agreement, or other license agreement which has been executed by you and Oracle and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

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

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

Sample Code

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

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

No Excessive Use of the Service

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

Beta Features

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

SuiteScript 2.0 API Introduction .......................................................... 1
SuiteScript 2.0 Hello World ................................................................. 1
SuiteScript 2.0 Script Basics ............................................................... 10
SuiteScript 2.0 Anatomy of a Script ...................................................... 12
SuiteScript 2.0 Script Creation Process ................................................. 14
SuiteScript 2.0 Advantages ................................................................. 15
SuiteScript 2.0 Terminology ............................................................... 18
SuiteScript 2.0 Developer Resources ..................................................... 19
SuiteScript Versioning Guidelines ....................................................... 20
SuiteScript 2.0 Script Types ............................................................... 23
SuiteScript 2.0 Bundle Installation Script Type ........................................ 25
  SuiteScript 2.0 Bundle Installation Script Entry Points ................................ 26
SuiteScript 2.0 Client Script Type ........................................................ 28
  SuiteScript 2.0 Client Script Reference ................................................ 32
  SuiteScript 2.0 Client Script Entry Points and API .................................. 36
SuiteScript 2.0 Map/Reduce Script Type ................................................ 43
  SuiteScript 2.0 Map/Reduce Script Reference ........................................ 54
  SuiteScript 2.0 Map/Reduce Script Entry Points ..................................... 85
SuiteScript 2.0 Mass Update Script Type .............................................. 122
  SuiteScript 2.0 Mass Update Script Entry Points ................................... 123
SuiteScript 2.0 Portlet Script Type ...................................................... 124
  SuiteScript 2.0 Portlet Script Reference .............................................. 128
  SuiteScript 2.0 Portlet Script Entry Points and API ................................ 130
SuiteScript 2.0 RESTlet Script Type .................................................... 140
  SuiteScript 2.0 Getting Started with RESTlets ..................................... 140
  SuiteScript 2.0 RESTlet Reference .................................................... 146
  SuiteScript 2.0 RESTlet Script and Request Samples ................................ 148
  SuiteScript 2.0 RESTlet Script Entry Points ....................................... 155
SuiteScript 2.0 Scheduled Script Type ................................................. 158
  SuiteScript 2.0 Scheduled Script Reference ......................................... 160
  SuiteScript 2.0 Script Entry Points and API ....................................... 170
SuiteScript 2.0 Suitelet Script Type .................................................... 171
  SuiteScript 2.0 Suitelet Script Reference ........................................... 173
  SuiteScript 2.0 Suitelet Script Entry Points and API .............................. 182
SuiteScript 2.0 User Event Script Type ................................................ 183
  SuiteScript 2.0 User Event Script Reference ......................................... 185
  SuiteScript 2.0 User Event Script Entry Points and API .......................... 195
