDateObject,
15         type: format.Type.DATE
16       });
17       return [parsedDateStringAsRawDateObject, formattedDateString];
18     }
19     function onRequest(context) {
20       var data = parseAndFormatDateString();
21
22       var form = serverWidget.createForm({
23         title: "Date"
24       );
25
26       var fldDate = form.addField({
27         type: serverWidget.FieldType.DATE,
28         id: "date",
29         label: "Date"
30       );
31       fldDate.defaultValue = data[0];
32
33       var fldString = form.addField({
34         type: serverWidget.FieldType.TEXT,
35         id: "dateastext",
36         label: "Date as text"
37       );
38       fldString.defaultValue = data[1];
39
40       context.response.writePage(form);
41     }
42     return {
43       onRequest: onRequest
44     };
45   });

```

Custom Lists

- [Create a Custom List](#)

Create a Custom List



Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript Debugger](#).

The following code creates a Suitelet that generates a custom list page. For steps to create this script, see [SuiteScript 2.x Custom List Pages](#).

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType Suitelet
4   */
5 define(['N/ui/serverWidget'], function(serverWidget) {
6     function onRequest(context) {
7       if(context.request.method === 'GET'){
8         //Section One - List - See 'Steps for Creating a Custom List Page', Step Five
9         var list = serverWidget.createList({
10           title: 'Purchase History'
11         });
12
13         list.style = serverWidget.ListStyle.REPORT;
14
15         list.addButton({
16           id: 'buttonid',
17           label: 'Test',
18           functionName: '' //the function called when the button is pressed
19         });
20
21         //Section Two - Columns - See 'Steps for Creating a Custom List Page', Step Seven
22         var datecol = list.addColumn({
23           id: 'column1',
24           type: serverWidget.FieldType.DATE,
25           label: 'Date',
26           align: serverWidget.LayoutJustification.RIGHT
27         );
28
29         list.addColumn({
30           id: 'column2',
31           type: serverWidget.FieldType.TEXT,
32           label: 'Product',
33           align: serverWidget.LayoutJustification.RIGHT
34         );
35
36         list.addColumn({
37           id: 'column3',
38           type: serverWidget.FieldType.INTEGER,
39           label: 'Quantity',
40           align: serverWidget.LayoutJustification.RIGHT
41         );
42
43         list.addColumn({
44           id: 'column4',
45           type: serverWidget.FieldType.CURRENCY,
46           label: 'Unit Cost',
47           align: serverWidget.LayoutJustification.RIGHT
48         );
49
50         list.addRows({
51           rows: [{column1: '05/30/2018', column2: 'Widget', column3: '4', column4: '4.50'},
52                   {column1: '05/30/2018', column2: 'Sprocket', column3: '6', column4: '11.50'},
53                   {column1: '05/30/2018', column2: 'Gizmo', column3: '9', column4: '1.25'}]
54         );
55         context.response.writePage(list);
56       } else{
57     }
58   }
59 }
```

```

60     return {
61         onRequest: onRequest
62     });
63 });

```

Custom Assistants

- Sample Custom Assistant Script

Sample Custom Assistant Script



Note: This topic applies to SuiteScript 2.0.

For more information about custom assistant scripts, see [Creating Custom Assistants](#). Note that assistants cannot be used on externally available Suitelets.



Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

The following sample creates an assistant with two steps. In the first one, you can select a new supervisor for the employee selected. The second step displays the selection and submits it.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define(['N/ui/serverWidget'], function(serverWidget) {
6     return {
7         onRequest: function(context) {
8             var assistant = serverWidget.createAssistant({
9                 title: 'New Supervisor',
10                hideNavBar: true
11            });
12            var assignment = assistant.addStep({
13                id: 'assignment',
14                label: 'Select new supervisor'
15            });
16            var review = assistant.addStep({
17                id: 'review',
18                label: 'Review and Submit'
19            });
20
21            var writeAssignment = function() {
22                assistant.addField({
23                    id: 'newsupervisor',
24                    type: 'select',
25                    label: 'Name',
26                    source: 'employee'
27                });
28                assistant.addField({
29                    id: 'assignedemployee',
30                    type: 'select',
31                    label: 'Employee',
32                    source: 'employee'
33                });
34            }
35
36            var writeReview = function() {
37                var supervisor = assistant.addField({
38                    id: 'newsupervisor',
39                    type: 'text',

```

```

40         label: 'Name'
41     });
42 supervisor.defaultValue = context.request.parameters.inpt_newsupervisor;
43
44     var employee = assistant.addField({
45         id: 'assignedemployee',
46         type: 'text',
47         label: 'Employee'});
48 employee.defaultValue = context.request.parameters.inpt_assignedemployee;
49 }
50
51     var writeResult = function() {
52         var supervisor = context.request.parameters.newsupervisor;
53         var employee = context.request.parameters.assignedemployee;
54         context.response.write('Supervisor: ' + supervisor + '\nEmployee: ' + employee);
55     }
56
57     var writeCancel = function() {
58         context.response.write('Assistant was cancelled');
59     }
60
61     if (context.request.method === 'GET') //GET method means starting the assistant
62     {
63         writeAssignment();
64         assistant.currentStep = assignment;
65         context.response.writePage(assistant)
66     } else //POST method - process step of the assistant
67     {
68         if (context.request.parameters.next === 'Finish') //Finish was clicked
69             writeResult();
70         else if (context.request.parameters.cancel) //Cancel was clicked
71             writeCancel();
72         else if (assistant.currentStep.stepNumber === 1) { //transition from step 1 to step 2
73             writeReview();
74             assistant.currentStep = assistant.getNextStep();
75             context.response.writePage(assistant);
76         } else { //transition from step 2 back to step 1
77             writeAssignment();
78             assistant.currentStep = assistant.getNextStep();
79             context.response.writePage(assistant);
80         }
81     }
82 }
83 });
84 });

```

Suitelets with Embedded HTML

See [Embedding HTML from a Linked HTML Page in a Suitelet](#) for a complete example.

Backend Suitelets

- [Find Plugin Implementations](#)
- [Generate a SuiteSignOn Token](#)
- [Load an XML File and Obtain Child Element Values](#)
- [Parse an XML File and Log Element Values](#)
- [Redirect to a New Sales Order](#)
- [Render a PDF](#)
- [Render Search Results into a PDF File](#)
- [Retrieve the Name of the City Based on a ZIP Code](#)
- [Write User and Session Information to the Response](#)

Find Plugin Implementations

Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

The following sample shows an implementation of a custom plug-in interface. To test this sample, you need a custom plug-in type with a script ID of `customscript_magic_plugin` and an interface with a single method, `int doTheMagic(int, int)`.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType pluginTypeimpl
4 */
5 define(function() {
6     return {
7         doTheMagic: function(operand1, operand2) {
8             return operand1 + operand2;
9         }
10    });
11});
```

The following Suitelet iterates through all implementations of the custom plug-in type `customscript_magic_plugin`. For the plug-in to be recognized, the Suitelet script record must specify the plug-in type on the Custom Plug-in Types subtab.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define(['N/plugin'], function(plugin) {
6     function onRequest(context) {
7         var impls = plugin.findImplementations({
8             type: 'customscript_magic_plugin'
9         });
10
11        for (var i = 0; i < impls.length; i++) {
12            var pl = plugin.loadImplementation({
13                type: 'customscript_magic_plugin',
14                implementation: impls[i]
15            });
16            log.debug('impl ' + impls[i] + ' result = ' + pl.doTheMagic(10, 20));
17        }
18
19        var pl = plugin.loadImplementation({
20            type: 'customscript_magic_plugin'
21        });
22        log.debug('default impl result = ' + pl.doTheMagic(10, 20));
23    }
24
25    return {
26        onRequest: onRequest
27    };
28});
```

Generate a SuiteSignOn Token

Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

The following sample shows how to use `generateSuiteSignOnToken(options)` in a Suitelet script.



Important: The value used in this sample for the suiteSignOnRecordId field is a placeholder. Before using this sample, replace the suiteSignOnRecordId field value with a valid value from your NetSuite account. If you run a script with an invalid value, an error may occur. Additionally, the SuiteSignOn record you reference must be associated with a specific script. You make this association in the SuiteSignOn record's Connection Points sublist. For help with SuiteSignOn records, see the help topic [Creating SuiteSignOn Records](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5
6 // This script uses generateSuiteSignOnToken in a Suitelet.
7 define(['N/sso'], function(sso) {
8     function onRequest(context) {
9         var suiteSignOnRecordId = 'customsso_test'; //Replace placeholder values
10        var url = sso.generateSuiteSignOnToken(suiteSignOnRecordId);
11        log.debug(url);
12    }
13    return {
14        onRequest: onRequest
15    };
16 });

```

Load an XML File and Obtain Child Element Values



Note: This sample script uses the require function so that you can copy it into the SuiteScript Debugger and test it. You must use the define function in an entry point script (the script you attach to a script record and deploy). For more information, see [SuiteScript 2.x Script Basics](#) and [SuiteScript 2.x Script Types](#).

The following sample loads the BookSample.xml file from the File Cabinet, iterates through the individual book nodes, and accesses the child node values.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 require(['N/xml', 'N/file'], function(xml, file) {
6     return {
7         onRequest: function(options) {
8             var sentence = '';
9             var xmlFileContent = file.load('SuiteScripts/BookSample.xml').getContents();
10            var xmlDoc = xml.Parser.fromString({
11                text: xmlFileContent
12            });
13            var bookNode = xmlDoc.XPath.select({
14                node: xmlDoc,
15                xpath: '//b:book'
16            });
17
18            for (var i = 0; i < bookNode.length; i++) {
19                var title = bookNode[i].firstChild.nextSibling.textContent;
20                var author = bookNode[i].getElementsByTagName({
21                    tagName: 'b:author'
22                })[0].textContent;
23                sentence += 'Author: ' + author + ' wrote ' + title + '\n';
24            }
25
26            options.response.write(sentence);
27        }
28    };
29 });

```

This script produces the following output when used with the BookSample.xml file:

```

1 Author: Giada De Laurentiis wrote Everyday Italian.
2 Author: J K. Rowling wrote Harry Potter.
3 Author: James McGovern wrote XQuery Kick Start.
4 Author: Erik T. Ray wrote Learning XML.
```

Parse an XML File and Log Element Values

i Note: This sample script uses the require function so that you can copy it into the SuiteScript Debugger and test it. You must use the define function in an entry point script (the script you attach to a script record and deploy). For more information, see [SuiteScript 2.x Script Basics](#) and [SuiteScript 2.x Script Types](#).

The following sample parses the XML string stored in the xmlString variable. The sample selects all config elements in the xmlDoc node, loops through them, and logs their contents.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5
6 require(['N/xml'], function(xml) {
7     return {
8         onRequest: function(options) {
9             var xmlString = '<xml version="1.0" encoding="UTF-8"?><config date="1465467658668" transient="false">Some content</config>';
10
11             var xmlDoc = xml.Parser.fromString({
12                 text: xmlString
13             });
14
15             var bookNode = xmlDoc.XPath.select({
16                 node: xmlDoc,
17                 xpath: '//config'
18             });
19
20             for (var i = 0; i < bookNode.length; i++) {
21                 log.debug('Config content', bookNode[i].textContent);
22             }
23         }
24     };
25});
```

Redirect to a New Sales Order

i Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see [SuiteScript 2.0 Global Objects](#).

! **Important:** The value used in this sample for the entity field is a placeholder. Before using this sample, replace the entity field value with a valid value from your NetSuite account. If you run a script with an invalid value, an error may occur.

The following sample shows how to use a Suitelet to redirect to a new sales order record and set the entity field (which represents the customer).

```
1 /**
2 */
```

```

2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4  */
5
6 // This script redirects a new sales order record and sets the entity.
7 define(['N/record', 'N/http'], (record, http) => {
8   function onRequest(context) {
9     context.response.sendRedirect({
10       type: http.RedirectType.RECORD,
11       identifier: record.Type.SALES_ORDER,
12       parameters: ({
13         entity: 6
14       })
15     });
16   }
17   return {
18     onRequest: onRequest
19   };
20 });

```

Render a PDF

The following Suitelet generates and renders a PDF directly to the response.

Note: This script sample uses the `define` function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the `require` function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4  */
5 define(['N/xml'], function(xml) {
6   return {
7     onRequest: function(context) {
8       var xml = "<?xml version='1.0' encoding='UTF-8'?>\n" +
9         "<!DOCTYPE pdf PUBLIC '-//big.faceless.org//report\\ \"report-1.1.dtd\\>\n" +
10        "<pdf lang='ru-RU\\\" xml:lang='ru-RU\\>\n" +
11        "<head>\n" +
12        "<link name='russianfont\\\" type='font\\\" subtype='opentype\\\" " + "src='NetSuiteFonts/verdana.ttf\\\" " + "src-
bold=NetSuiteFonts/verdanab.ttf\\\" " + "src-italic=NetSuiteFonts/verdanai.ttf\\\" " + "src-bolditalic=NetSuiteFonts/verdan-
abi.ttf\\\" " + "bytes='2\\\"/>\n" +
13        "</head>\n" +
14        "<body font-family='russianfont\\\" font-size='18\\\">\nРусский текст</body>\n" +
15        "</pdf>";
16       context.response.renderPdf(xml);
17     }
18   }
19 });

```

Render Search Results into a PDF File

Note: This script sample uses the `define` function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the `require` function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

The following sample shows how to render search results into a PDF file.

```

1 /**
2  * @NApiVersion 2.x

```

```

3  * @NScriptType Suitelet
4  */
5 // This sample shows how to render search results into a PDF file.
6 define(['N/render', 'N/search'], function(render, search) {
7     function onRequest(options) {
8         var request = options.request;
9         var response = options.response;
10
11         var xmlStr = '<?xml version="1.0" encoding="UTF-8"?>\n' +
12             '<!DOCTYPE pdf PUBLIC "-//big.faceless.org//report" "report-1.1.dtd">\n' +
13             '<pdf lang="ru-RU" xml:lang="ru-RU">\n' + "<head>\n" +
14             '<link name="russianfont" type="font" subtype="opentype" ' + 'src="NetSuiteFonts/verdana.ttf"' " + "src-
bold=NetSuiteFonts/verdanab.ttf"' + 'src-italic="NetSuiteFonts/verdanai.ttf"' " + "src-bolditalic=NetSuiteFonts/verdan-
abi.ttf"' + 'bytes="2"/>\n' + "</head>\n" +
15             '<body font-family="russianfont" font-size="18">\n?????? ?????</body>\n" + "</pdf>';
16
17         var rs = search.create({
18             type: search.Type.TRANSACTION,
19             columns: ['trandate', 'amount', 'entity'],
20             filters: []
21         }).run();
22
23         var results = rs.getRange(0, 1000);
24         var renderer = render.create();
25         renderer.templateContent = xmlStr;
26         renderer.addSearchResults({
27             templateName: 'exampleName',
28             searchResult: results
29         });
30
31         var newfile = renderer.renderAsPdf();
32         response.writeFile(newfile, false);
33     }
34
35     return {
36         onRequest: onRequest
37     };
38 });

```

Retrieve the Name of the City Based on a ZIP Code

Note: This script sample uses the `define` function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the `require` function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

The following sample shows how to use a Suitelet and a custom module to retrieve the name of a city based on a ZIP code. To speed up processing, the Suitelet uses a cache. In this sample, the ZIP code is the key used to retrieve city names from the cache. For any ZIP code provided, if the corresponding city value is not already stored in the cache, a loader function is called. This loader function is a custom module called `zipCodeDatabaseLoader` (shown in the second script sample) that loads a CSV file and uses it to find the requested value.

Note: This sample depends on a CSV file that must exist before the script is run. The sample CSV file is available [here](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4  */
5
6 // This script retrieves the name of a city based on a ZIP code, using a cache.
7 define(['N/cache', '/SuiteScripts/zipToCityIndexCacheLoader'], function(cache, lib) {
8     const ZIP_CODES_CACHE_NAME = 'ZIP_CODES_CACHE';
9     const ZIP_TO_CITY_IDX_JSON = 'ZIP_TO_CITY_IDX_JSON';

```

```

10    function getZipCodeToCityLookupObj() {
11        var zipCache = cache.getCache({
12            name: ZIP_CODES_CACHE_NAME
13        });
14        var zipCacheJson = zipCache.get({
15            key: ZIP_TO_CITY_IDX_JSON,
16            loader: lib.zipCodeDatabaseLoader
17        });
18        return JSON.parse(zipCacheJson);
19    }
20
21    function findCityByZipCode(options) {
22        return getZipCodeToCityLookupObj()[String(options.zip)];
23    }
24
25    function onRequest(context) {
26        var start = new Date();
27        if (context.request.parameters.purgeZipCache === 'true') {
28            var zipCache = cache.getCache({
29                name: ZIP_CODES_CACHE_NAME
30            });
31            zipCache.remove({
32                key: ZIP_TO_CITY_IDX_JSON
33            });
34        }
35        var cityName = findCityByZipCode({
36            zip: context.request.parameters.zipcode
37        });
38
39        context.response.writeLine(cityName || 'Unknown :(');
40
41        if (context.request.parameters.auditPerf === 'true') {
42            context.response.writeLine('Time Elapsed: ' + (new Date().getTime() - start.getTime()) + ' ms');
43        }
44    }
45    return {
46        onRequest: onRequest
47    };
48};
49});

```

The following custom module returns the loader function used in the preceding Suitelet script sample. The loader function uses a CSV file to retrieve a value that was missing from a cache. This custom module does not need to include logic for placing the retrieved value into the cache. Whenever a value is returned through the options.loader parameter, the value is automatically placed into the cache. This allows the loader function to serve as the sole method of populating a cache with values.

```

1 /**
2  * zipToCityIndexCacheLoader.js
3  * @NApiVersion 2.x
4  * @NModuleScope Public
5 */
6
7 //This custom module is a loader function that uses a CSV file to retrieve a value that was missing from a cache.
8 define(['N/file', 'N/cache'], function(file, cache) {
9     const ZIP_CODES_CSV_PATH = '/SuiteScripts/Resources/free-zipcode-CA-database-primary.csv';
10
11     function trimOuterQuotes(str) {
12         return (str || '').replace(/^"/, '').replace(/"/, '');
13     }
14
15     function zipCodeDatabaseLoader(context) {
16         log.audit('Loading Zip Codes for ZIP_CODES_CACHE');
17         var zipCodesCsvText = file.load({
18             id: ZIP_CODES_CSV_PATH
19         }).getContents();
20         var zipToCityIndex = {};
21         var csvLines = zipCodesCsvText.split('\n');
22         util.each(csvLines.slice(1), function(el) {
23             var cells = el.split(',');
24             var key = trimOuterQuotes(cells[0]);

```

```

25     var value = trimOuterQuotes(cells[2]);
26     if (parseInt(key, 10))
27       zipToCityIndex[String(key)] = value;
28   });
29   return zipToCityIndex;
30 }
31
32 return {
33   zipCodeDatabaseLoader: zipCodeDatabaseLoader
34 }
35 });

```

Write User and Session Information to the Response

i Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

The following sample shows how to use a Suitelet to write user and session information for the currently executing script to the response.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5
6 // This script writes user and session information for the currently executing script to the response.
7 define(['N/runtime'], function(runtime) {
8   function onRequest(context) {
9     var remainingUsage = runtime.getCurrentScript().getRemainingUsage();
10    var userRole = runtime.getCurrentUser().role;
11    var currentSession = runtime.getCurrentSession();
12
13    // Set the current session's scope
14    currentSession.set({
15      name: 'scope',
16      value: 'global'
17    });
18
19    var sessionScope = runtime.getCurrentSession().get({
20      name: 'scope'
21    });
22
23    log.debug('Remaining Usage:', remainingUsage);
24    log.debug('Role:', userRole);
25    log.debug('Session Scope:', sessionScope);
26
27    context.response.write('Executing under role: ' + userRole
28                           + ' Session scope: ' + sessionScope + '.');
29  }
30  return {
31    onRequest: onRequest
32  };
33});

```

SuiteScript 2.x User Event Script Type

User event scripts are executed on the NetSuite server. They are executed when users perform certain actions on records, such as create, load, update, copy, delete, or submit. Most standard NetSuite records and custom record types support user event scripts. Exceptions include records used for personal identification purposes (such as a Driver's License, Passport, or other Government-issued ID),

some revenue recognition records, and some timecard-related records. See the help topic [SuiteScript Supported Records](#) for more information on specific records.

User event scripts can be used to perform the following tasks:

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

For additional information about SuiteScript 2.x User Event Scripts, see the following:

- [SuiteScript 2.x User Event Script Reference](#)
 - [How User Events are Executed](#)
 - [SuiteScript 2.x User Event Script Best Practices](#)
 - [SuiteScript 2.x User Event Script Tutorial](#)
- [SuiteScript 2.x User Event Script Entry Points and API](#)
 - [afterSubmit\(scriptContext\)](#)
 - [beforeLoad\(scriptContext\)](#)
 - [beforeSubmit\(scriptContext\)](#)
 - [context.UserEventType](#)

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

You can use SuiteScript Analysis to learn about when the script was installed and how it performed in the past. For more information, see the help topic [Analyzing Scripts](#).

SuiteScript 2.x User Event Script Sample

The following sample shows a user event script. This script is designed for use in environments that do not use the Team Selling feature.

When you deploy this script on the customer record, this script creates a follow-up phone call record for every newly created customer record.



Important: Before running this script, you must replace the salesrep internal ID with one specific to your account. Specifically, use an ID that represents an employee who is classified as a sales rep. The sales rep option is located on the Human Resources subtab of the employee record. If you do not replace the ID, the script may not work as expected. Additionally, note that this script is designed to work in environments where the customer record includes a salesrep field. If the Team Selling feature is enabled, the customer record typically will not include a salesrep field.

For help with writing scripts in SuiteScript 2.x, see [SuiteScript 2.x Hello World](#) and [SuiteScript 2.x Entry Point Script Creation and Deployment](#).

```
1 | /**
```

```

2  /*@NApiVersion 2.x
3  *@NScriptType UserEventScript
4  */
5  define(['N/record'],
6    function(record) {
7      function beforeLoad(context) {
8        if (context.type !== context.UserEventType.CREATE)
9          return;
10       var customerRecord = context.newRecord;
11       customerRecord.setValue('phone', '555-555-5555');
12       if (!customerRecord.getValue('salesrep'))
13         customerRecord.setValue('salesrep', 46); // replace '46' with one specific to your account
14     }
15     function beforeSubmit(context) {
16       if (context.type !== context.UserEventType.CREATE)
17         return;
18       var customerRecord = context.newRecord;
19       customerRecord.setValue('comments', 'Please follow up with this customer!');
20     }
21     function afterSubmit(context) {
22       if (context.type !== context.UserEventType.CREATE)
23         return;
24       var customerRecord = context.newRecord;
25       if (customerRecord.getValue('salesrep')) {
26         var call = record.create({
27           type: record.Type.PHONE_CALL,
28           isDynamic: true
29         });
30         call.setValue('title', 'Make follow-up call to new customer');
31         call.setValue('assigned', customerRecord.getValue('salesrep'));
32         call.setValue('phone', customerRecord.getValue('phone'));
33         try {
34           var callId = call.save();
35           log.debug('Call record created successfully', 'Id: ' + callId);
36         } catch (e) {
37           log.error(e.name);
38         }
39       }
40     }
41     return {
42       beforeLoad: beforeLoad,
43       beforeSubmit: beforeSubmit,
44       afterSubmit: afterSubmit
45     };
46   });

```

SuiteScript 2.x User Event Script Reference

- [How User Events are Executed](#)
- [SuiteScript 2.x User Event Script Best Practices](#)
- [SuiteScript 2.x User Event Script Tutorial](#)

How User Events are Executed



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.



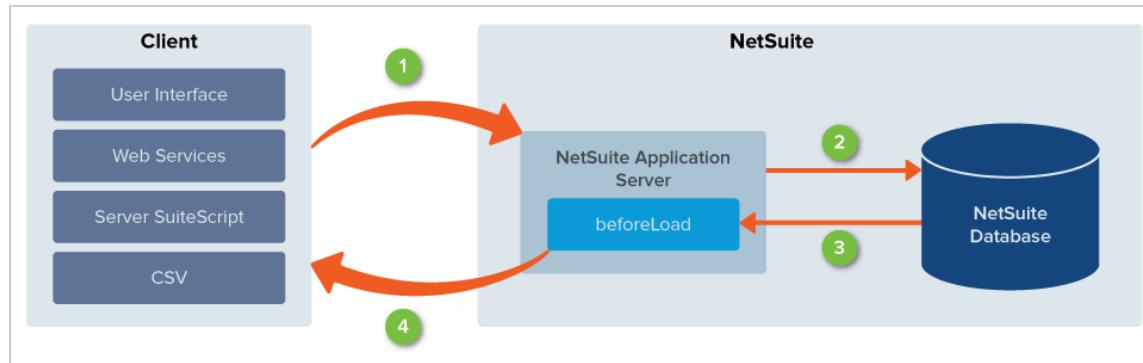
Important: User event scripts cannot be executed by other user event scripts or by workflows with a **Context of User Event Script**. In other words, you cannot chain user event scripts. You can, however, execute a user event script from a call within a scheduled script, a portlet script, or a Suitelet.

User event scripts are executed based on operations defined as beforeLoad, beforeSubmit, and afterSubmit.

 **Tip:** You can set the order in which user event scripts execute on the Scripted Records page. See the help topic [The Scripted Records Page](#).

beforeLoad

The following diagram shows an overview of what occurs during a beforeLoad operation:

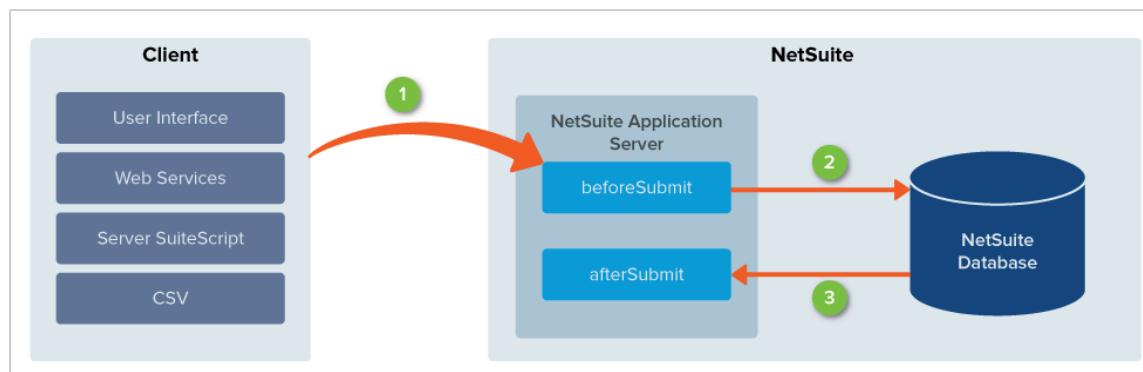


1. The client sends a read operation request for record data. The client request can come from the user interface, SOAP web services, CSV import, or server SuiteScript (except other user event scripts).
2. Upon receiving the request, the application server performs basic permission checks on the client.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

 **Note:** Standard records cannot be sourced during a beforeLoad operation. Use the pageInit client script for this purpose. See [pageInit](#).

beforeSubmit and afterSubmit

The following diagram shows an overview of what occurs on beforeSubmit and afterSubmit operations:



1. The client performs a write operation by submitting data to the application server. The client request can come from the user interface, SOAP web services, server SuiteScript calls, CSV imports, or XML. The application server:

- a. performs basic permission checks on the client
- b. processes the submitted data and performs specified validation checks during a beforeSubmit operation

The submitted data has **NOT** yet been committed to the database.

2. After the data has been validated, it is committed to the database.
3. If this (newly committed) data is then called by an afterSubmit operation, the data is taken from the database and is sent to the application server for additional processing. Examples of afterSubmit operations on data that are already committed to the database include, but are not limited to:
 - a. sending email notifications (regarding the data that was committed to the database)
 - b. creating child records (based on the data that was committed to the database)
 - c. assigning tasks to employees (based on data that was committed to the database)



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

SuiteScript 2.x User Event Script Best Practices



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.



Important: User event scripts cannot be executed by other user event scripts or by workflows with a **Context of User Event Script**. In other words, you cannot chain user event scripts.

The following list shows best practices for user event script development:

- Use the type argument and context object to define and limit the scope of your user event logic. See [context.UserEventType](#).
- Use execution context filtering to specify how and when a user event script is executed. Execution contexts provide information about how a script is triggered to execute. For example, a script can be triggered in response to an action in the NetSuite application, or an action occurring in another context, such as a web services integration. You can use execution context filtering to ensure that your scripts are triggered only when necessary. For more information, see the help topic [Execution Contexts](#).
- Limit the amount of script execution in user event scripts (less than 5 seconds, if possible) since they run often and run in-line. You can use the Script Performance Monitor SuiteApp to test the performance of your scripts deployed on a specific record type. See the help topic [Application Performance Management \(APM\)](#).
- Mission-critical business logic implemented using user events should be accompanied by a clean-up scheduled script to account for any unexpected errors or misfires.
- Any operation that depends on the submitted record being committed to the database should happen in an afterSubmit script.
- Be careful when updating transaction line items in a beforeSubmit script because you have to ensure that the line item totals net taxes and discounts are equal to the summarytotal, discounttotal, shippingtotal, and taxtotal amounts.
- Activities (user events) on a hosted website can trigger server SuiteScripts. In addition to sales orders, scripts on case records and customer records also execute in response to web activities.
- Assigning many executable functions to one record type is discouraged because this could negatively affect the user experience with that record type. For example, if there are ten beforeLoad scripts that

must complete their execution before the record loads into the browser, the time needed to load the record may increase significantly. Be aware of the number of user events scripts used, including bundled user event scripts.

- To set a field on a record or make any changes to a record being submitted, use the `beforeSubmit` event.
- Perform all post-processing operations of the current record on an `afterSubmit` event.
- Do not try to execute a user event script from another user event script. Instead, create a module containing the code that is common between the two user event scripts, and use the module in both scripts.
- If you want to store a value during a `beforeLoad` operation and then read that value during an `afterSubmit` operation in the same script, consider using a hidden custom field to store the value. You can add a hidden custom field to the form and store your value during the `beforeLoad` operation, and you can retrieve the value from the same field during the `afterSubmit` operation.
- Make sure that the user event script does not read a sensitive field value.

SuiteScript 2.x User Event Script Tutorial

A user event script is a server script that is triggered by actions taken on a record. The script type can include logic that is executed in three situations:

- After the user has started the process of opening the record but before the record loads.
- After the user clicks Save to save the record.
- After the record is submitted to the database.

This tutorial walks you through the implementation of a basic user event script. It includes the following sections:

- [Sample Script Overview](#)
- [Step One: Check Your Prerequisites](#)
- [Step Two: Create the Script File](#)
- [Step Three: Review the Script \(Optional\)](#)
- [Step Four: Upload the Script File to NetSuite](#)
- [Step Five: Create a Script Record and Script Deployment Record](#)
- [Step Six: Test the Script](#)



Note: Before proceeding, review [SuiteScript 2.x Script Basics](#) for an explanation of terms used in this tutorial. For an overview of the basic structural elements required in any SuiteScript 2.x entry point script, see [SuiteScript 2.x Anatomy of a Script](#).

Sample Script Overview

The sample script is meant to be deployed on the employee record. When it is, it takes the following actions:

- When a user begins the process of creating a new employee record, the script disables the Notes field before the page loads.
- After the user enters all required information about the employee and clicks Save, the script adds a value to the Notes field that states that no date has yet been set for the new employee's orientation.

- After the record has been submitted, the script creates a task record for scheduling the new employee's orientation session. The script assigns the task to the new employee's supervisor.

The script uses all three of the entry points available with the [SuiteScript 2.x User Event Script Type](#). For details about these entry points, see [beforeLoad\(scriptContext\)](#), [beforeSubmit\(scriptContext\)](#), and [afterSubmit\(scriptContext\)](#).

Check Your Prerequisites

To complete this tutorial, your system has to be set up properly. Review the following before you proceed:

- [Enable the Feature](#)
- [Create an Employee Record](#)

Enable the Feature

Before you can complete the rest of the steps listed in this topic, the Client and Server SuiteScript features must be enabled in your account. For help enabling these features, see the help topic [Enabling SuiteScript](#).

Create an Employee Record

To complete this tutorial, your system must have an employee record that you can designate as the supervisor of a new test employee.

To view your existing employee records, select Lists > Employees. If your account does not already have an employee record you can use, create one. For help creating an employee record, see the help topic [Adding an Employee](#).

Create the Script File

Before proceeding, you must create the entry point script file. To create the file, copy and paste the following code into the text editor of your choice. Save the file and name it `createTask.js`.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType UserEventScript
4  */
5
6 // Load two standard modules.
7 define ( ['N/record', 'N/ui/serverWidget'],
8   // Add the callback function.
9   function(record, serverWidget) {
10     // In the beforeLoad function, disable the Notes field.
11     function myBeforeLoad (context) {
12       if (context.type !== context.UserEventType.CREATE)
13         return;
14       var form = context.form;
15       var notesField = form.getField({
16         id: 'comments'
17       });
18       notesField.updateDisplayType({
19         displayType: serverWidget.FieldDisplayType.DISABLED
20       });
21     }
22
23     // In the beforeSubmit function, add test to the Notes field.
24     function myBeforeSubmit(context) {

```

```

25     if (context.type !== context.UserEventType.CREATE)
26         return;
27     var newEmployeeRecord = context.newRecord;
28     newEmployeeRecord.setValue({
29         fieldId: 'comments',
30         value: 'Orientation date TBD.'
31     });
32 }
33
34 // In the afterSubmit function, begin creating a task record.
35 function myAfterSubmit(context) {
36     if (context.type !== context.UserEventType.CREATE)
37         return;
38     var newEmployeeRecord = context.newRecord;
39     var newEmployeeFirstName = newEmployeeRecord.getValue ({
40         fieldId: 'firstname'
41     });
42     var newEmployeeLastName = newEmployeeRecord.getValue ({
43         fieldId: 'lastname'
44     });
45     var newEmployeeSupervisor = newEmployeeRecord.getValue ({
46         fieldId: 'supervisor'
47     });
48     if (newEmployeeSupervisor) {
49         var newTask = record.create({
50             type: record.Type.TASK,
51             isDynamic: true
52         });
53         newTask.setValue({
54             fieldId: 'title',
55             value: 'Schedule orientation session for ' +
56                 newEmployeeFirstName + ' ' + newEmployeeLastName
57         });
58         newTask.setValue({
59             fieldId: 'assigned',
60             value: newEmployeeSupervisor
61         });
62         try {
63             var newTaskId = newTask.save();
64             log.debug({
65                 title: 'Task record created successfully',
66                 details: 'New task record ID: ' + newTaskId
67             });
68         } catch (e) {
69             log.error({
70                 title: e.name,
71                 details: e.message
72             });
73         }
74     }
75 }
76
77 // Add the return statement that identifies the entry point functions.
78 return {
79     beforeLoad: myBeforeLoad,
80     beforeSubmit: myBeforeSubmit,
81     afterSubmit: myAfterSubmit
82 };
83 });

```

Review the Script (Optional)

If you want to understand more about how this script is structured, review the following subsections.

- [JSDoc Tags](#)
- [beforeLoad Function](#)
- [beforeSubmit Function](#)
- [afterSubmit Function](#)

- [return Statement](#)

JSDoc Tags

The following image shows the JSDoc block used in this sample script. For an explanation of the numbered callouts, see the table that follows the image.



| Callout | Description |
|---------|--|
| 1 and 3 | The @NApiVersion tag and its value (2.x). This tag is required in an entry point scripts. Valid values are 2.0, 2.x, and 2.X. |
| 2 and 4 | The @NScriptType tag and its value (UserEventScript). This tag identifies the script type being used. The value is not case sensitive, but using Pascal case, as shown in this example, is recommended for better readability. |

beforeLoad Function

The following image shows the myBeforeLoad function. Because of the way the script's [return Statement](#) is structured, this function executes when the beforeLoad entry point is invoked.



| Callout | Description |
|---------|--|
| 1 | The context object that is made available when the beforeLoad entry point is invoked. You can see a list of the properties that are available to this context object in beforeLoad(scriptContext) . This function uses two of those properties: type and form . |
| 2 | This statement tells the system that the rest of the logic in the function should execute only if the user is creating a new record. The statement uses the context object's type property to identify the type of action the user is taking. To see a list of the possible actions, review context.UserEventType . |
| 3 | These statements tell the system to disable the Notes field. They do so in part by using the context object's form property, which gives the script access to the form. These statements also use N/ui/serverWidget Module APIs, which let the script access specific fields and change how they appear and behave. |

beforeSubmit Function

The following image shows the myBeforeSubmit function. Because of the way the script's [return Statement](#) is structured, this function executes when the beforeSubmit entry point is invoked.

```
function myBeforeSubmit (context) {
    if (context.type !== context.UserEventType.CREATE)
        return;

    var newEmployeeRecord = context.newRecord;
    newEmployeeRecord.setValue({
        fieldId: 'comments',
        value: 'Orientation date TBD.'
    });
}
```

| Callout | Description |
|---------|--|
| 1 | <p>The context object that is made available when the beforeSubmit entry point is invoked. You can see a list of the properties that are available to this context object in beforeSubmit(scriptContext). This function uses two of those properties: type and newRecord.</p> <p>Note that the properties available to this context object are different from those available to the beforeLoad Function, which is linked to the beforeLoad entry point. The beforeLoad entry point has access to a form property, but the beforeSubmit object does not, which means that the beforeSubmit entry point function does not have access to the form used by the record.</p> |
| 2 | This statement tells the system that the rest of the logic in the function should execute only if the user is creating a new record. The statement uses the context object's type property to identify the type of action the user is taking. To see a list of the possible values, review context.UserEventType . |
| 3 | These statements add text to the Notes field of the new record. They do so by using the context object's newRecord property, which gives the script access to the record that is about to be created. This code uses the Record.setValue(options) method to add text to the Notes field of the new record. |

afterSubmit Function

The following image show the myAfterSubmit function. Because of the way the script's [return Statement](#) is structured, this function executes when the afterSubmit entry point is invoked.

The first part of the function retrieves data from the employee record:

```
function myAfterSubmit (context) {
    if (context.type !== context.UserEventType.CREATE)
        return;

    var newEmployeeRecord = context.newRecord;
    var newEmployeeFirstName = newEmployeeRecord.getValue ({
        fieldId: 'firstname'
    });

    var newEmployeeLastName = newEmployeeRecord.getValue ({
        fieldId: 'lastname'
    });

    var newEmployeeSupervisor = newEmployeeRecord.getValue ({
        fieldId: 'supervisor'
    });
}
```

| Callout | Description |
|---------|--|
| 1 | The context object that is made available when the afterSubmit entry point is invoked. You can see a list of the properties that are available to this context object in afterSubmit(scriptContext) . This function uses two of those properties: type and newRecord . |
| 2 | This statement tells the system that the rest of the logic in the function should execute only if the user is creating a new record. The statement uses the context object's type property to identify the type of action the user is taking. To see a list of the possible actions, review context.UserEventType . |
| 3 | These statements retrieve several pieces of data from the record. They do so by using the context object's newRecord property, which gives the script access to the record that has just been submitted. This code also uses the Record.getValue(options) method to retrieve the data. |

The rest of the function creates the task record:

```

4
if (newEmployeeSupervisor) {
    var newTask = record.create({
        type: record.Type.TASK,
        isDynamic: true
    });

    newTask.setValue({
        fieldId: 'title',
        value: 'Schedule orientation session for '
            + newEmployeeFirstName + ' ' + newEmployeeLastName
    });

    newTask.setValue({
        fieldId: 'assigned',
        value: newEmployeeSupervisor
    });
}

try {
    var newTaskId = newTask.save();

    log.debug({
        title: 'Task record created successfully',
        details: 'New task record ID: ' + newTaskId
    });

} catch (e) {
    log.error({
        title: e.name,
        details: e.message
    });
}
}

```

| Callout | Description |
|---------|--|
| 4 | A conditional statement. The expression inside the parentheses evaluates to true if a value has been set for the Supervisor field. If this condition is met, the system executes the logic within the curly braces. |
| 5 | These statements use N/record Module APIs to create a task record (record.create) and set values (newTask.setValue) on the record. |
| 6 | This code attempts to save the new task record. If any errors are encountered, details are written to the script deployment record's execution log. |

return Statement

The following image shows the callback function's return statement.



| Callout | Description |
|---------|---|
| 1 | Entry points. The callback function's return statement must use at least one entry point that belongs to the script type identified by the @NScriptType tag (Callout 4 in JSDoc Tags). This script uses all three of the user event script type's entry points. |
| 2 | Entry points functions. For each entry point used, your script must identify an entry point function that is defined elsewhere within the script. |

Upload the Script File to NetSuite

After you have created your entry point script file, upload it to your NetSuite File Cabinet.

To upload the script file:

1. In the NetSuite UI, go to Documents > Files > SuiteScripts.
2. In the left pane, select the SuiteScripts folder and click **Add File**.
3. Follow the prompts to locate the createTask.js file (created in Step Two) in your local environment and upload it.

Note that even after you upload the file, you can edit it from within the File Cabinet, if needed. For details, see the help topic [Editing Files in the File Cabinet](#).

Create a Script Record and Script Deployment Record

In general, before an entry point script can execute in your account, you must first create a script record that represents the entry point script file. You must also create a script deployment record.

To create the script record and script deployment record:

1. Go to Customization > Scripting > Scripts > New.
2. In the **Script File** dropdown list, select createTask.js.

Note that, if you had not yet uploaded the file, as described in Step Four, you could upload the file from this page. Clicking the plus sign icon to the right of the dropdown list opens a window that lets you select and upload a file. You may have to move your cursor over the area to the right of the dropdown list to display the plus sign icon.

3. After you have selected the script file and it is displayed in the dropdown list, click the **Create Script Record** button.

The system displays a new script record if the script file passes validation checks, and the createTask.js file is listed on the **Scripts** subtab.

4. Fill out the required body fields as follows:
 - In the **Name** field, enter **Create Task Record User Event**.
 - In the **ID** field, enter **_ues_create_task_record**.
5. Click the **Deployments** subtab.
6. Add a line to the sublist, as follows:
 - Set the **Applies to** dropdown list to **Employee**.
 - In the **ID** field, enter **_ues_create_task_record**.

Leave the other fields set to their default values. Note that the **Status** field is set to **Testing**, which means that the script does not deploy for other users. (If you wanted to change the deployment later and make the customization's behavior available to all users, you could edit the deployment and set the status to **Released**.)
7. Click **Save**.

The system creates the script and script deployment records.

Test the Script

Now that the script is deployed, you should verify that it executes as expected.

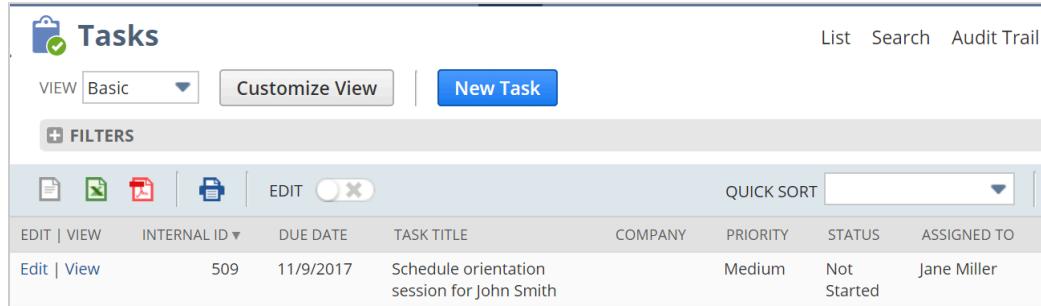
To test the script:

1. Begin the process of creating a new employee record by selecting **Lists > Employees > Employees > New**.
2. In the new employee form, check to see whether the Notes field is disabled. If the **beforeLoad** entry point function worked as expected, the field is gray and cannot be edited.
3. Enter value for required fields. These fields may vary depending on the features enabled in your account and any customizations that exist. At a minimum, you must enter values for the following:
 - **Name** — Enter a first name of John and last name of Smith.
 - **Subsidiary** — (OneWorld only) Choose an appropriate value from the dropdown list.
4. To make sure that the **afterSubmit** function executes correctly, enter a value in the **Supervisor** field, if you have not already.
5. Click **Save**. A success message appears, and the system displays the new record in View mode.
6. Verify that the **beforeSubmit** function was successful: Look at the Notes field. It should include the value **Orientation date TBD** as shown below.

The screenshot shows the NetSuite Employee view page. At the top, a green confirmation message says "✓ Confirmation" and "Employee successfully Saved". Below this, the employee record for "John Smith" is displayed. The record is marked as "ACTIVE". The "Edit" button is highlighted in blue. The "Actions" dropdown menu is open, showing options like "Print", "Email", and "Actions". The "Primary Information" section includes fields for "EMPLOYEE ID" (John Smith), "JOB TITLE", "NOTES" (which contains the value "Orientation date TBD." and is highlighted with a red border), "SUPERVISOR" (Jane Miller), and "IMAGE".

7. Verify that the **afterSubmit** function was successful:
 - a. Select Activities > Scheduling > Tasks.

- b. Check the filtering options to make sure that your view includes tasks assigned to all users.
- c. Verify that a task was created and assigned to the person you named as a supervisor in Step 4.



Next Steps

If you want to learn how to customize this entry point script so that it calls a custom module, see [SuiteScript 2.x Custom Module Tutorial](#).

SuiteScript 2.x User Event Script Entry Points and API

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

| Script Entry Point | Description |
|--|---|
| <code>afterSubmit(scriptContext)</code> | Executes immediately after a write operation on a record. |
| <code>beforeLoad(scriptContext)</code> | Executes whenever a read operation occurs on a record, and prior to returning the record or page. |
| <code>beforeSubmit(scriptContext)</code> | Executes prior to any write operation on the record. |
| <code>context.UserEventType</code> | Holds the string values for user event execution contexts. |

afterSubmit(scriptContext)

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|--|
| Description | Defines the function that is executed after a record is submitted. The afterSubmit operation is useful for performing any actions that need to occur following a write operation on a record. Examples of these actions include email notification, browser redirect, creation of dependent records, and synchronization with an external system. |
| Notes: | <ul style="list-style-type: none"> ■ The approve, cancel, and reject argument types are only available for record types such as sales orders, expense reports, timebills, purchase orders, and return authorizations. |

| | |
|----------------|---|
| | <ul style="list-style-type: none"> ■ Attaching a child custom record to its parent or detaching a child custom record from its parent triggers an edit event. ■ Asynchronous afterSubmit user events are only supported during webstore checkout. <p>This event can be used with the following context.UserEventType:</p> <ul style="list-style-type: none"> ■ create ■ edit ■ xedit (inline editing; only returns the fields edited and not the full record) ■ delete ■ approve (only available for certain record types) ■ cancel (only available for certain record types) ■ reject (only available for certain record types) ■ pack (only available for certain record types, for example Item Fulfillment records) ■ ship (only available for certain record types, for example Item Fulfillment records) ■ dropship (for purchase orders with items specified as "drop ship") ■ specialorder (for purchase orders with items specified as "special order") ■ orderitems (for purchase orders with items specified as "order item") ■ paybills (use this type to trigger afterSubmit user events for Vendor Payments from the Pay Bill page. Note that no sublist line item information will be available. Users must do a lookup/search to access line item values.) |
| Returns | void |
| Since | Version 2015 Release 2 |

Parameters

| | |
|--|---|
|  Note: | The scriptContext parameter is a JavaScript object. |
|--|---|

| Parameter | Type | Required / Optional | Description | Since |
|-------------------------|-------------------------------|---------------------|--|------------------------|
| scriptContext.newRecord | record.Record | required | The new record in read only mode. To edit a record, use the record.load(options) method to load the newly submitted record. Make changes, and submit the record again. | Version 2015 Release 2 |
| scriptContext.oldRecord | record.Record | required | The old record in read only mode. | Version 2015 Release 2 |
| scriptContext.type | string | required | The trigger type. Use the context.UserEventType enum to set this value. | Version 2015 Release 2 |

For an example of the afterSubmit entry point, see [SuiteScript 2.x User Event Script Sample](#).

beforeLoad(scriptContext)

| | |
|--|--|
|  Note: | This topic applies to SuiteScript 2.0 and SuiteScript 2.1. |
|--|--|

| | |
|--------------------|--|
| Description | Defines the function that is executed before a record is loaded; that is, whenever a read operation occurs on a record, and prior to returning the record or page. |
|--------------------|--|

| | |
|----------------|---|
| | <p>These operations include navigating to a record in the UI, reading a record in SOAP web services, and loading a record.</p> <p>The beforeLoad event cannot be used to source standard records. Use the pageInit client script for this purpose. See pageInit.</p> <p>Notes:</p> <ul style="list-style-type: none"> ■ beforeLoad user events cannot be triggered when you load/access an online form. ■ Data cannot be manipulated for records that are loaded in beforeLoad scripts. If you attempt to update a record loaded in beforeLoad, the logic is ignored. ■ Data can be manipulated for records created in beforeLoad user events. ■ Attaching a child custom record to its parent or detaching a child custom record from its parent triggers an edit event. <p>This event can be used with the following context.UserEventType:</p> <ul style="list-style-type: none"> ■ create ■ edit ■ view ■ copy ■ print ■ email ■ quick view |
| Returns | void |
| Since | Version 2015 Release 2 |

Parameters

| | |
|--|---|
|  Note: | The scriptContext parameter is a JavaScript object. |
|--|---|

| Parameter | Type | Required/ Optional | Description | Since |
|-------------------------|------------------------------------|-----------------------|---|------------------------|
| scriptContext.form | serverWidget.Form | required | The current form. | Version 2015 Release 2 |
| scriptContext.newRecord | record.Record | required | The new record. | Version 2015 Release 2 |
| scriptContext.request | http.ServerRequest | optional | The HTTP request information sent by the browser. If the event was triggered by a server action, this value is not present. | Version 2015 Release 2 |
| scriptContext.type | string | required | <p>The type of operation invoked by the event (the trigger type).</p> <p>The type can be any of the possible values for the context.UserEventType enum.</p> <p>This parameter allows the script to branch out to different logic depending on the operation type. For example, a script with deltree logic that deletes a record and all of its child records</p> | Version 2015 Release 2 |

| Parameter | Type | Required/ Optional | Description | Since |
|-----------|------|-----------------------|---|-------|
| | | | should only be invoked when type equals delete. User event scripts should always check the value of the type argument to avoid indiscriminate execution. | |

For an example of the beforeLoad entry point, see [SuiteScript 2.x User Event Script Sample](#).

beforeSubmit(scriptContext)

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|--|
| Description | <p>Defines the function that is executed before a record is submitted; that is, prior to any write operation on the record.</p> <p>Changes made to the current record in this script persist after the write operation.</p> <p>The beforeSubmit event can be used to validate the submitted record, perform any restriction and permission checks, and perform any last-minute changes to the current record.</p> <p>Notes:</p> <ul style="list-style-type: none"> ■ The approve, cancel, and reject argument types are only available for record types such as sales orders, expense reports, timebills, purchase orders, and return authorizations. ■ Only beforeLoad and afterSubmit user event entry point functions execute on the Message record type when a message is created by an inbound email case capture. Scripts set to execute on a beforeSubmit event do not execute. ■ User Event Scripts cannot override custom field permissions. For instance, if a user's role permissions and a custom field's permissions differ, beforeSubmit cannot update the custom field, even if the script is set to execute as Administrator. ■ Attaching a child custom record to its parent or detaching a child custom record from its parent triggers an edit event. <p>This event can be used with the following <code>context.UserEventType</code>:</p> <ul style="list-style-type: none"> ■ create ■ edit ■ xedit (inline editing) ■ delete ■ approve (only available for certain record types) ■ cancel (only available for certain record types) ■ reject (only available for certain record types) ■ pack (only available for certain record types, for example Item Fulfillment records) ■ ship (only available for certain record types, for example Item Fulfillment records) ■ markcomplete (specify this type for a beforeSubmit script to execute when users click Mark Complete on call and task records) ■ reassign (specify this type for a beforeSubmit script to execute when users click Grab on case records) ■ editforecast (specify this type for a beforeSubmit script to execute when users update opportunity and estimate records using the Forecast Editor) |
| Returns | void |

| | |
|--------------|------------------------|
| Since | Version 2015 Release 2 |
|--------------|------------------------|

Parameters

| |
|--|
| Note: The scriptContext parameter is a JavaScript object. |
|--|

| Parameter | Type | Required / Optional | Description | Since |
|-------------------------|---------------|---------------------|---|------------------------|
| scriptContext.newRecord | record.Record | required | The new record. | Version 2015 Release 2 |
| scriptContext.oldRecord | record.Record | required | The old record. | Version 2015 Release 2 |
| scriptContext.type | string | required | The trigger type. Use the context.UserEventType enum to set this value. | Version 2015 Release 2 |

For an example of the beforeSubmit entry point, see [SuiteScript 2.x User Event Script Sample](#).

context.UserEventType

| |
|---|
| Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1. |
|---|

| | |
|-------------------------|---|
| Enum Description | Holds the string values for user event execution contexts. The following user events types are supported for user event entry points: <ul style="list-style-type: none">▪ afterSubmit(scriptContext)▪ beforeLoad(scriptContext)▪ beforeSubmit(scriptContext) |
| Module | SuiteScript 2.x User Event Script Type |
| Since | Version 2015 Release 2 |

Values

| |
|--|
| <ul style="list-style-type: none"> ▪ APPROVE ▪ CANCEL ▪ CHANGEPASSWORD |
| Note: The CHANGEPASSWORD user event is triggered when a user updates their NetSuite password in the NetSuite UI. The password change event is only visible in the NetSuite account in which the password was changed. |

- CREATE
- DELETE
- DROPSHIP
- EDIT
- EDITFORECAST
- EMAIL
- MARKCOMPLETE
- ORDERITEMS
- PACK
- PAYBILLS
- PRINT
- QUICKVIEW
- REASSIGN
- REJECT
- SHIP
- SPECIALORDER
- TRANSFORM
- VIEW
- XEDIT

For an example of the context.UserEventType, see [SuiteScript 2.x User Event Script Sample](#).