SuiteScript 2.0 Workflow Action Script Type ........................................ 200
  SuiteScript 2.0 Workflow Action Script Entry Points and API .................... 201
SuiteScript 2.0 SDF Installation Script Type ......................................... 202
  Using an SDF Installation Script ..................................................... 202
SuiteScript 2.0 Record Actions and Macros .......................................... 205
  Overview of Record Action and Macro APIs ......................................... 205
  Supported Record Actions .............................................................. 205
  Supported Record Macros ............................................................... 206
SuiteScript 2.0 JSDoc Validation ......................................................... 207
  Controlling Access to Scripts and Custom Modules ................................ 209
SuiteScript 2.0 Entry Point Script Creation and Deployment ....................... 212
  Record-Level and Form-Level Scripts ............................................... 212
SuiteScript 2.0 Entry Point Script Validation ......................................... 212
  Entry Point Script Validation Guidelines ........................................... 213
  Entry Point Script Validation Examples ............................................ 216
  Entry Point Script Validation Error Reference ..................................... 218
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SuiteScript 2.0 Record-Level Scripts</td>
<td>221</td>
</tr>
<tr>
<td>Script Record Creation</td>
<td>221</td>
</tr>
<tr>
<td>Script Deployment</td>
<td>224</td>
</tr>
<tr>
<td>Viewing System Notes</td>
<td>228</td>
</tr>
<tr>
<td>SuiteScript 2.0 Form-Level Scripts</td>
<td>229</td>
</tr>
<tr>
<td>Attaching a Client Script to a Form</td>
<td>229</td>
</tr>
<tr>
<td>Configuring a Custom Action</td>
<td>230</td>
</tr>
<tr>
<td>SuiteScript 2.0 Custom Modules</td>
<td>232</td>
</tr>
<tr>
<td>SuiteScript 2.0 Anatomy of a Custom Module Script</td>
<td>233</td>
</tr>
<tr>
<td>SuiteScript 2.0 Custom Module Tutorial</td>
<td>234</td>
</tr>
<tr>
<td>Module Dependency Paths</td>
<td>244</td>
</tr>
<tr>
<td>Naming a Custom Module</td>
<td>245</td>
</tr>
<tr>
<td>Custom Module Examples</td>
<td>246</td>
</tr>
<tr>
<td>Troubleshooting Errors</td>
<td>250</td>
</tr>
<tr>
<td>Frequently Asked Questions: Custom Modules</td>
<td>251</td>
</tr>
<tr>
<td>SuiteScript 2.0 Scripting Records and Subrecords</td>
<td>252</td>
</tr>
<tr>
<td>SuiteScript 2.0 Scripting Records</td>
<td>252</td>
</tr>
<tr>
<td>SuiteScript 2.0 – Standard and Dynamic Modes</td>
<td>252</td>
</tr>
<tr>
<td>SuiteScript 2.0 – Record Modules</td>
<td>253</td>
</tr>
<tr>
<td>SuiteScript 2.0 Scripting Subrecords</td>
<td>253</td>
</tr>
<tr>
<td>Understanding Subrecords</td>
<td>254</td>
</tr>
<tr>
<td>Subrecord Scripting in SuiteScript 2.0 Compared With 1.0</td>
<td>269</td>
</tr>
<tr>
<td>Scripting Subrecords that Occur on Sublist Lines</td>
<td>270</td>
</tr>
<tr>
<td>Scripting Subrecords that Occur in Body Fields</td>
<td>294</td>
</tr>
<tr>
<td>SuiteScript 2.0 Custom Pages</td>
<td>315</td>
</tr>
<tr>
<td>SuiteScript 2.0 Custom Forms</td>
<td>315</td>
</tr>
<tr>
<td>Sample Custom Form Script</td>
<td>321</td>
</tr>
<tr>
<td>SuiteScript 2.0 Custom List Pages</td>
<td>336</td>
</tr>
<tr>
<td>Sample Custom List Page Script</td>
<td>337</td>
</tr>
<tr>
<td>SuiteScript 2.0 Working with UI Objects</td>
<td>341</td>
</tr>
<tr>
<td>Custom UI development</td>
<td>342</td>
</tr>
<tr>
<td>UI Component Overview</td>
<td>342</td>
</tr>
<tr>
<td>SuiteScript 2.0 UI Modules</td>
<td>344</td>
</tr>
<tr>
<td>Using HTML</td>
<td>344</td>
</tr>
<tr>
<td>Creating Custom Assistants</td>
<td>349</td>
</tr>
</tbody>
</table>
SuiteScript 2.0 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 on both the client and server sides, as well as robust debugging.

The latest version, SuiteScript 2.0, extends the capabilities provided by the initial SuiteScript 1.0 version, with an API architecture familiar to JavaScript developers. SuiteScript 2.0'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 2.0 is designed to support all standard JavaScript. The supplied SuiteScript 2.0 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 2.0 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 2.0 also provides asynchronous client-side processing through promises, map/reduce scripts that provide a structured framework for server-side 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 2.0 API:

- SuiteScript 2.0 Hello World
- SuiteScript 2.0 Script Basics
- SuiteScript 2.0 Anatomy of a Script
- SuiteScript 2.0 Script Creation Process
- SuiteScript 2.0 Advantages
- SuiteScript 2.0 Terminology
- SuiteScript 2.0 Developer Resources
- SuiteScript 2.0 Versioning Guidelines

SuiteScript 2.0 Hello World

SuiteScript 2.0 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 2.0 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 One: Enable the Feature
- Step Two: Create the Script File
- Step Three: Upload the Script File to NetSuite
- Step Four: Create a Script Record and Script Deployment Record
- Step Five: 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 2.0 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 One: Enable the Feature.
SuiteScript 2.0 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 2.0, 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 2.0 Script Types:

- The **SuiteScript 2.0 Client Script Type** is designed for scripts that should run in the browser.
- The **SuiteScript 2.0 Scheduled Script Type** is for server-side scripts that should run at a specific time or on a recurring schedule.
- The **SuiteScript 2.0 RESTlet Script Type** is for server-side 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 2.0 Modules**

SuiteScript 2.0 has a Modular Architecture. As one indication of this, all SuiteScript 2.0 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 2.0 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 2.0, 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 2.0 Custom Modules.
Step One: 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 Two: 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 Two: 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.
These tags reflect the following:

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

Every SuiteScript 2.0 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 2.0 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.

```javascript
... define(['N/ui/dialog'],

  // In Step 4, you put additional code here.

):;
```

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`.

```javascript
... function(dialog) {

  // In Step 5, you put additional code here.

}...
```

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`.

```javascript
... function helloWorld() {

  // In steps 6-10, you put additional code here.

}...
```

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 2.0 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`.

```javascript
var options = {
    title: 'Hello!',
    message: 'Hello, World!'
};
```

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 2.0.

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.

```javascript
try {
    // In steps 8 and 9, you put additional code here.
}
catch (e) {
    // In Step 10, you put additional code here.
}
```

8. 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`.

```javascript
dialog.alert(options);
```

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.
9. 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.

   ```javascript
   ... 
   log.debug({
     title: 'Success',
     details: 'Alert displayed successfully'
   });
   ...
   
   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 Five: Test the Script.

10. 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.

   ```javascript
   ... 
   log.error({
     title: e.name,
     details: e.message
   });
   ...
   
   11. Immediately after the entry point function, add a return statement. In every SuiteScript 2.0 entry point script, the return statement must include at least one line that has two components:

   ```javascript
   ...
   return {
     pageInit: helloWorld
   };
   ...
   
   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.

```javascript
/**
 * @NApiVersion 2.0
 * @NScriptType ClientScript
 */

define(['N/ui/dialog'],

  function (dialog) {

    function helloWorld() {

      var options = {
        title: 'Hello!',
        message: 'Hello, World!'
      };

      try {
        dialog.alert(options);

        log.debug (
          {
            title: 'Success',
            details: 'Alert displayed successfully'
          }
        );

      } catch (e) {

        log.error (
          {
            title: e.name,
            details: e.message
          }
        );

      }

    return {
      pageInit: helloWorld
    };

  });

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

Step Three: 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. 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 Four: 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.
   Note that, 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. (Note that if you wanted the dialog to appear when a different type of record loads, you could 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 wanted to change the deployment later and make this customization 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.

Step Five: 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 the corresponding **View** link.
   c. Click the **Execution Log** subtab.

   The subtab should show an entry similar to the following.

   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 2.0 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-side scripts that should execute when users take certain actions with records. To try a simple user event script, see **SuiteScript 2.0 User Event Script Tutorial**.
SuiteScript 2.0 Script Basics

Certain components are common to all SuiteScript 2.0 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 2.0 Hello World, paying particular attention to the Key Concepts section.

Modular Architecture

SuiteScript 2.0 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 2.0 entry point and custom module script must be structured so that it creates a module. If you have used SuiteScript 2.0 before, you may not have realized that your scripts were creating modules. However, every SuiteScript 2.0 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 2.0 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 2.0 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 2.0, consider reviewing the Asynchronous Module Definition (AMD) Specification, which SuiteScript 2.0 uses. However, you do not have to be deeply familiar with AMD before you can use SuiteScript 2.0.

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 2.0, 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

Many SuiteScript 2.0 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:

```javascript
var myObject = {
    fieldId: 'Hello!'
    value: 'Hello, World!'
};
myRecord.setValue(myObject);
```

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

```javascript
myRecord.setValue={['Hello!'
    value: 'Hello, World!'
]);
```

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 2.0 User
Event Script Tutorial.

# Context Objects Passed to Standard and Custom Entry Points

Every SuiteScript 2.0 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 2.0 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 2.0 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 2.0 Custom Module Tutorial.

Note: To see an example of how the components described in this topic appear in scripts, review SuiteScript 2.0 Anatomy of a Script and SuiteScript 2.0 Anatomy of a Custom Module Script.