 **Note:** Attaching a child custom record to its parent or detaching a child custom record from its parent triggers an edit event.

SuiteScript 2.x Workflow Action Script Type

Workflow action scripts allow you to create custom Workflow Actions that are defined on a record in a workflow. Workflow action scripts are useful for performing actions on sublist because sublist fields are not currently available through the Workflow Manager. Workflow action scripts are also useful when you need to create custom actions that execute complex computational logic that is beyond what can be done with the built-in actions.

For information about SuiteFlow workflows, see the following topics:

- [SuiteFlow Overview](#)
- [Working with Workflows](#)
- [SuiteFlow Reference and Examples](#)

For information about scripting with workflow action scripts, see the help topics [Creating and Using Workflow Action Scripts](#) and [SuiteScript 2.x Workflow Action Script Entry Points and API](#).

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

You can use SuiteScript Analysis to learn about when the script was installed and how it performed in the past. For more information, see the help topic [Analyzing Scripts](#).

Workflow Action Script Sample

This sample shows how to store a return value from a custom action script into a workflow field. This sample could be used if:

- You want to get a value from the Item sublist and use the value as a condition in the workflow. You use record.getSublistValue in the script and return this in the workflow.
- You want to check if a certain item exists in the Item sublist. The script returns "0" if item is not existing and "1" if it does.
- You want to make sure that all items in the Item sublist have a quantity equal to or greater than 1 (similar case as above).

Before using this script, the following must be done:

- Ensure that the script returns a value. You can specify this on the Parameters tab of the Script record page.
- In SuiteFlow, create a workflow field. The field should be of the same type as the return parameter of the Workflow Action script.
- Within a state, add the custom action (this is the Workflow Action script).
- Add the return value from the Workflow Action script to the Store Result In field. This field is found in the custom action's Parameters. See the help topic [Storing a Return Value from a Custom Action Script in a Workflow Field](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType WorkflowActionScript
4 */
5 define([], function() {
6   function onAction(scriptContext){
7     log.debug({
8       title: 'Start Script'
9     });
10    var newRecord = scriptContext.newRecord;
11    var itemCount = newRecord.getLineCount({
12      sublistId: 'item'
13    });
14    log.debug({
15      title: 'Item Count',
16      details: itemCount
17    });
18    for (var i = 0; i < itemCount; i++){
19      var quantity = newRecord.getSublistValue({
20        sublistId: 'item',
21        fieldId: 'quantity',
22        line: i
23      });
24      log.debug({
25        title: 'Quantity of Item ' + i,
26        details: quantity
27      });
28      if (quantity === 0){
29        return 0;
30      }
31    }
32    log.debug({
33      title: 'End Script'
34    });
35    return 1;
36  }
37  return {
38    onAction: onAction
39  }
40 });

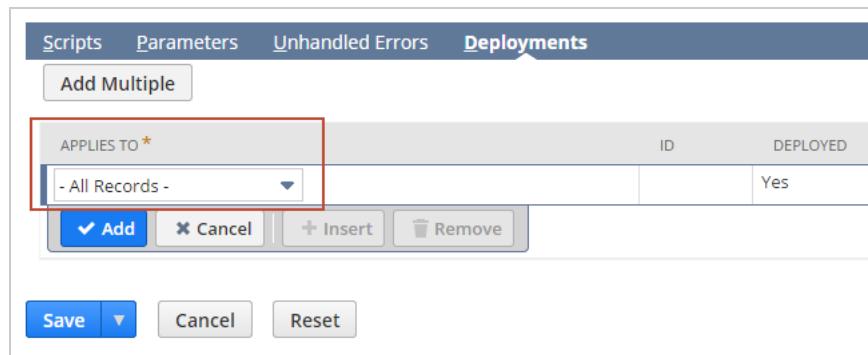
```

See the SuiteFlow help topics for additional samples, such as [Storing a Return Value from a Custom Action Script in a Workflow Field](#). Additional samples are also available by searching “workflow action script” on Suite Answers. Note that some samples may be in SuiteScript 1.0.

Creating and Using Workflow Action Scripts

You define a workflow action script like you would any other SuiteScript type: go to Customization > Scripting > Scripts > New.

Using a workflow action script, you can create generic [Custom Action](#) that are available to all record types. Do this by selecting **All Records** in the Applies To dropdown list on the Script Deployment page.



These custom actions then become available to all workflows, regardless of the underlying record type of the workflow. Through generic custom actions you can (for example) create a parameterized, generic action to set sales rep values. You can then set the parameters from within a workflow and invoke the generic “Set Sales Rep (Custom)” action, which will contain values specific to *that* workflow.

Be aware that if you set a workflow action script to deploy to All Records, and then you try to specify another record type on the script's Script Deployment page, you will receive an error. Also note that if you set the deployment of a workflow action script to All Records, the script will appear in the palette of actions (labeled as **custom**) for all workflows.

You can create parameters for a workflow definition in the customization details in the workflow manager.

You can also set the type of the value returned by a workflow action script. You make this choice on the Parameters subtab on the script record. If there are fields of the same type defined in the workflow (or workflow state), you can configure this value to be saved as the value of a specified field.

Additional Example

The following is a custom action workflow action script that sets the sales rep on the record in the workflow.

```

1 function onAction(scriptContext)
2 {
3     var newRecord = scriptContext.newRecord;
4     var myScript = runtime.getCurrentScript();
5     var scriptParam_salesrep = myScript.getParameter ({
6         name: 'custscript_salesrep'
7     });
8     newRecord.setValue ({
9         fieldId: 'salesrep'
10        value: scriptParam_salesrep
11    });
12 }
```

Notice there is no use of the record type and record id parameters (they are still sent, however); instead, the scriptContext is used to return all necessary data for the record currently in the workflow.

Also note that when executing workflow action scripts, the current record context is **workflow**. See the help topic [runtime.executionContext](#) for details on returning context information about what triggered the current script.

SuiteScript 2.x Workflow Action Script Entry Points and API

[onAction\(scriptContext\)](#)

onAction(scriptContext)

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

| | |
|--------------------|---|
| Description | Defines a Workflow Action script trigger point. For a script sample, see Workflow Action Script Sample . |
| Returns | void |
| Since | Version 2016 Release 1 |

Parameters

 **Note:** The scriptContext parameter is a JavaScript object.

| Parameter | Type | Required / Optional | Description | Since |
|--------------------------|-------------------|---------------------|--|--------|
| scriptContext.newRecord | record.Record | required | The new record. record.Record.save() is not permitted. | 2016.1 |
| scriptContext.oldRecord | record.Record | required | The old record. record.Record.save() is not permitted. | 2016.1 |
| scriptContext.form | serverWidget.Form | optional | The current form that the script uses to interact with the record. This parameter is available only in the beforeLoad context. | 2016.2 |
| scriptContext.type | string | optional | An event type, such as create, edit, view, or delete. | 2016.2 |
| scriptContext.workflowId | integer | optional | The internal ID of the workflow that calls the script. | 2016.2 |

SuiteScript 2.x SDF Installation Script Type

The SDF installation script type is used to perform tasks during deployment of a SuiteApp from SuiteCloud Development Framework (SDF) to your target account. You can use this script type to perform setup,

configuration, and data management tasks that would otherwise have to be completed by account administrators, such as:

- Setting up and verifying the target account
- Verifying that required permissions and features are enabled before installation
- Preventing installation if proper setup has not occurred
- Stopping installation if a requirement is missing in the account
- Migrating existing data (for instance, data that will be needed by a SuiteApp)
- Programmatically creating fields on a custom record depending on enable features in the account

The SDF installation script type is similar to the bundle installation script type. However, this script type is designed for use with SuiteApps that are developed and deployed with the SDF, rather than with SuiteBundler.

For details about SDF installation scripts, see the following help topics:

- [SDF Installation Script Requirements](#)
- [Managing SDF Installation Scripts in SDF](#)
- [SDF Installation Script Samples](#)
- [SuiteScript 2.x SDF Installation Script Entry Points and API](#)
-  [Executing Installation Scripts](#)

SDF Installation Script Requirements

To execute an SDF installation script during your SuiteApp project deployment, the script must be associated with a script record and a script deployment record. For more information, see the help topic [Creating a Script Record](#).

Script records and script deployment records can be created in several ways: (1) within your SDF project as XML representations, (2) using SuiteScript, and (3) from within your NetSuite account. For more information, see the help topics [SDF Installation Scripts as XML Definitions](#) and [SuiteCloud Development Framework XML Reference](#).

SDF installation scripts are governed by a maximum of 10,000 units per execution. For more information on governance, see the help topics [Script Type Usage Unit Limits](#) and [SuiteScript Governance and Limits](#).

Managing SDF Installation Scripts in SDF

You can use SuiteCloud Development Framework to manage SDF installation scripts as part of file-based customization projects. You can also incorporate SDF installations scripts into your SuiteApp project deployment. And, SuiteApp projects can include SDF installation scripts, which are automatically executed when the project is deployed. For more information, see the following help topics:

- [SuiteCloud Development Framework Overview](#)
- [Customizing SuiteApp Project Deployment using SDF Installation Scripts](#)
- [SDF Installation Scripts as XML Definitions](#)
- [Defining a Script Record and Deployment for an SDF Installation Script](#)
- [Customizing SuiteApp Project Deployment using SDF Installation Scripts](#)

SDF installation scripts and deployments can be monitored using the deploy log in the SuiteCloud IDE or the Execution Log on the Script or Script Deployment page. For more information, see the help topics [Deployment Logs](#) and [Handling Exceptions and Monitoring SDF Installation Scripts](#).

Copy to Account Support for SDF Installation Scripts

You can use the Copy to Account feature to copy an individual SDF installation script to another of your accounts. Each SDF installation script page has a clickable Copy To Account option in the upper right corner. For information on Copy to Account, see the help topic [Copy to Account Overview](#).

SDF Installation Script Samples

SDF installation script samples are available in the following SuiteCloud Development Framework help topics:

- [SDF Installation Script Example of Validating an Account Preference Before Installation](#)
- [SDF Installation Script Example of Customizing a SuiteApp Update](#)

SuiteScript 2.x SDF Installation Script Entry Points and API

The following entry point is defined for the SDF installation script type:

| Script Entry Point | Description |
|------------------------------------|---|
| run(scriptContext) | Defines what is executed when the script is specified to be run by the SDF deployment (in the deploy.xml file of an SDF project). |

run(scriptContext)

| | |
|--------------------|---|
| Description | Defines what is executed when the script is specified to be run by the SDF deployment (in the deploy.xml file of an SDF project). |
| Returns | void |
| Since | 2015.2 |

Parameters

| |
|--|
| Note: The scriptContext parameter is a JavaScript object. |
|--|

| Parameter | Type | Description | Since |
|-------------|--------|--|--------|
| fromVersion | string | The version of the SuiteApp currently installed on the account. Specify Null if this is a new installation. | 2015.2 |
| toVersion | string | The version of the SuiteApp that will be installed on the account. | 2015.2 |

For examples of the run entry point, see [SDF Installation Script Samples](#).

You can also define custom script parameters by adding a parameter to the script record (defined as a custom script field) and setting the parameter to a particular value on the deployment. The following code example shows how to retrieve a script parameter from within an SDF installation script:

```
1 | /**
```

```
2  * @NApiVersion 2.0
3  * @NScriptType SDFInstallationScript
4  */
5
6 define(['N/runtime'], function(runtime) {
7     function run(params) {
8         var customParam = runtime.getCurrentScript().getParameter({name: 'custscript1'});
9     }
10    return {
11        run: run
12    };
13 });
});
```

See the help topic [Creating Script Parameters \(Custom Fields\)](#) for more information on script parameters.
See the help topic [SDF Installation Script Example of Customizing a SuiteApp Update](#) for an example.

SuiteScript 2.x Record Actions and Macros

For more information, see the following topics:

- [Overview of Record Action and Macro APIs](#)
- [Supported Record Macros](#)
- [Supported Record Actions](#)

Overview of Record Action and Macro APIs



Note: The content in this topic pertains to SuiteScript 2.0.

SuiteScript 2.0 supports APIs that provide the programmatic equivalent of clicking a button in the NetSuite user interface. With the record action and macro APIs, you can use SuiteScript to trigger the same business logic that a UI button click triggers. The record action and macro APIs can increase productivity by automating regular tasks that previously had to be done manually in the UI.

NetSuite records offer two alternatives for executing native NetSuite logic: a user either clicks a UI button or runs a script that calls the API corresponding to the button. These script and UI alternatives both produce the same results. Macro and action APIs provide ease and flexibility for your scripting. These APIs are supported for all SuiteScript 2.0 client and server script types. Macro and action APIs also can lower governance usage because they can execute complex business logic in one API call instead of multiple API calls. For actions, governance is charged per individual action executed, and varies depending upon the type of action. For macros, no governance is charged, because changes executed by macros are saved as part of record submits.

Record actions provide a convenient way to update the state of one or more records that are in view mode. Changes that the execution of an action API makes to records are persisted in the database immediately. It is not necessary to take into account required roles, permissions, or other conditions for a record action to execute. The conditions required to execute an action are embedded in the record action API. If the conditions are not met, the action does not execute. Approve and reject are two example use cases for record actions. These actions can be applied to a single record or to multiple records of the same type, as a bulk process. When an approve or reject action is executed on a record, the approval status of the record is saved immediately.

Record macros provide an automated way to execute business logic on a record as it is edited. Changes that the execution of a macro API makes to a record are not persisted until the record is saved. An example use case for a record macro is a preview of the calculated tax amount for a sales order's items. This macro API executes after items are entered on a sales order. It results in the display of the calculated tax amount on the sales order. However, the tax amount is not saved until the record is saved. A macro API is applied only to a single record at a time. After changes to the record are saved, changes to other dependent records may occur as a result.

You need to use two different types of APIs to call record macros and actions in your scripts:

- Generic APIs to get and execute actions or macros:
 - Record action APIs are part of the [N/action Module](#). They include generic members for getting and executing actions on a record type. For details, see the help topic [N/action Module](#).
 - Record macro APIs are part of the [N/record Module](#). They include generic members for getting and executing macros on a record type. For details, see the help topics [Record Object Members](#) and [Macro Object Members](#).
- Individual APIs that implement specific logic on a specific record type:

- A limited number of individual actions for specific record types are supported. For details, see [Supported Record Actions](#).
- A limited number of individual macros for specific record types. For details, see [Supported Record Macros](#).



Note: For information about working with record actions in REST web services, see the help topic [Executing Record Actions \(Beta\)](#). REST web services support for actions is available as a BETA feature.



Executing Actions and Macros Using SuiteScript 2.0

Supported Record Actions



Note: The content in this topic pertains to SuiteScript 2.0.

Record actions provide a convenient way to update the state of one or more records that are in view mode. For more information, see [Overview of Record Action and Macro APIs](#).

Record action APIs are part of the [N/action Module](#). For more information, see the help topic [N/action Module](#).

The following table lists currently supported record actions. They are ordered by record type and include the label on the corresponding UI button. For details about a specific action, click its Action ID.



Note: For information about working with record actions in REST web services, see the help topic [Executing Record Actions \(Beta\)](#). In 20.1, REST web services support for actions is available as a BETA feature.

| Record Type | Action ID | UI Button Label |
|---------------------|------------------------------|----------------------|
| Bonus | cancel | Exclude from Payroll |
| Charge | clearBudgetAmounts | Clear Values |
| Charge | distributeBudgetTotalAmount | Distribute Total |
| Charge | selectAllBudgetLines | Mark All |
| Charge | setBudgetAmountsToCalculated | Set to Calculated |
| Charge | unselectAllBudgetLines | — |
| Invoice | markforgrouping | Mark for Grouping |
| Invoice | ungroupinvoice | Remove From Grouping |
| Invoice | unmarkforgrouping | Unmark for Grouping |
| Invoice Group | groupinvoices | Group Invoices |
| Invoice Group | unlinkinvoices | Unlink Invoices |
| Revenue Arrangement | allocate | Allocate |
| Supply Change Order | approve | Approve all pending |
| Supply Change Order | reject | Reject all pending |

| | | |
|------------------------|---------|---------------------|
| Supply Plan Definition | launch | Launch |
| Time (TimeBill) | approve | Approve |
| Time (TimeBill) | reject | Reject |
| Time (TimeBill) | retract | Retract |
| Time (TimeBill) | submit | Submit |
| Weekly Timesheet | approve | Approve All Pending |
| Weekly Timesheet | reject | Reject All Pending |
| Weekly Timesheet | retract | Retract |
| Weekly Timesheet | submit | Submit |

Supported Record Macros

 **Note:** The content in this topic pertains to SuiteScript 2.0.

Record macros provide an automated way to execute business logic on a record as it is edited. For more information, see [Overview of Record Action and Macro APIs](#).

Record macro APIs are part of the [N/record Module](#). For more information, see the help topics [Record Object Members](#) and [Macro Object Members](#).

The following table lists currently supported record macros. They are ordered by record type and include the label on the corresponding UI button. For details about a specific macro, click its Macro ID.

| Record Type | Macro ID | UI Button Label |
|---|----------------------|---------------------------|
| Billing Rate Card | modifyPriceByPercent | Recalculate |
| Project Charge Rule | copyResources | Copy Resources from Tasks |
| Project Work Breakdown Structure Macros | getAmountFieldValue | Get Amount Field Value |
| Project Work Breakdown Structure Macros | getAmountsFields | Get Amounts Fields |
| Project Work Breakdown Structure Macros | getUnmatchedActuals | Get Unmatched Actuals |
| Project Work Breakdown Structure Macros | setAmountFieldValue | Set Amount Field Value |
| Sales Order | autoAssignLocations | Auto Assign Locations |
| Transaction Record Macros | calculateTax | Preview Tax |
| Transaction Record Macros | getSummaryTaxTotals | Get Summary Tax Totals |
| Weekly Timesheet | checkTimeLimits | Check Time Limits |
| Weekly Timesheet | copyFromWeek | Copy From Week |

SuiteScript 2.x JSDoc Validation

Note: If you are currently using SuiteScript 2.0 for your scripts, consider updating your scripts to use SuiteScript 2.1. This version includes new language features that are provided by the ECMAScript 2019 specification. For more information, see [SuiteScript 2.1](#) and [SuiteScript Versioning Guidelines](#).

JSDoc 3 is a documentation generator for JavaScript source code. Developers insert specific comment blocks into their source code, and these comment blocks can be extracted to generate documentation. These comment blocks start with `/**` and end with `*/`.

JSDoc also includes its own markup language, which is made up of JSDoc tags. These tags are prefaced with the `@` symbol. The JSDoc tags are added to the comment blocks, and they are used to specify the various entities of a JavaScript file (for example, `@param`).

```

1 /**
2  * Creates a file
3  * @param {string} name - file name
4  * @param {string} fileType - file type
5  * @param {string} contents - file content
6  * @returns {Object} file.File
7 */

```

JSDoc parses the source code for each comment block. JSDoc 3 also lets users create custom JSDoc tags. These tags can be defined to trigger events (for example, displaying a certain page).

SuiteScript 2.x includes several custom tags, such as `@NApiVersion` and `@NScriptType`. Some of these tags are required in each entry point script uploaded to NetSuite. To learn about these tags, see [SuiteScript 2.x JSDoc Tags](#).

When a SuiteScript 2.x script record is requested, NetSuite uses JSDoc 3 to evaluate the entry point script and parse the code for the required JSDoc tag. This tag is used to validate the SuiteScript version.

Note: SuiteScript 2.x users can use JSDoc 3 to create their own documentation for scripts, custom modules, and SuiteApps. To take advantage of this tool, developers must download JSDoc 3 from the official website. For additional information on JSDoc 3, see <http://usejsdoc.org/>.

JSDoc Comment Blocks

To be recognized as valid JSDoc content, JSDoc tag comment blocks must start with `/**` and end with `*/`. JSDoc tags consist of a key-value pair. The key is a string starting with `@`, and the key ends with the first white space after this string. The value starts with the next non-whitespace character and ends with the next carriage return. Each comment line in the block starts with `*`.

The following table lists examples of valid and invalid JSDoc formatting.

| Valid Examples | Invalid Examples |
|---|---|
| <pre> 1 /** 2 * @NApiVersion 2.x 3 */ </pre> | <pre> 1 /* 2 * @NApiVersion 2.x 3 */ </pre> <p>The JSDoc tag comment block does not start with <code>/**</code>.</p> |

| Valid Examples | Invalid Examples |
|----------------|--|
| | <pre>1 // @NApiVersion 2.x</pre> <p>The JSDoc tag comment is a single-line comment and not a block.</p> |
| | <pre>1 /** 2 * @NApiVersion 2.x*/</pre> <p>The JSDoc tag comment block does not include a carriage return after the value.</p> |

SuiteScript 2.x JSDoc Tags

The following table describes the available SuiteScript 2.x JSDoc tags. SuiteScript 2.x entry point scripts must include the following two tags:

- `@NApiVersion`
- `@NScriptType`

For more information about entry point validation, including possible errors, see [SuiteScript 2.x Entry Point Script Validation](#).

| JSDoc Tag | Possible Values | Required/ Optional | Description |
|---------------------------|---|---|---|
| <code>@NApiVersion</code> | 2.0 2.1 2.x 2.X | Required for entry point scripts Optional for custom modules | <p>This tag is used in two ways:</p> <ul style="list-style-type: none"> ■ For SuiteScript entry point scripts, this tag is a required declaration. It indicates to NetSuite the SuiteScript version to use. ■ For SuiteScript custom modules that are not entry point scripts, this tag is an optional declaration. It can serve as a defense against future incompatible versions of SuiteScript (versions 3.x and higher) attempting to load it. <p>The 2.x value usually represents the latest version of SuiteScript that is generally available and does not represent any versions that are released as beta features. However, this does not apply to SuiteScript 2.1. In this release, the 2.x value indicates that a script uses SuiteScript 2.0, not SuiteScript 2.1. You can still use SuiteScript 2.1 and all of its features in your scripts, but your 2.x scripts will not run as SuiteScript 2.1 scripts until a future release. For more information about SuiteScript versioning, see SuiteScript Versioning Guidelines.</p> |
| <code>@NScriptType</code> | BundleInstallationScript ConsolidatedRateAdjustorPlugin CustomGLPlugin ClientScript EmailCapturePlugin MapReduceScript | Required for entry point scripts | This tag identifies the type of script defined in the file. |

| JSDoc Tag | Possible Values | Required/ Optional | Description |
|---------------|--|-----------------------|---|
| | MassUpdateScript PaymentGatewayPlugin PluginTypeImpl Portlet PromotionsPlugin Restlet ScheduledScript ShippingPartnersPlugin Suitelet TaxCalculationPlugin UserEventScript WorkflowActionScript | | |
| @NModuleScope | SameAccount TargetAccount Public | Optional | <p>This tag is used to control access to scripts and custom modules.</p> <ul style="list-style-type: none"> ■ If the value is set to SameAccount, access to the module is limited to other modules from the same bundle, as well as modules native to the same source account and any associated sandboxes and Release Preview accounts. Source code is not hidden at runtime. ■ If the value is set to TargetAccount, access to the module is limited to other modules from the same bundle, as well as modules native to the same source account, target account, and any associated sandboxes and Release Preview accounts. Source code is hidden at runtime. ■ If the value is set to Public, any script in the account can load and use the module. Source code is hidden at runtime. <p>The default value is SameAccount.</p> <p>For more information, see Controlling Access to Scripts and Custom Modules.</p> |



Tip: If you are currently using SuiteScript 2.0 for your scripts, consider updating your scripts to use SuiteScript 2.1. This version includes new language features that are provided by the ECMAScript 2019 specification. For more information, see [SuiteScript 2.1](#) and [SuiteScript Versioning Guidelines](#)

Controlling Access to Scripts and Custom Modules



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

You can define the scope of environments (associated NetSuite accounts, sandboxes, and bundles) that may access a script. Access refers to whether the module can be loaded and invoked by another module.

Modules are either loaded by the NetSuite system (via entry point modules) or by other modules (as libraries).



Important: Before deploying a SuiteApp, make sure that you review the module scope. This practice will help you to avoid:

Deploying a SuiteApp that doesn't work as expected because it can't access a necessary module in another bundle.

Providing wider access than desired to third parties and other accounts.

To set the scope for a script, you must include the appropriate value using the JSDoc tag. This tag is optional but recommended.

If you do not set the scope, SameAccount applies as the default. For information about adding a JSDoc tag, see [SuiteScript 2.x JSDoc Validation](#).

The following table lists and describes the access control modifiers (supported module scopes) that are available in SuiteScript 2.x:

| ModuleScope Value | Description | Access Disallowed When: | Visibility of Source Code | Examples |
|-------------------|---|--|---------------------------|---|
| SameAccount | <p>Limits script access to modules native to the same environment in which the script was created and uploaded.</p> <p>This environment includes the account from which a bundle is installed and can also include the related family of sandbox accounts. For more information about sandbox accounts, see the help topic Understanding NetSuite Account Types.</p> <p>A script file that is 'native' to the environment is added to an account using an authenticated user session.</p> | A bundled module from a different source account attempts to import the module. | Visible during runtime | <ul style="list-style-type: none"> ■ Installing a customization from a single sandbox to a production environment that is linked to the same account. ■ Distributing a bundle with private business logic as an ISV (Independent Software Vendor). |
| TargetAccount | <p>Limits script access to modules native to the same source or target environment as the script.</p> <p>A source environment includes the NetSuite account (and any associated sandboxes and Release Preview accounts) from which a bundle is installed into another account.</p> <p>A target environment includes the NetSuite account (and any</p> | A bundled module that is not native to the target or source account attempts to import the module. | Hidden during runtime | <ul style="list-style-type: none"> ■ Account administrators distributing private business logic between accounts they control, and the modules in the bundle are needed by other modules created in the target account ■ Distributing a bundle with public APIs intended for import only by modules native to the target account. |

| ModuleScope Value | Description | Access Disallowed When: | Visibility of Source Code | Examples |
|-------------------|--|---|---------------------------|--|
| | associated sandboxes and Release Preview accounts) into which a bundle is installed (whether via Bundle Copy or Bundle Install). | | | |
| Public | Any bundle (native and third party) that is installed in the account can run the script. | The script is not installed in the account. | Hidden during runtime | <ul style="list-style-type: none"> ■ Customers installing customizations from multiple sandbox accounts to production, where modules from different sandboxes need to load each other. ■ Installing a customization from a development account to a sandbox or production account. ■ Developing open source code. |

For more information about packaging and distributing bundles (SuiteApps), see the help topic [SuiteBundler Overview](#).

NetSuite Development Accounts and Module Scope



Warning: If you use a NetSuite Development account to work on a module that you intend to distribute or test, NetSuite recommends that you choose a module scope value rather than accept the default.

Keep in mind that development accounts are limited to 5 users and isolated from production and sandbox accounts. Set the value to Public if other bundles or accounts depend on using a module that is sourced from a development account.

If the default module scope is used on a script in a development account that is packaged into a bundle and installed to production or sandbox accounts, that script will not be accessible to modules installed from other NetSuite accounts. For example, when developers use multiple developer accounts to collaborate on a project, the SameAccount module scope might not be an appropriate access control level. This scope prevents modules installed from different accounts from loading one another.

For more information about development accounts, see the help topics [The Development Account](#), [NetSuite Development Accounts](#), and the [SuiteApp Development Process with SuiteBundler](#).

SuiteScript 2.x Entry Point Script Creation and Deployment

After you write an entry point script, you must take the following steps to run the script in your NetSuite account:

1. Upload the script file to your File Cabinet. The file must have all required script elements as described in [SuiteScript 2.x Entry Point Script Validation](#).
2. Deploy your script:
 - Deploy your script on one or more record types, as described in [SuiteScript 2.x Record-Level Scripts](#).
 - If your script is a client script that is to be deployed only at the form level, follow the steps described in [SuiteScript 2.x Form-Level Scripts](#).

To better understand the difference between deploying a client script at the record level versus the form level, see [Record-Level and Form-Level Scripts](#). Note that only client scripts can be deployed at the form level. All other types of entry point script types can be deployed at the record level only.

Record-Level and Form-Level Scripts



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

A client script can be deployed either at the record level or at the form level.

When you deploy a client script at the record level, you deploy it globally on one or more record types. The script runs on all forms associated with that record type. By contrast, with a form-level deployment, you attach the script to a custom form associated with a record type, and the script runs only when that custom form is used.

For example, you could use record-level deployment to configure a client script to run on the employee record type. With this approach, the script runs whenever the employee record is used, regardless of which form is being used. With record-level deployment, it is also possible to limit the script by configuring it to be available only to certain audiences. With this approach, the script runs only when the record is used by people in certain roles, groups, or other classifications, as configured on the Audience subtab of the script deployment record. However, with record-level deployment, you cannot limit the script to only one form. The script behaves the same way on all forms associated with the record type.

By contrast, you could attach the client script to only one custom entry form for the employee record type. If a user then opened an employee record using the standard entry form, the script would not run. You can attach a client script to any custom entry form, custom transaction form, or custom address form. However, you cannot limit the audience for a form-level script.

Hidden fields from the form cannot be accessed or manipulated by client scripts.

Note that only client scripts can be deployed at the form level. All other entry point scripts can be deployed at the record level only.

For details about deploying a script at the record level, see [SuiteScript 2.x Record-Level Scripts](#). For details about deploying a client script at the form level, see [SuiteScript 2.x Form-Level Scripts](#).



Note: Record-level client scripts can also be used on forms and lists that have been generated through the use of Suitelets. Form-based client scripts cannot be used by Suitelets.

SuiteScript 2.x Entry Point Script Validation



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

For a SuiteScript 2.x entry point script to be usable, the script must contain certain required elements. The system parses your file for these elements when you upload the file to the File Cabinet. At that time, the system also saves data about your file, which is used later when you create a script record based on the script.

If NetSuite detects that an element within the script file is missing or formatted incorrectly, it returns an error. Errors can be returned at the time you upload the file to the File Cabinet, or when you try to create a script record for the script file. Errors can also be returned when you attach a client script to a custom form.

For more details, see the following help topics:

- [Entry Point Script Validation Guidelines](#)
- [Entry Point Script Validation Examples](#)
- [Entry Point Script Validation Error Reference](#)

Entry Point Script Validation Guidelines



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

For an entry point script to be valid, its structure must meet certain requirements and the script must include two required JSDoc tags: @NApiVersion and @NScriptType.

If the script does not meet all requirements, you cannot create a script record for the script, nor can you attach it to a form. In some cases, you are not permitted to upload the script file. And, if a correctly formatted script has been attached to a script record or a custom form, you are not permitted to edit and save changes to the script file that would introduce errors.

For more information, see the following help topics:

- [Entry Point Script Validation Terms and Overview](#)
- [Correct Structure for Entry Point Scripts](#)
- [Required JSDoc Tags for Entry Point Scripts](#)

Entry Point Script Validation Terms and Overview

When learning about script validation requirements and errors, it is important to understand certain terms. These terms are described in the following illustration and table.

```


    /**
     * @NApiVersion 2.x
     * @NScriptType UserEventScript
     */
    define(['N/record'],
        function(record) {
            function beforeLoad(context) {
                if (context.type !== context.UserEventType.CREATE)
                    return;
                var customerRecord = context.newRecord;
                customerRecord.setValue('phone', '555-555-5555');
                if (!customerRecord.getValue('salesrep'))
                    customerRecord.setValue('salesrep', 46);
            }
            function beforeSubmit(context) {
                if (context.type !== context.UserEventType.CREATE)
                    return;
                var customerRecord = context.newRecord;
                customerRecord.setValue('comments', 'Please follow up with this customer!');
            }
            function afterSubmit(context) {
                if (context.type !== context.UserEventType.CREATE)
                    return;
                var customerRecord = context.newRecord;
                if (customerRecord.getValue('salesrep')) {
                    var call = record.create({
                        type: record.Type.PHONE_CALL,
                        isDynamic: true
                    });
                    call.setValue('title', 'Make follow-up call to new customer');
                    call.setValue('assigned', customerRecord.getValue('salesrep'));
                    call.setValue('phone', customerRecord.getValue('phone'));
                    try {
                        var callId = call.save();
                        log.debug('Call record created successfully', 'Id: ' + callId);
                    } catch (e) {
                        log.error(e.name);
                    }
                }
            }
        }
    );
    return {
        beforeLoad: beforeLoad,
        beforeSubmit: beforeSubmit,
        afterSubmit: afterSubmit
    };
});


```

| Callout | Description |
|---------|---|
| 1 | JSDoc tags |
| 2 | JSDoc tag values |
| 3 | entry point function definitions |
| 4 | interface (also called script type interface) |
| 5 | entry points |
| 6 | entry point functions (also called script type functions) |

At a high level, the following rules apply:

- Each script must have a structure that meets certain structural requirements, as described in [Correct Structure for Entry Point Scripts](#).
- Each script file must include two JSDoc tags: @NApiVersion and @NScriptType, and they must have valid values. Additionally, the @NScriptType tag must be compatible with the script included in the file.. For more details, see [Required JSDoc Tags for Entry Point Scripts](#).

Correct Structure for Entry Point Scripts

An entry point script cannot be associated with a script record or custom form unless the script meets certain requirements. For the script to be valid, all of the following must be true:

- The script must include one and only one properly structured define statement. The script cannot include multiple define statements.
- The define statement cannot include direct references to SuiteScript 2.0 objects or enums. All such references must be wrapped in a function.
- The script's return statement must include an interface whose entry points are associated with one and only one script type. In other words, the return statement must include only one script type interface.
- The interface for the return statement must include at least one entry point.
- Each entry point must correspond with an entry point function.
- All entry point functions must be defined in the same file.
- The script type interface implemented must match the @NScriptType value.
- The script must not contain any JavaScript syntax errors.

Required JSDoc Tags for Entry Point Scripts

Every entry point script must contain the JSDoc tags described in the following table. This table includes details about some possible validation errors. For full details about validation errors, see [Entry Point Script Validation Error Reference](#).

| JSDoc Tag | Possible Values | Purpose | Possible Validation Errors |
|--------------|--|---|--|
| @NApiVersion | 2.0 2.1 2.x 2.X | Identifies the SuiteScript version to use. 2.x and 2.X are equivalent. | If you omit this tag within a file that is otherwise properly formatted for SuiteScript 2.0, the upload of the script fails. For details, see FAILED_TO_VALIDATE_SCRIPT_FILE . If you use an invalid value for this tag, the upload of the script fails. For details, see INVALID_API_VERSION . |
| @NScriptType | BundleInstallationScript ConsolidatedRateAdjustorPlugin CustomGLPlugin ClientScript EmailCapturePlugin MapReduceScript MassUpdateScript PaymentGatewayPlugin PluginTypeImpl Portlet PromotionsPlugin Restlet ScheduledScript ShippingPartnersPlugin | Identifies the type of script defined in the file. | If you use an @NScriptType value that is incompatible with the entry points included in the script, the upload of the script fails. For details, see WRONG_SCRIPT_TYPE . If you omit this tag within a file that is otherwise properly formatted for SuiteScript 2.0, you can upload the file, but you cannot create a script record for it, nor can you attach it to a form. For details, see FAILED_TO_VALIDATE_SCRIPT_FILE . |

| JSDoc Tag | Possible Values | Purpose | Possible Validation Errors |
|-----------|---|---------|----------------------------|
| | Suitelet TaxCalculationPlugin UserEventScript WorkflowActionScript | | |

For full details about working with JSDoc tags in SuiteScript 2.x, including details on the correct format, see [SuiteScript 2.x JSDoc Validation](#).

 **Tip:** If you are currently using SuiteScript 2.0 for your scripts, consider updating your scripts to use SuiteScript 2.1. This version includes new language features that are provided by the ECMAScript 2019 specification. For more information, see [SuiteScript 2.1](#) and [SuiteScript Versioning Guidelines](#).

Entry Point Script Validation Examples

 **Note:** This topic applies to SuiteScript 2.0.

The following examples show correct and incorrect code samples.

Includes All Required Elements Correct Example

The following script is valid. It includes a define statement, a return statement with an entry point, and an entry point function that corresponds with the entry point. Additionally, the value of the @NScriptType JSDoc tag matches the script type interface used, and the @NApiVersion JSDoc tag is correct.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType UserEventScript
4 */
5
6 define(['N/record'], (record) => {
7     function myBeforeSubmitFunction(context) {
8         if (context.type !== context.UserEventType.CREATE)
9             return;
10        let customerRecord = context.newRecord;
11        customerRecord.setValue({
12            fieldId: 'comments',
13            value: 'Please follow up with this customer!'
14        });
15    }
16    return {
17        beforeSubmit: myBeforeSubmitFunction
18    };
19});
```

Missing Return Statement Incorrect Example

The following script is invalid because it does not have a return statement. If you try to upload a script structured this way, the system returns the error "SuiteScript 2.0 entry point scripts must implement one script type function." For details, see [SCRIPT_OF_API_VERSION_20_MUST](#)

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType UserEventScript
4 */
5
6 define(['N/record'], (record) => {
```

```

7  function myBeforeSubmitFunction(context) {
8      if (context.type !== context.UserEventType.CREATE)
9          return;
10     let customerRecord = context.newRecord;
11     customerRecord.setValue({
12         fieldId: 'comments',
13         value: 'Please follow up with this customer!'
14     });
15 }
16 });

```

Missing Define Statement Incorrect Example

The following script is invalid because it uses a require statement instead of a define statement. If you try to upload a script structured like this, the system returns the error "SuiteScript 2.0 entry point scripts must implement one script type function." For details, see [SCRIPT_OF_API_VERSION_20_MUST](#) For more information on using the require statement, see the help topic [require Function](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType UserEventScript
4 */
5
6 require(['N/record'], (record) => {
7     function myBeforeSubmitFunction(context) {
8         if (context.type !== context.UserEventType.CREATE)
9             return;
10        var customerRecord = context.newRecord;
11        customerRecord.setValue('comments','Please follow up with this customer!');
12    }
13 });

```

Missing @NScriptType JSDoc tag Incorrect Example

The following script is invalid because it does not have an @NScriptType JSDoc tag. The system would permit you to upload this script file. However, if you try to create a script record for this script, the system returns the error "@NScriptType is mandatory for 2.0 entry point script." For details, see [MISSING_SCRIPT_TYPE](#).

```

1 /**
2  * @NApiVersion 2.x
3  */
4
5 define([ 'N/record' ], (record) => {
6     function myBeforeSubmitFunction(context) {
7         if (context.type !== context.UserEventType.CREATE)
8             return;
9         let customerRecord = context.newRecord;
10        customerRecord.setValue('comments','Please follow up with this customer!');
11    }
12    return {
13        beforeSubmit : myBeforeSubmitFunction
14    };
15 });

```

Missing @NApiVersion JSDoc tag Incorrect Example

The following script is invalid because it does not have an @NApiVersion JSDoc tag. If you try to upload a script structured like this, the system returns the error "Failed to validate script file." For details, see [FAILED_TO_VALIDATE_SCRIPT_FILE](#).

```

1 /**
2  * @NScriptType UserEventScript

```

```

3  */
4
5 define(['N/record'], function(record) {
6     function myBeforeSubmitFunction(context) {
7         if (context.type !== context.UserEventType.CREATE)
8             return;
9         var customerRecord = context.newRecord;
10        customerRecord.setValue('comments','Please follow up with this customer!');
11    }
12    return {
13        beforeSubmit : myBeforeSubmitFunction
14    };
15 });

```

Entry Point Script Validation Error Reference

 **Note:** This topic applies to SuiteScript 2.0.

The following table describes validation errors that can be returned when working with entry point scripts. These errors can be thrown when you upload an entry point script, when you create a script record, or when you attach a client script to a custom form. Some errors can also be returned when you edit script files that have already been uploaded to NetSuite, attached to script records, or attached to custom forms.

| Error Code | Full Description |
|--|---|
| CANNOT_CHANGE_API_VERSION | This file is used by a SuiteScript {1} script; you cannot change the API version of the file. {1} indicates the SuiteScript version (i.e., 1.0, 2.0, etc). |
| FAILED_TO_VALIDATE_SCRIPT_FILE | Failed to validate script file: {1} {1} indicates the script file name. |
| INVALID_API_VERSION | Invalid JSDoc tag value; valid values are: @NApiVersion [2.X, 2.x, 2.0, 2.1] |
| INVALID_JSDOC_TAG_VALUE | Invalid JSDoc tag value; valid values are: {1} {1} indicates the valid values based on the specific invalid JSDoc tag. |
| MISSING_SCRIPT_TYPE | The @NScriptType JSDoc tag is required for 2.0 entry point scripts. |
| MULTIPLE_DEFINE_CALLS | Invalid define call, define should only be called one time per module. Define calls found at the following line numbers: {1} {1} indicates the lines in the script file. |
| SCRIPT_OF_API_VERSION_20_CANNOT_IMPLEMENT_MORE_THAN_ONE_SCRIPT_TYPE_INTERFACES (See SCRIPT_OF_API_VERSION_20_CANNOT ...) | SuiteScript 2.0 entry point scripts cannot implement functions for more than one script type. |
| SCRIPT_OF_API_VERSION_20_MUST_IMPLEMENT_A_SCRIPT_TYPE_INTERFACE (See SCRIPT_OF_API_VERSION_20_MUST) | SuiteScript 2.0 entry point scripts must implement one script type function. |
| SS_V2_FILE_USED_FOR_FORM_SCRIPTS_MUST_IMPLEMENT_CLIENT_SCRIPT_TYPE (See SS_V2_FILE_USED_FOR_FORM_SCRIPT) | SuiteScript version 2 file used for form scripts must implement client script type |
| SYNTAX_ERROR | Syntax error: {1} {1} indicates additional information specific to the error. |

| Error Code | Full Description |
|--|---|
| THE_FILE_IS_USED_AS_SCRIPT_TYPE_1_CANNOT_BE_CHANGED_TO_2 (See THE_FILE_IS_USED_AS_SCRIPT_TYPE_1 ...) | The file is used as script type {1}, cannot be changed to {2} {1} and {2} indicate the script types. |
| WRONG_SCRIPT_TYPE | Script file includes @NScriptType {1}; this script type cannot be used to {2}. {1} indicates the script type specified. {2} indicates the script's entry points. |

CANNOT_CHANGE_API_VERSION

| | |
|------------------|---|
| Full Error Text: | This file is used by a SuiteScript {1} script; you cannot change the API version of the file. {1} indicates the SuiteScript version (i.e., 1.0, 2.0, etc). |
| Occurs: | This error is displayed when you attempt to edit a script file that is already associated with a script record or a custom form. Specifically, if you try to edit and save changes to the @NApiVersion JSDoc tag value. |
| Example: | If you try to remove the expression @NApiVersion 2.0 from a version 2.0 script, the system displays this error. If you try to add @NApiVersion 2.0 expression to a version 1.0 script, the system displays this error. |
| Correction: | NA |
| Other Notes: | In some cases, this error is preceded by the INVALID_API_VERSION error, which is included if the new @NApiVersion JSDoc tag value you are trying to use is invalid. Note that SuiteScript 1.0 files do not use the @NApiVersion JSDoc tag. If you are still using SuiteScript 1.0 scripts, you should consider converting them to SuiteScript 2.0. For help creating and uploading SuiteScript 1.0 files, see the help topic Running SuiteScript 1.0 in NetSuite Overview . |

FAILED_TO_VALIDATE_SCRIPT_FILE

| | |
|------------------|--|
| Full Error Text: | Failed to validate script file: {1} {1} indicates the script file name. |
| Occurs: | This error is displayed if your script file is not structured correctly. For example, you see this error if your script is formatted correctly for SuiteScript 2.0 in all ways except that it lacks the @NApiVersion JSDoc tag. Also, this error may be displayed if you do not correctly initialize global variables, as described in Resolve Error: "SuiteScriptModuleLoaderError FAIL_TO_EVALUTE_SCRIPT_1 . |
| Correction: | Check your script file to make sure that it contains all required elements and is structured correctly. |
| Other Notes: | See SuiteScript 2.x Entry Point Script Validation for more information on creating valid scripts. |

INVALID_API_VERSION

| | |
|------------------|--|
| Full Error Text: | Invalid JSDoc tag value; valid values are: @NApiVersion [2.X, 2.x, 2.0, 2.1] |
|------------------|--|

| | |
|--------------|---|
| Occurs: | This error is displayed when you attempt to upload a script file with an invalid @NApiVersion JSDoc tag value. It can also be displayed if you try to modify the value of the @NApiVersion JSDoc tag in a script file that has already been uploaded. |
| Correction: | Check your script file to make sure the @NApiVersion JSDoc tag has a valid value. Valid values are: 2.0, 2.x, and 2.X; 2.x and 2.X are equivalent. |
| Other Notes: | For more information on the @NApiVersion JSDoc tag values, see SuiteScript 2.x JSDoc Tags . |

INVALID_JSDOC_TAG_VALUE

| | |
|------------------|---|
| Full Error Text: | Invalid JSDoc tag value; valid values are: {1} {1} indicates the valid values based on the specific invalid JSDoc tag. |
| Occurs: | This error is displayed if your script file uses an invalid value for any JSDoc tag. |
| Correction: | Check your script file to make sure the value you used for the JSDoc tag is valid. Verify that it does not include typos or other errors. |
| Other Notes: | For more information on JSDoc tag values, see SuiteScript 2.x JSDoc Tags . |

MULTIPLE_DEFINE_CALLS

| | |
|------------------|---|
| Full Error Text: | Invalid define call, define should only be called one time per module. Define calls found at the following line numbers: {1} {1} indicates the lines in the script file. |
| Occurs: | This error is displayed if your script includes more than one define call. Each script can have only one define call. |
| Correction: | Check your script file to make sure there is only one define call. If your script file includes multiple define calls, it must be reworked. |
| Other Notes: | For more information on using define calls, see the help topic define Object . |

MISSING_SCRIPT_TYPE

| | |
|------------------|---|
| Full Error Text: | @NScriptType is mandatory for 2.0 entry point script |
| Occurs: | This error is displayed if you attempt to create a script record based on a SuiteScript 2.0 script that does not include the @NScriptType JSDoc tag. This error is also displayed if you remove the @NScriptType JSDoc tag from a script file that is attached to a script record. |
| Correction: | Edit the script file and add the @NScriptType JSDoc tag with the appropriate value. And, do not remove the @NScriptType JSDoc tag from any script file that is already attached to a script record. |
| Other Notes: | This error is displayed when you create a script record for your script file. You will be able to successfully upload your script file to the File Cabinet, but you will not be able to create a script record for it. In this case, successfully uploading a script file does not mean that the file is valid. |

SCRIPT_OF_API_VERSION_20_CANNOT ...

| | |
|------------------|---|
| Full Error Text: | SuiteScript 2.0 entry point scripts cannot implement functions for more than one script type. |
|------------------|---|

| | |
|--------------|---|
| | The full code for this error is: SCRIPT_OF_API_VERSION_20_CANNOT_IMPLEMENT_MORE_THAN_ONE_SCRIPT_TYPE_INTERFACES |
| Occurs: | This error is displayed if the interface portion of your script file includes entry points from more than one script type. |
| Correction: | Check your script to make sure the interface contains only entry points for one script type (the script type indicated in the @NScriptType JSDoc tag). If necessary, create additional script files to implement addition script type entry points. |
| Other Notes: | See SuiteScript 2.x Script Types for descriptions of all entry points for each script type. |

SCRIPT_OF_API_VERSION_20_MUST ...

| | |
|------------------|--|
| Full Error Text: | SuiteScript 2.0 entry point scripts must implement one script type function. The full error code for this error is: SCRIPT_OF_API_VERSION_20_MUST_IMPLEMENT_A_SCRIPT_TYPE_INTERFACE. |
| Occurs: | This error signifies that your script is missing an interface or that an error exists within the interface. This error can be returned when you try to upload a file, or when you try to edit a file that was previously uploaded. |
| Correction: | Check your script file to make sure that at least one entry point for the script type (as indicated in the @NScriptType JSDoc tag) is included in the interface. |
| Other Notes: | See SuiteScript 2.x Script Types for descriptions of all entry points for each script type. |

SS_V2_FILE_USED_FOR_FORM_SCRIPT

| | |
|------------------|--|
| Full Error Text: | SuiteScript 2.0 file used for form scripts must implement client script type. The full error code for this error is: SS_V2_FILE_USED_FOR_FORM_SCRIPTS_MUST_IMPLEMENT_CLIENT_SCRIPT_TYPE. |
| Occurs: | This error is displayed if you try to attach a script type other than a client script to a custom form. Only a client script can be deployed at the form level (i.e., attached to a custom form). All other types of entry point scripts can be deployed only at the record level. |
| Correction: | Make sure that the script you are attempting to attach to a custom form is a client script. Otherwise, deploy the script at the record level. |
| Other Notes: | For details about working with client scripts on custom form, see SuiteScript 2.x Form-Level Scripts . For details about deploying other types of entry point scripts at the record level, see SuiteScript 2.x Record-Level Scripts . |

SYNTAX_ERROR

| | |
|------------------|--|
| Full Error Text: | Syntax error: {1} {1} indicates additional information specific to the error. |
| Occurs: | This error is displayed if your script contains JavaScript syntax errors. |
| Correction: | Check to make sure your script follows all JavaScript syntax rules. |
| Other Notes: | There are several only resources available for JavaScript syntax rules including two JavaScript tutorial sites at: https://www.w3schools.com/js/default.asp and https://developer.mozilla.org/en-US/docs/Web/JavaScript . |

THE_FILE_IS_USED_AS_SCRIPT_TYPE_1 ...

| | |
|------------------|---|
| Full Error Text: | The file is used as script type {1}, cannot be changed to {2}. {1} and {2} indicate the script types. The full error code for this error is: THE_FILE_IS_USED_AS_SCRIPT_TYPE_1_CANNOT_BE_CHANGED_TO_2. |
| Occurs: | This error may be displayed if you edit a script file that is attached to an existing script record to use a different script type interface from what it previously used, and a different @NScriptType value. |
| Correction: | To avoid this problem, make your changes in a script file that is not already associated with a script record. Then create a new script record based on the updated script file. |
| Other Notes: | NA |

WRONG_SCRIPT_TYPE

| | |
|------------------|--|
| Full Error Text: | Script file includes @NScriptType {1}; this script type cannot be used to {2} {1} indicates the script type specified. {2} indicates the script's entry points. |
| Occurs: | This error is displayed if the @NScriptType JSDoc tag value is not compatible with the entry points in the script's interface. |
| Correction: | Check your script to make sure the @NScriptType JSDoc tag value in your script corresponds to the entry points used by your script. If the tag's value does not correspond to the script type interface, you cannot upload the file to the File Cabinet. |
| Other Notes: | See SuiteScript 2.x Script Types to access script type definitions which include descriptions of all entry points for the script type. |

SuiteScript 2.x Record-Level Scripts

After you have written a valid entry point script, as described in [SuiteScript 2.x Entry Point Script Validation](#), you must take the following steps if you want to deploy the script on records in your NetSuite account:

1. Create a script record for your script file, as described in [Script Record Creation](#). The script record identifies the script's internal ID, whether or not the script is active, and other details.
2. Deploy the script, as described in [Script Deployment](#). You use script deployments to configure details regarding how and when the script runs. The types of choices you make vary depending on the script's type.

Script Record Creation



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

After you have written an entry point script, you can create a script record for it. You must create a script record before you can deploy your script.

When you create a script record, you set some fields manually and the system automatically fills in other fields based on data in the script file. These automatically filled fields are described in the following table.

| Field on the Script Record | Value Taken From |
|----------------------------------|---|
| API Version | The @NApiVersion JSDoc tag in your script file. |
| Script File | The name of your script file. |
| Type | The @NScriptType JSDoc tag in your script file. |
| Functions (on the Script subtab) | The entry points defined within your script. |

Note: Your script must be properly structured and annotated before it can be used to create a script record. For details, see [SuiteScript 2.x Entry Point Script Validation](#).

To create a script record, go to Customization > Scripting > Scripts > New. Select the appropriate file in the Script File list, then click Create Script Record. A new script record is created.

The Type, API Version, and Script File fields are already populated as described in the table above. To complete the script record, enter a name and set any optional fields as needed. Then click Save.

After you save the script record, the system shows the new script record on the Script page in view mode. The Scripts subtab lists all possible entry point functions that could exist for the script type used in your script. The entry points used in your script are checked.

The screenshot shows the 'Script' creation page. At the top, there are buttons for 'Edit', 'Back', 'Deploy Script', 'Download XML', and 'Actions'. Below these are sections for 'TYPE' (User Event), 'NAME' (New User Event Script), 'PACKAGE', 'ID' (customscript_usereventscript), and 'API VERSION' (2.0). On the right, there are 'DESCRIPTION', 'OWNER' (John Smith), and an 'INACTIVE' checkbox. At the bottom, a 'Scripts' subtab is selected, showing a list of entry point functions: BEFORE LOAD FUNCTION, BEFORE SUBMIT FUNCTION, and AFTER SUBMIT FUNCTION, all of which are checked.

The entry point functions listed on the Scripts subtab cannot be edited. To add or remove an entry point function, you will need to edit the script and add or remove the appropriate functions. When you save your changes, the script record's list of entry point functions is updated to match the contents of the script file.

For more detailed help creating a script record, see the following help topics:

- [Creating a Script Record](#)

- Working With Script Parameters

Creating a Script Record



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The process of creating a script record varies depending on the script's type. The following procedure shows general steps that apply to all types. For certain script types, additional steps may be required.

To create a script record:

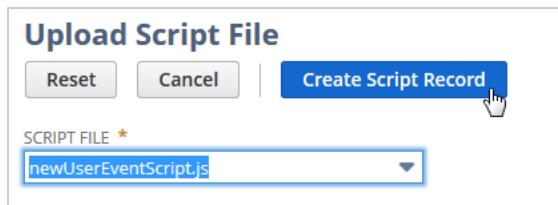
- Go to Customization > Scripting > Scripts > New.
The system displays the Upload Script File page.
- In the **Script File** list, select the appropriate SuiteScript (.js) file. The list shows all versions of SuiteScript script files that you have uploaded to your File Cabinet.
If the file you want to use has not been uploaded yet, you can upload it from this page. Point to the area at the right of the list to display a Plus icon.

The screenshot shows a 'Upload Script File' dialog box. At the top are 'Reset' and 'Cancel' buttons, followed by a 'Create Script Record' button. Below is a dropdown menu labeled 'SCRIPT FILE *' with a plus sign icon. A 'New' button is located at the bottom right of the dropdown menu.

Click the Plus icon to display a popup window that allows you to select and upload your .js script file from your local environment. Fill in the required fields to select your file and then click **Save**.

The screenshot shows a 'File' dialog box in a browser window. It includes fields for 'ATTACH FROM *' (set to 'Computer'), 'FILE NAME', 'FOLDER *' (set to 'SuiteScripts'), 'URL', 'SELECT FILE' (with a 'Choose File' button), and 'CHARACTER ENCODING' (set to 'Unicode (UTF-8)'). To the right are several checkboxes for file properties: INACTIVE, AVAILABLE FOR SUITEBUNDLES, HIDE IN SUITEBUNDLE, AVAILABLE WITHOUT LOGIN, COMPANY-WIDE USAGE, and GENERATE URL TIME STAMP. At the top left are 'Save' and 'Cancel' buttons.

3. After you have selected the file, click **Create Script Record**.



The system displays the Script page, with your .js script file listed on the Scripts subtab. Fields that are automatically filled in based on your script (API Version, Script File, Type, and Functions) are read-only and cannot be changed. Remember, these fields are automatically filled in based on the content of your script file.



Note: If the system redirects you to a page that prompts you to select a script type, that means that in Step 2 you identified a SuiteScript 1.0 file. You can click the Back button in your browser to return to the previous page and select a different file. If the script you want to use is a 1.0 version script, you should convert it to SuiteScript 2.0.

4. Enter a name in the **Name** field.
 5. In the **ID** field, optionally enter a custom ID for the script record. If the **ID** field is left blank, a system-generated internal ID is created for you when you save the record.
 6. If the **Portlet Type** field is displayed, select the appropriate value. This field is available only for the portlet script type, and it is a required field.
 7. In the **Description** field, optionally enter a description for the script.
 8. If appropriate, change the value of the **Owner** field. By default, this field is set to the currently logged-in user.
- Note that after the script record is saved, only the owner or a system administrator can modify the script record.
9. If appropriate, check the **Inactive** box. Marking a script record as inactive, prevents the script from executing. When a script is set to Inactive, any deployments associated with the script are also inactive.
 10. The **Scripts** subtab shows your script file name. You can select a different script file using icons next to the **Script File** field.
 11. On the **Parameters** subtab, define any parameters (custom fields) that are used by functions in the script. For information on script parameters, see [Working With Script Parameters](#).

12. On the **Unhandled Errors** subtab, optionally define the people to be notified if any script errors occur.

Three types of error notifications are sent:

- An initial email is sent on the first occurrence of an error within an hour.
- A second email with aggregated error information for every hour is sent. (The error counter is reset when this email is sent.)
- A third email is sent for the first 100 errors that have occurred after the error counter is set to 0.

For example, if an error is thrown 130 times within an hour, an initial email is sent. Then, after the 100th occurrence, another email is sent. Since there are an additional 30 occurrences within the same hour, a final summary email is sent at the end of the hour. During the second hour, if there are only 50 occurrences of the error, only one summary email is sent at the end of that hour.

By default, the **Notify Script Owner** box is checked. You can also identify other people to receive notifications, as follows:

- Check the **Notify All Admins** box, if all administrators should be notified.
- Select one or more groups from the **Groups** list. To define new groups, go to Lists > Relationships > Groups > New.
- Enter the email addresses of any other users who should be notified. You can enter a comma-separated list of email addresses.

13. Optionally, define a deployment for the script record by clicking the **Deployments** subtab and adding a line to the sublist. For details on adding a line to this list, see [Deploying a Script by using the Deployments Sublist](#). You can also add a script deployment without adding a deployment to this list and clicking **Save and Deploy** when you are ready to save your script. This will display the Script Deployment page where you will define your deployment.
14. If this is a client script, optionally add lines to the **Buttons** sublist, which appears on the **Scripts** subtab. You can translate the labels for these buttons. This feature is available in accounts with multi-language feature only.
15. If the **Scripts** subtab includes the **Custom Plug-in Types** sublist, optionally add lines to this list.
16. To save the new record, click one of the following
 - **Save** – to save and close the record.
 - **Save and Deploy** – to save the script record and open the Script Deployment page that lets you create a new script deployment record.

Working With Script Parameters

Script parameters can be used if you want your script to be configurable, either through script deployment or by the end user. Script parameters should be created whenever you need to parameterize a script that is deployed multiple times. Using script parameters makes it easy to customize the behavior of the script for each deployment. For more information on using script parameters, see the help topic [Creating Script Parameters \(Custom Fields\)](#).

Script Deployment



Note: This topic applies to SuiteScript 2.0.

Before an entry point script will run in your NetSuite account, it must be deployed. You can deploy a script when you create a script record, or you can deploy it later. The deployment settings available vary depending on the script type and on how you deploy the script.

When you deploy a script, NetSuite creates a script deployment record. These records are listed at Customization > Scripting > Script Deployments. Deployments are also represented as lines on the Deployments subtab of the script record.

Ensure that you have set the correct permissions and features. For more information, see the help topic [Setting Roles and Permissions for SuiteScript](#).

In a script deployment record, you can set up context filtering to determine how and when the associated script runs. You set up context filtering on the Context Filtering tab when you create or edit a script deployment record. There are two types of context filtering:

- **Execution context filtering** — Execution contexts provide information about how a script is triggered to execute. For example, a script can be triggered in response to an action in the NetSuite application, or an action occurring in another context, such as a web services integration. For more information, see the help topic [Execution Contexts](#).
- **Localization context filtering** — Localization contexts provide information about the countries or regions in which a script can execute. For example, you can specify that a script should run only on records that are associated with Canada. NetSuite automatically determines the localization context for records and transactions based on their values for country fields such as subsidiary and tax nexus. For more information, see the help topic [Record Localization Context](#).

For more information about setting up context filtering, see the help topic [Using the Context Filtering Tab](#).

Multiple deployments can be created for the same script record. When multiple deployments exist, they are executed in the order in which they are listed on the Deployments sublist. This sequence typically corresponds with the order in which the deployments were created.

For more details, see the following topics:

- [Methods of Deploying a Script](#)
- [Deploying a Script by using the Deployments Sublist](#)
- [Updating a Script Deployment](#)
- [Managing Web Store Performance Impact](#)

Methods of Deploying a Script



Note: This topic applies to SuiteScript 2.0.

To run an entry point script in your NetSuite account, you must create a script deployment for it. The deployment determines how and when the script runs. You can create a script deployment in any of the following ways:

- Using the Deployments sublist (see [Work with the Deployments Sublist](#)).
- Using the script deployment record (see [Use the Script Deployment Record](#)).
- Using N/record module methods (see [Use N/record Module Methods](#)).

In most cases, you can also use the above methods to edit a script deployment.

Work with the Deployments Sublist

You can deploy a script when you create a script record by using the Deployments sublist on the Script record page. This approach lets you define values for many deployment fields, although not all. For some script types, you can also use the Deployments sublist when you are editing an existing script record. However, for other script types, the Deployments sublist is read-only when you are editing the script record. For more information on using the Deployment sublist, see [Deploying a Script by using the Deployments Sublist](#).

Use the Script Deployment Record

You can deploy a script by using the script deployment record. You may want to use this approach for the following reasons:

- The script deployment record lets you access a greater number of fields than the Deployments sublist.
- If you have already created a script record, in some cases it is not possible to edit the Deployments sublist.

To define or edit a script deployment:

1. When you save your script record, you can immediately create a script deployment record by selecting **Save and Deploy** from the script record **Save** button.
If you want to update a deployment that already exists, go to Customization > Scripting > Script Deployments and select **Edit** next to the deployment.
2. On the Script Deployment page:
 - For Suitelet, scheduled, and portlet scripts, provide a name for the deployment in the **Title** field.
 - For user event and client scripts, select the record the script will run against in the **Applies To** field. In the **Applies To** field, you can also select **All Records** to deploy the script to all records that officially support SuiteScript. (For a list of these records, see the help topic [SuiteScript Supported Records](#).)
3. In the **ID** field, if desired, enter a custom scriptId for the deployment. If you do not create a scriptId, a system-generated internalId is created for you.
4. (Optional) Clear the **Deployed** box if you do not want to deploy the script. The box is checked by default. A script will only run in NetSuite when the **Deployed** box is selected.
5. In the **Status** field, set the script deployment status. See the help topic [Setting Script Deployment Status](#).
6. (Optional) In the **Event Type** list, specify an event type for the script execution. See the help topic [Setting Script Execution Event Type from the UI](#).
7. (Optional) In the **Log Level** field, specify which log messages will appear on the **Execution Log** tab after the script is executed. See the help topic [Setting Script Execution Log Levels](#).
8. In the **Execute as Role** field, select whether you want the script to execute using Administrator privileges, regardless of the permissions of the currently logged in user. See the help topic [Executing Scripts Using a Specific Role](#).
9. Check the **Available Without Login** box (for Suitelets only) to generate an External URL on save of this deployment. See the help topic [Setting Available Without Login](#).



Note: External URLs use account-specific domains. As a best practice, use `url.resolveScript(options)` to discover the Suitelet URL instead of hard-coding when possible.

10. On the **Audience** tab, specify the audiences for the script. See the help topic [Defining Script Audience](#).
11. (Optional) On the **Links** tab (for Suitelets only), create a menu link for the Suitelet if you want to launch your Suitelet from the UI. See [Suitelet Script Deployment Page Links Subtab](#).
12. On the **Context Filtering** tab, specify the execution and localization contexts for the script. See the help topic [Using the Context Filtering Tab](#).
13. (Optional) On the **Execution Log** tab, create custom views for all script logging details. See the help topic [Using the Script Execution Log Tab](#).
14. Click **Save**.

Note that for portlet scripts, you must enable the portlet to display on your dashboard (see the help topic [Custom Portlets](#)).

Use N/record Module Methods

You can create a deployment programmatically by using the `record.create(options)` method. When creating a script deployment record, set the `options.type` parameter to `record.Type.SCRIPT_DEPLOYMENT`. Similarly, if you want to modify a script deployment record programmatically, you can load it by using `record.load(options)`. You can also modify an existing deployment record by using other [N/record Module](#) methods.

For help with the field IDs available on the script deployment record, refer to the [SuiteScript Records Browser](#).

Deploying a Script by using the Deployments Sublist

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

You can deploy a script when you create a script record by using the Deployments sublist on the Script record page. This approach lets you define values for many deployment fields, although not all. For example, if you want to set up context filtering (execution context filtering or localization context filtering), you must do it in the script deployment record. For more information, see the help topic [Using the Context Filtering Tab](#). For some script types, you can also use the Deployments sublist when you are editing an existing script record. However, for other script types, the Deployments sublist is read-only when you are editing the script record.

You can use the Deployments sublist when you create a script record that you want to deploy. Depending on the script type, you can also use the Deployments sublist when you are editing an existing script record.

Ensure that you have set the correct permissions and features. For more information, see the help topic [Setting Roles and Permissions for SuiteScript](#).

 **Note:** For information on other ways of deploying a script, see [Methods of Deploying a Script](#).

Be aware that the script deployment process varies somewhat depending on the script type. The following procedure describes basic steps. For certain script types, additional steps may be required.

To deploy a script by using the Deployments sublist:

1. If the script record is not already open for editing, open it. Go to Customization > Scripting > Scripts. Locate the script for which you want to create a script deployment. Click the corresponding **Edit** link.
2. Click the **Deployments** subtab.
3. Add a value to the sublist, as follows:
 - If the **Title** column is displayed, enter a title.
 - If the **Applies to** column is displayed, select the appropriate value in the list. Specifically, select the record type where you want to deploy the script. To deploy the script on all record types supported in SuiteScript, select **All Records**.
 - Enter a meaningful name in the **ID** column. The record's ID lets you work with the deployment programmatically. Note that the system automatically adds a prefix of `customdeploy` to the value you enter. If you do not specify an ID, a system-generated ID is created for you.
 - Check or clear the **Deployed** box, as appropriate. For most script types, the default value is **Yes**. However, for map/reduce and scheduled scripts, the default is **Not Scheduled**.
 - In the **Status** column, select the appropriate deployment status. Note that the available values vary depending on the script type.

- If the **Event Type** list is displayed, optionally select a value from the list. This value identifies the event type that triggers the script execution.
- Optionally, in the **Log Level** field, specify which type of log messages will appear on the **Execution Log** tab when the script is executed.
- If the **Execute as Role** column appears, optionally select whether you want the script to execute using Administrator privileges, regardless of the permissions of the currently logged-in user.
- Click **Add** to add the new line to the sublist.

4. Click **Save**.

The system saves the deployment. If you view the Deployments subtab again, the deployment you just created is represented as a link. You can click the link to view the script deployment record that has been created. In some cases, this page lets you configure additional fields. For details, see [Updating a Script Deployment](#).

Updating a Script Deployment



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

You can edit an existing script deployment. For some script types, you can make changes by editing the script record's Deployments sublist. However, for other types, the Deployments sublist is read-only. For these script types, you must open the full script deployment record for editing. You can do this programmatically using the [N/record Module](#), or you can do this using the script deployment record entry form. This topic describes updating the deployment using the script deployment record entry form.

When you open the script deployment record for editing, you have access to more fields than are available on the Deployments sublist. For example, the script deployment record includes the following subtabs:

- **Audience** — When a script is deployed, it runs only in the accounts of the specified audience. This subtab lets you specify the roles that make up the script audience. In SuiteScript 2.0, this subtab is available for the following script types: client, portlet, RESTlet, Suitelet, and user event.
- **Scripts** — For a client or user event script, you can view a summary of the scripts that are associated with the script deployment record.
- **Links** — For a Suitelet, you can specify the menu paths that permit users to access the Suitelet.
- **Schedule** — For a map/reduce or scheduled script, you can configure the schedule that determines when the script runs.
- **Context Filtering** — You can set up context filtering to determine how and when a script runs. See the help topic [Using the Context Filtering Tab](#).
- **Execution Log** — You can use to log data when your script executes.

Additionally, the script deployment record lets you view details of how the deployment has changed over time. You can view these details on the History and System Notes subtabs. The System Notes tab also lists the contexts that applied each time the script executed. For more information, see the help topic [Execution Contexts](#).



Important: If the script associated with the deployment is running in NetSuite, you cannot edit the deployment. You must wait until the script stops running.

To update a script deployment:

1. Go to Customization > Scripting > Script Deployments.

2. Locate the deployment you want to edit, and click the corresponding **Edit** link.
3. Make the appropriate changes.
4. Click **Save**.

Managing Web Store Performance Impact

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

When deploying a user event script, you can permit or prevent the deployment from being triggered by web store activity. You configure this behavior by specifying execution contexts on the Context Filtering tab of the script deployment record. Execution contexts let you control whether the deployment is triggered by particular events, such as UI actions or web store activity. For more information, see the help topic [Execution Contexts](#).

The following execution contexts apply to web store activity:

- Web Application
- Web Store

To prevent a deployment from being triggered by web store activity, you should not select the Web Application or Web Store on the Context Filtering tab. Scripts can significantly slow web store performance. By disassociating scripts from web store activity, you can improve web store response times.

You can set up execution context filtering when you deploy a script. You can also set up execution context filtering by editing a script deployment record, as described in the following procedure.

To enable or disable web store triggers for a user event script deployment:

1. Open the deployment record for editing. For example, go to Customization > Scripting > Script Deployments. Locate the appropriate deployment record and click the corresponding **Edit** link.
2. On the script deployment record, on the **Context Filtering** tab, specify the execution contexts for the script. You can cancel the selection of web store contexts (Web Application and Web Store) to prevent the script from triggering in response to web store activity.
3. Click **Save**.

 **Note:** Another option for optimizing web store performance is the Asynchronous afterSubmit Sales Order Processing feature. When you enable this feature, all afterSubmit user events and workflows triggered by web store checkout run asynchronously. For details, see the help topic [Commerce Features](#).

Viewing System Notes

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

 **Note:** This topic applies to System Notes only. For information about viewing System Notes v2, see the help topic [Viewing System Notes v2](#).

You can review details about the history of a script or script deployment record. These details identify the user who created the record, the context in which the record was created, and the date of the last change to the record.

To view a script deployment's history:

1. Do one of the following:
 - To find a script, go to Customization > Scripting > Scripts.
 - To find a script deployment, go to Customization > Scripting > Script Deployments.
2. Locate the record you want to view and click **View**.
3. Click the **System Notes** subtab.

For more information about system notes, see the help topic [System Notes Overview](#).

 **Note:** In previous releases, details about these records was listed on its **History** subtab. However, that subtab is no longer updated. New activity is captured on the record's System Notes subtab.

SuiteScript 2.x Form-Level Scripts

In some cases, you might want a client script to deploy on only one form. To do this, you attach the script to the form. You can attach a script to a custom entry form, a custom transaction form, or a custom address form.

Only client scripts can be attached to forms. All other types of entry point scripts can be deployed at the record level. A script deployed at the record level is used on all forms associated with that record type. By contrast, when you attach a client script to a form, it runs on that form only. For more details about the differences between form-level and record-level deployments, see [Record-Level and Form-Level Scripts](#). For help with record-level deployment, see [SuiteScript 2.x Record-Level Scripts](#).

With both custom entry forms and custom transaction forms, you can also include logic in the script to create a custom action such as a button or a menu item. For custom address forms, you can deploy the script on the form, but you cannot configure custom actions.

For details on deploying a client script at the form level and creating custom actions, see the following help topics:

- [Attaching a Client Script to a Form](#)
- [Configuring a Custom Action](#)

Attaching a Client Script to a Form

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

You can attach a client script to a form after you have made sure that it is a valid script, as described in [SuiteScript 2.x Entry Point Script Validation](#).

After you attach a script to a form, it is deployed on that form and will run when the form is used. You can attach a client script to a custom entry form, a custom transaction form, or a custom address form.

 **Important:** Users must have at least the Edit level SuiteScript permission to attach a script to a custom form by editing the Custom Code tab of the form record. Users with the View level SuiteScript permission can see the Custom Code tab, but they cannot edit it.

To attach a client script to a form:

1. Navigate to the form to which you want to attach the script. For example, go to Customization > Forms > Entry Forms. Locate the appropriate form and click the corresponding **Edit** link.

2. Go to the **Custom Code** subtab.
3. In the **Script File** list, select the SuiteScript 2.x script that you want to attach. If the file you want to use has not yet been uploaded to the File Cabinet, you can upload it from this page. To upload a file, point to the area at the right of the list to display a Plus icon. Click this icon to display a dialog box. You can use this dialog box to upload a file from your local environment.

After you select the file, it is displayed in the Script File field. The **SuiteScript API Version** field gets updated based on your selected file. The page also updates to include a list of all possible client script entry point functions. Check marks are displayed next to the functions that are used by your script.

| Subtabs | Field Groups | Fields | Actions | Lists |
|---|--------------|--------|---------|-------|
| SCRIPT FILE | | | | |
| <input type="text" value="VendorPageInit.js"/> ▼ | | | | |
| SUITESCRIPT API VERSION | | | | |
| 2.0 | | | | |
| <input checked="" type="checkbox"/> PAGE INIT FUNCTION <input type="checkbox"/> SAVE RECORD FUNCTION <input type="checkbox"/> VALIDATE FIELD FUNCTION <input type="checkbox"/> FIELD CHANGED FUNCTION <input type="checkbox"/> POST SOURCING FUNCTION <input type="checkbox"/> LINE INIT FUNCTION <input type="checkbox"/> VALIDATE LINE FUNCTION <input type="checkbox"/> VALIDATE INSERT FUNCTION <input type="checkbox"/> VALIDATE DELETE FUNCTION <input type="checkbox"/> RECALC FUNCTION | | | | |

The boxes on the **Custom Code** subtab cannot be edited directly. To check or clear the boxes, edit the script to add or remove functions as appropriate. When you save your changes, the script record's boxes are updated to match the contents of the script file.

4. Click **Save**.

Configuring a Custom Action



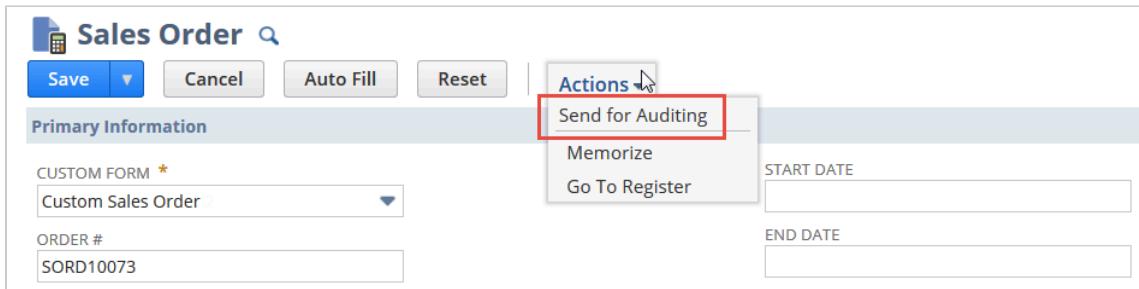
Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

In some cases, you might want to configure a custom action for a custom form. A custom action can be either a custom button or a custom menu item.

A custom button appears at the top of the page:

| Save | Cancel | Auto Fill | Reset | Actions |
|--|--------|-----------|-------|--|
| Primary Information | | | | <input type="button" value="Send for Auditing"/> <input type="button" value="Memorize"/> <input type="button" value="Go To Register"/> |
| CUSTOM FORM * | | | | START DATE |
| <input type="text" value="Custom Sales Order"/> ▼ | | | | <input type="text"/> |
| ORDER # | | | | END DATE |
| <input type="text" value="SORD10073"/> | | | | <input type="text"/> |

A custom menu item appears as an option when the user points to the Actions label:



You configure custom action elements by using logic contained in the client script attached to your form.

Custom actions can be configured for custom entry and custom transaction forms only. A custom action cannot be configured for a custom address form.

To configure a custom action on a custom form:

1. If the custom form is not already open for editing, open it. For example, go to Customization > Forms > Transaction Forms. Locate the appropriate form and click the corresponding **Edit** link.
2. Go to the **Actions** subtab.
3. Go to the **Custom Actions** subtab.
4. In the **Label** column, enter a label for your button or menu item.
5. In the **Function** column, enter the name of the appropriate entry point function from your client script.
6. In the **Display as** column, select **Button** or **Menu** as appropriate.
7. Click **Save**.



Note: For more information about both custom actions and standard actions, see the help topic [Configuring Buttons and Actions](#).

SuiteScript 2.x Custom Modules

- [Custom Modules Overview](#)
- [Writing a Custom Module](#)
- [Preparing to Add a Custom Module](#)
- [Module Dependency Paths](#)
- [Naming a Custom Module](#)
- [Custom Module Examples](#)
- [Troubleshooting Errors](#)
- [Custom Modules Frequently Asked Questions](#)

Custom Modules Overview

With SuiteScript 2.x, you have the ability to create custom modules (including third-party, AMD, and non-AMD modules). This supports modular and well-structured organization of code that is easier to read and modify. It also lets you build and access additional, customized API functionality.

Build a custom module to help you do the following:

- Group reusable functions into a common library file. Supporting code (such as helper functions and custom APIs) can be organized into reusable sections. These custom modules are loaded within your entry point script.
- Add custom modules to SuiteApps and expose those modules to third parties.
- Import a third-party API.
- Organize and separate utilities from business logic.



Note: Custom modules are not supported in bundle installation scripts.

Writing a Custom Module

From the source suitescript file that defines your custom module, you will need to use the [define Object](#). Then, to load and use the module, specify the dependency with a [require Function](#) from an entry point script. You must also define a function that instantiates the module.

If desired, you can add standard and custom JSDoc tags to your custom module scripts. JSDoc tags are optional (unless the script is an entry point script). However, NetSuite recommends using the @NApiVersion and @NModuleScope tags in your custom modules. The SuiteScript API version tag can help protect against loading from future incompatible versions of SuiteScript (versions 3.x and higher). The module scope tag determines whether other scripts and accounts can access the custom module.

For custom JSDoc tags, SuiteScript 2.x users can use JSDoc to create their own documentation for scripts, custom modules, and SuiteApps. To take advantage of this tool, developers must download JSDoc from the official website. For additional information on JSDoc, see <http://usejsdoc.org/>.

For script samples, see [Custom Module Examples](#).

To learn about setting up a custom module name, see [Naming a Custom Module](#).

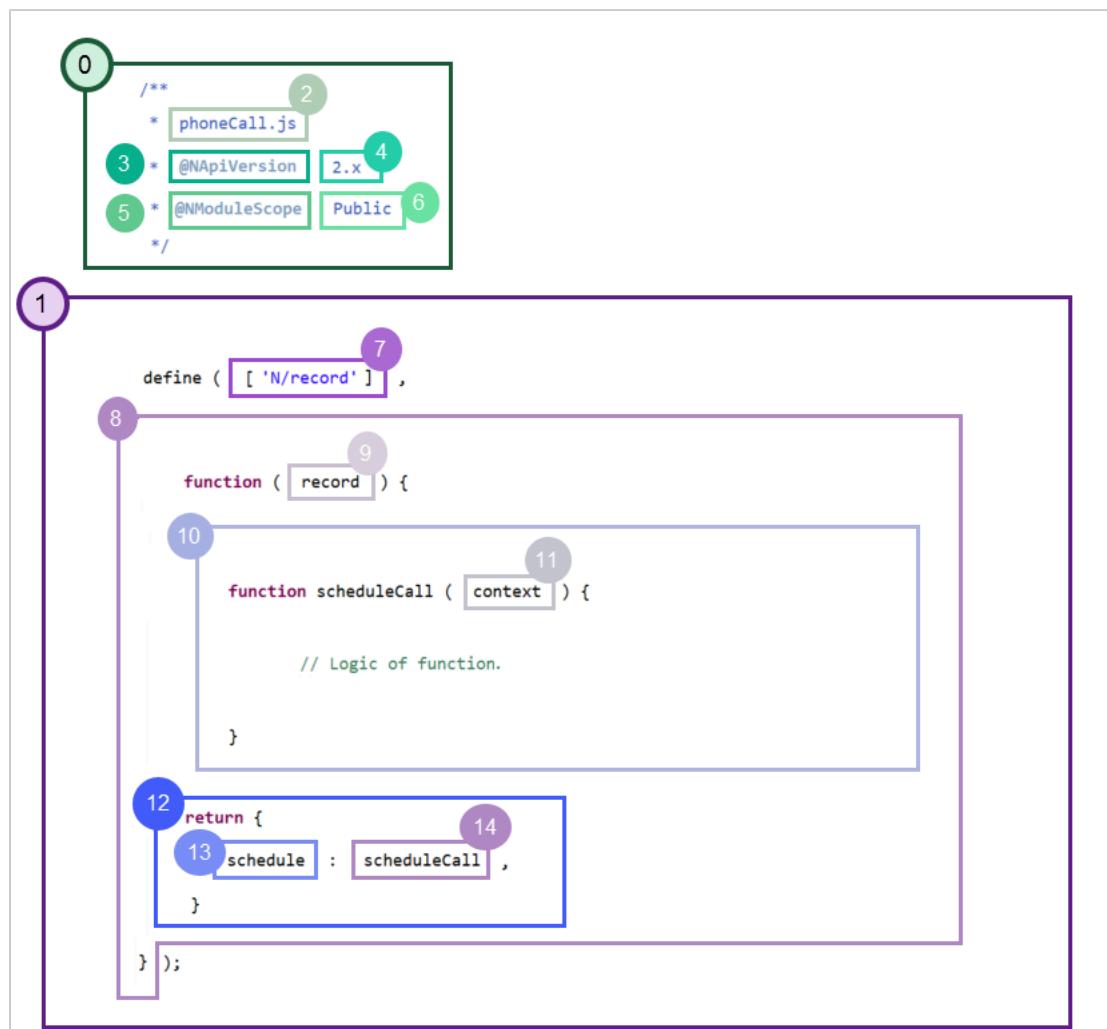
Preparing to Add a Custom Module

Before you work on a custom module, make sure you're familiar with the following topics:

- [SuiteScript 2.x Script Basics](#)
- [SuiteScript 2.x Entry Point Script Validation](#)
- [SuiteScript 2.x Global Objects and Module Dependency Paths](#)
- [Controlling Access to Scripts and Custom Modules](#)

SuiteScript 2.x Anatomy of a Custom Module Script

All SuiteScript 2.x custom module scripts must conform to the same basic structure. The following diagram illustrates that structure. For an explanation of the numbered components of the script, refer to the table that follows the diagram.



| General Area | Callout | Description |
|--------------------------------------|---------|---|
| 0 — JSDoc tags and other annotations | 2 | The title of the file that holds this script. This annotation is not required, but it can be useful. Any script that loads this module must refer to this name. |

| General Area | Callout | Description |
|----------------------|---------|--|
| | 3 and 4 | The @NApiVersion tag and its value. This tag is not required in a custom module script, but you may want to include it to prevent compatibility problems with scripts that use future versions of SuiteScript. |
| | 5 and 6 | The @NModuleScope tag and its value. This tag is optional. You can use it to limit the access that other scripts have to this module. For more information about this tag, see Controlling Access to Scripts and Custom Modules . |
| 1 — define statement | 7 | The define function's first argument, which is a list of modules required by the script. This script uses only one: the N/record Module , which lets the script interact with records. |
| | 8 | The define function's second argument, which is a callback function. |
| | 9 | The arguments of the callback function. Because this script loads only one module, the callback function takes only one argument. This argument represents the N/record Module and can be used anywhere in the callback function to access the module's APIs. You can give this object any name you prefer but, as a best practice, use a name that is similar to the module name. |
| | 10 | The script's entry point function. As you can see in the return statement (Callout 12), this entry point function is associated with the entry point named schedule. |
| | 11 | The context object that is made available when the schedule entry point is invoked. The values of this object's properties are defined in the script that calls this custom module. For an example, see SuiteScript 2.x Custom Module Tutorial . |
| | 12 | The callback function's return statement. |
| | 13 | The custom module's entry point. As with an entry point script, every custom module script must use at least one entry point. The difference is that an entry point script must use a standard entry point that is part of the script type identified by the @NScriptType JSDoc tag. A custom module entry point is your creation, so you can use this return statement to give the entry point any name you prefer. |
| | 14 | The entry point function returned when the schedule entry point is invoked. For each entry point used, your custom module script must identify some object, usually a function, that is returned. |



Note: For details on the standard structure for an entry point script, see [SuiteScript 2.x Anatomy of a Script](#).

SuiteScript 2.x Custom Module Tutorial

A custom module script holds logic that can be used by other scripts. If you have logic that is used in multiple scripts, you can create a custom module script to hold that logic. This approach is more efficient than copying the logic into each script where it is needed.

This topic walks you through the process of creating a custom module script and modifying an entry point script so that it uses the custom module.

This tutorial includes several steps:

- [Sample Script Overview](#)
- Step One: [Deploy the Prerequisite Script](#)
- Step Two: [Create the Custom Module Script File](#)

- Step Three: [Review the Script \(Optional\)](#)
- Step Four: [Upload the Custom Module Script File to NetSuite](#)
- Step Five: [Modify the Entry Point Script](#)
- Step Six: [Upload the Revised User Event Script](#)
- Step Seven: [Test the Script](#)

Sample Script Overview

The custom module script in this tutorial automatically creates a phone call record. After you create this module and upload it to your NetSuite File Cabinet, you can call this module from an entry point script.

When you have finished the updates to the entry point script (as described in this tutorial), the custom module script is triggered each time a new employee record is created. If the new employee record includes values in the Phone and Supervisor fields, the custom module schedules a phone call between the supervisor and the new employee.

Deploy the Prerequisite Script

If you have not deployed the user event script described in [SuiteScript 2.x User Event Script Tutorial](#), deploy it now.

Create the Custom Module Script File

Copy and paste the following code into the text editor of your choice. Save the file and name it phoneCall.js.

```

1 /**
2  * phoneCall.js
3  * @NApiVersion 2.x
4  * @NModuleScope Public
5 */
6
7
8 // This script creates a record, so it loads the N/record module.
9
10 define(['N/record'],
11
12     // The next line marks the beginning of the callback
13     // function. The 'record' argument is an object that
14     // represents the record module.
15
16     function(record) {
17
18         // The next line marks the beginning of the entry point
19         // function.
20
21         function scheduleCall(context) {
22             var newPhoneCall = record.create({
23                 type: record.Type.PHONE_CALL,
24                 isDynamic: true
25             });
26
27             newPhoneCall.setValue({
28                 fieldId: 'title',
29                 value: context.phoneCallTitle

```

```

30     });
31
32     newPhoneCall.setValue({
33         fieldId: 'assigned',
34         value: context.phoneCallOwner
35     });
36
37     newPhoneCall.setText({
38         fieldId: 'phone',
39         text: context.phoneNumber
40     });
41
42     try {
43         var newPhoneCallId = newPhoneCall.save();
44
45         log.debug({
46             title: 'Phone call record created successfully',
47             details: 'New phone call record ID: ' + newPhoneCallId
48         });
49
50     } catch (e) {
51         log.error({
52             title: e.name,
53             details: e.message
54         });
55     }
56 }
57
58 // Add the return statement that identifies the entry point function.
59 return {
60     schedule: scheduleCall,
61 }
62 });

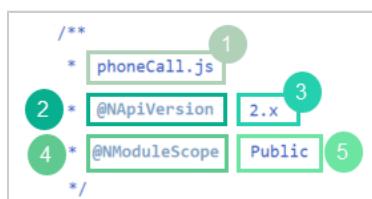
```

Review the Script (Optional)

If you want to understand more about how this script is structured, review the following subsections. Note that these images do not show the entire script. For more details, refer to the comments in [Create the Custom Module Script File](#).

JSDoc Tags

A custom module script is not required to have JSDoc tags, but they are recommended. The following illustration shows this sample's JSDoc block.



| Callout | Description |
|---------|---|
| 1 | The title of the file that holds this script. This annotation is not required, but it can be useful to list here. Any script that loads this module must refer to the name of the file. |
| 2 and 3 | The @NApiVersion tag and its value. This tag is not required in a custom module script, but you may want to include it to prevent compatibility problems with scripts that use future versions of SuiteScript. Valid values for this tag are 2.0, 2.x, and 2.X. |

| Callout | Description |
|---------|--|
| 4 and 5 | The @NModuleScope tag and its value. This tag is optional. You can use it to limit the access of other scripts to this module. For more information about this tag, see Controlling Access to Scripts and Custom Modules . |

Entry Point Function

Every custom module script must define an object that is returned when the script's entry point is invoked. This object could be a static value, such as a string. However, it is far more common for custom module scripts to return a function, as is the case with this script. Because of the way the [Return Statement](#) is set up, the entry point function shown in the following diagram is invoked when the script's schedule entry point is invoked.

As with an entry point function in a standard entry point script, this function takes a context object as its argument. For more details, see the table that follows the diagram.

```
function scheduleCall ( context ) {
    var newPhoneCall = record.create({
        type: record.Type.PHONE_CALL,
        isDynamic: true
    });

    newPhoneCall.setValue({
        fieldId: 'title',
        value: context.phoneCallTitle
    });

    newPhoneCall.setValue({
        fieldId: 'assigned',
        value: context.phoneCallOwner
    });

    newPhoneCall.setText({
        fieldId: 'phone',
        text: context.phoneNumber
    });

    try {
        var newPhoneCallId = newPhoneCall.save();

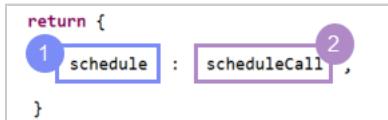
        log.debug({
            title: 'Phone call record created successfully',
            details: 'New Phone call record ID: ' + newPhoneCallId
        });
    } catch (e) {
        log.error({
            title: e.name,
            details: e.message
        });
    }
}
```

| Callout | Description |
|---------|---|
| 1 | The context object that is made available when the schedule entry point is invoked. The values of this object's properties are defined in the script that calls this custom module. |
| 2 | This statement uses the record.create(options) method to begin the process of creating a phone call record. |

| Callout | Description |
|---------|--|
| 3 | <p>These statements set fields on the phone call record. They use the properties of the context object, along with the Record.setValue(options) and Record.setText(options) methods.</p> <p>The values of the context object — phoneCallTitle, phoneCallOwner, and phoneCallNumber — must be defined in the script that calls the custom module.</p> |
| 4 | This try/catch block attempts to save the new phone call record by using the Record.save(options) method. If the save attempt fails, the block catches and logs the error that caused the problem. |

Return Statement

As with an entry point script, the callback function in a custom module script must include a return statement. The return statement must include at least one entry point.



| Callout | Description |
|---------|--|
| 1 | The custom module's entry point. As with an entry point script, every custom module script must use at least one entry point. The difference is that an entry point script must use a standard entry point that is part of the script type used by the script. A custom module entry point is your creation, so you can use this return statement to give the entry point any name you prefer. |
| 2 | <p>A reference to an object. This example references a function. This structure is probably the most common. However, the entry point could reference another object, such as a static value.</p> <p>Whatever object is referenced, it must be defined within the same script file.</p> |

Upload the Custom Module Script File to NetSuite

After you have created your custom module script file, upload it to your NetSuite File Cabinet.

To upload the script file:

1. In the NetSuite UI, go to Documents > File > SuiteScripts.
2. In the left pane, select the SuiteScripts folder and click **Add File**.
3. Follow the prompts to locate the phoneCall.js file in your local environment and upload it to the SuiteScripts folder.

Be aware that even after you upload the file, you can edit it from within the File Cabinet, if needed. For details, see the help topic [Editing Files in the File Cabinet](#).

Modify the Entry Point Script

After you upload a custom module script file, as described in the last section, you do not need to take any other actions for it to be available. You do not have to create a script record or script deployment record for it. However, for the script to be used, the script must be referenced by a script that has been deployed.

In the next procedure, you modify an entry point script so that it uses the phoneCall.js module. You will update the user event script described in [SuiteScript 2.x User Event Script Tutorial](#). To update the file, you can use either of the following approaches:

- If you want to copy and paste the updated script directly from this help topic, skip ahead to [Copy the Full Script](#).
- If you want to read about how to make the needed edits yourself, refer to [Update the Script Step by Step](#).

Update the Script Step by Step

This procedure tells you how to update createTask.js so that it uses the phoneCall.js custom module file.

To update the script step by step:

1. Open the createTask.js file for editing.
 2. Update the list of dependencies by adding /phoneCall.js to the first argument of the define function.
- ```
define (['N/record', 'N/ui/serverWidget', './phoneCall'] ,
```
3. Add an additional dependency to the callback function's first argument. This object represents the phone call module. It can be used to access the phone call module's API.
- ```
function(record, serverWidget, phone ) {
```
4. In the script's afterSubmit function, add logic to retrieve the employee's phone number.

```
function myAfterSubmit(context) {
    if (context.type !== context.UserEventType.CREATE)
        return;

    var newEmployeeRecord = context.newRecord;

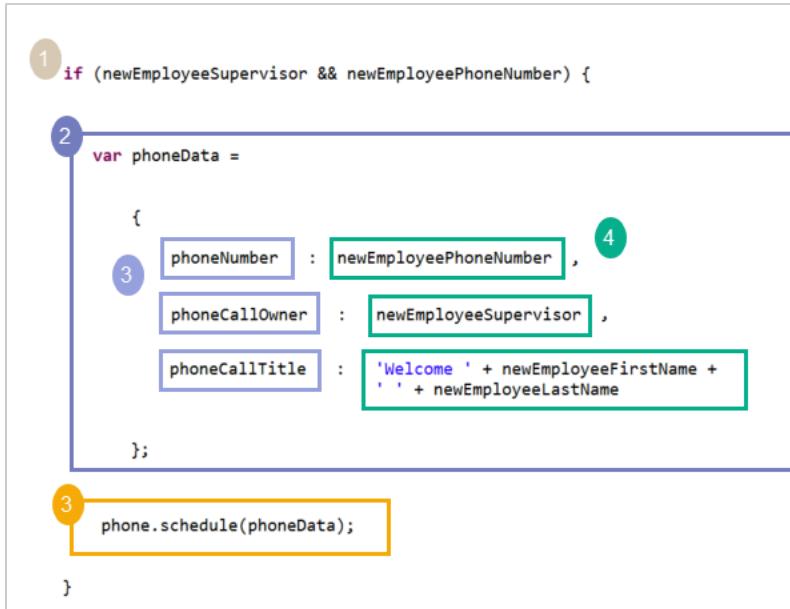
    var newEmployeeFirstName = newEmployeeRecord.getValue ({
        fieldId: 'firstname'
    });

    var newEmployeeLastName = newEmployeeRecord.getValue ({
        fieldId: 'lastname'
    });

    var newEmployeeSupervisor = newEmployeeRecord.getValue ({
        fieldId: 'supervisor'
    });

    var newEmployeePhoneNumber = newEmployeeRecord.getValue({
        fieldId: 'phone'
    })
```

5. Create a block of code that does the following:
 - Create a conditional test that checks to see whether the employee record has values in both the phone number and supervisor fields (Callout 1).
 - Create an object called phoneData that sets values for all fields needed by the phone call module's schedule method (Callout 2).
 - Call the schedule method, passing in the phoneCall object as its argument (Callout 3).



6. Upload the revised script file to your NetSuite File Cabinet, overwriting the prior version.

Copy the Full Script

The following shows the fully updated user event script. If you haven't already created the script file using the steps described in [Create the Script Step by Step](#), copy and paste the following code into the text editor of your choice. Save the file and name it createTask.js.

```

1 /**
2 *
3 * @NApiVersion 2.x
4 * @NScriptType UserEventScript
5 */
6
7 define(['N/record', 'N/ui/serverWidget', './phoneCall'],
8     function(record, serverWidget, phone) {
9
10     // In the beforeLoad function, disable the
11     // Notes field on the record.
12
13     function myBeforeLoad(context) {
14         if (context.type !== context.UserEventType.CREATE)
15             return;
16         var form = context.form;
17
18         var notesField = form.getField({
19             id: 'comments'
20         });
21
22         notesField.updateDisplayType({
23             displayType: serverWidget.FieldDisplayType.DISABLED
24         });
25     }
26
27     // In the beforeSubmit function, add a message to the Notes field.
28
29     function myBeforeSubmit(context) {
30         if (context.type !== context.UserEventType.CREATE)
31             return;
32         var newEmployeeRecord = context.newRecord;
33         newEmployeeRecord.setValue('comments', 'Orientation date TBD.');
34     }
35

```

```

36 // In the afterSubmit function, take several actions culminating
37 // in creating both a task record and a phone call record.
38
39 function myAfterSubmit(context) {
40
41 // If the user is not creating a new record, then stop executing.
42
43 if (context.type !== context.UserEventType.CREATE)
44     return;
45
46 // Use the context object's newRecord property to retrieve values
47 // from the new record.
48
49 var newEmployeeRecord = context.newRecord;
50 var newEmployeeFirstName = newEmployeeRecord.getValue ({
51     fieldId: 'firstname'
52 });
53
54 var newEmployeeLastName = newEmployeeRecord.getValue ({
55     fieldId: 'lastname'
56 });
57
58 var newEmployeeSupervisor = newEmployeeRecord.getValue ({
59     fieldId: 'supervisor'
60 });
61
62 var newEmployeePhoneNumber = newEmployeeRecord.getValue({
63     fieldId: 'phone'
64 });
65
66 // If the user entered a value for the supervisor field,
67 // create a task record for the supervisor.
68
69 if (newEmployeeSupervisor) {
70     var newTask = record.create({
71         type: record.Type.TASK,
72         isDynamic: true
73     });
74
75     newTask.setValue({
76         fieldId: 'title',
77         value: 'Schedule orientation session for ' +
78             newEmployeeFirstName + ' ' + newEmployeeLastName
79     });
80
81     newTask.setValue({
82         fieldId: 'assigned',
83         value: newEmployeeSupervisor
84     });
85
86     try {
87         var newTaskId = newTask.save();
88         log.debug({
89             title: 'Task record created successfully',
90             details: 'New task record ID: ' + newTaskId
91         });
92
93     } catch (e) {
94         log.error({
95             title: e.name,
96             details: e.message
97         });
98     }
99 }
100
101 // If the user entered values for both the supervisor and
102 // phone number fields, use the phoneCall module to schedule a phone call.
103
104 if (newEmployeeSupervisor && newEmployeePhoneNumber) {
105     var phoneData =
106     {
107         phoneNumber: newEmployeePhoneNumber,
108         phoneCallOwner: newEmployeeSupervisor,

```

```

109         phoneCallTitle: 'Welcome ' + newEmployeeFirstName +
110             ' ' + newEmployeeLastName
111     );
112   }
113 }
114
115 return {
116   beforeLoad: myBeforeLoad,
117   beforeSubmit: myBeforeSubmit,
118   afterSubmit: myAfterSubmit
119 };
120
121 });
122 });

```

Upload the Revised User Event Script

After you have updated createTask.js, upload it to your NetSuite File Cabinet.