SuiteScript 2.0 Anatomy of a Script

All SuiteScript 2.0 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.
SuiteScript 2.0 Anatomy of a Script

Note: For help with the terms defined in this topic, review SuiteScript 2.0 Hello World.

<table>
<thead>
<tr>
<th>General Area</th>
<th>Callout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 — JSDoc tags</td>
<td>2 and 3</td>
<td>The @NapiVersion tag, which is required in an entry point script, and its value. Valid values are 2.0, 2.x, and 2.X.</td>
</tr>
<tr>
<td></td>
<td>4 and 5</td>
<td>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.</td>
</tr>
<tr>
<td>General Area</td>
<td>Callout</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>1 — define statement</td>
<td>6</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>The define function’s second argument, which is a callback function.</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>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.</td>
</tr>
<tr>
<td>9, 10, and 11</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>12, 13, and 14</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>The callback function’s return statement.</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>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).</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>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.</td>
</tr>
</tbody>
</table>

### SuiteScript 2.0 Script Creation Process

The following is a very basic process flow for SuiteScript 2.0 script creation. For a more detail explanation with a sample script, see SuiteScript 2.0 Hello World.

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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>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.</td>
<td>Modular Architecture SuiteScript 2.0 Global Objects</td>
</tr>
<tr>
<td>2</td>
<td>Add required JSDoc tags to your entry point script.</td>
<td>SuiteScript 2.0 JSDoc Validation</td>
</tr>
<tr>
<td>3</td>
<td>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. <strong>Important:</strong> 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.</td>
<td>SuiteScript 2.0 Script Types</td>
</tr>
<tr>
<td>4</td>
<td>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.</td>
<td>SuiteScript 2.0 Global Objects</td>
</tr>
</tbody>
</table>
SuiteScript 2.0 Script Creation Process

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Upload and deploy your script to NetSuite.</td>
<td>SuiteScript 2.0 Entry Point Script Creation and Deployment</td>
</tr>
</tbody>
</table>

### SuiteScript 2.0 Advantages

SuiteScript 2.0 is a complete redesign of the SuiteScript model. This topic goes over several of the advantages SuiteScript 2.0 has over SuiteScript 1.0.

**Note:** Use SuiteScript 2.0 for new scripts that you develop, and consider converting your SuiteScript 1.0 scripts to SuiteScript 2.0. 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 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 2.0 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 2.0 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 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 2.0 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 topics:

- SuiteScript 2.0 Hello World
- SuiteScript 2.0 Script Basics
- SuiteScript 2.0 Modules
- SuiteScript 2.0 Entry Point Script Creation and Deployment
- SuiteScript 2.0 Custom Modules

**Note:** SuiteScript 2.0 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 2.0 gives you built-in dependency management. With SuiteScript 2.0, you define the SuiteScript 2.0 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 topics:

# SuiteScript 2.0 Hello World
# SuiteScript 2.0 Script Basics
# SuiteScript 2.0 Entry Point Script Creation and Deployment
# SuiteScript 2.0 Custom Modules

Modern Programming Syntax and Behavior

The underlying design principle of SuiteScript 2.0 is that SuiteScript 2.0 === JavaScript. This results in a decreased learning curve for experienced 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 2.0 is designed to support all standard JavaScript. The supplied SuiteScript 2.0 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 2.0 is modeled to look and behave like modern JavaScript. To meet that objective, SuiteScript 2.0 methods and objects are not prefixed with nlapi and nlobj. This change also reflects the modular organization of SuiteScript 2.0. SuiteScript 1.0 methods and objects respectively belong to the nlapi and nlobj namespaces. SuiteScript 2.0 methods and objects are encapsulated within various modules.

# Usage of properties and enumerations: SuiteScript 2.0 adopts the usage of properties and enumerations. Most SuiteScript 1.0 getter and setter methods are replaced with properties. Enumerations encapsulate common constants (for example, standard record types).

Note: JavaScript does not include an enumeration type. The SuiteScript 2.0 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 2.0 begins at 0.