To upload the script file:

1. In the NetSuite UI, go to Documents > File > SuiteScripts.
2. In the left pane, select the SuiteScripts folder and click **Add File**.
3. Follow the prompts to locate the createTask.js file in your local environment and upload it to the SuiteScripts folder.
4. When the system notifies you that a file with that name already exists, click OK to overwrite the existing file.

Test the Script

Now that the both files have been uploaded, you should verify that the custom module executes as expected.

To test the script:

1. Begin the process of creating a new employee record by selecting Lists > Employees > Employees > New.
2. In the new employee form, check to see whether the Notes field is disabled. If the beforeSubmit entry point function works as expected, the field is gray and cannot be edited.
3. Enter value for required fields. These fields may vary depending on the features enabled in your account and any customizations that exist. Minimally, they include:
 - **Name** — Enter a first name of Susan and last name Johnson.
 - **Subsidiary** — (OneWorld only) Choose a value from the dropdown list.
4. To make sure that the custom module logic is used, enter values in the **Supervisor** and **Phone** fields.
5. Click **Save**. A success message appears, and the system displays the new record in View mode.
6. Verify that the phone call record was created successfully:
 - a. Select Activities > Scheduling > Phone Calls.
 - b. Check the filtering options to make sure that phone calls assigned to all employees are being displayed.
 - c. Verify that a phone call was scheduled to welcome the new employee.

Module Dependency Paths

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

To specify module paths and module names, you can use the [define Object](#) or the [require Function](#).

Avoid using any file extensions in the path.

SuiteScript supports the following:

| Path Type | Description |
|------------------------|--|
| Absolute Paths | <p>Specifies the path from the root folder in the File Cabinet.</p> <p>Absolute paths start with a forward slash (/). For example, "/SuiteScripts/MyApp/Util".</p> |
| Relative Paths | <p>Specifies the path relative to the dependent module's location in the File Cabinet. This provides greater flexibility when moving nested folders containing modules in the File Cabinet.</p> <p>Relative Paths can start with a period (.) or two periods (..).</p> <p>For example, assume that the dependent module is located in the following folder: "/SuiteScripts/MyApp". The relative path for a sibling file under "SuiteScripts/MyApp" could be "./Util". The equivalent absolute path in this case would be "/SuiteScripts/MyApp/Util".</p> <p>Note: Do not use relative paths in global contexts.</p> |
| Bundle Virtual Paths | <p>Specifies a bundle path that the NetSuite system can resolve to a File Cabinet pointer. A valid bundle ID is required in the bundle virtual path.</p> <p>Bundle virtual paths start with a forward slash followed by a period (/). For example:</p> <ul style="list-style-type: none"> ■ ./bundle/<bundle id>/SS2_CustomModuleTest.js/ |
| Naming a Custom Module | Defines a global identifier used to reference a module by name and not path. Use a require Configuration to set the name. |

Absolute Paths

Use an initial forward slash (/) to denote the top-level File Cabinet directory.

In the following example, the Module Loader expects the custom module files lib1.js and lib2.js to be located in the SuiteScripts top-level directory in the File Cabinet.

```

1 // myModule.js
2 define(['/SuiteScripts/lib1', '/SuiteScripts/lib2'], // myModule has a dependency on modules lib1 and lib2
3     ...
4 );

```

Relative Paths

You can specify relative paths to modules in subdirectories from the directory of the current module.

In the following example, the Module Loader expects the custom module files lib1.js and lib2.js to be located in the lib subdirectory, the same File Cabinet directory that contains myModule.js.

```

1 // myModule.js
2 define(['./lib/lib1', './lib/lib2'], // myModule has a dependency on modules lib1 and lib2
3     ...
4 );

```

Bundle Virtual Paths

Use a bundle virtual path to point to a bundle File Cabinet location. Bundle virtual paths start with a forward slash followed by a period (/.). A valid bundle ID is required in the bundle virtual path.

In the following example, the Module Loader is looking for common libraries in shared bundles.

```

1 // myModule.js
2 require(['/.bundle/101'], // myModule has a dependency on modules with the bundle id 100 and 101
3     ...
4 );

```

In certain situations, a virtual path can help when a bundle is moved or the bundle id changes due to deprecation or copying. At the time of copying or deprecation, SuiteScript 2.x checks the deprecation or copy chain and looks for the file in multiple places. For example, a created bundle is deprecated by a newer version of the bundle, and the newer version is installed in the target account with a different ID. In this case, the scripts using the virtual path of the deprecated bundle in the target account use the new bundle's ID, because the virtual bundle path automatically resolves to the new bundle.

Be aware that the ID for the created bundle in the source account could not be used to specify a virtual bundle path. The virtual bundle path depends on a file path to an existing bundle. At the time the bundle is created in the source account, no such valid file path is created yet.

Naming a Custom Module



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Loading a custom module by name

You can set up a custom module name to aid re-use. After you configure a module name, you can require it without knowing the path.

Custom module loading by name also enables better interoperability with third-party libraries and may offer some protection against naming conflicts.

You will need to call [define\(id, \[dependencies,\] moduleObject\)](#) and configure a [require Function](#).

To load a custom module by name:

1. Create your module file. For example, you create a JavaScript file containing the following code and name it math.js. You save it under SuiteScripts/Example in the File Cabinet.

```

1 ...
2 let myMath = {
3     add: function(num1, num2) {
4         return num1 + num2
5     },
6     subtract: function(num1, num2) {
7         return num1 - num2
8     },
9     multiply: function(num1, num2) {
10        return num1 * num2
11    },
12    divide: function(num1, num2) {
13        return num1 / num2
14    },
15 }
16 ...

```

- To load the module by name, specify the module name (alias) and its path by configuring the paths parameter in a JSON file (myconfig.json). See the help topic [require Configuration](#) for more information.

```

1 ...
2 {
3     "paths": {
4         "math": "/SuiteScripts/Example/math.js"
5     }
6     "shim": {
7         "math": {
8             "exports": "myMath"
9         }
10    }
11 }
12 ...

```

- Load the custom module in your SuiteScript 2.x script by including the NAmdConfig tag and passing the name to the [define Object](#).

```

1 /**
2 * @NApiVersion 2.x
3 * @NScriptType usereventscript
4 * @NAmdConfig ./myconfig.json
5 */
6
7 define(['math'], function (math) {
8     return {
9         beforeLoad: function beforeLoad() {
10             log.debug({
11                 title: 'test',
12                 details: myMath.add(1,2)
13             });
14         }
15     });
16 });

```

Custom Module Examples



Note: This topic applies to SuiteScript 2.0.

The following examples demonstrate using the define and require functions when working with custom modules.

- Define a custom utility module

- Import the custom utility module
- Define a custom module for a SuiteApp
- Import a third-party JavaScript library

Define a custom utility module

The following file holds the definition of a custom module with a custom API. For example, to allow multiple scripts that depend on an incremental counter utility to require and import this functionality.

To protect against version incompatibility, this script includes the `@NApiVersion` tag.

```

1 /**
2  * counter.js
3  * @NApiVersion 2.x
4 */
5 define(function(){
6     let counter = 0;
7
8     function incrementValue() {
9         counter++;
10    }
11    function getValue() {
12        return counter;
13    }
14
15    return {
16        increment: incrementValue,
17        value: getValue
18    }
19 });

```

Import the custom utility module

This example uses the `define Object` to include a native SuiteScript module and a custom module.

The module's file path is used to pass in the custom utility as a dependency. As a best practice, it does not include the `.JS` file extension.

```

1 /**
2  * customRecord.js
3  * @NApiVersion 2.x
4 */
5 define(['N/record','./counter'],
6     function(record,counter){
7         function createCustomRec() {
8             record.create(..);
9             counter.increment();
10        }
11
12        return {
13            createCustomRecord: createCustomRec
14        }
15 });

```

Define a custom module for a SuiteApp

To define a bundled custom module that can be exposed to third parties, you must add the `@NModuleScope` JSDoc tag and assign it the value `public`.

```

1 /**
2  * @NApiVersion 2.x
3  * @NModuleScope public

```

```
4 | */
```

Next, use the `define Object` so that your custom module is recognized as an AMD module. To use a module that is bundled within an external SuiteApp, you need to pass the bundle's file path, containing a valid bundle ID, within the `define()` function as follows:

```
1 | define(['./.bundle/<bundle ID>/<module path>'],
2 |   function(<module name>){
3 |     <logic goes here>
4 |   }
5 | );
```

Import a third-party JavaScript library

- Import a third-party library
- Add a non-AMD library

Import a third-party library

Some third-party libraries register as AMD compatible.

In that case, you can specify a require configuration that sets up the path where the module is found. For example, you can create a JSON configuration file (`JsLibraryConfig.json`) containing the following code and save it in the File Cabinet:

```
1 | ...
2 | {
3 |   "paths": {
4 |     "coolthing": "/SuiteScripts/myFavoriteJsLibrary"
5 |   }
6 | }
```

Then, you could use the `NAmdConfig JSDoc` tag to provide a relative path from the script that needs the `coolthing` module:

```
1 | /**
2 | * @NApiVersion 2.x
3 | * @NScriptType userevents
4 | * @NAmdConfig ./jsLibraryConfig.json
5 | */
6 |
7 | define(['coolthing'],
8 |   function (coolthing) {
9 |     return {
10 |       beforeLoad: function beforeLoad(ctx) {
11 |         coolthing.times(2, function () {
12 |           log.debug({
13 |             title: 'log',
14 |             details: 'log'
15 |           });
16 |         });
17 |       }
18 |     };
19 |   });
});
```

Similarly, to use a version of jQuery, you can form your `require` configuration as follows:

Add a JSON configuration file (`jQueryConfig.json`) to the File Cabinet and configure the path:

```
1 | ...
```

```

2  {
3    "paths": {
4      "jquery": "/SuiteScripts/myjQueryLib"
5    }
6  }
7 ...

```

From the script using the jQuery module, reference the path to your JSON configuration file in the `@NAmdConfig` JSDoc tag. For example, `* @NAmdConfig /SuiteScripts/myjQueryLib.json`.

```

1 ...
2 * @NAmdConfig ./jQueryConfig.json
3 */
4
5 define(['jquery'], function(jquery) {
6 });
7 ...

```

Note: If you want to use jQuery with a Suitelet, import it via an on demand client script that is attached to the Suitelet using [Form.clientScriptFileId](#) or [Form.clientScriptModulePath](#).

Important: SuiteScript does not support direct access to the NetSuite UI through the Document Object Model (DOM). The NetSuite UI should only be accessed using SuiteScript APIs.

Add a non-AMD library

In this example, the following file would be uploaded to the File Cabinet as a JavaScript file titled `math.js` in the `/SuiteScripts/ShimExample` folder. It does not register as AMD compatible.

```

1 var myMath = {
2   add: function(num1, num2) {
3     return num1 + num2;
4   },
5   subtract: function(num1, num2) {
6     return num1 - num2;
7   },
8   multiply: function(num1, num2) {
9     return num1 * num2;
10  },
11  divide: function(num1, num2) {
12    return num1 / num2;
13  },
14}

```

When the library does not register as an AMD module, you need to use a require configuration that specifies the configuration parameters in JSON format. In this case, the `shim` parameter and `paths` parameter are configured in the `MathConfig` JSON file and also stored in the `ShimExample` folder in the File Cabinet:

```

1 ...
2 {
3   "paths": {
4     "math": "SuiteScripts/ShimExample/math.js"
5   },
6   "shim":{
7     "math": {
8       "exports": "myMath"
9     }
10  }
11}
12 ...

```

The module can be loaded as a dependency in an entry point script, such as in a user event script, provided that a valid @NAmConfig JSDoc tag with a path to the JSON configuration file is included.

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType userevents
4   * @NAmConfig /SuiteScript/ShimExample/MathConfig.json
5   */
6
7  define(['math'], function (math) {
8      return {
9          beforeLoad: function beforeLoad() {
10              log.debug({
11                  title: 'test',
12                  details: myMath.add(1,2)
13              });
14          }
15      });
16 });

```

Troubleshooting Errors



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Module not found

You may see this error if you are using the require function. The require function has no global context. Consequently, relative paths do not work for the require function unless you import require() as a dependency of define().

If you receive the Module does not exist error, try replacing any relative paths with an absolute path.

For more information, see the help topic [require Function](#).

You do not have permission to load this module.

Review your module scope settings. For a full description of support module scopes, see [Controlling Access to Scripts and Custom Modules](#).

Custom Modules Frequently Asked Questions



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

When should I use require versus define?

Always use the [define Object](#) in entry point scripts and when creating new modules. Use the [require Function](#) for loading and using existing modules.

There is a performance advantage to using the require function. Generally, give preference to using the require Function whenever you can. The define Function will imports all dependencies. The require function loads dependencies only as they are needed.

Here is a summary of when to use or not use these functions:

| Function | Example | Caveat |
|----------|---|--|
| define | To define entry point scripts To create new modules. To export modules To import modules via relative path | If you use a define function, you do not receive the performance benefit of progressive loading. |
| require | To import modules via absolute path (relative path requires context) To step through code in the SuiteScript debugger To support building a test framework For progressive loading of dependencies | If you use a require function, you may need to import a define that provides correct context. For example, to import a custom module with a relative path: <pre>1 define(['require'], function(require){ 2 require('./ParentFolder/customFile'), function(customFile){ 3 //your code 4 }); 5});</pre> |

Are third-party libraries supported?

Yes. You can define the library as a custom module, set up a global identifier to reference it, and configure it as a dependency. See [Import a third-party library](#) and [Import a third-party JavaScript library](#).

SuiteScript 2.x Scripting Records and Subrecords

For information on using SuiteScript 2.x to work with records and subrecords, see the following topics:

- [SuiteScript 2.x Scripting Records](#)
- [SuiteScript 2.x Scripting Subrecords](#)

SuiteScript 2.x Scripting Records

For details on using SuiteScript 2.x to work with records, see the following sections:

- [SuiteScript 2.x Standard and Dynamic Modes](#)
- [SuiteScript 2.x Record Modules](#)

SuiteScript 2.x Standard and Dynamic Modes

When you create, copy, load, or transform records in SuiteScript, you can work with records in standard or dynamic mode.

Standard mode:

When a SuiteScript 2.x script creates, copies, loads, or transforms a record in standard mode, the record's body fields and sublist line items are not sourced, calculated, and validated until the record is saved with [Record.save\(options\)](#). Standard mode is also called deferred dynamic mode and you see both terms in the SuiteScript help.

When you work with a record in standard mode, in most cases, you don't need to set values in any particular order. After a record is submitted, NetSuite processes the record's body fields and sublist line items in the correct order, regardless of the organization of your script. See [Getting Text in SuiteScript 2.x Record Modes](#).

Dynamic mode:

When a SuiteScript 2.x script creates, copies, loads, or transforms a record in dynamic mode, the record's body fields and sublist line items are sourced, calculated, and validated in real time. A record in dynamic mode emulates the behavior of a record in the UI.

When you work with a record in dynamic mode, the order in which you set field values matters. For some developers, this aspect might feel constraining. It is likely that scripting in dynamic mode will require you to refer back to the UI often. For example, on an invoice in the UI, you would not set the Terms field before setting the Customer field. The reason is that as soon as you set the Customer field, the value of Terms will be overridden. On an invoice, the value of Terms is sourced from the terms specified on the Customer record. The same behavior happens in dynamic scripting. In your scripts, if you do not set field values in the order that they are sourced in the UI, some of the values you set could be overridden.

How you can tell if a record is in dynamic mode:

You can determine if a record is in dynamic mode using these two SuiteScript 2.x properties:

- [Record.isDynamic](#) (for server scripts)
- [CurrentRecord.isDynamic](#) (for client scripts)

Record Modes and User Event Scripts

Standard mode is always used for user event scripts that instantiate records with the newRecord or oldRecord object provided by the script context. For that reason, the SSS_INVALID_API_USAGE error appears when a user event executes on one of these objects in the following situations:

- When the user event script executes on a record that is being created, and the script attempts to use [Record.getText\(options\)](#) without first using [Record.setText\(options\)](#) for the same field.
- When the user event script executes on an existing record or on a record being created through copying, and the script uses [Record.setValue\(options\)](#) on a field before using [Record.getText\(options\)](#) for the same field.

Getting Text in SuiteScript 2.x Record Modes

In dynamic mode, you can use [Record.getText\(options\)](#) without limitation but, in standard mode, limitations exist. In standard mode, you can use this method **only** in the following cases:

- You can use [Record.getText\(options\)](#) on any field where the script has already used [Record.setText\(options\)](#).
- If you are loading or copying a record, you can use [Record.getText\(options\)](#) on any field except those where the script has already changed the value with [Record.setValue\(options\)](#).

Record Module Method Considerations

Be aware that the [record.create\(options\)](#), [record.copy\(options\)](#), [record.load\(options\)](#), and [record.transform\(options\)](#) methods work in standard mode by default. If you want these methods to work in dynamic mode, you must set the `.isDynamic` property for each method.

SuiteScript 2.x Record Modules

With SuiteScript 2.x, you use the [N/record Module](#) and [N/currentRecord Module](#) to script with records.

In **server scripts**, use the [N/record Module](#). See the following topics for examples of working with records in server scripts:

- [SuiteScript 2.x User Event Script Sample](#)
- [SuiteScript 2.x User Event Script Tutorial](#)
- [SuiteScript 2.x Custom Module Tutorial](#)

In **client scripts**:

- Use [N/currentRecord Module](#) methods to interact with the record that is active in the current client context.
- Use the [N/record Module](#) to load and interact with remote records.

See the following topics for examples of working with records in client scripts:

- [SuiteScript Client Script Sample](#)
- [SuiteScript 2.x Client Scripts on currentRecord](#)
- [SuiteScript 2.x Remote Objects in Client Scripts](#)

- Disable Field on Client PageInit using SuiteScript 2.0

SuiteScript 2.x Scripting Subrecords

For details on using SuiteScript 2.x to work with subrecords, see the following topics:

- [About Subrecords](#)
- [Subrecord Scripting in SuiteScript 2.x Compared With 1.0](#)
- [Scripting Subrecords that Occur on Sublist Lines](#)
- [Scripting Subrecords that Occur in Body Fields](#)

About Subrecords



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1 and SuiteScript 2.1.

When you use SuiteScript to interact with records, you typically interact with many types of fields. During this process, you may come across fields with a data type of summary. These fields can be populated in only one way: by saving a subrecord to the field. Therefore, if you want to interact with these fields, you must understand how to script with subrecords.

In general, saving subrecords to summary fields is more complex than setting values for other types of fields. However, subrecords are similar to records. So if you know how to work with records, you already know a great deal about working with subrecords. This topic summarizes the similarities and differences.



Note: Subrecords are not compatible with the Advanced Employee Permissions feature. For more information, see the help topic [Before Enabling the Advanced Employee Permissions Feature](#).

Subrecords

Subrecords represent a way of storing data in NetSuite.

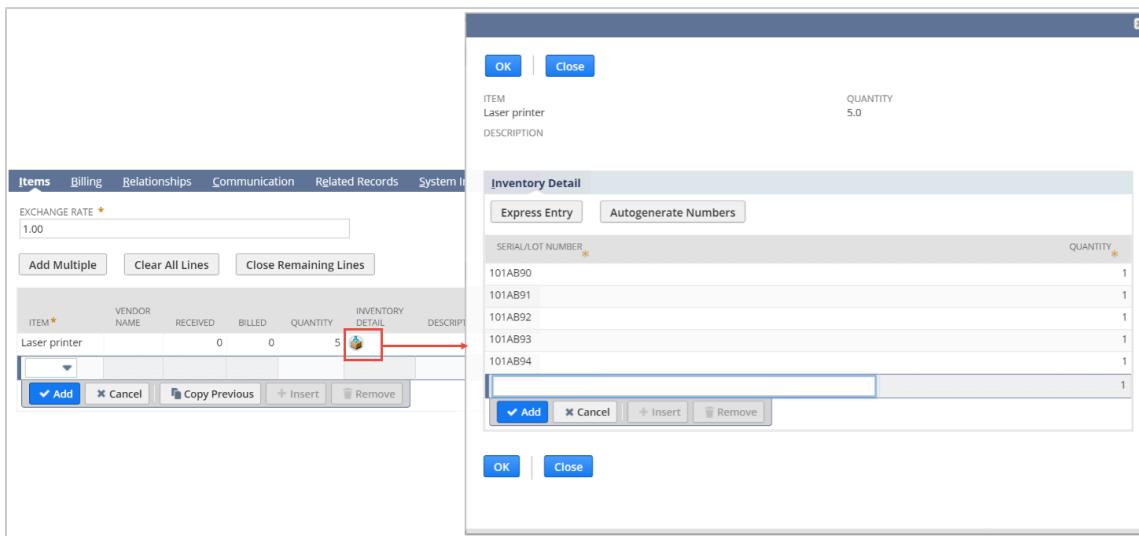
Like records, subrecords are classified by type. Some common types of subrecord include address, inventory detail, and order schedule.

Each subrecord type has a different purpose and includes different fields. For example:

- An address subrecord stores an address. It has fields such as city, state, and zip.
- An order schedule subrecord represents a purchase schedule. It has fields such as startdate and enddate.
- An inventory detail subrecord holds data, such as serial numbers, that describe inventory items. It contains a sublist that holds this data.

At the same time, subrecords differ from records in some ways. For example, while records can exist independently, a subrecord exists solely to hold information about a specific record. You cannot interact with a subrecord outside the context of a parent record.

In the UI, typically you open a subrecord by clicking an icon on the parent record. The subrecord form opens in a separate window. For example, the following illustration shows, at left, a purchase order with one line in its Items sublist. If you click the icon associated with that item, the system opens a window that represents the inventory detail subrecord.



To access this same subrecord from a script, first you would load the purchase order. Then you would use the [Record.getSublistSubrecord\(options\)](#) method to open the subrecord. For example:

```

1 ...
2 // Load the purchase order.
3 var rec = record.load({
4   type: record.Type.PURCHASE_ORDER,
5   id: 7,
6   isDynamic: false
7 });
8
9 // Retrieve the subrecord. For sublistId, use the ID of the relevant sublist on
10 // the purchase order record type. For fieldId, use the ID of the summary field
11 // on the sublist that holds the subrecord.
12 var subrec = rec.getSublistSubrecord({
13   sublistId: 'item',
14   line: 0,
15   fieldId: 'inventorydetail'
16 });
17
18 ...

```

For a full script example, see [Creating an Inventory Detail Subrecord Example](#).

Subrecord Scripting Overview

Use the following guidelines when scripting with subrecords:

- [When Subrecords Are Read-Only](#)
- [Look Up Details About the Summary Field](#)
- [Look Up the Subrecord's Field and Sublist IDs](#)
- [Use Record Methods to Get and Set Values](#)
- [Do Not Explicitly Save a Subrecord](#)

When Subrecords Are Read-Only

You can work with subrecords in both client and server scripts. However, subrecords are read-only when their parent records are retrieved in either of the following ways:

- Through the context object provided to a client script, or through [currentRecord.get\(\)](#).

- Through the context object provided to a beforeLoad user event script.

For more details, see [Supported Deployments for Subrecord Scripting](#).

Look Up Details About the Summary Field

Before you can script with a subrecord, you must have some knowledge of the summary field that holds the subrecord. In addition to the summary field's ID, you must know whether the summary field is situated on the body of the parent record or on one of its sublists. You need both pieces of information to instantiate the subrecord. If the summary field is a sublist field, you also need the relevant sublist ID.

If you are not sure where the field is situated, you can review the record in the UI. You can also check the reference page for the record in the [SuiteScript Records Browser](#). On the reference page for each record type, the **Fields** table lists all of the record type's body fields. The tables listed under the heading **Sublists** show sublist fields. For more details, see [Finding Details About Parent Record Types](#).

Look Up the Subrecord's Field and Sublist IDs

Like records, all subrecords have required and optional fields. To set values for these fields, you must have the field IDs. To work with a subrecord's sublist fields, you must also have the sublist ID. You can find both types of information in the [SuiteScript Records Browser](#). Each subrecord type is listed in the browser alongside the available record types. For each subrecord type, a reference page includes the IDs for all of the elements on the subrecord, including a sublist, if one exists, and all of the subrecord's fields. For more details, see [Finding Details About Subrecord Types](#).

Additionally, if you have the Show Internal IDs preference enabled, you can use the UI to find the IDs for subrecord body fields. To view a field's ID, click its label. In response, the system displays a popup window that shows the ID. For help enabling this preference, see the help topic [Setting the Internal ID Preference](#).

Use Record Methods to Get and Set Values

When you script with a subrecord, many aspects of the scripting process are identical to the process of scripting with records.

For example, when your script instantiates a subrecord, the system returns one of the same objects that it uses to represent records. These objects include `record.Record` and `currentRecord.CurrentRecord`.

Additionally, you use many of the same methods to interact with subrecords as you do with records. For example:

- In a server script, you use `Record.setValue(options)` to set a value on either a record or subrecord body field.
- In a client script, you can use `CurrentRecord.getValue(options)` to retrieve the value stored in either a record or subrecord body field.

A small number of record methods and properties are unavailable to subrecords. These exceptions are noted in the property descriptions in the [N/record Module](#) and [N/currentRecord Module](#) topics.

Do Not Explicitly Save a Subrecord

After you have created or updated a subrecord, you do not explicitly save it. Rather, after you have set all required fields on the subrecord, you simply save the parent record. When you save the record, the subrecord is also saved.

Creating an Inventory Detail Subrecord Example

The following example shows a typical approach to creating a subrecord.

This example creates a purchase order with one inventory item in its sublist. The example also creates an inventory detail subrecord to store a lot number for the item.

Before using this example in your NetSuite account, do the following:

- Make sure the [Advanced Bin / Numbered Inventory Management](#) feature is enabled, at Setup > Company > Enable Features, on the Items & Inventory subtab.
- Verify that you have at least one vendor, one location, and one lot-numbered inventory item defined in your system. Make a note of each record's internal ID.
- Where noted in the script example comments, replace the hardcoded IDs with valid values from your NetSuite account.

This example uses standard mode, but you could also add the subrecord using dynamic mode. For more details about both approaches, see [Using SuiteScript 2.x to Create a Subrecord in a Sublist Field](#).

 **Note:** For help deploying a user event script, see [SuiteScript 2.x Entry Point Script Creation and Deployment](#).

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType UserEventScript
4   */
5
6 define(['N/record'], function(record){
7     function myAfterSubmit(context){
8         // Create the purchase order.
9         var rec = record.create({
10             type: record.Type.PURCHASE_ORDER,
11             isDynamic: false
12         });
13
14         // Set body fields on the purchase order. Replace both of these
15         // hardcoded values with valid values from your NetSuite account.
16         rec.setValue({
17             fieldId: 'entity',
18             value: '2'
19         });
20
21         rec.setValue({
22             fieldId: 'location',
23             value: '2'
24         });
25
26         // Insert a line in the item sublist.
27         rec.insertLine({
28             sublistId: 'item',
29             line: 0
30         });
31
32         // Set the required fields on the line. Replace the hardcoded value
33         // for the item field with a valid value from your NetSuite account.
34         rec.setSublistValue({
35             sublistId: 'item',
36             fieldId: 'item',
37             line: 0,
38             value: '7'
39         });
40
41         rec.setSublistValue({
42             sublistId: 'item',
43             fieldId: 'quantity',
44             line: 0,
45             value: 1
46         });
47
48         // Instantiate the subrecord. To use this method, you must
49         // provide the ID of the sublist, the number of the line you want

```

```

50 // to interact with, and the ID of the summary field.
51
52 var subrec = rec.getSublistSubrecord({
53   sublistId: 'item',
54   line: 0,
55   fieldId: 'inventorydetail'
56 });
57
58 // Insert a line in the subrecord's inventory assignment sublist.
59 subrec.insertLine({
60   sublistId: 'inventoryassignment',
61   line: 0
62 });
63
64 subrec.setSublistValue({
65   sublistId: 'inventoryassignment',
66   fieldId: 'quantity',
67   line: 0,
68   value: 1
69 });
70
71 // Set the lot number for the item. Although this value is
72 // hardcoded, you do not have to change it, because it doesn't
73 // reference a record in your account. For the purpose of this
74 // example, the value can be any string.
75 subrec.setSublistValue({
76   sublistId: 'inventoryassignment',
77   fieldId: 'receiptinventorynumber',
78   line: 0,
79   value: '01234'
80 });
81
82 // Save the record. Note that the subrecord object does
83 // not have to be explicitly saved.
84
85 try {
86   var recId = rec.save();
87   log.debug({
88     title: 'Record created successfully',
89     details: 'Id: ' + recId
90   });
91
92 } catch (e) {
93   log.error({
94     title: e.name,
95     details: e.message
96   });
97 }
98 }
99
100 return {
101   afterSubmit: myAfterSubmit
102 };
103 });

```

For additional script samples, see [Scripting Subrecords that Occur on Sublist Lines](#) and [Scripting Subrecords that Occur in Body Fields](#).

Supported Deployments for Subrecord Scripting



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1 and SuiteScript 2.1.

A script can interact with subrecord instances only if it uses a supported deployment.

To understand supported deployments, it's important to understand that every subrecord has a parent record. For more details on this relationship, see [About Subrecords](#).

The following types of deployments are supported:

- [Server Scripts Deployed on Parent Records](#)
- [Client Scripts Deployed on Parent Records \(with Limitations\)](#)
- [Client Scripts Deployed on Custom Address Forms](#)

Server Scripts Deployed on Parent Records

If you want to create a server script that interacts with a subrecord, you can deploy the script on the parent record type. Alternatively, you can deploy the script on a different record type — but then use the script to load the parent record. After loading the parent record, you can interact with the subrecord in the context of its parent.

Subrecord methods are not supported in beforeLoad user event scripts, except if the script creates or loads another record that is a parent, and interacts with the subrecord in the context of that parent.

You cannot deploy a server script directly to a subrecord type.

Client Scripts Deployed on Parent Records (with Limitations)

A client script may not create subrecords on the current record and is limited to read-only access of existing subrecords on the current record. The client script may remove the subrecord from the current record.

You cannot deploy a client script directly on a subrecord type. However, you can customize an address form, as described in the following section.

You cannot deploy a client script directly to a subrecord type.

Client Scripts Deployed on Custom Address Forms

If appropriate, you can create custom forms for the address subrecord. This process is described in [Customizing Address Forms](#). When working with a custom address form, you can attach a client script to the form with the Custom Code subtab. In these types of scripts, you can interact with the subrecord using the same methods as you would with a record. This process is not covered in this chapter.

Body Field Subrecords and Sublist Subrecords

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1 and SuiteScript 2.1.

When a subrecord occurs on a record, it is always represented by a single field on the record. In the [SuiteScript Records Browser](#), this field is always listed as a field of type summary.

A field that contains a subrecord can exist either as a body or sublist field. For example, the subsidiary record has a body field called mainaddress, which stores an address subrecord. By contrast, the employee record permits the creation of multiple addresses, and each address is described in a sublist line. The sublist includes a field that contains the address subrecord instance.

This difference in where the summary field is placed affects the way you instantiate the subrecord.

For an example of each type of placement, see the following sections:

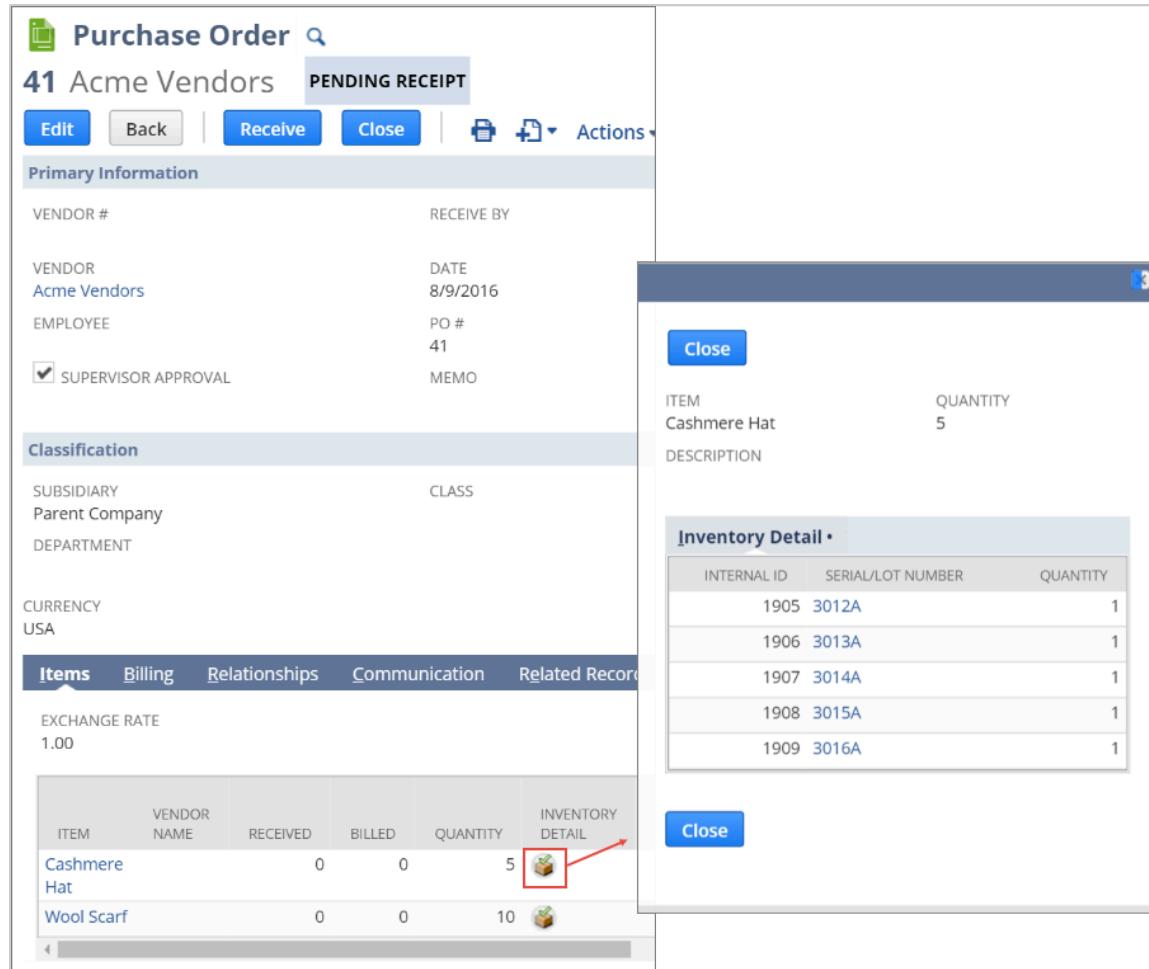
- [Subrecord that Occurs in a Sublist Field Example](#)
- [Subrecord that Occurs in a Body Field Example](#)

Subrecord that Occurs in a Sublist Field Example

Many subrecord types can occur in a sublist field.

For example, depending on the features enabled in your account, the item sublist of a purchase order record can include an Inventory Detail column. If the item on the line is a serialized or lot-numbered item, you can create a subrecord instance in this column.

In the UI, you can view and set values in this subrecord by clicking the icon in the Inventory Detail column. Clicking this icon opens a new window that represents the subrecord.



Be aware that other fields on the sublist line are not part of the subrecord. For example, in the preceding screenshot, the values in the Item, Vendor Name, Received, Billed, and Quantity columns are not part of the subrecord.

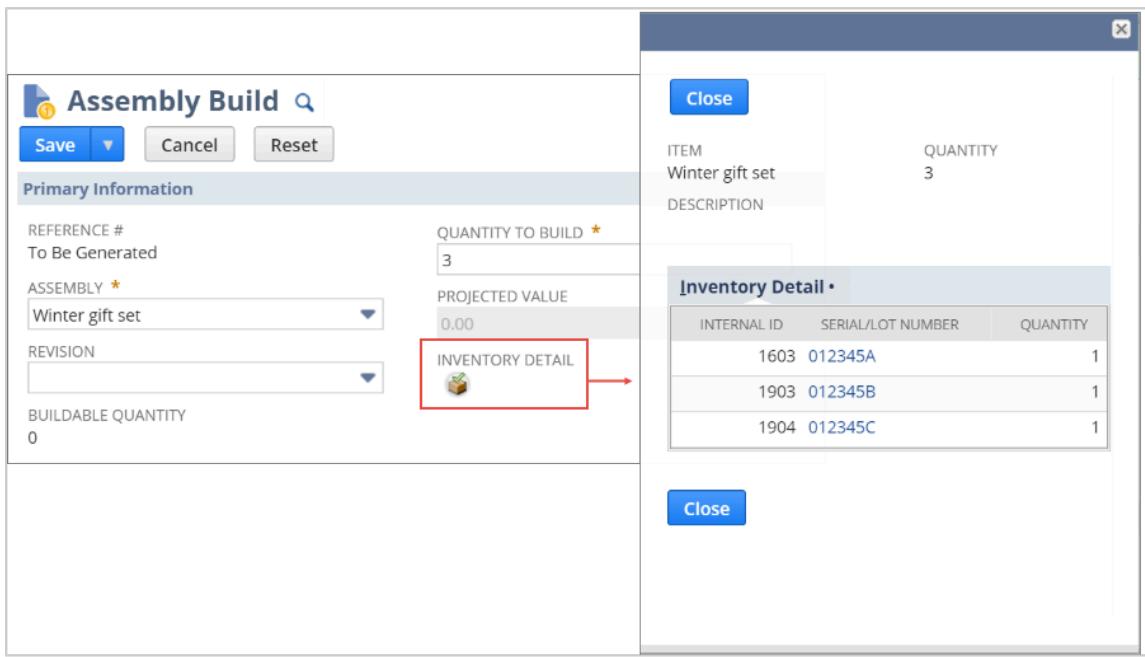
For details on using SuiteScript 2.x to work with subrecords that exist on sublist lines, see [Scripting Subrecords that Occur on Sublist Lines](#).

Subrecord that Occurs in a Body Field Example

A subrecord can also occur in a body field. In fact, the same type of subrecord that appears as a sublist field on one record type can appear as a body field on another record type.

For example, depending on the configuration of your account, the assembly build record can include an Inventory Detail body field. If the item in the record's Assembly field is a serialized or lot-numbered inventory item, you can create an Inventory Detail subrecord in the Inventory Detail field.

In the UI, you can view and set values in this subrecord by clicking the icon under the Inventory Detail label and opening a new window.



For details on using SuiteScript 2.x to work with subrecords that exist in body fields, see [Scripting Subrecords that Occur in Body Fields](#).

Structure of a Subrecord



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1 and SuiteScript 2.1.

The fields available in a subrecord vary depending on the subrecord's type. A subrecord can have body fields, a sublist, or both.

When working with a subrecord's fields, be aware of the following:

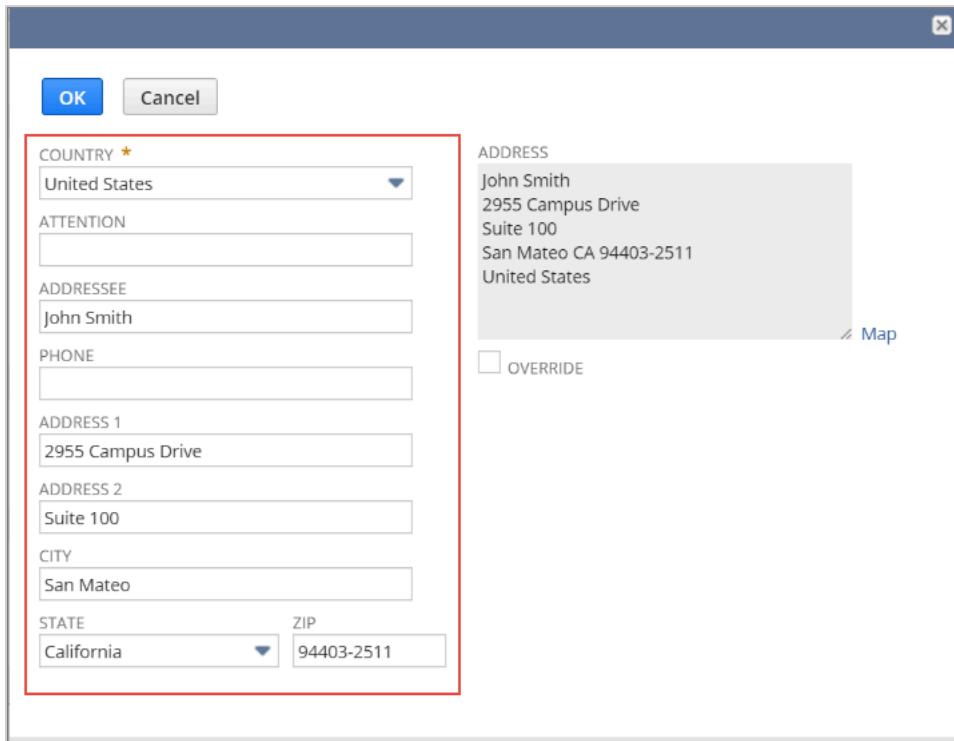
- After you instantiate a subrecord, you can set values on the subrecord's body and sublist fields with the same methods as you would on a record.
- In general, the sublists that exist within subrecords are not labeled in the UI. To find the name of a subrecord's sublist, refer to the [SuiteScript Records Browser](#). For details on using the Records Browser to find details on subrecords, see [Finding Subrecord Details in the Records Browser](#).

For examples of subrecords that are structured in different ways, see the following sections:

- [Writable Body Fields Example](#)
- [Writable Sublist Example](#)
- [Writable Body Fields and Sublist Example](#)

Writable Body Fields Example

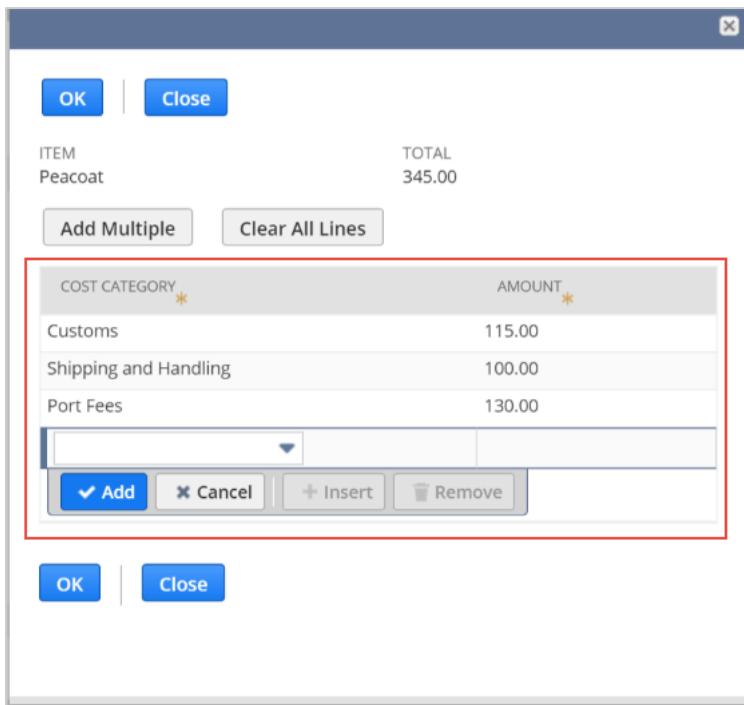
The address subrecord has several writable body fields, such as city, state, and zip. It has no sublist.



After you instantiate an address subrecord, you can set values for its body fields with the `Record.setValue(options)` method, the same as if you were setting values on an instance of a record. For more details, see [Creating an Address Sublist Subrecord Example](#) and [Creating an Address on a Subsidiary Record Example](#).

Writable Sublist Example

The landed cost subrecord has a sublist that lets you list individual expenses associated with merchandise you have received. The body fields on this subrecord are read-only, but the sublist is writable. To add details about an expense, you add a line to the sublist.

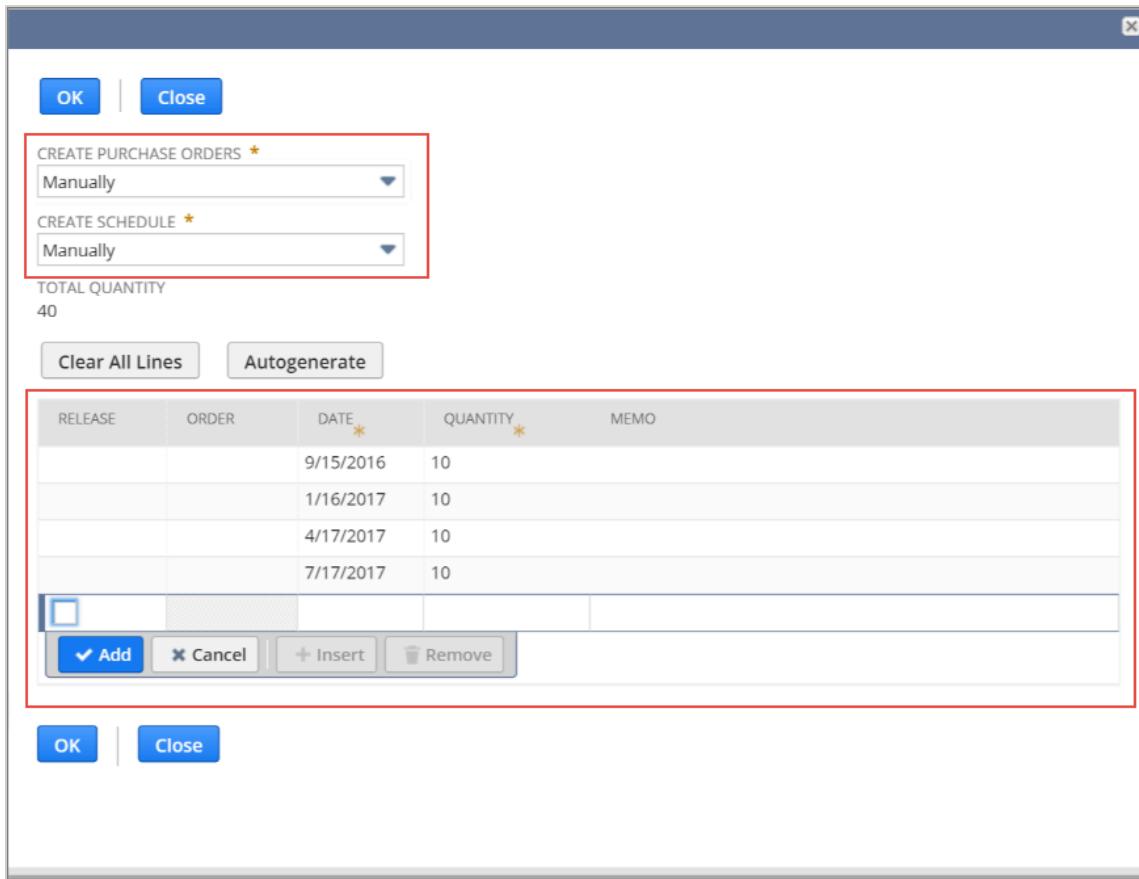


After you instantiate a landed cost subrecord, you can set values for its sublist fields with the same methods you would to set values on a record's sublist: `setSublistValue()` and `setCurrentSublistValue()`.

For more details, see [Creating a Landed Cost Sublist Subrecord Example](#).

Writable Body Fields and Sublist Example

The order schedule subrecord has a sublist that lets you configure how and when upcoming purchase orders are to be created. This subrecord has body fields that let you specify various qualities of the schedule, such as whether individual purchase orders must be created manually. It also has a sublist that lets you enter dates for the upcoming purchase orders.



Again, after you instantiate an order schedule subrecord, you can set values its fields with the same methods you would use to set values on a record. For body fields, use `setValue()`. For sublist fields, use `setCurrentSublistValue()` or `setSublistValue()`.

For more details, see [Creating an Order Schedule Sublist Subrecord Example](#).

Finding Subrecord Details in the Records Browser



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1 and SuiteScript 2.1.

As with records, you can find important information about subrecords in the [SuiteScript Records Browser](#). The browser includes details about each subrecord type. It also includes information about record types that can be parents to subrecords. For more guidance, see the following sections:

- [Finding Details About Parent Record Types](#)
- [Finding Details About Subrecord Types](#)

Finding Details About Parent Record Types

Every subrecord instance must have a parent record. An instance of a subrecord exists only to provide information about an instance of a record.

The parent record includes a field that contains, or references, the subrecord. To create a subrecord instance on a record, you must reference this field. These fields are always identified in the Records Browser as fields of type summary.

| item - Item | | | |
|--------------------|----------|------------------|----------|
| Internal ID | Type | Label | Required |
| amount | currency | Amount | false |
| billvariancestatus | text | | false |
| catchupperiod | select | | false |
| class | select | Class | false |
| deferrevrec | checkbox | | false |
| department | select | Department | false |
| description | textarea | Description | false |
| id | text | | false |
| isvsoebundle | text | | false |
| item | select | Item | true |
| itemsubtype | text | | false |
| itemtype | text | | false |
| line | text | | false |
| linenumber | integer | | false |
| location | select | Location | false |
| matrixtype | text | | false |
| options | text | | false |
| orderschedule | summary | Schedule | false |
| quantity | float | Quantity | false |
| quantityordered | float | Quantity Ordered | false |
| rate | rate | Rate | false |
| rateschedule | text | | false |
| units | select | Units | false |
| vendorname | text | Vendor Name | false |

Summary fields can occur either on the body of the parent record or in a sublist. It is important to know where the field occurs, because this distinction affects how you instantiate the subrecord. In the reference page for each record type in the Records Browser, the **Fields** table lists all of the record type's body fields. The tables listed under the heading **Sublists** shows sublist fields. See also [Body Field Subrecords](#) and [Sublist Subrecords](#).

In some cases, the values set for other fields on a record can affect the availability or behavior of the summary field. For example:

- On an assembly build record, the availability of the inventorydetail summary field varies depending on the value of the record's item field. The summary field is available only if the item field references a serialized or lot-numbered assembly item. For an example of working with this record-subrecord combination, see [Creating an Inventory Detail Subrecord on a Body Field Example](#).
- On a vendor bill record, the landedcost summary field is available only if the landedcostperline body field is set to true. For an example of working with this record-subrecord combination, see [Creating a Landed Cost Sublist Subrecord Example](#).
- On a sales transaction, the value of the shippingaddress summary field is affected by the shipaddresslist field. For general details about shipping and billing addresses, see [Scripting Transaction Shipping and Billing Addresses](#). For a script example, see [Using SuiteScript 2.x to Create a New Shipping Address Example](#).

Finding Details About Subrecord Types

As with record types, the Records Browser includes a reference page for each subrecord type. Subrecords are listed alphabetically with records.

Each subrecord is identified by a label displayed beneath the internal ID.

Order Schedule

Internal ID: orderschedule

Subrecord

Fields

| Internal ID | Type | nlapiSubmitField | Label | Required |
|---------------------|---------|------------------|------------------------|----------|
| createpurchaseorder | select | false | Create Purchase Orders | true |
| createschedule | select | false | Create Schedule | true |
| currencyprecision | integer | false | | false |
| enddate | date | false | End Date | false |
| externalid | text | false | ExternalId | false |
| item | integer | false | | false |
| releasefrequency | select | false | Release Frequency | false |
| startdate | date | false | Start Date | false |
| total | float | false | Total Quantity | false |

Sublists

schedule - Schedule

| Internal ID | Type | Label | Required |
|---------------|-------------|----------|----------|
| amount | poscurrency | | false |
| id | integer | | false |
| memo | text | Memo | false |
| orderschedule | integer | | false |
| purchaseorder | select | Order | false |
| quantity | posfloat | Quantity | true |
| release | checkbox | Release | false |
| trandate | date | Date | true |

As with record types, the reference page shows the subrecord's scriptable fields and sublists.

Note that in the UI, the sublists of subrecords typically are not labeled. However, the Records Browser displays the name of each sublist.

About the Address Subrecord



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The address subrecord has certain qualities that are unique. These characteristics can make the process of interacting with the address subrecord different from other subrecords. For details, see the following sections:

- Billing and Shipping Addresses Can Be Sourced from Other Records
- The Address Subrecord Can Have Custom Forms
- Address Data Is Summarized on the Parent Record



Note: See also [Scripting Transaction Shipping and Billing Addresses](#).

Billing and Shipping Addresses Can Be Sourced from Other Records

With most subrecord types, an instance of the subrecord is unique to the record where it was created. However, in some cases, a single address subrecord instance can be referenced by multiple records.

For example: You can create multiple addresses for an entity. If you later create a transaction for that entity, you can use one of the addresses defined on the entity record as the shipping or billing address for the transaction. For this reason, setting a value for a shipping or billing address is in some cases slightly different from the way that you set other address summary fields. For details, see [Addresses Can Be Sourced from Entity Records](#).

The Address Subrecord Can Have Custom Forms

Compared with other subrecords, the address subrecord is unique in that you can create custom entry forms for it. For example, you may want to create custom forms for different countries.

If your account has multiple address forms, and if you are not seeing the expected results, the reason could be related to the form. In particular, if your script is using dynamic mode, the first value you set on the subrecord form should be country. The reason is that if you set a value for country that differs from the default, the form resets when the country value changes. Therefore, as a best practice, set the country value first.



Note: The **Override**, **Phone**, and **Zip** fields of the address subrecord are not exposed as search filters from the customer record.

Address Data Is Summarized on the Parent Record

With most types of subrecords, details that you enter on the subrecord are not summarized on the parent record when you view it in the UI. You have to open the subrecord in its own window to view its data.

The address subrecord is an exception to this rule. After you enter details into an address subrecord and return to the main view of the record, typically the system displays a summary of the details you entered. For example, in the following screenshot, subrecord data is summarized in the Address column.

| Relationships | Communication | Address | Sales | Marketing | Support | Financial | Preferences | System Information | |
|---------------|------------------|-----------------|-------|---------------------|----------------------|-----------|--|--------------------|------|
| ID | DEFAULT SHIPPING | DEFAULT BILLING | | RESIDENTIAL ADDRESS | LABEL | | ADDRESS | | EDIT |
| 2359 | Yes | Yes | | | Company Headquarters | | 2955 Campus Drive Suite 100 San Mateo CA 94403-2511 United States | | |

This summary represents the value of one field on the address subrecord called `addrtext`. This value is created through the use of a template and generated from other values entered on the subrecord, such as the values for the city and state fields. Each address form can have its own template for determining the `addrtext` value. You can view the template for any address form by viewing its record at Customization > Forms > Address Forms.

An error in the `addrtext` field does not necessarily signify an error in the other values saved to the subrecord. If an error exists in the summary, review the template to make sure that it is capturing the values you intend.

To view the values for all of the fields on the subrecord, do one of the following:

- Use one of the following methods to retrieve the subrecord: `getCurrentSublistSubrecord()`, `getSublistSubrecord()`, or `getSubrecord()`. For an example, see [Retrieving an Address Subrecord Example](#).

- In the UI, open the subrecord for editing in its own window.

Client Scripts Attached to the Address Subrecord

Client scripts attached to address subrecords on transaction records may execute on the server and on the client. Be aware that this is expected behavior. For logic in a client script attached to an address subrecord to execute only once, wrap the logic in an if statement that immediately exits the script on the server. For example:

```
1 if (typeof document!=='undefined') {
2 // client script logic
3 }
```

If a client script attached to an address subrecord loads the [N/currentRecord Module](#), the script fails. Although you cannot load the [N/currentRecord Module](#) in a client script that is attached to the address subrecord, you can obtain currentRecord from context.currentRecord:

```
1 define([],function(){
2   function saveRecord(context){
3     var currentRecord = context.currentRecord;
4     return true;
5   }
6   return {
7     saveRecord: saveRecord
8   }
9});
```

See the help topic [currentRecord.CurrentRecord](#).

You can attach client scripts to custom entry forms, custom transaction forms, or custom address forms. See [Attaching a Client Script to a Form](#).

For information about client scripts, see [SuiteScript 2.x Client Script Type](#).

Subrecord Scripting in SuiteScript 2.x Compared With 1.0

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Compared with SuiteScript 1.0, SuiteScript 2.x introduces the following changes in how you script subrecords:

- [A Single Method Lets You Create and Load Subrecords](#)
- [Subrecords Do Not Have to Be Explicitly Saved](#)
- [To Create Addresses, You Must Use Subrecord Methods](#)

A Single Method Lets You Create and Load Subrecords

In SuiteScript 1.0, you use one set of APIs to create subrecords and another set to edit subrecords. For example, you could use nlapiCreateSubrecord to create a subrecord and you could use nlapiEditSubrecord to edit a subrecord.

By contrast, in SuiteScript 2.x, any method that creates a subrecord can also be used to load that subrecord for the purpose of editing it. These methods all have the word **get** in their names. For example, you use the getSubrecord() method to create or load a subrecord that exists on the body of a record. You use the getSublistSubrecord() or getCurrentSublistSubrecord() method to create or load a subrecord that exists on a sublist line.

When you use any of these methods, the system responds with the following logic:

- If a subrecord instance already exists in the specified field, the subrecord is loaded.
- If no subrecord instance exists, the system creates one. You can then set values on the field. The subrecord is saved when you save the record.

Subrecords Do Not Have to Be Explicitly Saved

In SuiteScript 1.0, you had to explicitly save a subrecord prior to saving the record. However, in SuiteScript 2.x, after you create a subrecord or make changes to one, you are not required to explicitly save the subrecord (and no methods exist for that purpose). Your new subrecord is saved at the time you save the record. The same rule applies if you make changes to an existing subrecord. Your updates are saved at the time you save the record.

To Create Addresses, You Must Use Subrecord Methods

The address subrecord was introduced in version 2014.2. Prior to that time, each address was represented on a record as a series of body fields or as a line in a sublist.

After the introduction of the address subrecord, SuiteScript 1.0 was enhanced to support two methods of interacting with addresses: In 1.0, you have the choice of interacting with addresses using subrecord APIs, which is the preferred method. But in 1.0 you can also interact with addresses using the legacy approach: setting values for the address body and sublist fields that used to exist. This support was made possible by logic added to the system that read the values set in this manner and created an address subrecord on behalf of the 1.0 script. Because this support exists in 1.0, these deprecated fields are displayed in the SuiteScript Records Browser as available fields.

However, in SuiteScript 2.x, to create an address, you **must** use subrecord methods. The system does not provide logic for the legacy address body and sublist fields. For that reason, to create, edit, or load an address in SuiteScript 2.x, you must instantiate the address subrecord by referencing the appropriate summary field.

Scripting Subrecords that Occur on Sublist Lines

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

In many cases, a subrecord is accessed through a field on a sublist line. For example:

- Every purchase order must include a list of items. Depending on the configuration of each item, the sublist line may be required to include an inventory detail subrecord.
- A vendor bill may include a list of items. If the bill is configured to track landed cost per line, each line in the Items sublist may include a landed cost subrecord.
- An employee may have multiple addresses. Each address is stored in a subrecord, and each subrecord is associated with a line in the Address sublist.

In each case, the sublist line also has fields that are not part of the subrecord. The subrecord itself is associated with only one field on the sublist line. In the SuiteScript Records Browser, this field is always identified as a field of type summary.

For more details, see the following sections:

- [Using SuiteScript 2.x to Create a Subrecord in a Sublist Field](#)
- [Using SuiteScript 2.x to Edit a Subrecord that Occurs in a Sublist Field](#)
- [Using SuiteScript 2.x to Retrieve a Sublist Subrecord](#)

Using SuiteScript 2.x to Create a Subrecord in a Sublist Field

Depending on the record type and other variables, a line on a record's sublist can include a field that references a subrecord. In many cases, you must add the subrecord at the time you are creating the sublist line. In other cases, you can go back and add the subrecord later.

To create a sublist subrecord, your script must use the [N/record Module](#). The script can use either dynamic or standard mode. For details, see the following sections:

- [Creating a Sublist Subrecord in Dynamic Mode](#)
- [Creating a Sublist Subrecord in Standard Mode](#)

Subrecords can also occur in the body field of a record. For details on working with subrecords when occur in body fields, see [Scripting Subrecords that Occur in Body Fields](#). For an overview of the difference between these two types of placement, see [Body Field Subrecords and Sublist Subrecords](#).

 **Note:** For more details about the methods referenced in this topic, see the help topic [Record Object Members](#).

Creating a Sublist Subrecord in Dynamic Mode

If your script uses dynamic mode, you can use the following procedure to create a subrecord in a sublist field.

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#).

To create a sublist subrecord in dynamic mode:

1. If you want to add a new record, create it and set the required body fields. If you want to update an existing record, load the record.
2. Do one of the following:
 - If you want to create a new sublist line, create the line using the [Record.selectNewLine\(options\)](#) method. Set any required values on the sublist line with the [Record.setCurrentSublistValue\(options\)](#) method.
 - If you want to add a new subrecord to an existing sublist line, identify that line using the [Record.selectLine\(options\)](#) method.
3. Create the new subrecord with the [Record.getCurrentSublistSubrecord\(options\)](#) method. This method takes two arguments:
 - A sublistId, which identifies the sublist.
 - A fieldId, which identifies the field on the sublist that contains the subrecord. In the Records Browser, the field that holds the subrecord is always identified as a field of type summary.

For example, you could use an expression like the following to create an order schedule subrecord on an item sublist:

```

1 ...
2 var orderScheduleSubrecord = blanketPurchaseOrder.getCurrentSublistSubrecord({
3   sublistId: 'item',
4   fieldId: 'orderschedule'
5 });
6 ...

```

4. As appropriate, set body fields on the subrecord with the [Record.setValue\(options\)](#) method. For example, on an order schedule subrecord, you could use the following expression to set a value for the Create Purchase Orders select field.

```
1 | ...
```

```

2 | orderScheduleSubrecord.setValue({
3 |   fieldId: 'createpurchaseorder',
4 |   value: 'LEAD'
5 | });
6 |

```

Be aware that not all subrecords have writable body fields.

5. If the subrecord has a sublist, generally you are required to add at least one line to the sublist. For each line, use the following guidelines:
 - Create the line with the [Record.selectNewLine\(options\)](#) method.
 - Set required values on the line with the [Record.setCurrentSublistValue\(options\)](#) method.
 - Save the subrecord's sublist line with the [Record.commitLine\(options\)](#) method.

For example, if you create an order schedule subrecord, you could use the following expressions to create a line on the subrecord's schedule sublist:

```

1 | ...
2 | orderScheduleSubrecord.selectNewLine
3 |   sublistId: 'schedule',
4 | );
5 |
6 | orderScheduleSubrecord.setCurrentSublistValue({
7 |   sublistId: 'schedule',
8 |   fieldId: 'quantity',
9 |   value: 1
10 | });
11 |
12 | orderScheduleSubrecord.setCurrentSublistValue({
13 |   sublistId: 'schedule',
14 |   fieldId: 'trandate',
15 |   value: dateVariable
16 | });
17 |
18 | orderScheduleSubrecord.commitLine({
19 |   sublistId: 'schedule'
20 | });
21 |

```

6. Save the sublist line that holds the subrecord with the [Record.commitLine\(options\)](#) method.
7. Save the record with the [Record.save\(options\)](#) method.



Note: For a full script examples, see [Creating an Inventory Detail Sublist Subrecord Example](#) and [Creating an Address Sublist Subrecord Example](#).

Creating a Sublist Subrecord in Standard Mode

If your script uses standard mode, you can use the following procedure to create a subrecord in a sublist field.

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

To create a sublist subrecord in standard mode:

1. If you want to add a new record, create it and set the required body fields. If you want to update an existing record, load the record.
2. If you want to create a new sublist line, create the line with the [Record.insertLine\(options\)](#) method. Set any required fields on the sublist line.
3. Create the new subrecord with the [Record.getSublistSubrecord\(options\)](#) method. This method takes three arguments:
 - A sublistId, which identifies the sublist.

- A fieldId, which identifies the field on the sublist that contains the subrecord. In the Records Browser, the field that holds the subrecord is always identified as a field of type summary.
- A line number.

For example, you could use an expression like the following to create an inventory detail subrecord on the first line in an item sublist:

```

1 ...
2 inventoryDetailSubrecord = rec.getSublistSubrecord({
3   sublistId: 'item',
4   fieldId: 'inventorydetail',
5   line: 0
6 });
7 ...

```

4. Set body fields on the subrecord with the [Record.setValue\(options\)](#) method. Be aware that not all subrecords have writable body fields.
5. If the subrecord has a sublist, generally you are required to add at least one line to the sublist. For each line, use the following guidelines:
 - Use the [Record.insertLine\(options\)](#) method to create the line.
 - Set values on the line with the [Record.setSublistValue\(options\)](#) method.

For example, if you were creating an order schedule subrecord, you could use the following expressions to create a line on the subrecord's schedule sublist:

```

1 ...
2 subrecordInvDetail.setSublistValue({
3   sublistId: 'inventoryassignment',
4   fieldId: 'receiptinventorynumber',
5   line: 0,
6   value: '012345'
7 });
8 ...

```

6. Save the record with the [Record.save\(options\)](#) method.

Note: For a full script example of creating a sublist subrecord in standard mode, see [Creating a Landed Cost Sublist Subrecord Example](#).

Creating an Inventory Detail Sublist Subrecord Example

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example shows how to create a purchase order record that includes an inventory detail subrecord. The script adds one line to the item sublist and creates an inventory detail subrecord on that line.

To use this example, you must meet the following prerequisites:

- The Advanced Bin / Numbered Inventory Management feature must be enabled at Setup > Company > Enable Features , on the Items & Inventory subtab.
- The item you add to the sublist should be a lot-numbered inventory item.
- The receiptinventorynumber value must be unique in your system.

This example uses dynamic mode, but you could also add the subrecord using standard mode. For general details about using either approach to add a sublist subrecord, see [Using SuiteScript 2.x to Create a Subrecord in a Sublist Field](#).

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType UserEventScript
4   */
5
6 define([ 'N/record' ],function(record){
7     function afterSubmit(context)
8       // Create the purchase order.
9       var rec = record.create({
10         type: record.Type.PURCHASE_ORDER,
11         isDynamic: true
12       });
13
14       // Set body fields on the purchase order.
15       rec.setValue({
16         fieldId: 'entity',
17         value: '1663'
18       });
19
20       rec.setValue({
21         fieldId: 'location',
22         value: '6'
23       });
24
25       // Create one line in the item sublist.
26       rec.selectNewLine({
27         sublistId: 'item'
28       });
29
30       rec.setCurrentSublistValue({
31         sublistId: 'item',
32         fieldId: 'item',
33         value: '299'
34       });
35
36       rec.setCurrentSublistValue({
37         sublistId: 'item',
38         fieldId: 'quantity',
39         value: 1
40       });
41
42       // Create the subrecord for that line.
43       var subrec = rec.getCurrentSublistSubrecord({
44         sublistId: 'item',
45         fieldId: 'inventorydetail'
46       });
47
48       // Add a line to the subrecord's inventory assignment sublist.
49       subrec.selectNewLine({
50         sublistId: 'inventoryassignment'
51       });
52
53       subrec.setCurrentSublistValue({
54         sublistId: 'inventoryassignment',
55         fieldId: 'quantity',
56         value: 2
57       });
58
59       subrec.setCurrentSublistValue({
60         sublistId: 'inventoryassignment',
61         fieldId: 'receiptinventorynumber',
62         value: '01234'
63       });
64
65       // Save the line in the subrecord's sublist.
66       subrec.commitLine({
67         sublistId: 'inventoryassignment'
68       });
69
70       // Save the line in the record's sublist
71       rec.commitLine({

```

```

72     sublistId: 'item'
73   });
74
75   // Save the record.
76   try {
77     var recId = rec.save();
78     log.debug({
79       title: 'Record created successfully',
80       details: 'Id: ' + recId
81     });
82   } catch (e) {
83     log.error({
84       title: e.name,
85       details: e.message
86     });
87   }
88 }
89 return {
90   afterSubmit: afterSubmit
91 };
92 });

```

Creating an Address Sublist Subrecord Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example shows how to create an employee record and populate the Address sublist with one line. The script also creates an address subrecord on the sublist line.

This example uses dynamic mode, but you could also add the subrecord using standard mode. For general details about using either approach to add a sublist subrecord, see [Using SuiteScript 2.x to Create a Subrecord in a Sublist Field](#).

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType UserEventScript
4 */
5
6 define([ 'N/record' ],function(record){
7   function afterSubmit(context){
8
9     // Create the record.
10    var rec = record.create({
11      type: record.Type.EMPLOYEE,
12      isDynamic: true
13    });
14
15    // Set the required body fields.
16    rec.setValue({
17      fieldId: 'firstname',
18      value: 'John'
19    });
20
21    rec.setValue({
22      fieldId: 'lastname',
23      value: 'Smith'
24    });
25
26    rec.setValue({
27      fieldId: 'subsidiary',
28      value: '1'
29    });
30
31    // Create a line in the Address sublist.
32    rec.selectNewLine({
33      sublistId: 'addressbook'
34    });

```

```

35     // Set an optional field on the sublist line.
36     rec.setCurrentSublistValue({
37         sublistId: 'addressbook',
38         fieldId: 'label',
39         value: 'Primary Address'
40     });
41
42
43     // Create an address subrecord for the line.
44     var subrec = rec.getCurrentSublistSubrecord({
45         sublistId: 'addressbook',
46         fieldId: 'addressbookaddress'
47     });
48
49     // Set body fields on the subrecord. Because the script uses
50     // dynamic mode, you should set the country value first. The country
51     // value determines which address form is to be used, so by setting
52     // this value first, you ensure that the values for the rest
53     // of the form's fields will be set properly.
54     subrec.setValue({
55         fieldId: 'country',
56         value: 'US'
57     });
58
59     subrec.setValue({
60         fieldId: 'city',
61         value: 'San Mateo'
62     });
63
64     subrec.setValue({
65         fieldId: 'state',
66         value: 'CA'
67     });
68
69     subrec.setValue({
70         fieldId: 'zip',
71         value: '94403'
72     });
73
74     subrec.setValue({
75         fieldId: 'addr1',
76         value: '2955 Campus Drive'
77     });
78
79     subrec.setValue({
80         fieldId: 'addr2',
81         value: 'Suite 100'
82     });
83
84     // Save the sublist line.
85     rec.commitLine({
86         sublistId: 'addressbook'
87     });
88
89     // Save the record.
90     try{
91         var recId = rec.save();
92
93         log.debug({
94             title: 'Record created successfully',
95             details: 'Id: ' + recId
96         });
97     } catch (e){
98         log.error({
99             title: e.name,
100             details: e.message
101         });
102     }
103 }
104 return {
105     afterSubmit: afterSubmit
106 };
107 });

```

Creating an Order Schedule Sublist Subrecord Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example creates a blanket purchase order record. It creates one line in the item sublist and creates an order schedule subrecord on that line.

To use this example, the Blanket Purchase Order feature must be enabled at Setup > Company > Enable Features, on the Transactions subtab.

This example uses dynamic mode, but you could also add the subrecord using standard mode. For general details about using either approach to add a sublist subrecord, see [Using SuiteScript 2.x to Create a Subrecord in a Sublist Field](#).

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType UserEventScript
4 */
5
6 define([ 'N/record' ],function(record) {
7     function afterSubmit(context) {
8         // Create the record.
9         var rec = record.create({
10             type: record.Type.BLANKET_PURCHASE_ORDER,
11             isDynamic: true
12         });
13
14         // Set body fields on the record.
15         rec.setValue({
16             fieldId: 'entity',
17             value: '1663'
18         });
19
20         rec.setValue({
21             fieldId: 'location',
22             value: '6'
23         });
24
25         rec.setValue({
26             fieldId: 'memo',
27             value: '456789'
28         });
29
30         // Create one line in the item sublist.
31         rec.selectNewLine({
32             sublistId: 'item',
33             line: 0
34         });
35
36         rec.setCurrentSublistValue({
37             sublistId: 'item',
38             fieldId: 'item',
39             value: '500'
40         });
41
42         rec.setCurrentSublistValue({
43             sublistId: 'item',
44             fieldId: 'quantity',
45             value: '1'
46         });
47
48         // Create the subrecord for that line.
49         var subrec = rec.getCurrentSublistSubrecord({
50             sublistId: 'item',
51             fieldId: 'orderschedule'
52         });
53

```

```

54 // Set a field on the body of the subrecord.
55 subrec.setValue({
56   fieldId: 'createpurchaseorder',
57   value: 'LEAD'
58 });
59
60 // Create a line in the subrecord's sublist.
61 subrec.selectNewLine({
62   sublistId: 'schedule',
63 });
64
65 subrec.setCurrentSublistValue({
66   sublistId: 'schedule',
67   fieldId: 'quantity',
68   value: 1
69 });
70
71 var nextQuarter = new Date();
72 nextQuarter.setDate(nextQuarter.getDate() + 90);
73
74 subrec.setCurrentSublistValue({
75   sublistId: 'schedule',
76   fieldid: 'trandate',
77   value: nextQuarter
78 });
79
80 // Save the line in the subrecord's sublist.
81 subrec.commitLine({
82   sublistId: 'schedule'
83 });
84
85 // Save the line in the record's sublist
86 rec.commitLine({
87   sublistId: 'item'
88 });
89
90 // Save the record.
91 try {
92   var recId = rec.save();
93   log.debug({
94     title: 'Record created successfully',
95     details: 'Id: ' + recId
96   });
97 } catch (e) {
98   log.error({
99     title: e.name,
100    details: e.message
101  });
102 }
103
104 return {
105   afterSubmit: afterSubmit
106 };
107 });

```

Creating a Landed Cost Sublist Subrecord Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example creates a vendor bill record. This example creates one line in the item sublist. It also sets the Landed Cost per Line option to true. With this configuration, it is possible to create a landed cost subrecord for each line.

To use this example, the Landed Cost feature must be enabled at Setup > Company > Enable Features , on the Items & Inventory subtab.

This example uses standard mode, but you could also add the subrecord using dynamic mode. For general details about using either approach to add a sublist subrecord, see [Using SuiteScript 2.x to Create a Subrecord in a Sublist Field](#).

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType UserEventScript
4   */
5
6 define([ 'N/record' ],function(record) {
7     function afterSubmit(context) {
8         // Create the record.
9         var rec = record.create({
10             type : record.Type.VENDOR_BILL,
11             isDynamic: false
12         });
13
14         // Set body fields on the record.
15         rec.setValue({
16             fieldId: 'entity',
17             value: '1663'
18         });
19
20         rec.setValue({
21             fieldId: 'location',
22             value: '6'
23         });
24
25         rec.setValue({
26             fieldId: 'tranid',
27             value: '101A'
28         });
29
30         // Set the Landed Cost per Line field to true.
31         rec.setValue({
32             fieldId: 'landedcostperline',
33             value: true
34         });
35
36         // Add an item to the Item sublist.
37         rec.insertLine({
38             sublistId: 'item',
39             line: 0
40         });
41
42         // Set values on the sublist line.
43         rec.setSublistValue({
44             sublistId: 'item',
45             fieldId: 'item',
46             line: 0,
47             value: '599'
48         });
49
50         rec.setSublistValue({
51             sublistId: 'item',
52             fieldId: 'quantity',
53             line: 0,
54             value: 1
55         });
56
57         rec.setSublistValue({
58             sublistId: 'item',
59             fieldId: 'location',
60             line: 0,
61             value: '6'
62         });
63
64         // Create the subrecord.
65         var subrec = rec.getSublistSubrecord({
66             sublistId: 'item',
67             fieldId: 'landedcost',
68             line: 0
69         });
70
71         // Add a line to the subrecord's Landed Cost Data sublist.

```

```

72     subrec.insertLine({
73         sublistId: 'landedcostdata',
74         line: 0
75     });
76
77     // Set values on the subrecord's sublist line.
78     subrec.setSublistValue({
79         sublistId: 'landedcostdata',
80         fieldId: 'costcategory',
81         line: 0,
82         value: 2
83     });
84
85     subrec.setSublistValue({
86         sublistId: 'landedcostdata',
87         fieldId: 'amount',
88         line: 0,
89         value: 17.85
90     });
91
92     // Save the record.
93     try {
94         var recId = rec.save();
95         log.debug({
96             title: 'Record created successfully',
97             details: 'Id: ' + recId
98         });
99     } catch (e) {
100         log.error({
101             title: e.name,
102             details: e.message
103         });
104     }
105 }
106 return {
107     afterSubmit: afterSubmit
108 };
109 });

```

Using SuiteScript 2.x to Edit a Subrecord that Occurs in a Sublist Field

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

In some cases, your script can make changes to a subrecord that occurs in a sublist field.

To edit a sublist subrecord, your script must use the [N/record Module](#). The script can use either dynamic or standard mode. For details, see the following sections:

- [Editing a Subrecord in Dynamic Mode](#)
- [Editing a Subrecord in Standard Mode](#)

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

Editing a Subrecord in Dynamic Mode

If your script uses dynamic mode, you can use the following procedure to edit a subrecord that occurs in a sublist field.

To edit a subrecord in dynamic mode:

1. Load the record.

2. Use the [Record.selectLine\(options\)](#) method to identify the sublist and line that contain the subrecord that you want to update.
3. Retrieve the subrecord with the [Record.getCurrentSublistSubrecord\(options\)](#) method. This method takes two arguments:
 - A sublistId.
 - A fieldId, which identifies the field on the sublist that contains the subrecord. In the Records Browser, the field that holds the subrecord is always identified as a field of type summary.

For example, suppose you are working with an entity record, such as an employee or customer. You could use an expression like the following to load an address subrecord from the entity's Address sublist:

```

1 ...
2 var addressSubrecord = rec.getCurrentSublistSubrecord({
3   sublistId: 'addressbook',
4   fieldId: 'addressbookaddress'
5 });
6 ...

```

4. As appropriate, update body fields on the subrecord with the [Record.setValue\(options\)](#) method. For example, you could use an expression like the following to update a value on the address subrecord:

```

1 ...
2 addressSubrecord.setValue({
3   fieldId: 'city',
4   value: 'St. Petersburg'
5 });
6 ...

```

However, note that some subrecords do not have writable body fields.

5. If the subrecord has a sublist whose values you want to modify, use the following steps for each line you want to change:
 1. Identify the line you want to change with the [Record.selectLine\(options\)](#) method.
 2. For each value you want to change, use the [Record.setCurrentSublistValue\(options\)](#) method to identify the field and the new value.
 3. Save your changes to the subrecord's sublist line with the [Record.commitLine\(options\)](#) method.
6. Save the line that holds the subrecord with the [Record.commitLine\(options\)](#) method.
7. Save the record with the [Record.save\(options\)](#) method.



Note: For a full script example that shows editing a sublist subrecord in dynamic mode, see [Updating an Order Schedule Sublist Subrecord Example](#).

Editing a Subrecord in Standard Mode

If your script uses standard mode, you can use the following procedure to edit a subrecord that occurs in a sublist field.

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

To edit a subrecord in standard mode:

1. Load the record.

2. Retrieve the subrecord with the [Record.getSublistSubrecord\(options\)](#) method. This method takes three arguments:

- A sublistId.
- A fieldId, which identifies the field on the sublist that contains the subrecord. In the Records Browser, the field that holds the subrecord is always identified as a field of type summary.
- A line number, which identifies the sublist line that contains the subrecord you want to change.

For example, you could use an expression like the following to load an inventory detail subrecord from an item sublist:

```

1 ...
2 inventoryDetailSubrecord = rec.getSublistSubrecord({
3   sublistId: 'item',
4   fieldId: 'inventorydetail',
5   line: 0
6 });
7 ...

```

3. Update body fields on the subrecord with the [Record.setValue\(options\)](#) method. Be aware that not all subrecords have writable body fields.
4. If the subrecord has a sublist whose values you want to modify, use the [Record.setSublistValue\(options\)](#) method to update the appropriate value. This method takes four arguments:
 - A sublistId.
 - A fieldId, which identifies the field you want to change.
 - A line number, which identifies the sublist line you want to change.
 - The new value.

For example, if you were updating an inventory detail subrecord, you could use the following expression to update the serial number on the first line of the inventory assignment sublist:

```

1 ...
2 inventoryDetailSubrecord.setSublistValue({
3   sublistId: 'inventoryassignment',
4   fieldId: 'receiptinventorynumber',
5   line: 0,
6   value: '56789'
7 });
8 ...

```

5. Save the record with the `save()` method.



Note: For a full script example showing how to modify a subrecord in standard mode, see [Updating an Address Subrecord Example](#).

Updating an Order Schedule Sublist Subrecord Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example loads an existing blanket purchase order record. It selects a line on the record's item sublist, retrieves the subrecord associated with that line, and makes changes to the subrecord.

To use this example, you must meet the following prerequisites:

- The Blanket Purchase Order feature must be enabled at Setup > Company > Enable Features , on the Transactions subtab.

- The blanket purchase order record that you reference should have at least one line in the item sublist. The line should reference an existing order schedule subrecord.

This example uses dynamic mode, but you could also update the subrecord using standard mode. For general details about using either approach to update a sublist subrecord, see [Using SuiteScript 2.x to Edit a Subrecord that Occurs in a Sublist Field](#). To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

```

1 define([ 'N/record' ],function(record) {
2     function afterSubmit(context) {
3         // Load the blanket purchase order record.
4         var rec = record.load({
5             type: record.Type.BLANKET_PURCHASE_ORDER,
6             id: 3319,
7             isDynamic: true
8         });
9
10        // Select the sublist and line.
11        rec.selectLine({
12            sublistId: 'item',
13            line: 0
14        });
15
16        // Retrieve the subrecord.
17        var subrec = rec.getCurrentSublistSubrecord({
18            sublistId: 'item',
19            fieldId: 'orderschedule'
20        });
21
22        // Select the appropriate line in the subrecord's sublist.
23        subrec.selectLine({
24            sublistId: 'schedule',
25            line: 0
26        });
27
28        // Identify the field to be modified, and set new value.
29        var nextQuarter = new Date();
30        nextQuarter.setDate(nextQuarter.getDate() + 90);
31
32        subrec.setCurrentSublistValue({
33            sublistId: 'schedule',
34            fieldId: 'trandate',
35            value: nextQuarter
36        });
37
38        // Save the subrecord's sublist line.
39        subrec.commitLine({
40            sublistId: 'schedule'
41        });
42
43        // Save the item sublist line that contains the subrecord.
44        rec.commitLine({
45            sublistId: 'item'
46        });
47
48        // Save the record.
49        try {
50            var recId = rec.save();
51            log.debug({
52                title: 'Record updated successfully',
53                details: 'Id: ' + recId
54            });
55        } catch (e) {
56            log.error({
57                title: e.name,
58                details: e.message
59            });
60        }
61    }
62    return {
63        afterSubmit: afterSubmit

```

```

64     });
65 });

```

Updating an Address Subrecord Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example shows how to update an employee address. On the employee record, each address is stored in a subrecord that is associated with a line in the Address sublist.

This example uses standard mode, but you could also edit the subrecord using dynamic mode. For general details about using either approach to update a sublist subrecord, see [Using SuiteScript 2.x to Edit a Subrecord that Occurs in a Sublist Field](#).

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType UserEventScript
4 */
5
6 define([ 'N/record' ], function(record) {
7     function afterSubmit(context) {
8         // Load the record.
9         var rec = record.load({
10             type: record.Type.EMPLOYEE,
11             id: 1863,
12             isDynamic: false
13         });
14
15         // Retrieve the subrecord to be modified.
16         var subrec = rec.getSublistSubrecord({
17             sublistId: 'addressbook',
18             fieldId: 'addressbookaddress',
19             line: 0
20         });
21
22         // Change a field on the subrecord.
23         subrec.setValue({
24             fieldId: 'addr1',
25             value: '15 Main Street'
26         });
27
28         // Save the record.
29         try {
30             var recId = rec.save();
31             log.debug({
32                 title: 'Record updated successfully',
33                 details: 'Id: ' + recId
34             });
35         } catch (e) {
36
37             log.error({
38                 title: e.name,
39                 details: e.message
40             });
41         }
42     }
43     return {
44         afterSubmit: afterSubmit
45     };
46 });

```

Using SuiteScript 2.x to Retrieve a Sublist Subrecord

In some cases, you may want to retrieve data from a subrecord that occurs in a sublist field.

To retrieve a sublist subrecord, use the [N/record Module](#). Your script can use either dynamic or standard mode. For details, see the following sections:

- [Retrieving a Sublist Subrecord in Dynamic Mode](#)
- [Retrieving a Subrecord in Standard Mode](#)

Retrieving a Sublist Subrecord in Dynamic Mode

If your script uses dynamic mode, you can use the following procedure to retrieve a subrecord that occurs in a sublist field.

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

To retrieve a sublist subrecord in dynamic mode:

1. Load the parent record.
2. Select the line that contains the subrecord with the [Record.selectLine\(options\)](#).
3. Retrieve the subrecord with the [Record.getCurrentSublistSubrecord\(options\)](#) method. This method takes two arguments:
 - A sublistId.
 - A fieldId, which identifies the field on the sublist that contains the subrecord. In the Records Browser, the field that holds the subrecord is always identified as a field of type summary.

For example, suppose you are working with an entity record, such as an employee or customer. You could use an expression like the following to load an address subrecord from the entity's Address sublist:

```

1 ...
2 var addressSubrecord = rec.getCurrentSublistSubrecord({
3   sublistId: 'addressbook',
4   fieldId: 'addressbookaddress'
5 });
6 ...

```

4. If you want to retrieve a value from the body of the subrecord, use the [Record.getValue\(options\)](#). For example, you could use an expression like the following to retrieve a detail from the body of an address subrecord:

```

1 ...
2 var cityValue = addressSubrecord.getValue({
3   fieldId: 'city'
4 });
5 ...

```

5. If you want to retrieve a value from the subrecord's sublist, use the [Record.getSublistValue\(options\)](#) method.



Note: For a full script example showing how to retrieve a sublist subrecord in dynamic mode, see [Retrieving an Order Schedule Subrecord Example](#).

Retrieving a Subrecord in Standard Mode

If your script uses standard mode, use the following procedure to retrieve a sublist subrecord.

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

To retrieve a subrecord in standard mode

1. Load the parent record.
2. Retrieve the subrecord with the [Record.getSublistSubrecord\(options\)](#) method. This method takes three arguments:
 - A sublistId.
 - A fieldId, which identifies the field on the sublist that contains the subrecord. In the Records Browser, the field that holds the subrecord is always identified as a field of type summary.
 - A line number

For example, you could use an expression like the following to load an order schedule subrecord from a blanket purchase order record:

```

1 ...
2 var orderScheduleSubrecord = rec.getSublistSubrecord({
3   sublistId: 'item',
4   fieldId: 'orderschedule',
5   line: 0
6 });
7 ...

```

3. If you want to retrieve a value from the body of the subrecord, use the [Record.getValue\(options\)](#) method.
4. If you want to retrieve a value from the subrecord's sublist, use the [Record.getSublistValue\(options\)](#) method.

For example, the order schedule subrecord has a schedule sublist. If you wanted to retrieve the date value from the second line of the schedule sublist, you would use an expression like the following:

```

1 ...
2 var dateValue = orderScheduleSubrecord.getSublistValue({
3   sublistId: 'schedule',
4   fieldId: 'trandate',
5   line: 1
6 });
7 ...

```



Note: For a full script example showing how to retrieve a sublist subrecord in standard mode, see [Retrieving an Address Subrecord Example](#).

Retrieving an Order Schedule Subrecord Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example loads a blanket purchase order record. It selects a line on the item sublist, then retrieves the order schedule associated with that line. It reads two values from the subrecord and prints them to the script deployment execution log.

If you want to use this script, you must meet the following prerequisites:

- The Blanket Purchase Order feature must be enabled at Setup > Company > Enable Features , on the Transactions subtab.

- The blanket purchase order record you reference must have at least one line in the Item sublist, and that line should include an order schedule subrecord.

This example uses dynamic mode, but you could also load the subrecord using standard mode. For general details about using either approach to retrieve a sublist subrecord, see [Using SuiteScript 2.x to Retrieve a Sublist Subrecord](#).

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType userevents
4   */
5
6 define([ 'N/record' ],function(record) {
7     function afterSubmit(context) {
8       // Load the record.
9       var rec = record.load({
10         type: record.Type.BLANKET_PURCHASE_ORDER,
11         id: 3120,
12         isDynamic: true
13       });
14
15       // Retrieve the subrecord.
16
17       rec.selectLine({
18         sublistId: 'item',
19         line: 0
20       });
21
22       var subrec = rec.getCurrentSublistSubrecord({
23         sublistId: 'item',
24         fieldId: 'orderschedule',
25       });
26
27       // Create a variable and initialize it to the
28       // value of the trandate field.
29       var dateValue = subrec.getSublistValue({
30         sublistId: 'schedule',
31         fieldId: 'trandate',
32         line: 0
33       });
34
35       // Create a variable and initialize it to the
36       // value of the memo field.
37       var memoValue = subrec.getSublistValue({
38         sublistId: 'schedule',
39         fieldId: 'memo',
40         line: 0
41       });
42
43       // Print the retrieved values to the execution log.
44       try {
45         log.debug({
46           title: 'date value',
47           details: 'date value: ' + dateValue
48         });
49         log.debug({
50           title: 'memo value',
51           details: 'memo value: ' + memoValue
52         });
53       } catch (e) {
54         log.error({
55           title: e.name,
56           details: e.message
57         });
58       }
59     }
60
61     return {
62       afterSubmit: afterSubmit
63     };
64   }

```

63 | });

Retrieving an Address Subrecord Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

This example loads an employee record and selects a line on the address sublist. It retrieves a value from the sublist line that is not part of the subrecord. It also retrieves the address subrecord and reads one of the subrecord's fields. The script prints both values to the script deployment record's execution log.

This example uses standard mode, but you could also load the subrecord using dynamic mode. For general details about using either approach to retrieve a sublist subrecord, see [Using SuiteScript 2.x to Retrieve a Sublist Subrecord](#).

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType userevents
4 */
5
6 define([ 'N/record' ],function(record) {
7     function afterSubmit(context) {
8         // Load the record.
9         var rec = record.load({
10             type: record.Type.EMPLOYEE,
11             id: 1863,
12             isDynamic: false
13         });
14
15         // Retrieve a value from the sublist line
16         var labelValue = rec.getSublistValue({
17             sublistId: 'addressbook',
18             fieldId: 'label',
19             line: 0
20         });
21
22         // Retrieve the subrecord associated with that same line.
23         var subrec = rec.getSublistSubrecord({
24             sublistId: 'addressbook',
25             fieldId: 'addressbookaddress',
26             line: 0
27         });
28
29         // Create a variable to initialize it to the
30         // value of the subrecord's city field.
31         var cityValue = subrec.getValue({
32             fieldId: 'city'
33         });
34
35         // Print the retrieved values to the execution log.
36         try {
37             log.debug({
38                 title: 'label value',
39                 details: 'label value: ' + labelValue
40             });
41
42             log.debug({
43                 title: 'city value',
44                 details: 'city value: ' + cityValue
45             });
46         } catch (e) {
47             log.error({
48                 title: e.name,
49                 details: e.message
50             });
51     }
}

```

```

52     }
53     return {
54         afterSubmit: afterSubmit
55     };
56 });

```

Scripting Subrecords that Occur in Body Fields

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

In some cases, a subrecord is accessed through a body field on a record.

Most subrecords referenced from body fields behave in the same general way. However, there are some differences when working with subrecords that are used as the shipping or billing addresses on transactions.

For details, see the following sections:

- [Body Field Subrecords](#)
- [Transaction Shipping and Billing Addresses](#)

Body Field Subrecords

The following are examples of typical subrecords that are referenced from body fields:

- An assembly build record may need to include the serial numbers of each assembly being tracked. These details would be stored in an inventory detail subrecord referenced from a body field on the assembly build record.
- A subsidiary record may include a main address, a shipping address, and a return address. Each of these address subrecords is referenced by a field on the body of the record.
- The company information preferences page includes the preferred main address, shipping address, and return address for your organization. Each of these address subrecords is referenced by a field on the body of the record.

For details about working with these types of subrecords, see the following sections:

- [Using SuiteScript 2.x to Create a Body Field Subrecord](#)
- [Using SuiteScript 2.x to Update a Body Field Subrecord](#)
- [Using SuiteScript 2.x to Retrieve a Body Field Subrecord](#)

Transaction Shipping and Billing Addresses

Some transactions can have body fields that represent shipping and billing addresses. Each of these fields contains an address subrecord. The process of interacting with these subrecords can be different from working with other body field subrecords. For details, see [Scripting Transaction Shipping and Billing Addresses](#).

Using SuiteScript 2.x to Create a Body Field Subrecord

Some types of records have body fields that can reference subrecords. Scripting a subrecord that exists in a body field is slightly different from working with those that exist on sublists.

To create a subrecord, your script must use the [N/record Module](#). The script can use either dynamic or standard mode. For details on each approach, see the following sections:

- [Creating a Body Field Subrecord in Dynamic Mode](#)
- [Creating a Body Field Subrecord in Standard Mode](#)



Important: If you are scripting the shipping address or billing address on a transaction, see [Scripting Transaction Shipping and Billing Addresses](#).

Creating a Body Field Subrecord in Dynamic Mode

If your script uses dynamic mode, you can use the following procedure to create a subrecord in a body field.

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

To create a body field subrecord in dynamic mode:

1. If you want to add a new record, create it and set the required body fields. If you want to update an existing record, load the record.
2. Create the subrecord with the [Record.getSubrecord\(options\)](#). This method takes one argument: A fieldId, which identifies the field on the record that contains the subrecord. In the Records Browser, the field that holds the subrecord is always identified as a field of type summary.

For example, you could use an expression like the following to create an inventory detail subrecord on an assembly build record:

```

3 | ...
2 | var inventoryDetailSubrecord = rec.getSubrecord({
3 |   fieldId: 'inventorydetail'
4 | });
5 | ...

```

3. Set body fields on the subrecord with the [Record.setValue\(options\)](#). Be aware that not all subrecords have writable body fields.
4. If the subrecord has a sublist, generally you are required to add at least one line to the sublist. For each line, use the following guidelines:
 - Create the line with the [Record.selectNewLine\(options\)](#).
 - Set required values on the line with the [Record.setCurrentSublistValue\(options\)](#).
 - Save the subrecord's sublist line with the [Record.commitLine\(options\)](#).

For example, if you were creating an inventory detail subrecord, you could use the following expression to create a line on the subrecord's inventory assignment sublist:

```

1 | ...
2 | inventoryDetailSubrecord.selectNewLine({
3 |   sublistId: 'inventoryassignment',
4 | });
5 |
6 | inventoryDetailSubrecord.setCurrentSublistValue({
7 |   sublistId: 'inventoryassignment',
8 |   fieldId: 'receiptinventorynumber',
9 |   value: '012345'
10 | });
11 | inventoryDetailSubrecord.commitLine({

```

```

13     sublistId: 'inventoryassignment'
14   })
15 ...

```

5. Save the record.



Note: For full script example showing how to create a body field subrecord using dynamic mode, see [Creating an Address on a Subsidiary Record Example](#) and [Creating an Inventory Detail Subrecord on a Body Field Example](#).

Creating a Body Field Subrecord in Standard Mode

If your script uses standard mode, use the following procedure to create a subrecord on the body field of a record.

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

To create a body field subrecord in standard mode:

1. If you are adding a new record, create it and set the required body fields. If you are updating an existing record, load the record.
2. Create the subrecord with the [Record.getSubrecord\(options\)](#) method. This method takes one argument: A fieldId, which identifies the body field on the record that contains the subrecord. In the Records Browser, the field that holds the subrecord is always identified as a field of type summary.

For example, you could use an expression like the following to create an address subrecord on a location record:

```

1 ...
2 var addressSubrecord = rec.getSubrecord({
3   fieldId: 'mainaddress'
4 });
5 ...

```

3. Set body fields on the subrecord using the [Record.setValue\(options\)](#) method. Be aware that not all subrecords have writable body fields.
4. If the subrecord has a sublist, generally you are required to add at least one line to the sublist. For each line, use the following guidelines:
 - Create the line with the [Record.insertLine\(options\)](#) method.
 - Set required values on the line with the [Record.setSublistValue\(options\)](#) method.

For example, if you were creating an inventory detail subrecord, you could use the following expressions to create a line on the subrecord's inventory assignment sublist:

```

1 ...
2 inventoryDetailSubrecord.insertLine({
3   sublistId: 'inventoryassignment',
4   line: 0
5 });
6
7 inventoryDetailSubrecord.setSublistValue({
8   sublistId: 'inventoryassignment',
9   fieldId: 'receiptinventorynumber',
10  line: 0,
11  value: '12345'
12 });
13 ...

```

5. Save the record.

Creating an Address on a Subsidiary Record Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example shows how to create a subsidiary record that includes an address. In this case, the address data is contained in an address subrecord assigned to the mainaddress field.

To use this example, you must meet the following prerequisites:

- You must have a OneWorld account.
- The value you use for the subsidiary record's name field must be unique in your system.

This example uses dynamic mode, but you could also add the subrecord using standard mode. For general details about using either approach, see [Using SuiteScript 2.x to Create a Body Field Subrecord](#).

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType UserEventScript
4   */
5
6 define([ 'N/record' ],function(record) {
7     function afterSubmit(context) {
8       var rec = record.create({
9         type: record.Type.SUBSIDIARY,
10        isDynamic: true
11      });
12
13      // Set body fields on the record.
14      rec.setValue({
15        fieldId: 'name',
16        value: 'US Subsidiary'
17      });
18
19      rec.setValue({
20        fieldId: 'state',
21        value: 'CA'
22      });
23
24      // Create the address subrecord.
25      var subrec = rec.getSubrecord({
26        fieldId: 'mainaddress',
27      });
28
29      subrec.setValue({
30        fieldId: 'city',
31        value: 'San Mateo'
32      });
33
34      subrec.setValue({
35        fieldId: 'state',
36        value: 'CA'
37      });
38
39      subrec.setValue({
40        fieldId: 'zip',
41        value: '94403-2511'
42      });
43
44      subrec.setValue({
45        fieldId: 'addr1',
46        value: '2955 Campus Drive'

```

```

47     });
48
49     subrec.setValue({
50         fieldId: 'addr2',
51         value: 'Suite 100'
52     });
53
54     // Save the record.
55     try {
56         var recId = rec.save();
57         log.debug({
58             title: 'Record created successfully',
59             details: 'Id: ' + recId
60         });
61     } catch (e) {
62         log.error({
63             title: e.name,
64             details: e.message
65         });
66     }
67 }
68 return {
69     afterSubmit : afterSubmit
70 };
71 });

```

Creating an Inventory Detail Subrecord on a Body Field Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example shows how to create an assembly build record that includes an inventory detail subrecord. In this case, the subrecord is contained in a body field called inventory detail.

To use this example, you must meet the following prerequisites:

- The Advanced Bin / Numbered Inventory Management feature must be enabled at Setup > Company > Enable Features, on the Items & Inventory subtab.
- The item selected for the item body field should be a serialized assembly item.
- The receiptinventorynumber value must be unique in your system.

This example uses dynamic mode, but you could also add the subrecord using standard mode. For general details about using either approach to add a sublist subrecord, see [Using SuiteScript 2.x to Create a Body Field Subrecord](#).

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

```

1 require(['N/record'], function(record) {
2
3     // Create a lot numbered assembly item.
4     var item = record.create({
5         type: record.Type.LOT_NUMBERED_ASSEMBLY_ITEM,
6         isDynamic: true
7     });
8     item.setValue({
9         fieldId: 'itemid',
10        value: 'lot numbered item'
11    });
12    item.setValue({
13        fieldId: 'taxschedule',
14        value: 2
15    });
16    item.selectNewLine({
17        sublistId: 'member'

```

```

18    });
19    item.setCurrentSublistValue({
20      sublistId: 'member',
21      fieldId: 'item',
22      value: 233
23    });
24    item.commitLine({
25      sublistId: 'member'
26    });
27    var itemId = item.save();
28
29 // Create the assembly build record.
30 var rec = record.create({
31   type: record.Type.ASSEMBLY_BUILD,
32   isDynamic: true
33 });
34
35 // Set body fields.
36 rec.setValue({
37   fieldId: 'subsidiary',
38   value: '1'
39 });
40 rec.setValue({
41   fieldId: 'location',
42   value: '1'
43 });
44 rec.setValue({
45   fieldId: 'item',
46   value: itemId
47 });
48 rec.setValue({
49   fieldId: 'quantity',
50   value: 1
51 });
52
53 // Create the inventory detail subrecord.
54 var subrec = rec.getSubrecord({
55   fieldId: 'inventorydetail'
56 });
57
58 // Create a line on the subrecord's inventory assignment sublist.
59 subrec.selectNewLine({
60   sublistId: 'inventoryassignment',
61 });
62 subrec.setCurrentSublistValue({
63   sublistId: 'inventoryassignment',
64   fieldId: 'receiptinventorynumber',
65   value: '012345'
66 });
67 subrec.setCurrentSublistValue({
68   sublistId: 'inventoryassignment',
69   fieldId: 'quantity',
70   value: 1
71 });
72 subrec.commitLine({
73   sublistId: 'inventoryassignment'
74 });
75
76 // Save the record.
77 var recId = rec.save();
78 log.debug({
79   title: 'Record created successfully',
80   details: 'Id: ' + recId
81 });
82 });

```

Using SuiteScript 2.x to Update a Body Field Subrecord

If the business logic of the parent record permits it, your script can load an existing body field subrecord and make changes to it.