For additional information, see the following topics:

# SuiteScript 2.0 Hello World
# SuiteScript 2.0 Entry Point Script Creation and Deployment
# SuiteScript 2.0 Custom Modules

Functionality Enhancements

The following enhancements are exclusive to SuiteScript 2.0:

# Map/Reduce Script Type
# Asynchronous Client Side Processing (Promises)
# SFTP File Transfer API
# Cache API
# SuiteScript 2.0 Advantages

- Search Pagination API
- Flat File Streaming API
- Expanded Support for HTTP Content Type Headers
- New Encryption/Encoding Functionality

## Map/Reduce Script Type

SuiteScript 2.0 introduces a new server-side script type based on the map/reduce model. Map/ reduce scripts provide a structured framework for server-side 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 2.0 Map/Reduce Script Type](#).

## Asynchronous Client Side 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 2.0, all client scripts support the use of promises. With promises, developers can write asynchronous code that is intuitive and efficient. SuiteScript 2.0 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 2.0 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 2.0 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 2.0 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 2.0 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 2.0 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 2.0 adds enhanced encryption, decryption, and hashing functionality with the N/crypto Module. Additional encoding functionality is exposed in the N/encode Module.

SuiteScript 2.0 Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom Module</td>
<td>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. For additional information on custom modules, see SuiteScript 2.0 Custom Modules.</td>
</tr>
<tr>
<td>Deferred Dynamic Mode</td>
<td>See the definition for standard mode.</td>
</tr>
<tr>
<td>Dynamic Mode</td>
<td>See the definition for standard mode.</td>
</tr>
<tr>
<td>Entry Point</td>
<td>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. For additional information on entry points, see SuiteScript 2.0 Entry Point Script Validation, SuiteScript 2.0 Hello World, and SuiteScript 2.0 Script Types.</td>
</tr>
<tr>
<td>Entry Point Function</td>
<td>A function that executes when an entry point is invoked. For additional information on entry point functions, see SuiteScript 2.0 Entry Point Script Validation and SuiteScript 2.0 Hello World.</td>
</tr>
<tr>
<td>Entry Point Script</td>
<td>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. For additional information on entry point scripts, see SuiteScript 2.0 Entry Point Script Validation and SuiteScript 2.0 Hello World.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Enum</td>
<td>JavaScript does not include an enumeration type. The SuiteScript 2.0 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.</td>
</tr>
<tr>
<td>Governance</td>
<td>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. For additional information on governance, see the help topic SuiteScript Governance.</td>
</tr>
</tbody>
</table>
| Script        | A script refers to an aggregate of  
# An entry point script file  
# All custom modules used by the entry point script  
# The script record associated with the entry point script  
For additional information on the components that make up a script, see SuiteScript 2.0 Entry Point Script Validation, SuiteScript 2.0 Custom Modules, Script Record Creation, and SuiteScript 2.0 Hello World. |
| Script Deployment | A script deployment determines a portion of the associated script’s runtime behavior. The settings you can define on a script deployment record include:  
# The record types the script executes against  
# When the script is executed  
# The audience and role restrictions  
# The script log levels  
The settings found on the deployment record vary based on script type. For additional information on script deployments, see SuiteScript 2.0 Entry Point Script Creation and Deployment and SuiteScript 2.0 Hello World. |
| Script Type   | SuiteScript 2.0 scripts consist of several script types. Each script type is designed for a specific type of situation and specific types of triggering events.  
For additional information on script types, see SuiteScript 2.0 Script Types. |
| Standard Mode | Standard mode is also referred to as deferred dynamic mode.  
There are two modes you can operate in when you work with a record in SuiteScript: standard mode and dynamic mode.  
# 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.  
For additional information on standard and dynamic mode, see the help topic record.Record. |
| Usage Units   | See the definition for governance.                                                                                                                                                                         |

**SuiteScript 2.0 Developer Resources**

**SuiteScript 2.0 – Help**

See the following help sections for information about developing with SuiteScript 2.0:
# SuiteScript 2.0 – Internal Resources

The following internal resources are available in addition to the SuiteScript 2.0 help.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetSuite User Group</td>
<td>Official forum for the NetSuite community. Use the search field to find the SuiteScript 2.0 board.</td>
</tr>
<tr>
<td>SuiteScript 2.0: Extend NetSuite with JavaScript</td>
<td>Oracle Training Course</td>
</tr>
<tr>
<td>SuiteScript 2.0 for Experienced Developers</td>
<td>Oracle Training Course</td>
</tr>
<tr>
<td>SuiteScript Help Overview video</td>
<td>Video from the Oracle Learning Library</td>
</tr>
</tbody>
</table>