To edit a subrecord, your script must use the [N/record Module](#). The script can use either dynamic or standard mode.

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

To update a body field subrecord:

1. Load the record.
2. Retrieve the subrecord with the [Record.getSubrecord\(options\)](#) method. This method takes one argument: A fieldId, which identifies the field on the sublist that contains the subrecord. In the Records Browser, the field that holds the subrecord is always identified as a field of type summary.

For example, you could use an expression like the following to load the inventory detail subrecord associated with an assembly build record:

```

1 ...
2 var subrec = call.getSubrecord({
3   fieldId: 'inventorydetail'
4 });
5 ...

```

3. Update body fields on the subrecord using the [Record.setValue\(options\)](#) method. Be aware that not all subrecords have writable body fields.
4. If the subrecord has a sublist whose values you want to change, you can:
 - If your script uses dynamic mode, use the following steps for each value you want to change:
 1. Identify the line you want to change with the [Record.selectLine\(options\)](#) method.
 2. For each value you want to change, use the [Record.setCurrentSublistValue\(options\)](#) method to identify the field and the new value.
 3. Save your changes to the subrecord's sublist line with the [Record.commitLine\(options\)](#) method.
 - If your script uses standard mode, use the [Record.setSublistValue\(options\)](#) method to update each field that you want to change. This field takes four arguments:
 - A sublistId.
 - A fieldId, which identifies the field on the sublist that contains the subrecord.
 - A line number, which identifies the sublist line that contains the subrecord you want to change.
 - The new value.
5. Save the record with the [Record.save\(options\)](#) method.



Note: For full script example showing how to edit a body field subrecord using dynamic mode, see [Editing a Body Field Address Subrecord Example](#) and [Editing a Body Field Inventory Detail Subrecord Example](#).

Editing a Body Field Address Subrecord Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example loads a subsidiary record. It also loads and edits the address subrecord stored in the subsidiary's mainaddress body field.

To use this script, you must have a OneWorld account.

For general details on updating body field subrecords, see [Using SuiteScript 2.x to Update a Body Field Subrecord](#).

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType UserEventScript
4   */
5  define([ 'N/record' ],function(record) {
6      function afterSubmit(context) {
7          // Load the record.
8          var rec = record.load({
9              type: record.Type.SUBSIDIARY,
10             id: 1,
11             isDynamic: true
12         });
13
14         // Load the subrecord.
15
16         var subrec = rec.getSubrecord({
17             fieldId: 'mainaddress'
18         });
19
20         // Make changes to one field.
21         subrec.setValue({
22             fieldId: 'addr1',
23             value: '12331-A Riata Trace Parkway'
24         });
25
26         // Save the record.
27         try {
28             var recId = rec.save();
29             log.debug({
30                 title: 'Record updated successfully',
31                 details: 'Id: ' + recId
32             });
33
34         } catch (e) {
35             log.error({
36                 title: e.name,
37                 details: e.message
38             });
39         }
40     }
41     return {
42         afterSubmit : afterSubmit
43     };
44 });

```

Editing a Body Field Inventory Detail Subrecord Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

The following example loads an assembly build record. It also loads the subrecord stored in the assembly build's Inventory Detail body field and saves a change to the subrecord.

To use this example, you must meet the following prerequisites:

- The Advanced Bin / Numbered Inventory Management feature must be enabled at Setup > Company > Enable Features, on the Items & Inventory subtab.
- The assembly build record that you load must already have an inventory detail subrecord.
- The new value you choose for receiptinventorynumber must be unique in your system.

For general details on updating body field subrecords, see [Using SuiteScript 2.x to Update a Body Field Subrecord](#).

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType UserEventScript
4   */
5
6 define([ 'N/record' ],function(record) {
7     function afterSubmit(context) {
8         //Load the record.
9         var rec = record.load({
10             type: record.Type.ASSEMBLY_BUILD,
11             id: 4918,
12             isDynamic: true
13         });
14
15         // Load the subrecord.
16
17         var subrec = rec.getSubrecord({
18             fieldId: 'inventorydetail'
19         });
20
21         // Identify a line on the subrecord's sublist.
22
23         subrec.selectLine({
24             sublistId: 'inventoryassignment',
25             line: 0
26         });
27
28         // Make changes to one sublist field.
29
30         subrec.setCurrentSublistValue({
31             sublistId: 'inventoryassignment',
32             fieldId: 'receiptinventorynumber',
33             value: '890123'
34         });
35
36         // Save the record.
37         try {
38             var recId = rec.save();
39
40             log.debug({
41                 title: 'Record updated successfully',
42                 details: 'Id: ' + recId
43             });
44
45             } catch (e) {
46                 log.error({
47                     title: e.name,
48                     details: e.message
49                 });
50             }
51         }
52         return {
53             afterSubmit: afterSubmit
54         };
55     });
}

```

Using SuiteScript 2.x to Retrieve a Body Field Subrecord

In some cases, you may want to retrieve data from a subrecord that occurs in a body field.

To retrieve a subrecord, your script can use either the [N/record Module](#) or the [N/currentRecord Module](#). Your script can use either dynamic or standard mode.

To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)

To use SuiteScript 2.x to retrieve a body field subrecord:

1. Load the record.

2. Retrieve the subrecord with the `getSubrecord()` method. This method takes one argument: A `fieldId`, which identifies the field on the sublist that contains the subrecord. In the Records Browser, the field that holds the subrecord is always identified as a field of type summary.

For example, you could use an expression like the following to load an address subrecord stored in the body field of a location record:

```

1 ...
2 var subrec = call.getSubrecord({
3   fieldId: 'mainaddress'
4 });
5 ...

```

3. If you want to retrieve a value from the body of the subrecord, use the `getValue()` method. For example, you could use an expression like the following to retrieve a detail from the body of an address subrecord:

```

1 ...
2 var cityValue = subrec.getValue({
3   fieldId: 'city'
4 });
5 ...

```

4. If you want to retrieve a value from the subrecord's sublist, use the `getSublistValue()` method. For example, you could use an expression like the following to retrieve a detail from the sublist of an inventory detail subrecord:

```

1 ...
2 var cityValue = subrec.getValue({
3   sublistId: 'inventoryassignment',
4   fieldId: 'receiptinventorynumber',
5   line: 4,
6 });
7 ...

```

Retrieving a Body Field Address Subrecord Example



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Many record types have body fields that reference address subrecords. A similar design exists with the Company Information page, which is accessed at Setup > Company > Company Information. Technically, this page is not a record, but you can interact with it as you would interact with a record. And like a record, the Company Information page includes body fields that reference subrecords. Specifically, it includes a `mainaddress`, `shippingaddress`, and `returnaddress` subrecord. The following example shows how to retrieve values from each of these addresses.

Note also that this page includes body fields labeled **state** and **country**. These fields are set independently of the subrecords. In SuiteScript, you interact with them as you would any standard body field.

To get the most from this example, you should first populate the Company Information page with values in the following fields:

- State
- County
- Address
- Shipping address

- Return address

For general details on retrieving body field subrecords, see [Using SuiteScript 2.x to Update a Body Field Subrecord](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType UserEventScript
4 */
5 define(['N/config', 'N/record'], function(config, record) {
6     function afterSubmit(context) {
7         // The Company Information page is not a standard record. You access it by
8         // using the config.load method.
9
10        var companyInfo = config.load({
11            type: config.Type.COMPANY_INFORMATION
12        });
13
14        // Retrieve some of the preferences stored on the main part of the
15        // Company Information page. You interact with these preferences as if
16        // they were body fields.
17
18        var companyName = companyInfo.getValue({
19            fieldId: 'companyname'
20        });
21
22        var employerId = companyInfo.getValue({
23            fieldId: 'employerid'
24        });
25
26        // The Company Information page also includes some address fields on
27        // the body of the record. Retrieve these fields.
28        var state = companyInfo.getValue({
29            fieldId: 'state'
30        });
31
32        var country = companyInfo.getValue({
33            fieldId: 'country'
34        });
35
36        // Retrieve details from the subrecord that represents the main address.
37        // As a first step, instantiate the subrecord.
38        var mainAddress = companyInfo.getSubrecord ({
39            fieldId: 'mainaddress'
40        });
41
42        // Retrieve values from the main address subrecord.
43        var mainAddressCity = mainAddress.getValue({
44            fieldId: 'city'
45        });
46
47        var mainAddressState = mainAddress.getValue({
48            fieldId: 'state'
49        });
50
51        // Retrieve details from the subrecord that represents the shipping address.
52        // To start, instantiate the subrecord.
53        var shippingAddress = companyInfo.getSubrecord('shippingaddress');
54
55        // Retrieve values from the subrecord.
56        var shippingAddressCity = shippingAddress.getValue({
57            fieldId: 'city'
58        });
59
60        var shippingAddressState = shippingAddress.getValue({
61            fieldId: 'state'
62        });
63
64        // Retrieve details from the subrecord that represents the return address.
65        // To start, instantiate the shipping address subrecord.
66        var returnAddress = companyInfo.getSubrecord('returnaddress');
67

```

```

68 // Retrieve values from the subrecord.
69 var returnAddressCity = returnAddress.getValue({
70   fieldId: 'city'
71 });
72
73 var returnAddressState = returnAddress.getValue({
74   fieldId: 'state'
75 });
76
77 // Write selected details to the log.
78 log.debug ({
79   title: 'mainAddressState',
80   details: mainAddressState
81 });
82
83 log.debug ({
84   title: 'shippingAddressState',
85   details: shippingAddressState
86 });
87
88 log.debug ({
89   title: 'returnAddressState',
90   details: returnAddressState
91 });
92 }
93 return {
94   afterSubmit: afterSubmit
95 };
96 });

```

Scripting Transaction Shipping and Billing Addresses

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

A transaction's shipping address is stored in an address subrecord referenced from a body field on the transaction. A transaction's billing address is stored in the same way. The process of interacting with these subrecords is similar to how you work with other subrecords referenced by body fields. However, some differences exist. For details, see the following sections:

- [Addresses Can Be Sourced from Entity Records](#)
- [A Default Address May Be Used](#)
- [New Shipping and Billing Addresses Are Always Custom](#)

 **Note:** See also [About the Address Subrecord](#).

Addresses Can Be Sourced from Entity Records

Many transactions are associated with an entity. The entity record may already have addresses defined on its Address subtab. When working with a transaction, you can designate one of these existing addresses as the transaction's shipping or billing address.

If you choose to reference an existing address, your script does not have to create an address subrecord or use any subrecord methods. Instead, you can pick an address with a select field that is associated with the summary field. For example, to designate an existing address as the shipping address on a sales order, you use the shipaddresslist select field. This field identifies the subrecord being used by the transaction's shippingaddress summary field. For an example of how to select an existing address using the shipaddresslist select field, see [Using SuiteScript 2.x to Select an Existing Shipping Address Example](#).

A Default Address May Be Used

An entity can have a default shipping or billing address. For these types of entities, if you create a transaction and you don't set a value for shipping or billing address, the entity's default is used.

If the entity has a default shipping or billing address that you want to completely override, use `removeSubrecord()` to clear the summary field before you set the values for your new subrecord. As an alternative, you can set the `shipaddresslist` field to null before setting your new address values. For an example, see [Using SuiteScript 2.x to Create a New Shipping Address Example](#).

New Shipping and Billing Addresses Are Always Custom

In the UI, if you are creating a sales order and you want to enter a new shipping address, you must select either New or Custom in the Ship to Select field. If you select New, the address you enter is also saved to the customer record. If you select Custom, the new address is not saved to the customer record. It is saved only on the transaction.

By contrast, in SuiteScript, you cannot choose between New or Custom. Any new shipping or billing address created by your script is treated as a Custom address. The address is saved on the transaction, but it cannot be saved to the entity record.

Using SuiteScript 2.x to Create a New Shipping Address Example

The following example creates a sales order record with one line in the item sublist. It also creates a shipping address for the transaction. It overrides any default shipping address that might be defined on the customer record.



Note: For details about working with a transaction's shipping or billing address, see [Scripting Transaction Shipping and Billing Addresses](#).

To use this example, you must meet the following prerequisites:

- As already shown, when you create the new shipping address for the sales order, country field is set first (because the example uses dynamic mode – To learn about SuiteScript scripting modes, see [SuiteScript 2.x Standard and Dynamic Modes](#)). For more information, see [About the Address Subrecord](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType UserEventScript
4 */
5
6 define([ 'N/record' ],function(record) {
7     function afterSubmit(context){
8         // Create the record.
9         var rec = record.create({
10             type: record.Type.SALES_ORDER,
11             isDynamic: true
12         });
13
14         // Set body fields on the record.
15         rec.setValue({
16             fieldId: 'entity',
17             value: '2163'
18         });
19
20         rec.setValue({
21             fieldId: 'memo',
22             value: '102A'
23         });
24
25         // Create the subrecord.
26         var subrec = rec.getSubrecord({
27             fieldId: 'shippingaddress'
28         });
29

```

```

30 // Set values on the subrecord.
31 // Set country field first when script uses dynamic mode
32 subrec.setValue({
33   fieldId: 'country',
34   value: 'US'
35 });
36
37 subrec.setValue({
38   fieldId: 'city',
39   value: 'New York'
40 });
41
42 subrec.setValue({
43   fieldId: 'state',
44   value: 'New York'
45 });
46
47 subrec.setValue({
48   fieldId: 'zip',
49   value: '10018'
50 });
51
52 subrec.setValue({
53   fieldId: 'addr1',
54   value: '8 W 40th St.'
55 });
56
57 // Create a line in the item sublist.
58 rec.selectNewLine({
59   sublistId: 'item'
60 });
61
62 rec.setCurrentSublistValue({
63   sublistId: 'item',
64   fieldId: 'item',
65   value: '100'
66 });
67
68 rec.setCurrentSublistValue({
69   sublistId: 'item',
70   fieldId: 'quantity',
71   value: '11'
72 });
73
74 rec.commitLine({
75   sublistId: 'item'
76 });
77
78 // Save the record.
79 try {
80   var recId = rec.save();
81   log.debug({
82     title: 'Record created successfully',
83     details: 'Id: ' + recId
84   });
85 } catch (e) {
86   log.error({
87     title: e.name,
88     details: e.message
89   });
90 }
91
92 return {
93   afterSubmit: afterSubmit
94 };
95 });

```

Using SuiteScript 2.x to Select an Existing Shipping Address Example

The following script creates a sales order record with one in line in the item sublist. It also sets the value of the shipping address field to the value of an address from the customer record. Because the script

selects an existing address subrecord, it does not use subrecord methods. It identifies the address subrecord by internal ID.



Note: For details about working with a transaction's shipping or billing address, see [Scripting Transaction Shipping and Billing Addresses](#).

```

1  /**
2   * @NApiVersion 2.x
3   * @NScriptType UserEventScript
4   */
5
6 define([ 'N/record' ],function(record) {
7     function afterSubmit(context) {
8         var rec = record.create({
9             type: record.Type.SALES_ORDER,
10            isDynamic: true
11        });
12
13        // Set body fields on the record.
14        rec.setValue({
15            fieldId: 'entity',
16            value: '110'
17        });
18
19        // Set the shipping address to the value of
20        // an address on the customer record.
21        rec.setValue({
22            fieldId: 'shipaddresslist',
23            value: '395713'
24        });
25
26        // Create one line in the item sublist.
27        rec.selectNewLine({
28            sublistId: 'item'
29        });
30
31        rec.setCurrentSublistValue({
32            sublistId: 'item',
33            fieldId: 'item',
34            value: '100'
35        });
36
37        rec.setCurrentSublistValue({
38            sublistId: 'item',
39            fieldId: 'quantity',
40            value: '3'
41        });
42
43        rec.commitLine({
44            sublistId: 'item'
45        });
46
47        // Save the record.
48        try {
49            var recId = rec.save();
50            log.debug({
51                title: 'Record created successfully',
52                details: 'Id: ' + recId
53            });
54        } catch (e) {
55            log.error({
56                title: e.name,
57                details: e.message
58            });
59        }
60    }
61    return {
62        afterSubmit: afterSubmit
63    };
64 });

```

Using SuiteScript 2.x to Retrieve a Shipping Address Example

The following user event script gets all fields from the shipping address of a new transaction record.

To use this script, deploy it on a sales order or another transaction that has a shippingaddress summary field. After you create a new transaction that includes a shipping address, the address should be written to the execution log of your script deployment record.

```

1  /**
2  * @NApiVersion 2.x
3  * @NScriptType UserEventScript
4  */
5
6 define(['N/record'],function(record){
7     function beforeSubmit(context){
8         if(context.type==context.UserEventType.CREATE){
9             var salesOrder = context.newRecord;
10            var shipToAddress = salesOrder.getSubrecord({
11                fieldId: 'shippingaddress'
12            });
13
14            log.debug({
15                title: 'shipping address',
16                details: JSON.stringify(shipToAddress)
17            });
18        }
19    }
20    return {
21        beforeSubmit: beforeSubmit
22    };
23 });

```

Using SuiteScript 2.x to Retrieve one Value from a Shipping Address Example

The following script loads a sales order record. It retrieves a value from the shipping address and prints it to the execution log.

 **Note:** For details about working with a transaction's shipping or billing address, see [Scripting Transaction Shipping and Billing Addresses](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType ClientScript
4  */
5
6 define([ 'N/record' ],function(record){
7     function pageInit(){
8         //Load the record.
9         var rec = record.load({
10             type: record.Type.SALES_ORDER,
11             id: 5025,
12             isDynamic: true
13         });
14
15         // Retrieve the subrecord.
16         var subrec = rec.getSubrecord({
17             fieldId: 'shippingaddress'
18         });
19
20         // Create a variable and initialize it to the
21         // value of the subrecord's city field.
22         var cityValue = subrec.getValue({
23             fieldId: 'city'
24         });

```

```
25 // Print the value to the execution log.
26 try {
27     log.debug({
28         title: 'city value',
29         details: 'city value: ' + cityValue
30     });
31 } catch (e) {
32     log.error({
33         title: e.name,
34         details: e.message
35     });
36 }
37 return {
38     pageInit: pageInit
39 };
40 }
41 });
42});
```

SuiteScript 2.x Custom Pages

You can use SuiteScript 2.x methods to create custom forms and list pages. See the following help topics:

- [SuiteScript 2.x Custom Forms](#)
- [SuiteScript 2.x Custom List Pages](#)
- [SuiteScript 2.x Working with UI Objects](#)



Important: SuiteScript does not support direct access to the NetSuite UI through the Document Object Model (DOM). You should only access the NetSuite UI by using SuiteScript APIs.

SuiteScript 2.x Custom Forms

For information about using SuiteScript 2.0 to create custom forms, see:

- [Supported Script Types for Custom Form Creation](#)
- [Supported UI Components for Custom Forms](#)
- [Positioning Fields on Forms](#)
- [Sample Custom Form Script](#)

Supported Script Types for Custom Form Creation

You can use the following script types to create custom forms:

- **Suitelet:** Suitelets provide the most flexibility for creating a form. Suitelets can process requests and responses directly, giving you control over the data included in a form. For information about Suitelets, see [SuiteScript 2.x Suitelet Script Type](#).
- **Portlet:** Portlets are rendered within dashboard portlets. For information about portlet scripts, see [SuiteScript 2.x Portlet Script Type](#).
- **Before Load User Event Script:** User event scripts are executed on the server when a user performs certain actions on records. You can use a before load user event script to manipulate an existing form when a user edits a record. For information about before load user event scripts, see [SuiteScript 2.x User Event Script Type](#) and [beforeLoad\(scriptContext\)](#).

Supported UI Components for Custom Forms

You can use the following elements in your custom forms.

Buttons

You can use a button to trigger specific actions. The Submit button saves edits to a record. You also can attach a client script to a form and trigger it with a custom button.

- For information about adding a custom button to a form, see the help topic [Form.addButton\(options\)](#).
- For information about adding a submit button to a form, see the help topic [Form.addSubmitButton\(options\)](#).

Fields

Add a customized field to a form to collect specific information from a user. Use the [serverWidget.FieldType](#) enumeration to specify the field type. Most field types include basic error checking to ensure that a user correctly formats their data.

- For information about adding a custom field to a form, see the help topic [Form.addField\(options\)](#).

Configuring Fields

After you add a field to a form, you can customize the positioning of the fields on the form by placing them within a field group, or on a tab or subtab.

- For information about field positioning and layout, see [Positioning Fields on Forms](#).
- For information about placing a field on a tab or subtab, see [Steps for Adding a Tab to a Form](#).

You can define a variety of field configuration properties by setting enum values. The following are examples of commonly used field configuration properties.

- Use [serverWidget.FieldDisplayType](#) to indicate whether a field should be displayed and how it should appear. For example, you can disable fields, hide them, or make them read-only.
- Use [Field.isMandatory](#) to indicate whether a field is required.
- For other field configuration properties, see the help topic [serverWidget.Field](#).

Specialized Fields

The Secret Key field is a special field that can be used with the [N/crypto Module](#) to perform encryption or hashing.

- For information about using a Secret Key field, see the help topic [Form.addSecretKeyField\(options\)](#).

The Credential field is a special text field that you can use to store credentials, such as a password, used to invoke third-party services. Credential fields can be restricted to certain domains, scripts, and users. The data stored in a credential field is encrypted, not stored as clear text.

- For information about using a Credential field, see the help topic [Form.addCredentialField\(options\)](#).

Field Groups

Use field groups to organize and manage fields on a form. Some of the properties listed in [serverWidget.FieldGroup](#) can provide additional field group customization.

- For details, see the help topic [Form.addFieldGroup\(options\)](#).

Tabs and Subtabs

You can add tabs or subtabs to a form to organize large groups of fields so that a user can navigate the form more easily.

- For information about adding a tab to a form, see [Steps for Adding a Tab to a Form](#) and [Form.addTab\(options\)](#).

Sublists

A sublist is a list of child records that can be added to a parent record form.

- For information about adding a sublist to a form, see [Steps for Adding a Sublist to a Form](#), and [Form.addSublist\(options\)](#).

Page Links

A page link on a form can be either a breadcrumb link or a crosslink. Breadcrumb links display a series of links leading to the location of the current page. Crosslinks are used for navigation, including links such as Forward, Back, List, Search, and Customize.

- For more information about page link types, see the help topic [serverWidget.FormPageLinkType](#).
- For more information about adding page links to a form, see the help topic [Form.addPageLink\(options\)](#).

Positioning Fields on Forms

You can add field groups to position fields together with other closely related fields on custom forms. Use the properties in [serverWidget.FieldGroup](#) to add field groups to forms. You also can use [Field.updateLayoutType\(options\)](#) to define the positioning and placement of fields.

Note: Some field group properties are not supported for field groups on forms, such as properties for collapsing and hiding field groups.

The following screenshot shows a single column field group above a double column field group.

Field Layout Type

The [serverWidget.FieldLayoutType](#) enumerations contain values used by [Field.updateLayoutType\(options\)](#) to position fields outside of a field group and on the same row.

You can use the OUTSIDE value to position a field outside of a field group. You can further refine the field's position by using the OUTSIDEABOVE and OUTSIDE BELOW value to position a field above or below

a field group. The following screenshot displays three text fields positioned outside of a field group by setting these Field Layout Type values.

The following code sample illustrates how to set Field Layout values to produce the field placement in the screenshot.

```

1 //Add additional code
2 ...
3 var outsideAbove = form.addField({
4   id: 'outsideabovefield',
5   type: serverWidget.FieldType.TEXT,
6   label: 'Outside Above Field'
7 });
8 outsideAbove.updateLayoutType({
9   layoutType: serverWidget.FieldLayoutType.OUTSIDEABOVE
10 });
11 ...
12 //Add additional code

```

You can use the STARTROW, MIDROW, and ENDROW enumerations to position fields together in the same row. For example, you can group similar fields together, such as first name and last name. The following screenshot displays three text fields positioned together using these Field Layout Type values.

The following code sample illustrates how to set Field Layout values to produce the field placement in the screenshot.

```

1 //Add additional code
2 ...

```

```

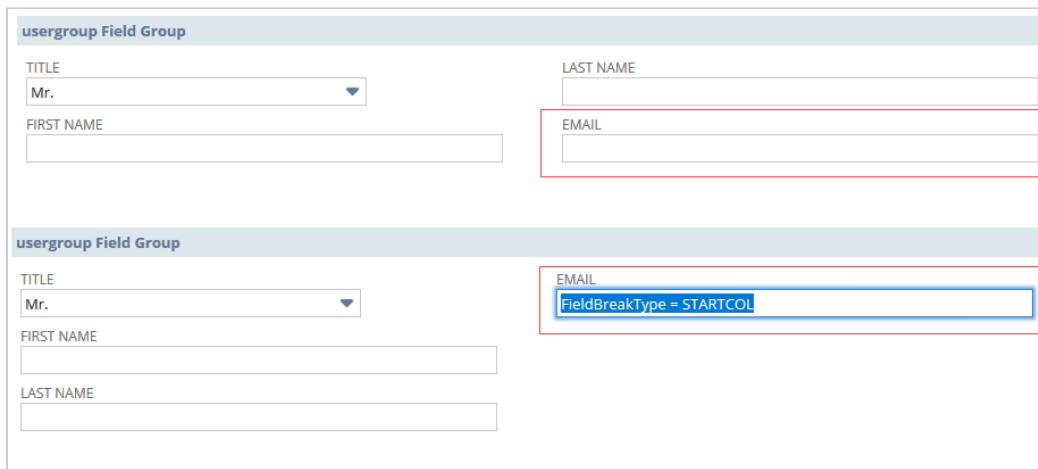
3  var startRow = form.addField({
4    id: 'startrow',
5    type: serverWidget.FieldType.TEXT,
6    label: 'STARTROW',
7    container: 'usergroup'
8  });
9
10 startRow.updateLayoutType({
11   layoutType: serverWidget.FieldLayoutType.STARTROW
12 });
13
14 var midRow = form.addField({
15   id: 'midrow',
16   type: serverWidget.FieldType.TEXT,
17   label: 'MIDROW',
18   container: 'usergroup'
19 });
20
21 midRow.updateLayoutType({
22   layoutType: serverWidget.FieldLayoutType.MIDROW
23 });
24
25 var endRow = form.addField({
26   id: 'endrow',
27   type: serverWidget.FieldType.TEXT,
28   label: 'ENDROW',
29   container: 'usergroup'
30 });
31
32 endRow.updateLayoutType({
33   layoutType: serverWidget.FieldLayoutType.ENDROW
34 });
35
36 ...
37 //Add additional code

```

Field Break Type

The [serverWidget.FieldBreakType](#) enumeration contains the values set by [Field.updateBreakType\(options\)](#) to define how fields are divided across columns and rows on forms.

- Use the STARTCOL value to move a field to a new column. The following screenshot shows the difference between the EMAIL field when the break type is set to NONE (top image) and when it is set to STARTCOL (bottom image).



The following code sample illustrates how to set the FieldBreakType values to produce the field placement in the screenshot.

```

1 //Add additional code
2 ...
3 var email = form.addField({
4   id: 'emailfield',
5   type: serverWidget.FieldType.EMAIL,
6   label: 'Email',
7   container: 'usergroup'
8 });
9
10 email.updateBreakType({
11   breakType : serverWidget.FieldBreakType.STARTCOL
12 });
13 ...
14 //Add additional code

```

Use the STARTROW value to move a field to a new row. The STARTROW value can only be used on fields that are positioned outside of a field group. The NONE value is the default configuration, and does not move a field to a new row. For information about how to position a field outside of a field group, see the help topics [serverWidget.FieldLayoutType](#) and [Field.updateLayoutType\(options\)](#).

The following screenshot shows the difference between a field with a break type set to STARTROW and NONE.

The following code sample illustrates how to set the FieldBreakType to STARTROW to produce the field placement in the screenshot.

```

1 //Add additional code
2 ...
3 var outsideAbove = form.addField({
4   id: 'outsideabovefield',
5   type: serverWidget.FieldType.TEXT,
6   label: 'OUTSIDEABOVE'
7 });
8
9 outsideAbove.updateLayoutType({
10   layoutType: serverWidget.FieldLayoutType.OUTSIDEABOVE
11 });
12
13 outsideAbove.updateBreakType({
14   layoutType: serverWidget.FieldLayoutType.STARTROW
15 });
16 ...
17 //Add additional code

```

Sample Custom Form Script



Note: The content in this topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

The following screenshot displays a form created by a Suitelet. You can use the sample code provided below to create this form.

The screenshot shows a custom form titled "Customer Information". It has three main sections: "User Information", "Company Information", and "Payment Information". The "User Information" section contains fields for Title (dropdown menu showing "Mr."), First Name, Last Name, and Email. The "Company Information" section contains fields for Company, Website, Phone Number, and Website2. The "Payment Information" section contains a dropdown for Payment Card 1 (set to "Payment Card 1") and a date selector for Expiry Date (set to Jan 2020). There is also a field for Credit Card Number. Below these sections is a "Purchase History" tab. At the bottom of the form are "Submit" and "Reset" buttons.

Steps for Creating a Custom Form with SuiteScript 2.x

Before proceeding, you must create a script file. To create this file, do one of the following:

- If you want to copy and paste the completed script directly from this help topic, go to the [Complete Custom Form Script Sample](#).
- If you want to walk through the steps for creating the script, complete the following procedure.

To create a custom form, complete the following steps:

1. Create a Suitelet and add the required JSDoc tags.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */

```

2. Add the define function to load the [N/ui/serverWidget Module](#) module.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define(['N/ui/serverWidget'],
6   function(serverWidget) {
7 });

```

3. To enable a Submit button, create an onRequest function. In the onRequest function create an if statement for the context request method and set it to 'GET'. After the if statement, create a return statement to support the GET request method.



Note: This sample is divided into four sections, as noted in the code comments. Each section identifies where to include different parts of the sample code. Section two of the sample code is referenced in [Steps for Adding a Tab to a Form](#). Section three of the sample code is referenced in [Steps for Adding a Sublist to a Form](#) sections.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define(['N/ui/serverWidget'],
6     function(serverWidget) {
7         function onRequest(context){
8             if(context.request.method === 'GET'){
9                 context.response.writePage(form);
10                // Section One - Forms - See "Steps for Creating a Form"
11                // Section Two - Tabs - See "Steps for Adding a Tab to a Form"
12                // Section Three - Sublist - See "Steps for Adding a Sublist to a Form"
13            }else{
14                // Section Four - Output - Used in all sections
15            }
16        }
17        return {
18        }
19    };
20 });

```

- In section one, call the `createForm` method using `serverWidget.createForm(options)`.

```

1 var form = serverWidget.createForm({
2     title: 'Enter Customer Information'
3 });

```

- In section one, create Field Groups to organize the fields using `Form.addFieldGroup(options)`. Use the properties defined in `serverWidget.FieldGroup` to control the appearance and behavior of the field groups.

```

1 var usergroup = form.addFieldGroup({
2     id : 'usergroup',
3     label : 'User Information'
4 });
5 usergroup.isSingleColumn = true;
6
7 var companygroup = form.addFieldGroup({
8     id : 'companygroup',
9     label : 'Company Information'
10 });

```

- In section one, use `Form.addButton(options)` to add the Submit button to a form. Change the text displayed on the button by modifying the `label` property.

```

1 form.addButton({
2     label: 'Submit'
3 });

```

- In section one, use `Form.addField(options)` to create the fields you need. Make sure to define the `id`, `type`, and `label` properties correctly. Use the `container` property to specify the field group.

Use the properties defined in `serverWidget.Field` to change the behavior and properties of the fields. This sample uses `Field.isMandatory` to define several required fields, and `Field.defaultValue` to set the default value of the Company name field.

Some fields, such as the email and phone fields, provide basic error checking to make sure the field contents meet the requirements for the field. For more information, see the help topic [serverWidget.FieldType](#).



Note: If you are modifying an existing form or adding custom fields, make sure you add the custpage_ prefix to the identifier to prevent conflicts with existing standard fields.

```

1 var select = form.addField({
2   id: 'titlefield',
3   type: serverWidget.FieldType.SELECT,
4   label: 'Title',
5   container: 'usergroup'
6 });
7
8 select.addSelectOption({
9   value: 'Mr.',
10  text: 'Mr.'
11 });
12
13 select.addSelectOption({
14   value: 'MS.',
15  text: 'Ms.'
16 });
17
18 select.addSelectOption({
19   value: 'Dr.',
20  text: 'Dr.'
21 });
22
23 var fname = form.addField({
24   id: 'fnamefield',
25   type: serverWidget.FieldType.TEXT,
26   label: 'First Name',
27   container: 'usergroup'
28 });
29 fname.isMandatory = true;
30
31 var lname = form.addField({
32   id: 'lnamefield',
33   type: serverWidget.FieldType.TEXT,
34   label: 'Last Name',
35   container: 'usergroup'
36 });
37 lname.isMandatory = true;
38
39 form.addField({
40   id: 'emailfield',
41   type: serverWidget.FieldType.EMAIL,
42   label: 'Email',
43   container: 'usergroup'
44 });
45
46 var companyname = form.addField({
47   id: 'companyfield',
48   type: serverWidget.FieldType.TEXT,
49   label: 'Company',
50   container: 'companygroup'
51 });
52 companyname.defaultValue = 'Company Name';
53
54 form.addField({
55   id: 'phonefield',
56   type: serverWidget.FieldType.PHONE,
57   label: 'Phone Number',
58   container: 'companygroup'
59 });
60
61 form.addField({
62   id: 'urlfield',

```

```

63 |     type: serverWidget.FieldType.URL,
64 |     label: 'Website',
65 |     container: 'companygroup'
66 | });

```

8. In section four, define a variable for each field in this sample.

```

1 | var delimiter = /\u0001/;
2 | var titleField = context.request.parameters.titlefield;
3 | var fnameField = context.request.parameters.fnamefield;
4 | var lnameField = context.request.parameters.lnamefield;
5 | var emailField = context.request.parameters.emailfield;
6 | var companyField = context.request.parameters.companyfield;
7 | var phoneField = context.request.parameters.phonefield;
8 | var urlField = context.request.parameters.urlfield;

```

9. In section four, create an output statement using `context.response.write`. This statement demonstrates the Submit button by generating an output displaying the variables created from the field data.

```

1 | context.response.write('You have entered:'
2 |   + '<br/> Name: ' + titleField + ' ' + fnameField
3 |   + '<br/> Email: ' + emailField
4 |   + '<br/> Company: ' + companyField
5 |   + '<br/> Phone: ' + phoneField + ' Website: ' + urlField);

```

Steps for Adding a Tab to a Form

This sample uses the code created in [Sample Custom Form Script](#) to show how to add tab and subtab UI components to a form.

When you add a tab to a form, consider the following limitations:

- You cannot add a single tab. If you create a single tab, the contents of the tab appear on the form without the tab bar.
- A tab does not appear until you have assigned an object to the form. After you have added a field, you can set the `container` property to indicate where the field appears on the tab.

1. In section two, use [Form.addTab\(options\)](#) to add two tabs to the form. You can use [Tab.helpText](#) to add inline help text to guide users.



Note: A value for `tab.id` must be all lowercase and contain no spaces. If you are adding a tab to an existing page, include the prefix `custpage` to prevent conflicts with existing standard tabs.

```

1 //Section Two - Tabs
2
3 var tab1 = form.addTab({
4   id : 'tab1id',
5   label : 'Payment'
6 });
7 tab1.helpText = 'Help Text Goes Here';
8
9 var tab2 = form.addTab({
10   id : 'tab2id',
11   label : 'Inventory'
12 });

```

2. In section two, use [Form.addSubtab\(options\)](#) to add two subtabs. Set the `tab` property to `tab1id` to position the subtab on the first tab you created.

```

1 | form.addSubtab({

```

```

2   id : 'subtab1id',
3   label : 'Payment Information',
4   tab: 'tab1id'
5 });
6
7 form.addSubtab({
8   id : 'subtab2id',
9   label : 'Transaction Record',
10  tab: 'tab1id'
11 });

```

- In section two, add more fields to the form. To assign a field to a tab, use the container property to indicate the tab where you want to place the field.

The following sample adds a Credential field to the form. Credential fields are text fields used to store credentials in NetSuite. Notice that when you click the Submit button in this sample, the credit card number is encrypted and is not displayed as clear text. For information about using Credential fields, see the help topic [Form.addCredentialField\(options\)](#).

```

1 //Subtab One Fields
2 var ccselect = form.addField({
3   id: 'cctypefield',
4   type: serverWidget.FieldType.SELECT,
5   label: 'Credit Card',
6   container: 'subtab1id'
7 });
8
9 ccselect.addSelectOption({
10   value: 'PayCard0',
11   text: 'Payment Card 0'
12 });
13
14 ccselect.addSelectOption({
15   value: 'PayCard1',
16   text: 'Payment Card 1'
17 });
18
19 ccselect.addSelectOption({
20   value: 'PayCard2',
21   text: 'Payment Card 2'
22 });
23
24
25 var expmonth = form.addField({
26   id: 'expmonth',
27   type: serverWidget.FieldType.SELECT,
28   label: 'Expiry Month',
29   container: 'subtab1id'
30 });
31
32 expmonth.updateLayoutType({
33   layoutType: serverWidget.FieldLayoutType.STARTROW});
34
35 expmonth.addSelectOption({
36   value: '01',
37   text: 'Jan'
38 });
39
40 expmonth.addSelectOption({
41   value: '02',
42   text: 'Feb'
43 });
44
45 expmonth.addSelectOption({
46   value: '03',
47   text: 'Mar'
48 });
49
50 expmonth.addSelectOption({
51   value: '04',
52   text: 'Apr'
53 });

```

```

53 });
54
55 expmonth.addSelectOption({
56   value: '05',
57   text: 'May'
58 });
59
60 expmonth.addSelectOption({
61   value: '06',
62   text: 'Jun'
63 });
64
65 expmonth.addSelectOption({
66   value: '07',
67   text: 'Jul'
68 });
69
70 expmonth.addSelectOption({
71   value: '08',
72   text: 'Aug'
73 });
74
75 expmonth.addSelectOption({
76   value: '09',
77   text: 'Sep'
78 });
79
80 expmonth.addSelectOption({
81   value: '10',
82   text: 'Oct'
83 });
84
85 expmonth.addSelectOption({
86   value: '11',
87   text: 'Nov'
88 });
89
90 expmonth.addSelectOption({
91   value: '12',
92   text: 'Dec'
93 });
94
95 var expyear = form.addField({
96   id: 'expyear',
97   type: serverWidget.FieldType.SELECT,
98   label: 'Expiry Year',
99   container: 'subtabId'
100 });
101
102 expyear.updateLayoutType({
103   layoutType: serverWidget.FieldLayoutType.ENDROW});
104
105 expyear.addSelectOption({
106   value: '2020',
107   text: '2020'
108 });
109
110 expyear.addSelectOption({
111   value: '2019',
112   text: '2019'
113 });
114
115 expyear.addSelectOption({
116   value: '2018',
117   text: '2018'
118 });
119
120 var credfield = form.addCredentialField({
121   id : 'credfield',
122   label : ' Credit Card Number',
123   restrictToDomains : 'www.mysite.com',
124   restrictToScriptIds : 'customscript_my_script',
125   restrictToCurrentUser : false,

```

```

126     container : 'subtab1id'
127   });
128   credfield.maxLength = 32;
129
130 //Subtab Two fields
131 form.addField({
132   id: 'transactionfield'
133   type: serverWidget.FieldType.LABEL,
134   label: 'Transaction History - Coming Soon',
135   container: 'subtab2id'
136 });
137
138 //Tab Two fields
139 form.addField({
140   id: 'inventoryfield',
141   type: serverWidget.FieldType.LABEL,
142   label: 'Inventory - Coming Soon',
143   container: 'tab2id'
144 });

```

- In section four, define a variable for each field you have added to the tabs. In this sample, these variables are used to demonstrate the function of the Submit button.

```

1 var ccField = context.request.parameters.cctypefield;
2 var ccNumber = context.request.parameters.credfield;
3 var expMonth = context.request.parameters.expmonth;
4 var expYear = context.request.parameters.expyear;

```

- In section four, update the output statement to display the variables created when the Submit button is clicked.

```

1 context.response.write('You have entered:'
2   + '<br/> Name: ' + titleField + ' ' + fnameField + ' ' + lnameField
3   + '<br/> Email: ' + emailField
4   + '<br/> Company: ' + companyField
5   + '<br/> Phone: ' + phoneField + ' Website: ' + urlField
6   + '<br/> Credit Card: ' + ccField
7   + '<br/> Number: ' + ccNumber
8   + '<br/> Expiry Date: ' + expMonth + '/' + expYear);

```

Steps for Adding a Sublist to a Form

This sample uses the code created in [Steps for Adding a Tab to a Form](#) to show how to add a sublist to a form.

- In section three, use [Form.addSublist\(options\)](#) to add a sublist to the form. Set the tab property to tab2id to position the sublist on the second tab. This sublist is an inline editor type sublist. For information about sublist types, see the help topic [serverWidget.SublistType](#).

```

1 var sublist = form.addSublist({
2   id : 'sublistid',
3   type : serverWidget.SublistType.INLINEEDITOR,
4   label : 'Inventory',
5   tab: 'tab2id'
6 });

```

- In section three, use [Sublist.addButton\(options\)](#) to add a button to the sublist. In this sample the button is not functional, but you can use the functionName property to specify the function triggered when the button is clicked.

```

1 sublist.addButton({
2   id : 'buttonId',
3   label : 'Print ',
4   functionName: '' //Add the function triggered on button click

```

```
5 | }));
```

3. In section three, use `Sublist.addField(options)` to add fields to the sublist. For more information about supported field types, see the help topic [serverWidget.FieldType](#).



Note: `Sublist.addField(options)` does not support the inline HTML field type.

```

1  sublist.addField({
2    id : 'datefieldid',
3    type : serverWidget.FieldType.DATE,
4    label : 'Date'
5  });
6
7  sublist.addField({
8    id : 'productfieldid',
9    type : serverWidget.FieldType.TEXT,
10   label : 'Product'
11 });
12
13 sublist.addField({
14   id : 'qtyfieldid',
15   type : serverWidget.FieldType.INTEGER,
16   label : 'Quantity'
17 });
18
19 sublist.addField({
20   id : 'upfieldid',
21   type : serverWidget.FieldType.CURRENCY,
22   label : 'Unit Cost'
23 });

```

Complete Custom Form Script Sample



Note: This script sample uses the `define` function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the `require` function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript Debugger](#).

The following code creates a Suitelet that generates a custom form containing several field types, tabs, a sublist, and a submit button. For steps to create this script, see [Sample Custom Form Script](#).

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define(['N/ui/serverWidget'], function(serverWidget) {
6   function onRequest(context) {
7     if (context.request.method === 'GET') {
8
9       //Section One - Forms - See 'Steps for Creating a Custom Form' in topic 'Sample Custom Form Script'
10      var form = serverWidget.createForm({
11        title: 'Customer Information'
12      });
13
14      var usergroup = form.addFieldGroup({
15        id: 'usergroup',
16        label: 'User Information'
17      });
18      usergroup.isSingleColumn = true;
19
20      var companygroup = form.addFieldGroup({
21        id: 'companygroup',
22        label: 'Company Information'
23      });
24
25      var select = form.addField({

```

```

26     id: 'titlefield',
27     type: serverWidget.FieldType.SELECT,
28     label: 'Title',
29     container: 'usergroup'
30   });
31   select.addSelectOption({
32     value: 'Mr.',
33     text: 'Mr.'
34   });
35   select.addSelectOption({
36     value: 'MS.',
37     text: 'Ms.'
38   });
39   select.addSelectOption({
40     value: 'Dr.',
41     text: 'Dr.'
42   });
43
44   var fname = form.addField({
45     id: 'fnamefield',
46     type: serverWidget.FieldType.TEXT,
47     label: 'First Name',
48     container: 'usergroup'
49   });
50   fname.isMandatory = true;
51
52   var lname = form.addField({
53     id: 'lnamefield',
54     type: serverWidget.FieldType.TEXT,
55     label: 'Last Name',
56     container: 'usergroup'
57   });
58   lname.isMandatory = true;
59
60   form.addField({
61     id: 'emailfield',
62     type: serverWidget.FieldType.EMAIL,
63     label: 'Email',
64     container: 'usergroup'
65   });
66
67   var companyname = form.addField({
68     id: 'companyfield',
69     type: serverWidget.FieldType.TEXT,
70     label: 'Company',
71     container: 'companygroup'
72   });
73   companyname.defaultValue = 'Company Name';
74
75   form.addField({
76     id: 'phonefield',
77     type: serverWidget.FieldType.PHONE,
78     label: 'Phone Number',
79     container: 'companygroup'
80   });
81
82   form.addField({
83     id: 'urlfield',
84     type: serverWidget.FieldType.URL,
85     label: 'Website',
86     container: 'companygroup'
87   });
88
89   form.addSubmitButton({
90     label: 'Submit'
91   });
92
93 // Section Two - Tabs - See 'Steps for Adding a Tab to a Form' in topic 'Sample Custom Form Script'
94 var tab1 = form.addTab({
95   id: 'tab1id',
96   label: 'Payment'
97 });
98 tab1.helpText = 'Help Text Goes Here';

```

```

99
100    var tab2 = form.addTab({
101        id: 'tab2id',
102        label: 'Inventory'
103    });
104
105    form.addSubtab({
106        id: 'subtab2id',
107        label: 'Payment Information',
108        tab: 'tab2id'
109    });
110
111    form.addSubtab({
112        id: 'subtab2id',
113        label: 'Transaction Record',
114        tab: 'tab2id'
115    });
116
117    //Subtab One Fields
118    var ccselect = form.addField({
119        id: 'cctypefield',
120        type: serverWidget.FieldType.SELECT,
121        label: 'Credit Card',
122        container: 'subtab1id'
123    });
124    ccselect.addSelectOption({
125        value: 'PayCard0',
126        text: 'Payment Card 0'
127    });
128    ccselect.addSelectOption({
129        value: 'PayCard1',
130        text: 'Payment Card 1'
131    });
132    ccselect.addSelectOption({
133        value: 'PayCard2',
134        text: 'Payment Card 2'
135    });
136
137    var expmonth = form.addField({
138        id: 'expmonth',
139        type: serverWidget.FieldType.SELECT,
140        label: 'Expiry Date:',
141        container: 'subtab1id'
142    });
143    expmonth.updateLayoutType({
144        layoutType: serverWidget.FieldLayoutType.STARTROW
145    });
146    expmonth.addSelectOption({
147        value: '01',
148        text: 'Jan'
149    });
150    expmonth.addSelectOption({
151        value: '02',
152        text: 'Feb'
153    });
154    expmonth.addSelectOption({
155        value: '03',
156        text: 'Mar'
157    });
158    expmonth.addSelectOption({
159        value: '04',
160        text: 'Apr'
161    });
162    expmonth.addSelectOption({
163        value: '05',
164        text: 'May'
165    });
166    expmonth.addSelectOption({
167        value: '06',
168        text: 'Jun'
169    });
170    expmonth.addSelectOption({
171        value: '07',
172    });

```

```

172         text: 'Jul'
173     });
174     expmonth.addSelectOption({
175         value: '08',
176         text: 'Aug'
177     });
178     expmonth.addSelectOption({
179         value: '09',
180         text: 'Sep'
181     });
182     expmonth.addSelectOption({
183         value: '10',
184         text: 'Oct'
185     });
186     expmonth.addSelectOption({
187         value: '11',
188         text: 'Nov'
189     });
190     expmonth.addSelectOption({
191         value: '12',
192         text: 'Dec'
193     });
194
195     var expyear = form.addField({
196         id: 'expyear',
197         type: serverWidget.FieldType.SELECT,
198         label: 'Expiry Year',
199         container: 'subtab1id'
200     });
201     expyear.updateLayoutType({
202         layoutType: serverWidget.FieldLayoutType.ENDROW
203     });
204     expyear.addSelectOption({
205         value: '2020',
206         text: '2020'
207     });
208     expyear.addSelectOption({
209         value: '2019',
210         text: '2019'
211     });
212     expyear.addSelectOption({
213         value: '2018',
214         text: '2018'
215     });
216
217     var credfield = form.addCredentialField({
218         id: 'credfield',
219         label: ' Credit Card Number',
220         restrictToDomains: 'www.mysite.com',
221         restrictToScriptIds: 'customscript_my_script',
222         restrictToCurrentUser: false,
223         container: 'subtab1id'
224     });
225     credfield.maxLength = 32;
226
227 // Subtab two Fields
228 form.addField({
229     id: 'transactionfield',
230     type: serverWidget.FieldType.LABEL,
231     label: 'Transaction History - Coming Soon',
232     container: 'subtab2id'
233 });
234
235 // Tab Two Fields
236 form.addField({
237     id: 'inventoryfield',
238     type: serverWidget.FieldType.LABEL,
239     label: 'Inventory - Coming Soon',
240     container: 'tab2id'
241 });
242
243 // Section Three - Sublist - See 'Steps for Adding a Sublist to a Form' in topic 'Sample Custom Form Script'
244 var sublist = form.addSublist({

```

```

245     id: 'sublistid',
246     type: serverWidget.SublistType.INLINEEDITOR,
247     label: 'Inline Sublist',
248     tab: 'tab2id'
249   });
250   sublist.addButton({
251     id: 'buttonId',
252     label: 'Print ',
253     functionName: '' // Add the function triggered on button click
254   });

255   // Sublist Fields
256   sublist.addField({
257     id: 'datefieldid',
258     type: serverWidget.FieldType.DATE,
259     label: 'Date'
260   });

261   sublist.addField({
262     id: 'productfieldid',
263     type: serverWidget.FieldType.TEXT,
264     label: 'Product'
265   });

266   sublist.addField({
267     id: 'qtyfieldid',
268     type: serverWidget.FieldType.INTEGER,
269     label: 'Quantity'
270   });

271   sublist.addField({
272     id: 'upfieldid',
273     type: serverWidget.FieldType.CURRENCY,
274     label: 'Unit Cost'
275   });

276   context.response.writePage(form);
277 } else {
278   // Section Four - Output - Used in all sections
279   var delimiter = /\u0001/;
280   var titleField = context.request.parameters.titlefield;
281   var fnameField = context.request.parameters.fnamefield;
282   var lnameField = context.request.parameters.lnamefield;
283   var emailField = context.request.parameters.emailfield;
284   var companyField = context.request.parameters.companyfield;
285   var phoneField = context.request.parameters.phonefield;
286   var urlField = context.request.parameters.urlfield;
287   var ccField = context.request.parameters.cctypefield;
288   var ccNumber = context.request.parameters.credfield;
289   var expMonth = context.request.parameters.expmonth;
290   var expYear = context.request.parameters.expyear;

291   context.response.write('You have entered:'
292     + '<br/> Name: ' + titleField + ' ' + fnameField + ' ' + lnameField
293     + '<br/> Email: ' + emailField
294     + '<br/> Company: ' + companyField
295     + '<br/> Phone: ' + phoneField + ' Website: ' + urlField
296     + '<br/> Credit Card: ' + ccField
297     + '<br/> Number: ' + ccNumber
298     + '<br/> Expiry Date: ' + expMonth + '/' + expYear);
299   }
300 }
301 return {
302   onRequest: onRequest
303 };
304 });
305 });
306 });
307 });
308 });
309 });
310 });

```

SuiteScript 2.x Custom List Pages

For information about using SuiteScript 2.0 to create custom list pages, see:

- [Supported Script Types for Custom List Page Creation](#)
- [Supported UI Components for Custom List Pages](#)
- [Sample Custom List Page Script](#)

Supported Script Types for Custom List Page Creation

You can use the following script types to create custom list pages:

- **Suitelet:** Suitelets provide the most flexibility for creating a list. Suitelets can process requests and responses directly, giving you control over the data included in a list. For information about Suitelets, see [SuiteScript 2.x Suitelet Script Type](#).
- **Portlet:** Portlets are rendered within dashboard portlets. For information about portlet scripts, see [SuiteScript 2.x Portlet Script Type](#).

Supported UI Components for Custom List Pages

You can use any of the following components on your custom list page.

Buttons

Add a button to a list page to trigger custom functions.

- For information about adding a button, see the help topic [List.addButton\(options\)](#).

Columns

A column contains an editable or read-only field that displays a record value. Column properties are defined using the supported field types specified in [serverWidget.FieldType](#). An Edit column is a special column that adds links to edit and view each record or row in the list.

- For information about adding columns to a list page, see the help topic [List.addColumn\(options\)](#).
- For information about adding an edit column, see the help topic [List.addEditColumn\(options\)](#).



Note: Use a URL column to add a website to each row in a list. Use [ListColumn.setURL\(options\)](#) to specify the base URL for the website, and [ListColumn.addParamToURL\(options\)](#) to assign additional properties to a URL column.

Page Link

Page links on list pages can be either a breadcrumb link or a crosslink. Breadcrumb links display a series of links leading to the location of the current page. Crosslinks are used for navigation, including links such as Forward, Back, List, Search, and Customize.

- For information about page link types, see the help topic [serverWidget.FormPageLinkType](#).
- For information about adding page links, see the help topic [List.addPageLink\(options\)](#).

Rows

Each row in a list represents a single record. You can add a single row or multiple rows to a custom list page by specifying the column ID and value for each row.

- For more information on adding a single row, see the help topic [List.addRow\(options\)](#).
- For more information on adding multiple rows, see the help topic [List.addRows\(options\)](#).

Sample

```

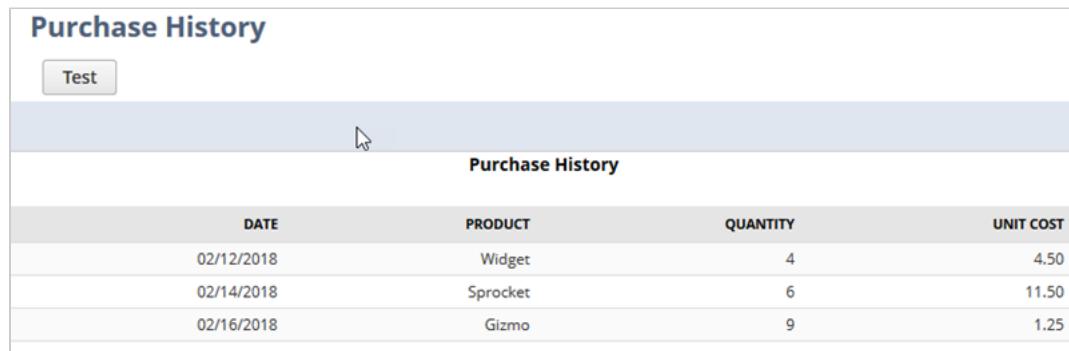
1 /**
2  * @NScriptType Suitelet
3  * @NApiVersion 2.x
4 */
5 define([ 'N/ui/serverWidget'], function(serverWidget) {
6     return {
7         onRequest : function(context) {
8             var list = serverWidget.createList({
9                 title:"List"
10            });
11            list.addColumn({
12                id: 'column1',
13                type: serverWidget.FieldType.TEXT,
14                label: 'Text',
15                align: serverWidget.LayoutJustification.LEFT
16            });
17            context.response.writePage(list);
18        }
19    });

```

Sample Custom List Page Script

 **Note:** The content in this topic pertains to SuiteScript 2.0.

The following screenshot displays a list page created by a Suitelet. You can use the sample code provided below to create this list yourself.



The screenshot shows a Suitelet interface with a header bar containing a 'Test' button. Below the header is a title 'Purchase History'. A table follows, with columns labeled 'DATE', 'PRODUCT', 'QUANTITY', and 'UNIT COST'. The table contains three rows of data:

| DATE | PRODUCT | QUANTITY | UNIT COST |
|------------|----------|----------|-----------|
| 02/12/2018 | Widget | 4 | 4.50 |
| 02/14/2018 | Sprocket | 6 | 11.50 |
| 02/16/2018 | Gizmo | 9 | 1.25 |

Steps for Creating a Custom List Page with SuiteScript 2.x

Before proceeding, you must create a script file. To create this file, you can do one of the following:

- If you want to copy and paste the completed script directly from this help topic, go to [Complete Custom List Page Sample Script](#).

- If you want walk through the steps for creating the script, complete the following procedure.

To create a list page, complete the following steps:

- Create a Suitelet, and add the required JSDoc tags.

```
1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
```

- Add the define function to load the [N/ui/serverWidget Module](#) module.

```
1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define(['N/ui/serverWidget'],
6         function(serverWidget) {
7});
```

- To enable a Submit button, create an onRequest function. In the onRequest function create an if statement for the context request method, and set it to 'GET'. After the if statement, create a return statement to support the GET request method.



Note: This sample is divided into two sections, as noted in the code comments. Each section identifies where to include different parts of the sample code.

```
1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define(['N/ui/serverWidget'],
6         function(serverWidget) {
7             function onRequest(context){
8                 if(context.request.method === 'GET'){
9                     // Section One - List - See 'Steps for Creating a List', Step Five
10                    // Section Two - Columns - See 'Steps for Creating a List', Step Seven
11                     context.response.writePage(list);
12                 }else{
13                 }
14             }
15             return {
16                 onRequest: onRequest
17             }
18         });
19});
```

- In section one, create a list page using [serverWidget.createList\(options\)](#). You can use [serverWidget.ListStyle](#) to define the style of the list page.

```
1 var list = serverWidget.createList({
2     title: 'Purchase History'
3 });
4 list.style = serverWidget.ListStyle.REPORT;
```

- In section one, below the declaration of the list, use [List.addButton\(options\)](#) to create a button on the list. Use the functionName property to call the function triggered when the button is clicked.

```
1 list.addButton({
2     id : 'buttonid',
3     label : 'Test',
4     functionName: '' //the function called when the button is pressed
5});
```

6. In section two, use [List.addColumn\(options\)](#) to define the columns in the list. The column type is specified according to the [serverWidget.FieldType](#) enumeration.

Use the align property to justify the position of the column.



Important: Not all field types are supported by [List.addColumn\(options\)](#), see the help topic [serverWidget.FieldType](#).

```

1 var datecol = list.addColumn({
2   id : 'column1',
3   type : serverWidget.FieldType.DATE,
4   label : 'Date',
5   align : serverWidget.LayoutJustification.RIGHT
6 });
7
8   list.addColumn({
9     id : 'column2',
10    type : serverWidget.FieldType.TEXT,
11    label : 'Product',
12    align : serverWidget.LayoutJustification.RIGHT
13 });
14
15   list.addColumn({
16     id : 'column3',
17     type : serverWidget.FieldType.INTEGER,
18     label : 'Quantity',
19     align : serverWidget.LayoutJustification.RIGHT
20 });
21
22   list.addColumn({
23     id : 'column4',
24     type : serverWidget.FieldType.CURRENCY,
25     label : 'Unit Cost',
26     align : serverWidget.LayoutJustification.RIGHT
27 });

```

7. In section two, use [List.addRow\(options\)](#) to define a single row on the list. Or use [List.addRows\(options\)](#) to define multiple rows at once.

```

1 list.addRow({
2   row : { column1 : '02/12/2018', column2 : 'Widget', column3: '4', column4:'4.50' }
3 });

```

```

1 list.addRows({
2   rows :[{column1 : '02/12/2018', column2 : 'Widget', column3: '4', column4:'4.50'},
3     {column1 : '02/14/2018', column2 : 'Sprocket', column3: '6', column4:'11.50'},
4     {column1 : '02/16/2018', column2 : 'Gizmo', column3: '9', column4:'1.25'}]
5 });

```

Complete Custom List Page Sample Script



Note: This script sample uses the define function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the require function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript Debugger](#).

The following code creates a Suitelet that generates a custom list page. For steps to create this script, see [SuiteScript 2.x Custom List Pages](#).

```

1 /**
2  * @NApiVersion 2.x

```

```

3  * @NScriptType Suitelet
4  */
5 define(['N/ui/serverWidget'], function(serverWidget) {
6     function onRequest(context){
7         if(context.request.method === 'GET'){
8             //Section One - List - See 'Steps for Creating a Custom List Page', Step Five
9             var list = serverWidget.createList({
10                 title: 'Purchase History'
11             });
12
13             list.style = serverWidget.ListStyle.REPORT;
14
15             list.addButton({
16                 id: 'buttonid',
17                 label: 'Test',
18                 functionName: '' //the function called when the button is pressed
19             });
20
21             //Section Two - Columns - See 'Steps for Creating a Custom List Page', Step Seven
22             var datecol = list.addColumn({
23                 id: 'column1',
24                 type: serverWidget.FieldType.DATE,
25                 label: 'Date',
26                 align: serverWidget.LayoutJustification.RIGHT
27             });
28
29             list.addColumn({
30                 id: 'column2',
31                 type: serverWidget.FieldType.TEXT,
32                 label: 'Product',
33                 align: serverWidget.LayoutJustification.RIGHT
34             });
35
36             list.addColumn({
37                 id: 'column3',
38                 type: serverWidget.FieldType.INTEGER,
39                 label: 'Quantity',
40                 align: serverWidget.LayoutJustification.RIGHT
41             });
42
43             list.addColumn({
44                 id: 'column4',
45                 type: serverWidget.FieldType.CURRENCY,
46                 label: 'Unit Cost',
47                 align: serverWidget.LayoutJustification.RIGHT
48             });
49
50             list.addRows([
51                 {column1: '05/30/2018', column2: 'Widget', column3: '4', column4: '4.50'},
52                 {column1: '05/30/2018', column2: 'Sprocket', column3: '6', column4: '11.50'},
53                 {column1: '05/30/2018', column2: 'Gizmo', column3: '9', column4: '1.25'}]
54             );
55             context.response.writePage(list);
56         } else{
57         }
58     }
59
60     return {
61         onRequest: onRequest
62     };
63 });

```

SuiteScript 2.x Working with UI Objects

For information about using SuiteScript 2.x UI Objects, see:

- [Custom UI development](#)
- [UI Component Overview](#)

- SuiteScript 2.x UI Modules
- Using HTML
- Creating Custom Assistants

Custom UI development



Note: This topic applies to SuiteScript 2.0.

You can build a custom user interface to optimize your NetSuite environment while maintaining the NetSuite look and feel. This gives you the flexibility to create the forms and pages you need for your organization. There are several ways to create and manage your custom UI allowing you to select the solution that is ideal for you.

SuiteBuilder

SuiteBuilder is a point and click tool used to customize the look and feel of your user interface. SuiteBuilder provides a lot of functionality without needing to know how to develop with SuiteScript. You can use SuiteBuilder to modify existing forms by reorganizing different fields using field groups or subtabs. It can also be used to hide or display fields.

While SuiteBuilder is easier to use than developing with SuiteScript, it is not as flexible and is limited to modifying or creating a copy of existing forms.



Note: For more information on using SuiteBuilder, see the help topic [SuiteBuilder Overview](#).

SuiteScript 2.x UI Components

SuiteScript 2.x UI components are a collection of objects and methods contained in the serverWidget API module. Using SuiteScript 2.x provides a lot of control and flexibility allowing you to create custom pages and forms from scratch. Additionally, you can control the placement and positioning of UI components such as fields with more precision. This may be useful when creating a form that needs to be printed on custom preprinted paper.

Additionally, using SuiteScript provides more flexibility and allow automation, as SuiteScript can be used to perform validations and calculations on forms as well as automatically create, access or update records.

While SuiteScript provides more control and flexibility than SuiteBuilder, it is harder to learn and use and does not allow you to use existing form templates.

For more information on SuiteScript 2.x UI Components, see [UI Component Overview](#).

HTML

NetSuite supports the use of inline and embedded HTML to use standard HTML to customize your user interface. Additionally, it is possible to develop your UI completely with HTML components and embed it into a Suitelet for processing. While using HTML may be useful for creating pages and forms that need to meet specific brand requirements, it is more difficult to process and manage form data from HTML than it is to manage data from a NetSuite UI Component.

For more information on using HTML components, see [Using HTML](#).

UI Component Overview



Note: The content in this help topic pertains to SuiteScript 2.0 and SuiteScript 2.1.

UI Components are a collection of objects that can be used to customize your NetSuite experience using SuiteScript 2.x. Using server side scripts you can create custom UIs while maintaining the NetSuite look and feel. SuiteScript UI components are generated on the server and are accessible through client scripts.



Note: SuiteScript does not support direct access to the NetSuite UI through the Document Object Model (DOM). NetSuite UI should only be accessed using SuiteScript APIs.

You can customize the NetSuite user interface using three script types:

Workflow Action scripts allow you to manipulate custom workflow actions that are defined on a record in a workflow. For more information on Workflow Action Scripts, see [SuiteScript 2.x Workflow Action Script Type](#).

Suitelets are the most commonly used script type, as they provide the most flexibility and functionality. Suitelets can be used to create forms, list, assistants as well as support the use of in-line HTML. For more information on Suitelets, see [SuiteScript 2.x Suitelet Script Type](#).

Portlets, like Suitelets, provide a variety of functionality including forms and lists. Portlets can also be used for displaying links and inline HTML. Unlike Suitelets, Portlets can only be displayed on the Dashboard pages and cannot be displayed on their own. For more information on Portlets, see [SuiteScript 2.x Portlet Script Type](#).

User Event scripts can also be used to manipulate custom forms. For more information on User Event Scripts, see [SuiteScript 2.x User Event Script Type](#).

The guide focuses on developing custom user interfaces using Suitelets.

Page Types

When creating a custom user interface with a Suitelet, there are three types of page types that can be combined with the standard NetSuite UI components and HTML components to build the UI you need.

Form

A form is used to display information within NetSuite. Forms are composed of fields that can be configured to display or collect data from the user. Information collected from forms can be submitted by the user and processed by a Suitelet. Additionally, you can generate forms to display and output the information stored in a record.

For information on forms, supported UI components and how to create a form, see [SuiteScript 2.x Custom Forms](#).

For information on the API module, see the help topic `serverWidget.Form`.

List

A list page is used to present information in a table containing rows and columns. Lists can be used to display search results as well as give the user the ability to update and edit the information contained within them.

For information on how to create a list, see [SuiteScript 2.x Custom List Pages](#).

For information on the API module, see the help topic `serverWidget.List`.

Assistant

An assistant is a multistep wizard that you can use to help the user accomplish a larger goal. Assistants are built out of steps, with each step containing a portion of the complete goal.

For information on how to create an assistant, see [Creating Custom Assistants](#).

For information on the API module, see the help topic [serverWidget.Assistant](#)

Standard UI Components

There are four key NetSuite UI components that can be used to build a page, list or assistant. These UI components are standard to the NetSuite experience though they may have different properties depending on the page types.

- **Button**

A customized button is button that you can add to your user interface. You can control the buttons visibility, label and functionality.

For information on the API module, see the help topic [serverWidget.Button](#)

- **Field**

Custom fields are used to record and display information in your user interface.

For more information on types of fields, see the help topic [serverWidget.FieldType](#)

For information on the API module, see the help topic [serverWidget.Field](#)

- **Tab**

A tab is a special section added to your user interface. Using multiple tabs allow you to display more information on a single page.

For information on the API module, see the help topic [serverWidget.Tab](#)

- **Sublist**

A sublist is a list that can be embedded on your page. Custom sublists can present information based on the results of a saved search.

For information on the API module, see the help topic [serverWidget.Sublist](#)

Properties and Enumerations

Each UI component has a series of properties and methods that can be used to customize the functionality and appearance of your NetSuite UI components. These configuration values are stored as enumerations and work with each UI component to give you the most flexibility in designing your UI.

Examples of enumerations and properties include:

- The [serverWidget.FieldType](#) enumerations are used to specify the field type, and define how they behave on your form.
- The [serverWidget.FieldLayoutType](#) enumerations are used to position a field outside a field group or together on the same row.
- The [serverWidget.FieldBreakType](#) enumerations are used to control field break types.
- The [Assistant.isNotOrdered](#) property is used to specify if your assistant is either sequential or non-sequential.

SuiteScript 2.x UI Modules

These three API modules are available for building a custom UI using SuiteScript 2.x.

| | |
|-------------------|---|
| n/ui/serverWidget | The serverWidget module contains the UI components used to work with user interfaces in NetSuite. This module is used to create custom pages, forms, lists and widgets that have the NetSuite look-and-feel. For more information, see the help topic N/ui/serverWidget Module . |
| n/ui/message | The message module is used to display and manage messages at the top of your User Interface. For more information, see the help topic N/ui/message Module . |
| n/ui/dialog | The dialog module is used to create modal dialog boxes that can be used to present additional options or alerts. This module uses JavaScript promises to manage dialogs asynchronously. For more information, see the help topic N/ui/dialog Module . |

Using HTML



Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Using HTML in a Suitelet lets you leverage the flexibility of HTML to add elements that are not available in the NetSuite UI components. With HTML you can customize and style your pages in accordance with your organization's brand and style.

Inline HTML

Use HTML to define a custom field to be included on custom pages generated by Suitelets. For more information, see [SuiteScript 2.x Suitelet Script Type](#).



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

To add HTML to your Suitelet form, use [Form.addField\(options\)](#) to add a field to the form. Specify the field type as INLINEHTML and use the [Field.defaultValue](#) property to set the HTML value.

```

1 var htmlImage = form.addField({
2     id: 'custpage_htmlfield',
3     type: serverWidget.FieldType.INLINEHTML,
4     label: 'HTML Image'
5 });
6     htmlImage.defaultValue = "<img src='https://<accountID>.app.netsuite.com/images/logos/netsuite-oracle.svg'
alt='NetSuite-Oracle logo'>"
```



Important: SuiteScript does not support direct access to the NetSuite UI through the Document Object Model (DOM). The NetSuite UI should only be accessed using SuiteScript APIs .

HTML Suitelets

HTML Suitelets use HTML elements to build a UI instead of NetSuite UI components. The HTML is embedded in the Suitelet using the [https.get\(options\)](#) method from the [N/https Module](#) and the data returned from the form is managed by the Suitelet. Unlike a custom form, which returns prebuilt UI objects, an HTML Suitelet returns a string containing the HTML code. As a result, manipulating and

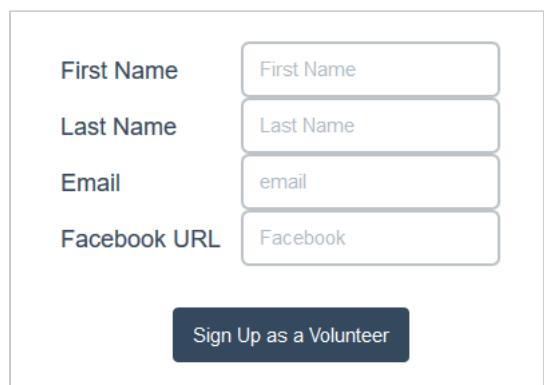
passing data is difficult. Since HTML is coded in a string, passing data to and from the HTML Suitelet requires that you modify the string and change values within the string. The more values you use, the more complex the string manipulation needs to be.

The following example demonstrates a simple volunteer sign-up sheet using an HTML Suitelet. This example can be examined in two parts.

UI Built in HTML

The UI is completely managed by the HTML. It uses standard HTML form elements instead of NetSuite UI components.

The following code sample and screenshot illustrates the form UI built in HTML.



```

1 <form method="post" class="form-horizontal" action="https://LinkToSuitelet.js">
2   <table>
3     <tbody><tr>
4       <td>First Name</td>
5       <td class="col-md-8"><input class="form-control" id="firstname" placeholder="First Name" name="firstname" required="" type="text"></td>
6     </tr>
7     <tr>
8       <td>Last Name</td>
9       <td class="col-md-8"><input class="form-control" id="lastname" placeholder="Last Name" name="lastname" required="" type="text"></td>
10    </tr>
11    <tr>
12      <td>Email</td>
13      <td class="col-md-8"><input class="form-control" id="email" placeholder="email" name="email" required="" type="email"></td>
14    </tr>
15    <tr>
16      <td>Facebook URL</td>
17      <td class="col-md-8"><input class="form-control" id="custentity_fb_url" placeholder="Facebook" name="custentity_fb_url" required="" type="text"></td>
18    </tr>
19    <input name="company" value="{{sponsorid}}" type="hidden">
20    <input name="project" value="{{projectid}}" type="hidden">
21  </tbody></table>
22  <br>
23  <button type="submit" class="btn btn-inverse">Sign Up as a Volunteer</button>
24 </form>

```

Form Processing Managed by the Suitelet

The Suitelet manages the information collected from the HTML form. This Suitelet uses the [https.get\(options\)](#) method from the [N/https Module](#) to access the content from the HTML page, and

uses methods from the [N/record Module](#), [N/email Module](#), and [N/search Module](#) to collect, process, and respond to the data the user submits.

```

1  /**
2  *
3  * @NApiVersion 2.x
4  * @NScriptType Suitelet
5  *
6  */
7
8 define(['N/https', 'N/record', 'N/email', 'N/search'],
9
10    function callbackFunction(https, record, email, search) {
11
12        function getFunction(context) {
13
14            var contentRequest = https.get({
15                url: "https://LinkToFormPage.html"
16            });
17            var contentDocument = contentRequest.body;
18            var sponsorid = context.request.parameters.sponsorid;
19
20            if (sponsorid) {
21                contentDocument = contentDocument.replace("{{sponsorid}}", sponsorid);
22                log.debug("Setting Sponsor", sponsorid)
23            }
24
25            var projectid = context.request.parameters.projectid;
26
27            if (projectid) {
28                contentDocument = contentDocument.replace("{{projectid}}", projectid);
29                log.debug("Setting Project", projectid);
30            }
31
32            context.response.write(contentDocument);
33        }
34
35        function postFunction(context) {
36
37            var params = context.request.parameters;
38
39            var emailString = "First Name: {{firstname}}\nLast Name: {{lastname}}\nEmail: {{email}}\nFacebook URL: {{custentity_fb_url}}"
40
41            var contactRecord = record.create({
42                type: "contact",
43                isDynamic: true
44            );
45
46            for (param in params) {
47                if (param === "company") {
48                    if (params[param] !== "{{sponsorid}}") {
49                        contactRecord.setValue({
50                            fieldId: param,
51                            value: params[param]
52                        );
53                        var lkpfld = search.lookupFields({
54                            type: "vendor",
55                            id: params["company"],
56                            columns: ["entityid"]
57                        );
58                        emailString += "\nSponsor: " + lkpfld.entityid;
59                    }
60                else {
61                    contactRecord.setValue({
62                        fieldId: "custentity_sv_shn_isindi",
63                        value: true
64                    )
65                }
66            else {
67                if (param !== "project") {

```

```

69         contactRecord.setValue({
70             fieldId: param,
71             value: params[param]
72         });
73         var replacer = "{{" + param + "}}";
74         emailString = emailString.replace(replacer, params[param]);
75     }
76 }
77
78 var contactId = contactRecord.save({
79     ignoreMandatoryFields: true,
80     enableSourcing: true
81 });
82
83 log.debug("Record ID", contactId);
84
85 if (params["project"]
86 ["project"] != "{{projectid}}") {
87
88     var lkpfld = search.lookupFields({
89         type: "job",
90         id: params["project"],
91         columns: ["companyname"]
92     });
93
94     emailString += "\nProject Name: " + lkpfld.companyname;
95
96     var participationRec = record.create({
97         type: "customrecord_project_participants",
98         isDynamic: true
99     });
100
101     participationRec.setValue({
102         fieldId: "custrecord_participants_volunteer",
103         value: contactId
104     })
105
106     participationRec.setValue({
107         fieldId: "custrecord_participants_project",
108         value: params["project"]
109     })
110
111     var participationId = participationRec.save({
112         enableSourcing: true,
113         ignoreMandatoryFields: true
114     })
115 }
116
117 log.debug("Email String", emailString);
118
119 email.send({
120     author: -5,
121     recipients: 256,
122     subject: "New Volunteer Signed Up",
123     body: "A new volunteer has joined:\n\n" + emailString
124 });
125
126
127 email.send({
128     author: -5,
129     recipients: params["email"],
130     subject: "Thank you!",
131     body: "Thank you for volunteering:\n\n" + emailString
132 });
133
134 var contentRequest = https.get({
135     url: "https://LinkToFormCompletePage.html"
136 });
137
138 var contentDocument = contentRequest.body;
139
140 context.response.write(contentDocument);
141

```

```

142     }
143     function onRequestFxn(context) {
144
145         if (context.request.method === "GET") {
146             getFunction(context)
147         }
148         else {
149             postFunction(context)
150         }
151     }
152     return {
153         onRequest: onRequestFxn
154     };
155 });
156

```

Note: It is possible to simplify the creation of HTML Suitelets through the use of an HTML templating engine. NetSuite servers run FreeMarker, a Java library used to generate text outputs based on templates, to create dynamic pages and forms. For more information about FreeMarker, see <http://freemarker.org/docs/index.html>

Creating Custom Assistants

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

You can use the assistant page type to create custom assistants. An assistant, also known as a wizard, contains a series of steps that a user must complete to accomplish a larger task.

For examples that show some of the built-in assistants that come with NetSuite, see [Understanding NetSuite Assistants](#).

Assistants are built with Suitelets and by using the [N/ui/serverWidget Module](#).

Note: The assistant cannot be used on externally available Suitelets.

Important: SuiteScript does not support direct access to the NetSuite UI through the Document Object Model (DOM). The NetSuite UI should only be accessed using SuiteScript APIs.

Assistant Creation Process

Note: This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

To develop an assistant, complete the following steps:

- Create a new assistant and define steps.**

Create a new assistant in the Suitelet and define each step for the assistant.

- Build assistant pages.**

Use fields, field groups, and sublists to build out the assistant pages for each step.

- Process assistant pages.**

For each assistant page, construct a response to the user's navigation. At a minimum, you must render each assistant step or page using a GET request. You can also use a POST request to process the user's data before redirecting the user to another step.

Supported UI Components for Assistants

 **Note:** This topic applies to SuiteScript 2.0 and SuiteScript 2.1.

Use the following UI components to build a custom assistant.

Fields

Use fields to display and collect data. Fields are added to the current step on an on demand basis. For more information, see the help topic [Assistant.addField\(options\)](#).

Field Groups

Use field groups to manage and organize the fields in a step. For more information about field groups, see the help topic [Assistant.addFieldGroup\(options\)](#). Additionally, field groups can be collapsed to save space on the page. For more information, see the help topics [FieldGroup.isCollapsed](#) and [FieldGroup.isCollapsible](#).

Sublists

Use a sublist to display the results of a saved search.

 **Note:** INLINEEDITOR is the only sublist type supported by the assistant page type.

For more information, see the help topic [Assistant.addSublist\(options\)](#).

Buttons

You do not need to add buttons to an assistant unlike the form and list page types. Buttons are automatically generated by the assistant.

Steps

Steps are the primary UI components used when creating an assistant because they define each page of the assistant. Step sequence, placement, and positioning can be specified using the following properties.

- To add a step to your assistant, see the help topic [Assistant.addStep\(options\)](#).
- To indicate whether the steps must be completed in a particular order, see the help topic [Assistant.isNotOrdered](#).

 **Note:** Ordered steps appear vertically along the left side of the assistant, while unordered steps appear horizontally across the top of the assistant.

- To add help text to your steps, see the help topic [AssistantStep.helpText](#).

Sample Custom Assistant Script

 **Note:** This topic applies to SuiteScript 2.0.

For more information about custom assistant scripts, see [Creating Custom Assistants](#). Note that assistants cannot be used on externally available Suitelets.



Note: This script sample uses the `define` function, which is required for an entry point script (a script you attach to a script record and deploy). You must use the `require` function if you want to copy the script into the SuiteScript Debugger and test it. For more information, see the help topic [SuiteScript 2.x Global Objects](#).

The following sample creates an assistant with two steps. In the first one, you can select a new supervisor for the employee selected. The second step displays the selection and submits it.

```

1 /**
2  * @NApiVersion 2.x
3  * @NScriptType Suitelet
4 */
5 define(['N/ui/serverWidget'], function(serverWidget) {
6     return {
7         onRequest: function(context) {
8             var assistant = serverWidget.createAssistant({
9                 title: 'New Supervisor',
10                hideNavBar: true
11            });
12            var assignment = assistant.addStep({
13                id: 'assignment',
14                label: 'Select new supervisor'
15            });
16            var review = assistant.addStep({
17                id: 'review',
18                label: 'Review and Submit'
19            });

20            var writeAssignment = function() {
21                assistant.addField({
22                    id: 'newsupervisor',
23                    type: 'select',
24                    label: 'Name',
25                    source: 'employee'
26                });
27                assistant.addField({
28                    id: 'assignedemployee',
29                    type: 'select',
30                    label: 'Employee',
31                    source: 'employee'
32                });
33            };
34        }

35        var writeReview = function() {
36            var supervisor = assistant.addField({
37                id: 'newsupervisor',
38                type: 'text',
39                label: 'Name'
40            });
41            supervisor.defaultValue = context.request.parameters.inpt_newsupervisor;
42

43            var employee = assistant.addField({
44                id: 'assignedemployee',
45                type: 'text',
46                label: 'Employee'});
47            employee.defaultValue = context.request.parameters.inpt_assignedemployee;
48        }
49    }

50    var writeResult = function() {
51        var supervisor = context.request.parameters.newsupervisor;
52        var employee = context.request.parameters.assignedemployee;
53        context.response.write('Supervisor: ' + supervisor + '\nEmployee: ' + employee);
54    }

55    var writeCancel = function() {
56        context.response.write('Assistant was cancelled');
57    }

58    if (context.request.method === 'GET') //GET method means starting the assistant
59    {
60
61
62

```

```
63 |         writeAssignment();
64 |         assistant.currentStep = assignment;
65 |         context.response.writePage(assistant)
66 |     } else //POST method - process step of the assistant
67 |     {
68 |         if (context.request.parameters.next === 'Finish') //Finish was clicked
69 |             writeResult();
70 |         else if (context.request.parameters.cancel) //Cancel was clicked
71 |             writeCancel();
72 |         else if (assistant.currentStep.stepNumber === 1) { //transition from step 1 to step 2
73 |             writeReview();
74 |             assistant.currentStep = assistant.getNextStep();
75 |             context.response.writePage(assistant);
76 |         } else { //transition from step 2 back to step 1
77 |             writeAssignment();
78 |             assistant.currentStep = assistant.getNextStep();
79 |             context.response.writePage(assistant);
80 |         }
81 |     }
82 | };
83 | });
84 |});
```

Transitioning from SuiteScript 1.0 to SuiteScript 2.x



Note: This topic applies to all versions of SuiteScript.

Oracle NetSuite encourages you to transition from using SuiteScript 1.0 to using SuiteScript 2.x. If you are using SuiteScript 1.0 for your scripts, consider converting these scripts to SuiteScript 2.0 or SuiteScript 2.1. SuiteScript 1.0 scripts continue to be supported, however, SuiteScript 1.0 functionality is no longer being updated, and no new feature development or enhancement work is being done for SuiteScript 1.0.

You should use SuiteScript 2.x for any new or substantially revised scripts to take advantage of new features, APIs, and functionality enhancements.

In general, SuiteScript 2.x provides support for all functionality in SuiteScript 1.0. However, there is not always a direct mapping between functions and objects in SuiteScript 1.0 and modules and methods available in SuiteScript 2.x. And some functionality provided by SuiteScript 1.0 can now be done using JavaScript.

See the following help topics for guidance on transitioning your scripts from SuiteScript 1.0 to SuiteScript 2.x:

- [Overview of the Differences Between SuiteScript 1.0 and SuiteScript 2.x](#)
- [Differences Between SuiteScript 1.0 and SuiteScript 2.x Script Types](#)
- [SuiteScript 1.0 APIs Not Directly Mapped to a SuiteScript 2.x Module](#)
- [Differences in Similar SuiteScript 1.0 and SuiteScript 2.x Capabilities](#)
- [Converting a SuiteScript 1.0 Script to a SuiteScript 2.x Script](#)

For more information on specific features of SuiteScript 2.x, see the following help topics:

- [SuiteScript 2.1](#)
- [SuiteScript 2.x Script Creation Process](#)
- [SuiteScript 2.x Terminology](#)
- [SuiteScript 2.x Script Types](#)
- [SuiteScript 2.x Global Objects](#)
- [SuiteScript 2.x Global Objects](#)
- [SuiteScript 2.x Scripting Records and Subrecords](#)
- [SuiteScript Reserved Words](#)
- [SuiteScript 2.x JSDoc Validation](#)
- [SuiteScript 2.x Entry Point Script Creation and Deployment](#)
- [SuiteScript 2.x Custom Modules](#)
- [SuiteScript 2.x API Reference](#)
- [SuiteScript Code Samples](#)
- [SuiteScript Developer Guide](#)
- [SuiteScript FAQ](#)

Overview of the Differences Between SuiteScript 1.0 and SuiteScript 2.x



Note: This topic applies to all versions of SuiteScript.

In general, SuiteScript 2.x provides support for all functionality in SuiteScript 1.0. However, there is not always a direct mapping between functions and objects in SuiteScript 1.0 and modules and methods available in SuiteScript 2.x. And some functionality provided by SuiteScript 1.0 can now be done using JavaScript.

Because SuiteScript 2.x is a modular language, there are several general differences between how scripts are written in SuiteScript 2.x and how they are written in SuiteScript 1.0. These differences include:

- Scripts now have a specified structure with entry points and entry point functions (see [Script Structure, Entry Points, Entry Point Functions](#))
- Scripts now have access to a context object (see [Context Objects](#))
- Scripts now use and support JSDoc (see [JSDoc](#))
- There are now reserved words that cannot be used in SuiteScript scripts (see [Reserved Words](#))
- Enumerated values are now available (see [Enumerated Values](#))
- Scripts now use modules instead of functions and objects (see [Modules](#))
- Arguments or parameters are now specified as objects of key: value pairs (see [Arguments and Parameters](#))
- Scripting subrecords in SuiteScript 2.x is fundamentally different from scripting subrecords in SuiteScript 1.0 (see [Scripting Subrecords](#))

There are also more specific differences concerning arguments and parameters, such as:

- Some argument or parameter names have changed.
- Some arguments or parameters to functions are now properties.
- Some SuiteScript 2.x module methods have additional parameters than their SuiteScript 1.0 function or object counterparts.

For more information on these differences and other specific differences, see [Differences in Similar SuiteScript 1.0 and SuiteScript 2.x Capabilities](#).

Also note that some SuiteScript 1.0 functions and objects are not directly mapped to a SuiteScript 2.x module. For a list of these functions and objects, see [SuiteScript 1.0 APIs Not Directly Mapped to a SuiteScript 2.x Module](#).

Script Structure, Entry Points, Entry Point Functions

Each SuiteScript 2.x script type includes one or more entry points that are exclusive to that script type. An entry point represents the juncture at which the system grants control of the NetSuite application to the script. When you include an entry point in your script, you tell the system that it should execute a function defined within the script. This function is called an entry point function. With many script types, the available entry points are analogous to the types of events that trigger your script. For example, the entry points in a client script represent events that can occur during a browser session, such as pageInit and fieldChanged. In comparison, the scheduled script type has only one entry point, execute, which represents the point at which a schedule executes the script or a user manually executes the script.

For more information on the required structure for SuiteScript 2.x scripts, see [SuiteScript 2.x Anatomy of a Script](#).

For more information on entry points and entry point functions, see [SuiteScript 2.x Entry Point Script Creation and Deployment](#).

For more information on entry points for each specific script type, see [SuiteScript 2.x Script Types](#).

Context Objects

Every SuiteScript 2.x script, whether it be an entry point script or a custom module, must use an entry point. Each entry point must also correspond to an object (usually a function) defined within the script. When the object is a function, it is referred to as an entry point function. In most cases, an entry point function has access to a system-provided object that lets your script access data and take actions specific to the context in which the script is executing. For that reason, this object is often referred to as a context object.

For more information on context objects, see [Context Objects Passed to Standard and Custom Entry Points](#).

JSDoc

SuiteScript 2.x supports the use of JSDoc tags and comment blocks. In fact, two SuiteScript JSDoc tags are required for SuiteScript 2.x scripts: `@NApiVersion` and `@ScriptType`. You can use JSDoc to document your scripts by inserting JSDoc comment blocks and tags within your script.

For more information, see the help topic [SuiteScript 2.x JSDoc Validation](#).

Reserved Words

Like most programming languages, SuiteScript includes a list of reserved words. You cannot use reserved words as variable names, labels, or function names in languages based on the ECMAScript specification, such as JavaScript. SuiteScript is based on the ECMAScript specification, so you also cannot use ECMAScript reserved words as variable names or function names in your SuiteScript scripts. If you use a reserved word as a variable name or function name, you may encounter a syntax error when your script runs.

There are also additional reserved parameter names when using SuiteScript Suitelets. For more information on these reserved parameter names, see [SuiteScript 2.x Reserved Parameter Names in Suitelet URLs](#).

For more information on SuiteScript reserved words, see [SuiteScript Reserved Words](#).

Enumerated Values

Although JavaScript does not include an enumeration type, in the SuiteScript 2.x documentation we use the term 'enumeration' (or enum) to describe a plain JavaScript object with a flat, map-like structure. Within this object, each key points to a read-only string value.

Enumerations typically take the form of a constant with an associated value. For example, the [N/runtime Module](#) includes the `runtime.Permission` enum that includes permission values of FULL, EDIT, CREATE,

VIEW, and NONE. Or, the [N/error Module](#) includes the error.Type enum that includes values for possible errors that may be generated from the execution of a script.

Some enumeration values are static and some are dynamic. Static values are available to all SuiteScript 2.x scripts regardless of the type of account or which features are enabled. Dynamic values are determined based on the specific account type and enabled features. In most cases, the help center documentation indicates that some values are dynamic and may not be available to all scripts in all accounts.

Modules

All SuiteScript 2.x APIs are organized into a set of modules. Each module's name reflects its functionality. For example, the [N/record Module](#) lets you interact with NetSuite records and the [N/https Module](#) module lets you make HTTPS requests to external web services. Most SuiteScript 2.x modules must be explicitly loaded by a script before the script can access that module's methods, properties, and enums. You load each module you want to use by specifying it as an argument to the define function. Some SuiteScript modules are also globally available and do not need to be explicitly loaded.

For more information on the modular architecture of SuiteScript 2.x, see [Modular Architecture](#).

For more information on the define function, see the help topic [define Object](#).

For more information on globally available objects, see the help topic [SuiteScript 2.x Global Objects](#).

Arguments and Parameters

There are several differences between how arguments or parameters are used in SuiteScript 1.0 and how they are used in SuiteScript 2.x. Some of these differences are:

- In SuiteScript 2.x, method parameters are specified as objects of key: value pairs. In SuiteScript 1.0, arguments are passed to API methods as individual variables or values. In SuiteScript 2.x, they are typically passed as objects. Standard methods expect all parameters to be passed in the form of a single object, that has one or many properties (key: values pairs).
- Some argument or parameter names are slightly different in SuiteScript 1.0 than in SuiteScript 2.x for the same comparable API method.
- Some arguments or parameters to functions in SuiteScript 1.0 are now properties in SuiteScript 2.x modules.
- Some SuiteScript 2.x module methods have additional parameters than their SuiteScript 1.0 function or object counterparts.

The [SuiteScript 1.0 to SuiteScript 2.x API Map](#) help topic provides links to each SuiteScript 1.0 API and each corresponding SuiteScript 2.x module, method, or property. You can use this map to compare arguments and parameters between similar functions. The map also includes notes that point out SuiteScript 1.0 functions that are now module properties in SuiteScript 2.x.

Scripting Subrecords

Scripting subrecords in SuiteScript 2.x is fundamentally different from scripting subrecords in SuiteScript 1.0. Compared with SuiteScript 1.0, SuiteScript 2.x introduces the following changes in how you script subrecords:

- In SuiteScript 2.x, you can use a single method to create or load a subrecord.

- In SuiteScript 2.x, subrecords do not have to be explicitly saved; they are saved when you save the parent record.
- In SuiteScript 2.x, you must use subrecord methods to work with the address subrecord; addresses are now implemented using subrecords rather than individual fields on a record.

For more information on scripting subrecords in SuiteScript 2.x, see the help topics [SuiteScript 2.x Scripting Subrecords](#) and [Subrecord Scripting in SuiteScript 2.x Compared With 1.0](#).

Differences Between SuiteScript 1.0 and SuiteScript 2.x Script Types



Note: This topic applies to all versions of SuiteScript.

All script types available in SuiteScript 1.0 are also available in SuiteScript 2.x. Plus, SuiteScript 2.x has two additional script types: map/reduce script and SDF installation script.

In SuiteScript 1.0, the term 'event types' is used to describe how a script was triggered. In SuiteScript 2.x, this term is now 'entry point'. All SuiteScript 2.x scripts have defined entry points, which generally map directly to the event types supported by the same script type in SuiteScript 1.0. For more information on SuiteScript 2.x entry points, see [SuiteScript 2.x Entry Point Script Creation and Deployment](#).

See the following help topics for specific information on the differences in each script type:

- [Bundle Installation Script Type Differences](#)
- [Client Script Type Differences](#)
- [Map/Reduce Script Type Differences](#)
- [Mass Update Script Type Differences](#)
- [Portlet Script Type Differences](#)
- [RESTlet Script Type Differences](#)
- [Scheduled Script Type Differences](#)
- [SDF Installation Script Type Differences](#)
- [Suitelet Script Type Differences](#)
- [User Event Script Type Differences](#)
- [Workflow Action Script Type Differences](#)

For more information on specific SuiteScript 2.x script types, see the following help topics:

- [SuiteScript 2.x Bundle Installation Script Type](#)
- [SuiteScript 2.x Client Script Type](#)
- [SuiteScript 2.x Map/Reduce Script Type](#)
- [SuiteScript 2.x Mass Update Script Type](#)
- [SuiteScript 2.x Portlet Script Type](#)
- [SuiteScript 2.x RESTlet Script Type](#)
- [SuiteScript 2.x Scheduled Script Type](#)

- [SuiteScript 2.x SDF Installation Script Type](#)
- [SuiteScript 2.x Suitelet Script Type](#)
- [SuiteScript 2.x User Event Script Type](#)
- [SuiteScript 2.x Workflow Action Script Type](#)

Bundle Installation Script Type Differences

The only difference between bundle installation scripts in SuiteScript 1.0 and bundle installation scripts in SuiteScript 2.x is that the `toversion` parameter in SuiteScript 1.0 event types is now called `version` in the `beforeInstall`, `afterInstall`, and `beforeUninstall` entry points in SuiteScript 2.x.

For more information on the bundle installation script type in SuiteScript 2.x, see [SuiteScript 2.x Bundle Installation Script Type](#).

Client Script Type Differences

The differences between client scripts in SuiteScript 1.0 and client scripts in SuiteScript 2.x are:

- In SuiteScript 2.x, a client script includes a new parameter: `currentRecord`.
- In SuiteScript 2.x, the `fieldChanged` entry point includes a new parameter: `column`.
- The `type` parameter in the SuiteScript 1.0 `pageInit` event type is called `mode` in the SuiteScript 2.x `pageInit` entry point.
- The `recalc` event type in SuiteScript 1.0 is now the `sublistChanged` entry point in SuiteScript 2.x.
- In SuiteScript 2.x, the `validateDelete` entry point includes a new parameter: `lineCount`.
- In SuiteScript 2.x, the `validateField` entry point includes a new parameter: `column`.
- In SuiteScript 2.x, there are two new entry points: `localizationContextEnter` and `localizationContextExit`.

For more information on the client script type in SuiteScript 2.x, see [SuiteScript 2.x Client Script Type](#).

Map/Reduce Script Type Differences

The map/reduce script is a new script type for SuiteScript 2.x that is designed for scripts that need to handle large amounts of data. It is best suited for situations where the data can be divided into small, independent parts. There is no equivalent SuiteScript 1.0 script type.

For more information on the map/reduce script type in SuiteScript 2.x, see [SuiteScript 2.x Map/Reduce Script Type](#).

Mass Update Script Type Differences

The differences between mass update scripts in SuiteScript 1.0 and mass update scripts in SuiteScript 2.x are:

- In SuiteScript 2.x, a mass update script includes the `each` entry point. There is no equivalent event type in SuiteScript 1.0.

- The `rec_type` and `rec_id` parameters used in SuiteScript 1.0 mass update scripts are replaced with `params.id` and `params.type` parameters for each entry point in SuiteScript 2.x.

For more information on the mass update script type in SuiteScript 2.x, see [SuiteScript 2.x Mass Update Script Type](#).

Portlet Script Type Differences

The differences between portlet scripts in SuiteScript 1.0 and portlet scripts in SuiteScript 2.x are:

- In SuiteScript 2.x, a portlet can now be used for a SuiteApp.
- In SuiteScript 2.x, a portlet script includes the `render` entry point. There is no equivalent event type in SuiteScript 1.0.
- The following methods are included in a SuiteScript 1.0 `nlobjPortlet` object, but are not included in the SuiteScript 2.x `Portlet` object: `setRefreshInterval(n)`, `setScript(scriptId)`.
- The following functions included in a SuiteScript 1.0 `nlobjPortlet` object are now properties in a SuiteScript 2.x `Portlet` object:
 - The `setHtml(html)` function in SuiteScript 1.0 is the `Portlet.html` property in SuiteScript 2.x.
 - The `setTitle(title)` function in SuiteScript 1.0 is the `Portlet.title` property in SuiteScript 2.x.
- The parameters for the `n1PortletObject.addColumn` method have different names in the `Portlet.addColumn` method in SuiteScript 2.x:
 - The `name` parameter in SuiteScript 1.0 is the `id` parameter in SuiteScript 2.x.
 - The `just` parameter in SuiteScript 1.0 is the `align` parameter in SuiteScript 2.x.
- The `Portlet.addEditColumn` method in SuiteScript 2.x has new parameters: `link`, `linkParam`, and `linkParamName`.
- The `name` parameter for the SuiteScript 1.0 `n1PortletObject.addField` method is now called `id` in the SuiteScript 2.x `Portlet.addField` method.
- The `indent` parameter for the SuiteScript 1.0 `n1PortletObject.addLine` method is now called `align` in the SuiteScript 2.x `Portlet.addField` method.
- In SuiteScript 2.x, the portlet script includes two new properties: `Portlet.clientScriptFileId` and `Portlet.clientScriptModulePath`.

For more information on the portlet script type in SuiteScript 2.x, see [SuiteScript 2.x Portlet Script Type](#).

RESTlet Script Type Differences

The only difference between RESTlet scripts in SuiteScript 1.0 and RESTlet scripts in SuiteScript 2.x is that the RESTlet in SuiteScript 2.x includes the `delete`, `get`, `post`, and `put` entry points. These entry points are equivalent to the `delete`, `get`, `put`, and `post` functions (http methods) in SuiteScript 1.0.

- In SuiteScript 2.x, a RESTlet includes the `delete`, `get`, `post`, and `put` entry points. These entry points are equivalent to the `delete`, `get`, `put`, and `post` functions (http methods) in SuiteScript 1.0.



Note: The SuiteScript 2.x documentation for RESTlet script error handling includes more detail than previously provided for SuiteScript 1.0. For more information on error handling with RESTlet scripts, see [SuiteScript 2.x RESTlet Error Handling](#).

For more information on the client script type in SuiteScript 2.x, see the help topic [SuiteScript 2.x RESTlet Script Type](#).

Scheduled Script Type Differences

The differences between scheduled scripts in SuiteScript 1.0 and scheduled scripts in SuiteScript 2.x are:

- In SuiteScript 2.x, a scheduled script includes the execute entry point. There is no equivalent event type in SuiteScript 1.0.
- The SuiteScript 1.0 nlapiScheduleScript function is replaced with the SuiteScript 2.x task.create(options) and task.ScheduledScriptTask
- The type parameter in the SuiteScript 1.0 nlapiScheduledScript method is replaced with the context.InvocationType enum in SuiteScript 2.x. The values are similar: SCHEDULED, ON_DEMAND, USER_INTERFACE, ABORTED, and SKIPPED.
- The nlapiSetRecoverPoint function in SuiteScript 1.0 has no SuiteScript 2.x equivalent. The functionality provided by this function is not needed because of the way scheduled scripts execute in SuiteScript 2.x.
- The nlapiYieldScript function in SuiteScript 1.0 has no SuiteScript 2.x equivalent. The functionality provided by this function is not needed because of the way scheduled scripts execute in SuiteScript 2.x.

For more information on the scheduled script type in SuiteScript 2.x, see [SuiteScript 2.x Scheduled Script Type](#).

SDF Installation Script Type Differences

The SDF installation script is a new script type for SuiteScript 2.x that is used to perform tasks during deployment of a SuiteApp from the SuiteCloud Development Framework (SDF) to a target account. There is no equivalent SuiteScript 1.0 script type.

For more information on the SDF installation script type in SuiteScript 2.x, see [SuiteScript 2.x SDF Installation Script Type](#).

Suitelet Script Type Differences

The only difference between Suitelet scripts in SuiteScript 1.0 and Suitelet scripts in SuiteScript 2.x is that in SuiteScript 2.x, a Suitelet includes the onRequest entry point. There is no equivalent event type in SuiteScript 1.0.

For more information on the Suitelet script type in SuiteScript 2.x, see [SuiteScript 2.x Suitelet Script Type](#).

User Event Script Type Differences

The differences between user event scripts in SuiteScript 1.0 and user event scripts in SuiteScript 2.x are:

- The beforeLoad entry point in SuiteScript 2.x includes a new parameter: newRecord.
- The beforeSubmit entry point in SuiteScript 2.x includes two new parameters: oldRecord and newRecord.
- The afterSubmit entry point in SuiteScript 2.x includes two new parameters: oldRecord and newRecord.
- The type parameter in each entry point in SuiteScript 2.x is now set using the context.UserEventType enum.

For more information on the user event script type in SuiteScript 2.x, see [SuiteScript 2.x User Event Script Type](#).

Workflow Action Script Type Differences

The differences between workflow action scripts in SuiteScript 1.0 and workflow action scripts in SuiteScript 2.x are:

- In SuiteScript 2.x, a workflow action script includes the onAction entry point. There is no equivalent event type for in SuiteScript 1.0.
- The id parameter in SuiteScript 1.0 is called workflowId in SuiteScript 2.x.
- There are two new parameters in SuiteScript 2.x: newRecord and oldRecord.

For more information on the workflow action script type in SuiteScript 2.x, see [SuiteScript 2.x Workflow Action Script Type](#).

Differences in Similar SuiteScript 1.0 and SuiteScript 2.x Capabilities



Note: This topic applies to all versions of SuiteScript.

Some SuiteScript 1.0 capabilities work differently in SuiteScript 2.x, such as logging, printing, scripting subrecords, and resolving URLs.

Logging

Instead of using the SuiteScript 1.0 nlapiLogExecution function to log all types of log messages, SuiteScript 2.x provides a different method for each type of log message: audit, debug, emergency, and error. Each method is available in the [N/log Module](#).

- The [log.audit\(options\)](#) method logs an audit message that appears on the Execution Log tab only if the Log Level on the script deployment is set to Audit or Debug.
- The [log.debug\(options\)](#) method logs a debug message that appears on the Execution Log tab only if the Log Level on the script deployment is set to Debug.
- The [log.emergency\(options\)](#) method logs an emergency message that appears on the Execution Log tab if the Log Level on the script deployment is set to Audit, Debug, Error, or Emergency. In other words, emergency messages always appear.
- The [log.error\(options\)](#) method logs an error message that appears on the Execution Log tab only if the Log Level on the script deployment is set to Audit, Debug, or Error.

For more information, see the help topics:

- [log Object](#)
- [N/log Module](#)

Printing and Template Creation

Instead of using the SuiteScript 1.0 nlapiPrintRecord function to print a record, in SuiteScript 2.x you use the methods in the [N/render Module](#) module, shown in the following table:

| SuiteScript 1.0 Function | SuiteScript 2.x Method |
|--|--|
| nlapiCreateEmailMerger(templateId) | <code>render.mergeEmail(options)</code> |
| nlapiCreateTemplateRenderer() | <code>render.create()</code> |
| nlapiPrintRecord(type, id, mode, properties) | <code>render.bom(options), render.packingSlip(options), render.pickingTicket(options), render.statement(options), render.transaction(options)</code> |
| nlapiXMLToPDF(xmlstring) | <code>render.xmlToPdf(options), TemplateRenderer.renderAsPdf()</code> |

For more information, see the help topic [N/render Module](#).

Scripting Subrecords

Scripting subrecords in SuiteScript 2.x is fundamentally different from scripting subrecords in SuiteScript 1.0. Compared with SuiteScript 1.0, SuiteScript 2.x introduces the following changes in how you script subrecords:

- You can use a single method to create or load a subrecord
- SuiteScript 2.x subrecords do not have to be explicitly saved; they are saved when you save the parent record
- You must use subrecord methods to work with the address subrecord; addresses are not implemented using subrecords rather than individual fields on a record

For more information on scripting subrecords in SuiteScript 2.x, see [SuiteScript 2.x Scripting Subrecords](#) and [Subrecord Scripting in SuiteScript 2.x Compared With 1.0](#).

Resolving URLs

Instead of using the SuiteScript 1.0 nlapiResolveURL function to resolve all URLs, SuiteScript 2.x provides a different method to resolve a record, a script, or a task. Each method is available in the [N/url Module](#).

- The `url.resolveDomain(options)` method used to resolve a domain name.
- The `url.resolveRecord(options)` method used to resolve a record URL.
- The `url.resolveScript(options)` method used to resolve a script URL.
- The `url.resolveTaskLink(options)` method is used to resolve a task link.

For more information on the methods available to resolve URLs, see the help topic [N/url Module](#)

SuiteScript 1.0 APIs Not Directly Mapped to a SuiteScript 2.x Module



Note: This topic applies to all versions of SuiteScript.

Some SuiteScript 1.0 functions and objects do not directly map to any SuiteScript 2.x module method, property, or enum. This includes SuiteScript 1.0 APIs that either do not have direct SuiteScript 2.x API equivalents or the mapping is not intuitive (such as a mapping from a method to a property, or to a new module altogether).

The following table provides the functions and objects that do not have direct or intuitive mapping to SuiteScript 2.x. This information is a summary of what is included in the [SuiteScript 1.0 to SuiteScript 2.x API Map](#).

| SuiteScript 1.0 Function or Object | What to use when writing SuiteScript 2.x scripts |
|---|---|
| Functions | |
| nlapiAddDays(d, days) | Standard JavaScript supplies date functions to add or subtract days from a Date object. SuiteScript 2.x is also compatible with third-party JavaScript APIs that provide date functions. |
| nlapiAddMonths(d, months) | Standard JavaScript supplies date functions to add or subtract months from a Date object. SuiteScript 2.x is also compatible with third-party JavaScript APIs that provide date functions. |
| nlapiEncrypt(s, algorithm, key) | Hashing and HMAC functionality are provided by the N/crypto Module . Encoding functionality is provided by the N/encode Module . |
| nlapiGetCurrentLineItemDateTimeValue(type, fieldId, timeZone) | This functionality is provided by the N/format Module . Specifically, the <code>format.Timezone</code> enum used in the <code>format.format(options)</code> and <code>format.parse(options)</code> methods. |
| nlapiGetDateTimeValue(fieldId, timeZone) | |
| nlapiGetLineItemDateTimeValue(type, fieldId, lineNum, timeZone) | |
| nlapiSetCurrentLineItemDateTimeValue(type, fieldId, dateTime, timeZone) | |
| nlapiSetDateTimeValue(fieldId, dateTime, timeZone) | |
| nlapiSetLineItemDateTimeValue(type, fieldId, lineNum, dateTime, timeZone) | |
| nlapiSetRecoveryPoint() | Scripting recovery points for scheduled scripts are no longer necessary. See SuiteScript 2.x Scheduled Script Type and Map/Reduce Script Type Differences . |
| Objects | |
| nlobjCredentialBuilder.replace(string1, string2) | This functionality is not provided by a SuiteScript 2.x module or method. You need to manually perform the replacement of credential strings within your script. |

| SuiteScript 1.0 Function or Object | What to use when writing SuiteScript 2.x scripts |
|---|---|
| nlobjPortlet.setRefreshInterval(n) | Although there is a Portlet object in SuiteScript 2.x, that object does not include a refresh function. It is no longer needed in SuiteScript 2.x scripts. |
| nlobjRecord.getCurrentLineItemDateTimeValue(type, fieldId, timeZone) | This functionality is provided by the N/format Module . Specifically, the <code>format.format(options)</code> and <code>format.parse(options)</code> methods. |
| nlobjRecord.getDateJsonValue(fieldId, timeZone) | |
| nlobjRecord.getLineItemDateTimeValue(type, fieldId, lineNum, timeZone) | |
| nlobjRecord.setCurrentLineItemDateTimeValue(type, fieldId, dateValue, timeZone) | |
| nlobjRecord.setDateJsonValue(fieldId, dateValue, timeZone) | |
| nlobjRecord.setLineItemDateTimeValue(type, fieldId, lineNum, dateValue, timeZone) | |
| nlobjResponse.setContentType(type, name, disposition) | The functionality to specify content type and encoding type is specified using HTTP headers in SuiteScript 2.x. For more information, see the help topics HTTP Header Information and HTTPS Header Information . |
| nlobjResponse.setEncoding(encodingType) | |
| nlobjSublist.setLineItemValues(values) | Use the Sublist.setSublistValue(options) method of the N/ui/serverWidget Module to set each value in the sublist individually. |
| nlobjSubrecord.cancel() and nlobjSubrecord.commit() | Scripting subrecords in SuiteScript 2.x is fundamentally different from scripting subrecords in SuiteScript 1.0. For more information on scripting subrecords in SuiteScript 2.x, see the help topics SuiteScript 2.x Scripting Subrecords and Subrecord Scripting in SuiteScript 2.x Compared With 1.0 . |

Along with functions and objects that are not directly mapped from SuiteScript 1.0 to SuiteScript 2.x, there are also some functions and objects in SuiteScript 1.0 that map to SuiteScript 2.x, but those mapping may not be intuitive. For example, the functions and objects provided to perform CSV imports in SuiteScript 1.0 are now provided by the [N/task](#) in SuiteScript 2.x. All mappings are provided in the [SuiteScript 1.0 to SuiteScript 2.x API Map](#). The map includes notes to aid in understanding mappings between SuiteScript 1.0 and SuiteScript 2.x that are not intuitive. Also refer to all help topics for the mapped SuiteScript 2.x modules to fully understand the difference (if any) in functionality provided by SuiteScript 1.0 and by SuiteScript 2.x.

Further, some SuiteScript 1.0 functions and objects map to SuiteScript 2.x module properties instead of module methods. The [SuiteScript 1.0 to SuiteScript 2.x API Map](#) clearly indicates these type of mappings. Look in the Notes column – it will indicate the mapping is to a SuiteScript 2.x module property.

Converting a SuiteScript 1.0 Script to a SuiteScript 2.x Script



Note: This topic applies to all versions of SuiteScript.

The general steps you take to convert a SuiteScript 1.0 script to a SuiteScript 2.x script depend on whether you are converting an entry point script or a custom module script.

- If your script is for a specific script type, such as user event or client script, you will first need to edit the script so that the correct script type structure is in place. This includes adding JSDoc comments, the define function, and a return statement at a minimum. For more information on proper script structure, see the help topic [SuiteScript 2.x Anatomy of a Script](#).
- After you place the script into the proper structure, you can start converting individual SuiteScript 1.0 calls into their corresponding SuiteScript 2.x calls according to the [SuiteScript 1.0 to SuiteScript 2.x API Map](#). Note that this order is not a strict order of steps to take, but you may find it easier to place the script into the proper script structure first and then convert each individual call statement.
- If your script is a custom module and not of a specific script type, you can simply convert each individual SuiteScript 1.0 call into its corresponding SuiteScript 2.0 call according to the [SuiteScript 1.0 to SuiteScript 2.x API Map](#).

Here are a few additional notes to consider when converting your scripts:

- Some functions and objects in SuiteScript 1.0 are now supported in SuiteScript 2.x by using JavaScript, such as Date functions.
- Some functions and objects in SuiteScript 1.0 operate a little differently in SuiteScript 2.x, such as logging and printing. See the help topic [Differences in Similar SuiteScript 1.0 and SuiteScript 2.x Capabilities](#).
- Some function and objects in SuiteScript 1.0 do not completely map to any SuiteScript 2.x module, method, property, or enum. See the help topic [SuiteScript 1.0 APIs Not Directly Mapped to a SuiteScript 2.x Module](#).

The [Sample SuiteScript 1.0 to SuiteScript 2.x Conversions](#) help topic shows you a side-by-side comparison of SuiteScript 1.0 and SuiteScript 2.x scripts for each script type and custom modules.

Sample SuiteScript 1.0 to SuiteScript 2.x Conversions



Note: This topic applies to all versions of SuiteScript.

This help topic provides a comparison of SuiteScript 1.0 scripts to SuiteScript 2.x scripts after they have been converted. Each comparison includes the SuiteScript 1.0 script on the left side and the SuiteScript 2.x script on the right side.

Conversions for the following scripts are included:

- Bundle installation script one: [Confirm bundle installation, enable features, create account](#)
- Client script one: [Set default fields values on a record](#)
- Client script two: [Disable fields on a record](#)
- Client script three: [Display user profile information](#)
- Mass update script one: [Update fields on sales order and estimate records](#)
- Portlet script one: [Build a simple portlet that posts data to a servlet](#)
- RESTlet script one: [GET, POST, PUT, and DELETE methods](#)
- Scheduled script one: [Fulfill and bill sales orders each day](#)
- Suitelet script one: [Create a simple form](#)
- User event script one: [Implement end of month sales order promotions](#)
- Workflow action script one: [Set the sales rep on the record in a workflow](#)

- Custom module script one: [Create an item receipt from a purchase order](#)
- Custom module script two: [Create and run a joined search](#)
- Custom module script three: [Create CSV imports](#)

Confirm bundle installation, enable features, create account

This sample is a bundle installation script that makes sure the bundle being installed is version 2.0, and that the Work Orders and Multiple Currencies features are enabled, and then creates an account record.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- JSDoc tags are added at the top of the script to indicate the script version and script type
- The define statement is added which imports the N/runtime and N/error modules.
- The return statement is added with the beforeInstall, beforeUpdate, and afterUpdate entry points.
- The beforeInstall, beforeUpdate, and afterUpdate entry point functions are created.
- The beforeInstall, beforeUpdate, and afterUpdate entry point functions are converted to SuiteScript 2.x.
 - The toversion argument is changed to the params.toVersion or params.version parameter.
 - The fromversion argument is changed to the params.fromVersion parameter.
 - The nlapiCreateRecord function is changed to the record.create method.
 - The setFieldValue function is changed to the Record.setValue method.
 - The nlapiSubmitRecord function is changed to the Record.save method.
- The checkFeatureEnabled function is converted to SuiteScript 2.x.
 - The nlapiGetContext function is no longer needed.
 - The getFeature function is changed to the runtime.isFeatureInEffect method.
 - The nlapiLogExecution function is changed to the log.debug method.
 - The nlobjError object is changed to the error.create method.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|---|
| <pre> 1 function beforeInstall(toversion){ 2 checkFeatureEnabled('WORKORDERS'); 3 if (toversion.toString() == "2.0"){ 4 checkFeatureEnabled('MULTICURRENCY'); 5 } 6 } 7 8 function afterInstall(toversion){ 9 var randomnumber=Math.floor(Math.random()*10000); 10 var objRecord = nlapiCreateRecord('account'); 11 objRecord.setFieldValue('accttype','Bank'); 12 objRecord.setFieldValue('acctnumber',randomnumber); 13 objRecord.setFieldValue('acctname','Acct '+toversion); 14 nlapiSubmitRecord(objRecord, true); 15 } 16 17 function beforeUpdate(fromversion, toversion){ 18 checkFeatureEnabled('WORKORDERS'); 19 if (toversion.toString() == "2.0"){ 20 checkFeatureEnabled('MULTICURRENCY'); 21 } 22 } 23 24 function afterUpdate(fromversion, toversion){ 25 if (fromversion.toString() != toversion.toString()){ </pre> | <pre> 1 /** 2 * @NApiVersion 2.x 3 * @NScriptType BundleInstallationScript 4 */ 5 6 define(['N/runtime', 'N/error'], (runtime, error) => { 7 function checkFeatureEnabled(featureId){ 8 isInEffect = runtime.isFeatureInEffect({ 9 feature: featureId 10 }); 11 12 if (isInEffect){ 13 log.debug({ 14 title: Feature, 15 details: featureId + ' enabled' 16 }); 17 } else { 18 let featureError = error.create({ 19 name: 'INSTALLATION_ERROR', 20 message: 'Feature' + featureId + ' must be enabled. Please enable the feature and try again.' 21 }); 22 throw featureError; 23 } 24 } </pre> |

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|--|
| <pre> 26 var randomnumber=Math.floor(Math.random()*10000); 27 var objRecord = nlapiCreateRecord('account'); 28 objRecord.setFieldValue('accttype','Bank'); 29 objRecord.setFieldValue('acctnumber',randomnumber); 30 objRecord.setFieldValue('acctname','Acct '+toversion); 31 nlapiSubmitRecord(objRecord, true); 32 } 33 } 34 35 function checkFeatureEnabled(featureId){ 36 var objContext = nlapiGetContext(); 37 var feature = objContext.getFeature(featureId); 38 39 if (feature){ 40 nlapiLogExecution('DEBUG','Feature',featureId+' enabled'); 41 } else { 42 throw new nlobjError('INSTALLATION_ERROR','Feature '+ 43 featureId+' must be enabled. Please enable the feature and try 44 again.'); 45 } 46 }</pre> | <pre> 25 return { 26 beforeInstall: function beforeInstall(params) { 27 checkFeatureEnabled('WORKORDERS'); 28 29 if (params.version.toString() === "2.0") { 30 checkFeatureEnabled('MULTICURRENCY'); 31 } 32 }, 33 34 beforeUpdate: function beforeUpdate(params) { 35 checkFeatureEnabled('WORKORDERS'); 36 37 if (params.toVersion.toString() === "2.0") { 38 checkFeatureEnabled('MULTICURRENCY'); 39 } 40 }, 41 42 afterUpdate: function afterUpdate(params) { 43 if (params.fromVersion.toString() !== tover 44 sion.toString()) { 45 let randomnumber = Math.floor(Math.random()*10000); 46 let objRecord = record.create({ 47 type: 'account' 48 }); 49 objRecord.setValue({ 50 fieldId: 'accttype', 51 value: 'Bank' 52 }); 53 objRecord.setValue({ 54 fieldId: 'acctnumber', 55 value: randomnumber 56 }); 57 objRecord.setValue({ 58 fieldId: 'acctname', 59 value: 'Acct '+toversion 60 }); 61 objRecord.save({ 62 enableSourcing: true 63 }); 64 } 65 } 66 }); 67 };</pre> |

Set default fields values on a record

This sample is a client script that sets default field values on a record.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- JSDoc tags are added at the top of the script to indicate the script version and script type.
- The define statement is added which imports the N/record module.
- The return statement is added to the pageInit entry point.
- A record is loaded using the record.load method
- The nlapiGetFieldValue function is changed to the Record.getValue method.
- The nlapiGetFieldText function is changed to the Record.getText method.
- The nlapiSetFieldText function is changed to the Record.setText method.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|---|
| <pre> 1 function pageInit() { 2 if ((nlapiGetFieldValue('fieldA').length === 0) (nlapiGetField 3 Text('fieldA') === "valueA")) 4 { 5 nlapiSetFieldText('fieldA', nlapiGetFieldText('valueB')); 6 } 7 }</pre> | <pre> 1 /** 2 * @NApiVersion 2.x 3 * @NScriptType ClientScript 4 */ 5 6 define(['N/record'], (record) => {</pre> |

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|------------------------|--|
| <pre> 6 } </pre> | <pre> 7 function pageInit(context) { 8 let myRecord = record.load({ 9 type: record.Type.TRANSACTION, 10 id: 7 11 }); 12 let myFieldValue = myRecord.getValue({ 13 fieldId: 'fieldA' 14 }); 15 let myFieldText = myRecord.getText({ 16 fieldId: 'fieldA' 17 }); 18 19 if ((myFieldValue.length === 0) (myFieldText 20 === 'valueA')) 21 { 22 myRecord.setValue({ 23 fieldId: 'fieldA', 24 value: 'valueB' 25 }); 26 } 27 28 return { 29 pageInit: pageInit 30 }; 31 }); </pre> |

Disable fields on a record

This sample is a client script that disables fields on a record.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- JSDoc tags are added at the top of the script to indicate the script version and script type.
- The define statement is added which imports the N/record module.
- The return statement is added with the pageInit entry point.
- A record is loaded using the record.load method.
- The nlapiDisableField function is changed to the Record.isDisabled property.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|--|---|
| <pre> 1 function pageInit() { 2 nlapiDisableField('custrecord_other_fieldA', true); 3 nlapiDisableField('custrecord_other_fieldB', true); 4 } </pre> | <pre> 1 /** 2 * @NApiVersion 2.x 3 * @NScriptType ClientScript 4 */ 5 6 define(['N/record'], (record) => { 7 function pageInit(context) { 8 let myRecord = record.load({ 9 type: record.Type.TRANSACTION, 10 id: 7 11 }); 12 let myRecordFieldA = myRecord.getField({ 13 fieldId: 'custrecord_other_fieldA' 14 }); 15 let myRecordFieldB = myRecord.getField({ 16 fieldId: 'custrecord_other_fieldB' 17 }); 18 19 myRecordFieldA.isDisabled = true; 20 myRecordFieldB.isDisabled = true; 21 } 22 23 return { 24 pageInit: pageInit 25 }; 26 }); </pre> |

Display user profile information

This sample is a client script that displays user profile information.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- JSDoc tags are added at the top of the script to indicate the script version and script type.
- The define statement is added which imports the N/runtime and N/ui/message modules.
- The return statement is added with the pageInit entry point.
- The nlapi GetUser function is changed to the runtime.getCurrentUser method and the runtime.User.name property.
- The nlapiGetRole function is changed to the runtime.User.role property.
- The nlapiGetDepartment function is changed to the runtime.User.department property.
- The nlapiGetLocation function is changed to the runtime.User.location property.
- The alert function is changed to the message.create and message.show functions.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|---|
| <pre> 1 function pageInit(){ 2 var userName = nlapi GetUser(); 3 var userRole = nlapi GetRole(); 4 var userDept = nlapi GetDepartment(); 5 var userLoc = nlapi GetLocation(); 6 7 alert("Current User Information" + "\n\n" + 8 "Name: " + userName + "\n" + 9 "Role: " + userRole + "\n" + 10 "Dept: " + userDept + "\n" + 11 "Loc: " + userLoc 12); 13 }</pre> | <pre> 1 /** 2 * @NApiVersion 2.x 3 * @NScriptType ClientScript 4 */ 5 6 define (['N/runtime', 'N/ui/message'], (runtime, message) => { 7 function pageInit(context) { 8 let myUser = runtime.getCurrentUser(); 9 let userName = myUser.name; 10 let userRole = myUser.role; 11 let userDept = myUser.department; 12 let userLoc = myUser.location; 13 14 let myMessageText = "Current User Information" + "\n\n" + 15 "Name: " + userName + "\n" + 16 "Role: " + userRole + "\n" + 17 "Dept: " + userDept + "\n" + 18 "Loc: " + userLoc; 19 20 let myMessage = message.create({ 21 type: INFORMATION, 22 title: 'User Information', 23 message: myMessageText, 24 duration: 10000 25 }); 26 myMessage.show(); 27 } 28 29 return { 30 pageInit: pageInit 31 }; 32 }); </pre> |

Update fields on sales order and estimate records

This sample is a mass update script that updates fields on sales order and estimate records.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- JSDoc tags are added at the top of the script to indicate the script version and script type.
- The define statement is added which imports the N/record and N/runtime modules.
- The return statement is added with the each entry point.
- The nlapiLoadRecord function is changed to the record.load method.

- The nlapiGetContext and getSetting functions are changed to the runtime.getCurrentScript and the runtime.Script.getParameter methods.
- The setFieldValue method is changed to the Record.setValue method.
- The nlapiSubmitRecord method is changed to the Record.save method.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|---|
| <pre> 1 function updateDepartment(rec_type, rec_id) { 2 var transaction = nlapiLoadRecord(rec_type, rec_id); 3 transaction.setFieldValue('department', nlapiGetContext().getSetting('SCRIPT', 'custscript_dept_update')); 4 nlapiSubmitRecord(transaction, false, true); 5 </pre> | <pre> 1 /** 2 * @NApiVersion 2.x 3 * @NScriptType MassUpdateScript 4 5 6 define(['N/record', 'N/runtime'], (record, runtime) => { 7 function each(params) { 8 let transaction = record.load({ 9 type: params.type, 10 id: params.id 11 }); 12 let scriptObj = runtime.getCurrentScript(); 13 14 let myDepartment = scriptObj.getParameter({ 15 name: 'custscript_dept_update' 16 }); 17 18 transaction.setValue({ 19 fieldId: 'department', 20 value: myDepartment 21 }); 22 transaction.save({ 23 enableSourcing: false, 24 ignoreMandatoryFields: true 25 }); 26 } 27 return { 28 each: each 29 }; 30 }); </pre> |

Build a simple portlet that posts data to a servlet

This sample is a portlet script builds a simple a simple for that posts data to a servlet.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- JSDoc tags are added at the top of the script to indicate the script version and script type.
- The define statement is added which imports the N/portlet module.
- The return statement is added with the render entry point.
- The addField, addSelectOption, and setSubmitButton functions are changed to methods that accept an object as the parameter.
- The setLayoutType function is changed to the updateLayout and updateBreakType methods.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|--|
| <pre> 1 function demoSimpleFormPortlet(portlet, column) { 2 portlet.setTitle('Simple Form Portlet') 3 4 var fld = portlet.addField('text','text','Text'); 5 6 fld.setLayoutType('normal','startcol'); 7 8 portlet.addField('integer','integer','Integer'); 9 10 portlet.addField('date','date','Date'); </pre> | <pre> 1 /** 2 * @NApiVersion 2.x 3 * @NScriptType Portlet 4 5 6 define(['N/portlet'], (portlet) => { 7 function render(params) { 8 let portlet = params.portlet; 9 10 portlet.title = 'Simple Form Portlet'; </pre> |

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|---|
| <pre> 11 12 var select = portlet.addField('fruit','select','Select'); 13 14 select.addSelectOption('a','Oranges'); 15 16 select.addSelectOption('b','Apples'); 17 18 select.addSelectOption('c','Bananas'); 19 20 portlet.addField('textarea','textarea','Textarea'); 21 22 portlet.setSubmitButton('http://httpbin.org/post','Submit'); 23 }</pre> | <pre> 11 12 let fld = portlet.addField({ 13 id: 'text', 14 type: 'text', 15 label: 'Text' 16 }); 17 18 fld.updateLayoutType({ 19 layoutType: 'normal' 20 }); 21 22 fld.updateBreakType({ 23 breakType: 'startcol' 24 }); 25 26 let select = portlet.addField({ 27 id: 'fruit', 28 type: 'select', 29 label: 'Select' 30 }); 31 32 select.addSelectOption({ 33 value: 'a', 34 text: 'Oranges' 35 }); 36 37 select.addSelectOption({ 38 value: 'b', 39 text: 'Apples' 40 }); 41 42 select.addSelectOption({ 43 value: 'c', 44 text: 'Bananas' 45 }); 46 47 portlet.addField({ 48 id: 'textarea', 49 type: 'textarea', 50 label: 'Textarea' 51 }); 52 53 portlet.setSubmitButton({ 54 url: 'http://httpbin.org/post', 55 label: 'Submit', 56 target: '_top' 57 }); 58 } 59 60 return { 61 render: render 62 }; 63 }); </pre> |

GET, POST, PUT, and DELETE methods

This sample is a RESTlet that shows how to use the GET, POST, PUT, and DELETE methods in a RESTlet script..

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- JSDoc tags are added at the top of the script to indicate the script version and script type.
- The define statement is added which imports the N/record and N/error modules.
- The return statement is added with the onRequest entry point.
- The nlapiLoadRecord function is changed to the record.load method.
- The nlapiCreateRecord function is changed to the record.create method.
- The record.setFieldValue function is changed to the Record.setValue method.
- The nlapiSubmitRecord function is changed to the Record.save method.

- The nlapiDeleteRecord function is changed to the record.delete method.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|--|--|
| <pre> 1 // (GET method sample - get a standard NetSuite record) 2 function getRecord(datain) { 3 return nlapiLoadRecord(datain.recordtype, datain.id); 4 } </pre> <pre> 1 // (POST method sample - create a standard NetSuite record) 2 function createRecord(datain) { 3 var err = new Object(); 4 5 if (!datain.recordtype) { 6 err.status = "failed"; 7 err.message= "missing recordtype"; 8 return err; 9 } 10 11 var record = nlapiCreateRecord(datain.recordtype); 12 13 for (var fieldname in datain) { 14 if (datain.hasOwnProperty(fieldname)) { 15 if (fieldname != 'recordtype' && fieldname != 'id') { 16 var value = datain[fieldname]; 17 if (value && typeof value != 'object') { 18 record.setFieldValue(fieldname, value); 19 } 20 } 21 } 22 } 23 var recordId = nlapiSubmitRecord(record); 24 return recordId; 25 } </pre> <pre> 1 // (DELETE method sample - delete a standard NetSuite record) 2 function deleteRecord(datain) { 3 nlapiDeleteRecord(datain.recordtype, datain.id); 4 } </pre> | <pre> 1 /** 2 * @NApiVersion 2.x 3 * @NScriptType Restlet 4 */ 5 define(['N/record', 'N/error'], (record, error) => { 6 7 // Support method called by other methods for validating 8 function doValidation(args, argNames, methodName) { 9 for (let i = 0; i < args.length; i++) { 10 if (!args[i] && args[i] != 0) { 11 throw error.create({ 12 name: 'MISSING_REQ_ARG', 13 message: 'Missing a required argument: [' + 14 argNames[i] + '] for method: ' + methodName 15 }); 16 } 17 } 18 19 // (GET method - get a standard NetSuite record) 20 function _get(context) { 21 doValidation([context.recordtype, context.id], ['record 22 type', 'id'], 'GET'); 23 return JSON.stringify(record.load({ 24 type: context.recordtype, 25 id: context.id 26 })); 27 } 28 29 // (POST method - create a NetSuite record) 30 function post(context) { 31 doValidation([context.recordtype], ['recordtype'], 'POST'); 32 let rec = record.create({ 33 type: context.recordtype 34 }); 35 for (var fldName in context) 36 if (context.hasOwnProperty(fldName)) 37 if (fldName !== 'recordtype') 38 rec.setValue(fldName, context[fldName]); 39 let recordId = rec.save(); 40 return String(recordId); 41 } 42 43 // (DELETE method - delete a standard NetSuite record) 44 function _delete(context) { 45 doValidation([context.recordtype, context.id], ['record 46 type', 'id'], 'DELETE'); 47 record.delete({ 48 type: context.recordtype, 49 id: context.id 50 }); 51 return String(context.id); 52 } 53 54 // (PUT method - upsert a NetSuite record) 55 function put(context) { 56 doValidation([context.recordtype, context.id], ['record 57 type', 'id'], 'PUT'); 58 let rec = record.load({ 59 type: context.recordtype, 60 id: context.id 61 }); 62 for (let fldName in context) 63 if (context.hasOwnProperty(fldName)) 64 if (fldName !== 'recordtype' && fldName != 'id') 65 rec.setValue(fldName, context[fldName]); 66 rec.save(); 67 return JSON.stringify(rec); 68 } 69 70 return { 71 get: _get, 72 delete: _delete, 73 post: post, 74 put: put 75 }; 76 } </pre> |

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|------------------------|------------------------|
| | 73 }); |

Fulfill and bill sales orders each day

This sample is a scheduled script that fulfills and bills sales orders created each day.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- JSDoc tags are added at the top of the script to indicate the script version and script type.
- The define statement is added which imports the N/search and N/record modules.
- The return statement is added with the onRequest entry point.
- The nlobjSearchFilter function is changed to the search.createFilter method.
- The nlobjSearchColumn function is changed to the search.createColumn method.
- The nlapiSearchRecord function is changed to the search.run.each method.
- The nlapiTransformRecord function is changed to the record.transform method.
- The nlapiSubmitRecord function is changed to the Record.save method.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|--|--|
| <pre> 1 function processOrdersCreatedToday(type) 2 { 3 //only execute when run from the scheduler 4 if (type != 'scheduled' && type != 'skipped') return; 5 6 var filters = new Array(); 7 filters[0] = new nlobjSearchFilter('mainline', null, 'is', 'T'); 8 filters[1] = new nlobjSearchFilter('trandate', null, 'equal To', 'today'); 9 10 var searchresults = nlapiSearchRecord('salesorder', null, filter s, null, new nlobjSearchColumn('terms')); 11 for (var i = 0; searchresults != null && i < searchresult s.length; i++) 12 { 13 var id = searchresults[i].getId(); 14 var fulfillRecord = nlapiTransformRecord('salesorder', id, 'itemfulfillment'); 15 nlapiSubmitRecord(fulfillRecord); 16 17 var billType = searchresults[i].getValue('paymentmethod') == null ? 'invoice' : 'cashSale'; 18 var billRecord = nlapiTransformRecord('salesorder', id, billType); 19 nlapiSubmitRecord(billRecord); 20 } 21 }</pre> | <pre> 1 /** 2 * @NApiVersion 2.x 3 * @NScriptType ScheduledScript 4 */ 5 6 // This script creates multiple sales records and logs the record creation progress. 7 define(['N/search', 'N/record'], (search, record) => { 8 function processOrdersCreatedToday() { 9 if ((context.type === context.InvocationType.SCHEDULED) (context.type === context.InvocationType.SKIPPED)) { 10 let filters = new Array(); 11 12 let filters[0] = search.createFilter({ 13 name: 'mainline', 14 operator: 'is', 15 values: 'T' 16 }); 17 let filters[1] = search.createFilter({ 18 name: 'trandate', 19 operator: 'equalTo', 20 values: 'today' 21 }); 22 23 let searchcolumn = search.createColumn({ 24 name: 'terms' 25 }); 26 27 let search = search.create({ 28 type: , 29 filters: filters, 30 columns: searchcolumn 31 }); 32 33 search.run().each(function(result) { 34 35 let id = result.getValue({ 36 name: 'id' 37 }); 38 39 let fulfillRecord = record.transform({ 40 fromType: 'salesorder', 41 fromId: id, 42 toType: 'itemfulfillment' 43 }); 44 }); 45 } 46 } 47 })(search, record); </pre> |

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|------------------------|---|
| | <pre> 45 fulfillRecord.save(); 46 47 let paymentmethod = result.getValue({ 48 name: 'paymentmethod' 49 }); 50 51 let billType = paymentmethod ? 'invoice' : 52 'cashsale'; 53 54 let billRecord = record.transform({ 55 fromType: 'salesorder', 56 fromId: id, 57 toType: billType 58 }); 59 60 billRecord.save(); 61 } 62 63 return { 64 execute: processOrdersCreatedToday 65 } 66 }); </pre> |

Create a simple form

This sample is a Suitelet that creates a simple form.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- JSDoc tags are added at the top of the script to indicate the script version and script type.
- The define statement is added which imports the N/ui/serverWidget module.
- The return statement is added with the onRequest entry point.
- The nlapiCreateForm function is changed to the serverWidget.createForm method.
- The field.setLayoutType function is changed to the serverWidget.FieldLayoutType.NORMAL and serverWidget.FieldBreakType.STARTCOL enums.
- The dumpResponse function is rewritten to use the context parameter.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|--|
| <pre> 1 function demoSimpleForm(request, response) { 2 if (request.getMethod() == 'GET') { 3 var form = nlapiCreateForm('Simple Form'); 4 var field = form.addField('textfield', 'text', 'Text'); 5 field.setLayoutType('normal', 'startcol') 6 form.addField('datefield', 'date', 'Date'); 7 form.addField('currencyfield', 'currency', 'Currency'); 8 form.addField('textareafield', 'textarea', 'Textarea'); 9 10 var select = form.addField('selectfield', 'select', 'Select'); 11 select.addSelectOption('', ''); 12 select.addSelectOption('a', 'Albert'); 13 select.addSelectOption('b', 'Baron'); 14 select.addSelectOption('c', 'Chris'); 15 select.addSelectOption('d', 'Drake'); 16 select.addSelectOption('e', 'Edgar'); 17 18 var sublist = form.addSubList('sublist', 'inlineeditor', 'In line Editor Sublist'); 19 sublist.addField('sublist1', 'date', 'Date'); 20 sublist.addField('sublist2', 'text', 'Text'); 21 sublist.addField('sublist3', 'currency', 'Currency'); 22 sublist.addField('sublist4', 'textarea', 'Large Text'); 23 sublist.addField('sublist5', 'float', 'Float'); 24 25 form.addButton('Submit'); 26 </pre> | <pre> 1 Noela already had this one converted 2 /** 3 * @NApiVersion 2.x 4 * @NScriptType Suitelet 5 */ 6 7 define(['N/ui/serverWidget'], (serverWidget) => { 8 function onRequest(context) { 9 if (context.request.method === 'GET') { 10 let form = serverWidget.createForm({ 11 title: 'Simple Form' 12 }); 13 14 let field = form.addField({ 15 id: 'custpage_text', 16 type: serverWidget.FieldType.TEXT, 17 label: 'Text' 18 }); 19 20 field.setLayoutType = serverWidget.FieldLayoutType.NORMAL; 21 22 field.updateBreakType({ 23 breakType: serverWidget.FieldBreakType.STARTCOL 24 }); 25 26 form.addField({ 27 id: 'custpage_date', 28 </pre> |

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|--|---|
| <pre> 27 response.writePage(form); 28 } 29 else { 30 dumpResponse(request,response); 31 } 32 }</pre> | <pre> 28 type: serverWidget.FieldType.DATE, 29 label: 'Date' 30 }); 31 32 form.addField({ 33 id: 'custpage_currencyfield', 34 type: serverWidget.FieldType.CURRENCY, 35 label: 'Currency' 36 }); 37 38 let select = form.addField({ 39 id: 'custpage_selectfield', 40 type: serverWidget.FieldType.SELECT, 41 label: 'Select' 42 }); 43 44 select.addSelectOption({ 45 value: 'a', 46 text: 'Albert' 47 }); 48 49 select.addSelectOption({ 50 value: 'b', 51 text: 'Baron' 52 }); 53 54 let sublist = form.addSublist({ 55 id: 'sublist', 56 type: serverWidget.SublistType.INLINEEDITOR, 57 label: 'Inline Editor Sublist' 58 }); 59 60 sublist.addField({ 61 id: 'sublist1', 62 type: serverWidget.FieldType.DATE, 63 label: 'Date' 64 }); 65 66 sublist.addField({ 67 id: 'sublist2', 68 type: serverWidget.FieldType.TEXT, 69 label: 'Text' 70 }); 71 72 form.addSubmitButton({ 73 label: 'Submit Button' 74 }); 75 76 context.response.writePage(form); 77 } else { 78 const delimiter = /\u0001/; 79 let textField = context.request.parameters.textfield; 80 let dateField = context.request.parameters.datefield; 81 let currencyField = context.request.parameters.currency 82 field; 83 let selectField = context.request.parameters.selectfield; 84 let sublistData = context.request.parameters.sublistdata; 85 ta.split(delimiter); 86 let sublistField1 = sublistData[0]; 87 let sublistField2 = sublistData[1]; 88 89 context.response.write('You have entered: ' + textField 90 + ' ' + dateField + ' ' 91 + currencyField + ' ' + selectField + ' ' + sublist 92 Field1 + ' ' + sublistField2); 93 } 94 95 return { 96 onRequest: onRequest 97 }; 98 }); 99 }); </pre> |

Implement end of month sales order promotions

This sample is a user event script that implements end of month sales order promotions.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- JSDoc tags are added at the top of the script to indicate the script version and script type
- The define statement is added which imports the N/currentRecord, N/runtime, and N/ui/serverWidget modules
- The return statement is added with the beforeLoad entry point
- The nlapiGetContext function is changed to the context.UserEventType property.
- The getExecutionContext function is changed to the runtime.executionContext property.
- The nlapiLogExecution function is changed to the log.debug method.
- The form function parameter is changed to the form context component.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|---|
| <pre> 1 function customizeUI_SalesOrderBeforeLoad(type, form) { 2 var currentContext = nlapiGetContext(); 3 4 if (type == 'create' & currentContext.getExecutionContext() 5 == 'userinterface') { 6 7 var fieldId = 'custpage_eom_promotion'; 8 var fieldLabel = 'Eligible EOM promotion'; 9 var today = new Date(); 10 var month = today.getMonth(); 11 var date = today.getDate(); 12 13 nlapiLogExecution('DEBUG', 'month date', month + ' ' + date); 14 15 //February 16 if (month==1) { 17 18 if (date==24 date==25 date==26 date==27 date==28 19 date==29) 20 form.addField(fieldId, 'checkbox', fieldLabel); 21 22 //31-day months 23 else if (month==0 month==2 month ==4 month==6 month==7 24 month==9 month==11) 25 { 26 27 if (date==27 date==28 date==29 date==30 date==31) 28 form.addField(fieldId, 'checkbox', fieldLabel); 29 30 //30-day months 31 else 32 { 33 34 if (date==26 date==27 date==28 date==29 date==30) 35 form.addField(fieldId, 'checkbox', fieldLabel); 36 37 } 38 } 39 } </pre> | <pre> 1 /* 2 * @NApiVersion 2.x 3 * @NScriptType UserEventScript 4 */ 5 6 define (['N/currentRecord', 'N/runtime', 'N/ui/serverWidget'], 7 (currentRecord, runtime, serverWidget) => { 8 9 function customizeUI_SalesOrderBeforeLoad(context) { 10 11 if (context.UserEventType === context.UserEventType.CREATE) { 12 13 if (runtime.executionContext === runtime.Context 14 Type.USEREVENT) { 15 16 const fieldId = 'custpage_eom_promotion'; 17 const fieldLabel = 'Eligible EOM promotion'; 18 let today = new Date(); 19 let month = today.getMonth(); 20 let date = today.getDate(); 21 22 log.debug({ 23 title: 'month date', 24 details: month + ' ' + date 25 }); 26 27 // February 28 if (month === 1) { 29 30 if (date === 24 date === 25 date === 26 31 date === 27 date === 28 date === 29) { 32 33 context.form.addField({ 34 id: fieldId, 35 label: fieldLabel, 36 type: serverWidget.FieldType.CHECKBOX 37 }); 38 39 // 31-day months 40 } else if (month === 0 month === 2 month === 4 41 month === 6 month === 7 42 month === 9 month === 11) { 43 44 if (date === 27 date === 28 date === 29 45 date === 30 date === 31) { 46 47 context.form.addField({ 48 id: fieldId, 49 label: fieldLabel, 50 type: serverWidget.FieldType.CHECKBOX 51 }); 52 53 } 54 55 } 56 57 } 58 59 } 60 61 } 62 63 } 64 } </pre> |

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|------------------------|--|
| | <pre> 54 return { 55 beforeLoad: customUI_SalesOrderBeforeLoad 56 }; 57 }); </pre> |

Set the sales rep on the record in a workflow

This sample is a user event script that sets the sales rep on the record in a workflow. To get a new record in SuiteScript 2.x, this code needs to be placed into a user event script in the beforeLoad, beforeSubmit, or afterSubmit entry point. For this example, the beforeLoad entry point is used.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- JSDoc tags are added at the top of the script to indicate the script version and script type.
- The define statement is added which imports the N/currentRecord and N/runtime modules.
- The return statement is added with the beforeLoad entry point.
- The nlapiGetNewRecord function is changed to context.newRecord.
- The nlapiGetContext and getSetting functions are changed to the runtime.getCurrentScript and the runtime.Script.getParameter methods.
- The setFieldValue function is changed to the Record.setValue method.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|--|
| <pre> 1 function changeSalesRep() { 2 nlapiGetNewRecord().setFieldValue('salesrep', nlapiGetContex 3 t().getSetting('SCRIPT', 'custscript_salesrep')); 4 </pre> | <pre> 1 /** 2 * @NApiVersion 2.x 3 * @NScriptType UserEventScript 4 */ 5 6 define(['N/currentRecord', 'N/runtime'], (currentRecord, runtime) => 7 { 8 function changeSalesRep(context) { 9 let myRecord = context.newRecord(); 10 let scriptObj = runtime.getCurrentScript(); 11 let myScriptParam = scriptObj.getParameter({ 12 name: 'custscript_salesrep' 13 }); 14 15 myRecord.setValue({ 16 fieldId: 'salesrep', 17 value: myScriptParam 18 }); 19 20 return { 21 beforeLoad: changeSalesRep 22 }; 23 }); </pre> |

Create an item receipt from a purchase order

This sample is a custom module script that creates an item receipt from a purchase order. This is either a custom module or an individual function within a user event or other server script, so no JSDoc or define statement or return statement or entry point function is needed. But you will need to load the N/record and N/email modules for the function to work.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- JSDoc tags are added at the top of the script to indicate the script version and script type.

- The nlapiTransformRecord function is changed to the record.transform method.
- The getLineItemValue function is changed to the Record.getSublistValue method.
- The setLineItemValue function is changed to the Record.setSublistValue method.
- The nlapiSubmitRecord function is changed to the Record.save method.
- The nlapiSendEmail function is changed to the email.send method.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|--|
| <pre> 1 function transformPurchaseOrder() { 2 var fromRecord; 3 var fromId; 4 var toRecord; 5 var trecord; 6 var qty; 7 8 fromRecord = 'purchaseorder'; 9 fromId = 26; 10 toRecord = 'itemreceipt'; 11 12 trecord = nlapiTransformRecord(fromRecord, fromId, toRecord); 13 qty = trecord.getLineItemValue('item', 'quantity', 1); 14 trecord.setLineItemValue('item', 'quantity', 1, '2'); 15 var idl = nlapiSubmitRecord(trecord, true); 16 17 nlapiSendEmail(-5, -5, 'Transform Email' + 'Original Qty = ' + qty + ' ' + 'Record Created = ' + idl , null); 18 } </pre> | <pre> 1 function transformPurchaseOrder() { 2 const fromRecord = 'purchaseorder'; 3 const fromId = 26; 4 const toRecord = 'itemreceipt'; 5 6 let trecord = record.transform({ 7 fromType: fromRecord, 8 fromId: fromId, 9 toType: toRecord, 10 }); 11 12 let qty = trecord.getSublistValue({ 13 sublistId: 'item', 14 fieldId: 'quantity', 15 line: 1 16 }); 17 18 trecord.setSublistValue({ 19 sublistId: 'item', 20 fieldId: 'quantity', 21 line: 1, 22 value: '2' 23 }); 24 25 let idl = trecord.save({ 26 enableSourcing: true 27 }); 28 29 email.send({ 30 author: -5, 31 recipients: -5, 32 subject: 'Transform Email' + 'Original Qty = ' + qty + ' ' + 'Record Created = ' + idl, 33 body: null, 34 }); 35 } </pre> |

Create and run a joined search

This sample is a custom module script that creates and runs a joined search. This sample can be used as a custom module or as an individual function within a user event or other server script, so no JSDoc or define statement or return statement or entry point function is needed. But you will need to load the N/search module for the function to work.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- The nllobjSearchFilter function is changed to the search.createFilter method.
- The nllobjSearchColumn function is changed to the search.createColumn method.
- The nlapiSearchRecord function is changed to the Search.run method.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|--|
| <pre> 1 function myJoinedSearch() { 2 // Create a filters array and define search filters for an item search 3 var filters = new Array(); </pre> | <pre> 1 function myJoinedSearch () { 2 let filters = new Array(); 3 </pre> |

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|---|
| <pre> 4 //filter by a specific customer (121) on the Pricing record 5 filters[0] = new nlobjSearchFilter('customer', 'pricing', 'is', '121'); 6 7 //filter by a currency type (USA) on the Pricing record 8 filters[1] = new nlobjSearchFilter('currency', 'pricing', 'is', '1'); 9 10 11 //set search return columns for Pricing search 12 var columns = new Array(); 13 14 //return data from pricelevel and unitprice fields on the Pricing record 15 columns[0] = new nlobjSearchColumn('pricelevel', 'pricing'); 16 columns[1] = new nlobjSearchColumn('unitprice', 'pricing'); 17 18 //specify name as a search return column. There is no join set in this field. 19 //This is the Name field as it appears on Item records. 20 columns[2] = new nlobjSearchColumn('name'); 21 22 //execute the Item search, which uses data on the Pricing record as search 23 filters 24 var searchresults = nlapiSearchRecord('item', null, filters, columns); 25 26 27 28 29 30 31 32 33 </pre> | <pre> 4 filters[0] = search.createFilter({ 5 name: 'customer', 6 join: 'pricing', 7 operator: 'is', 8 values: '121' 9 }); 10 11 filters[1] = search.createFilter({ 12 name: 'currency', 13 join: 'pricing', 14 operator: 'is', 15 values: '1' 16 }); 17 18 let columns = new Array(); 19 20 columns[0] = search.createColumn({ 21 name: 'pricelevel', 22 join: 'pricing' 23 }); 24 columns[1] = search.createColumn({ 25 name: 'unitprice', 26 join: 'pricing' 27 }); 28 columns[2] = search.createColumn({ 29 name: 'name' 30 }); 31 32 let searchresults = search.run(); 33 </pre> |

Create CSV imports

This sample includes two scripts used to creates CSV imports. These samples can be used as a custom module or as individual function within a user event or other server script, so no JSDoc or define statement or return statement or entry point function is needed. But you will need to load the N/task and N/file modules for the function to work.

The conversion of this script from SuiteScript 1.0 to SuiteScript 2.x includes the following:

- The nlapiLoadFile function is changed to the file.load method.
- The nlapiCreateCSVImport function is changed to the task.create method using the task.TaskType.CSV_Import type value.
- The job.setMapping function is changed to the job.mappingId property.
- The job.setsetPrimaryFile function is changed to the job.importFile property.
- The job.setLinkedFile function is changed to the job.linkedFiles property (which is an Object).
- The job.setOption function is changed to the job.name property.
- The nlapiSubmitCSVImport function is changed to the job.submit method.

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|---|--|
| <pre> 1 var mapping fileId = 2; 2 var primaryFile = nlapiLoadFile(73); 3 4 var job = nlapiCreateCSVImport(); 5 job.setMapping(mapping fileId); 6 job.setPrimaryFile(primaryFile); 7 job.setOption("jobName", "jobImport"); 8 9 var jobId = nlapiSubmitCSVImport(job); </pre> | <pre> 1 function CSVImport() { 2 const mapping fileId = 2; 3 let primaryFile = file.load({ 4 id: 73 5 }); 6 7 let job = task.create({ 8 taskType: task.TaskType.CSV_IMPORT 9 }); 10 11 job.mappingId = mapping fileId; 12 job.importFile = primaryFile; </pre> |

| SuiteScript 1.0 Script | SuiteScript 2.x Script |
|--|---|
| <pre> 1 var mapping fileId = "CUSTIMPORTentityMultiFile"; 2 3 var job = nlapiCreateCSVImport(); 4 job.setMapping(mapping fileId); 5 6 job.setPrimaryFile(nlapiLoadFile(73)); 7 8 job.setLinkedFile("addressbook", nlapiLoadFile(74)); 9 job.setOption("jobName", "jobImport"); 10 11 var jobId = nlapiSubmitCSVImport(job); </pre> | <pre> 13 job.name = 'jobImport'; 14 15 let jobId = job.submit(); 16 1 2 function CSVMultiFileImport() { 3 const mapping fileId = 'CUSTIMPORTentityMultiFile'; 4 5 let job = task.create({ 6 taskType: task.TaskType.CSV_IMPORT 7 }); 8 9 job.mappingId = mapping fileId; 10 11 let primaryFile = file.load({ 12 id: 73 13 }); 14 job.primaryFile = primaryFile; 15 16 let linkedFile = file.load({ 17 id: 74 18 }); 19 20 job.linkedFiles = {'addressbook': linkedFile}; 21 22 job.name = 'jobImport'; 23 24 let jobId = job.submit(); </pre> |