# SuiteScript 2.0 – External Resources

The following external resources are available in addition to the SuiteScript 2.0 help.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Specification</td>
<td>SuiteScript 2.0 implements its modular architecture with the Asynchronous Module Definition (AMD) specification. AMD is used to define and load JavaScript modules and their dependencies.</td>
</tr>
<tr>
<td>RequireJS</td>
<td>RequireJS also implements the AMD specification.</td>
</tr>
<tr>
<td><a href="https://www.promisejs.org/">https://www.promisejs.org/</a></td>
<td>Tutorials on JavaScript promises</td>
</tr>
<tr>
<td>Eloquent JavaScript</td>
<td>Free online JavaScript book</td>
</tr>
<tr>
<td>You Don’t Know JS</td>
<td>Free online JavaScript book series</td>
</tr>
<tr>
<td>Object-oriented JavaScript for Beginners</td>
<td>MDN article on object oriented JavaScript</td>
</tr>
<tr>
<td>StackOverflow – SuiteScript 2.0</td>
<td>Online community for developers</td>
</tr>
</tbody>
</table>

# SuiteScript Versioning Guidelines

See the following sections for information about SuiteScript versioning:
SuiteScript 2.1 Beta

In NetSuite 2019.2, a new version of SuiteScript is available as a beta feature. 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 6 (ES6) edition of the ECMAScript specification. It supports all standard 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 the help topic SuiteScript 2.1 Beta.

SuiteScript Versioning

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

<table>
<thead>
<tr>
<th>Version Type</th>
<th>Numbering Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major</td>
<td>Version 2.0, 3.0, 4.0</td>
<td>Major versions of SuiteScript include significant functionality changes and improvements. Major versions are not backward compatible with previously released versions.</td>
</tr>
<tr>
<td>Minor</td>
<td>Version 2.0, 2.1, 2.2</td>
<td>Minor versions of SuiteScript include enhancements to existing features. 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.</td>
</tr>
<tr>
<td>Patch</td>
<td>Does not apply</td>
<td>Patch versions of SuiteScript are included with regular NetSuite bug fix releases. Patch versions are backward compatible with all versions released since the last major version.</td>
</tr>
</tbody>
</table>

You can also specify a SuiteScript version of 2.x in your scripts. The 2.x value always represents the latest version of SuiteScript that is generally available. It does not represent any versions that are released as beta features. For example, in NetSuite 2019.2, SuiteScript 2.1 is being released as a beta feature. In this release, the 2.x value indicates that a script uses SuiteScript 2.0 because SuiteScript 2.1 is a beta feature and not generally available. When SuiteScript 2.1 is generally available in a future release, the 2.x value will indicate that a script uses SuiteScript 2.1. For more information about specifying a version for your script, see SuiteScript 2.0 JSDoc Validation.

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 2.0 Map/Reduce Script Type for more information.
SuiteScript 2.0 Script Types

The following script types are supported:

<table>
<thead>
<tr>
<th>Script Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SuiteScript 2.0 Bundle Installation Script Type</td>
<td>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. Bundle installation scripts execute on the server.</td>
</tr>
<tr>
<td>SuiteScript 2.0 Client Script Type</td>
<td>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. Client scripts execute in the user’s browser.</td>
</tr>
<tr>
<td>SuiteScript 2.0 Map/Reduce Script Type</td>
<td>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. Map/reduce scripts execute on the server.</td>
</tr>
<tr>
<td>SuiteScript 2.0 Mass Update Script Type</td>
<td>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. Mass update scripts execute on the server.</td>
</tr>
<tr>
<td>SuiteScript 2.0 Portlet Script Type</td>
<td>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. Portlet scripts execute on the server.</td>
</tr>
<tr>
<td>SuiteScript 2.0 RESTlet Script Type</td>
<td>RESTlets can be used to define custom RESTful integrations to NetSuite. RESTlets execute on the server.</td>
</tr>
<tr>
<td>SuiteScript 2.0 Scheduled Script Type</td>
<td>Scheduled scripts can be scheduled for submission or submitted on an on-demand basis. Scheduled scripts execute on the server.</td>
</tr>
<tr>
<td>SuiteScript 2.0 Suitelet Script Type</td>
<td>Suitelets enable the creation of dynamic web content and build NetSuite-looking pages. Suitelets can be used to implement custom front and backends. Suitelets execute on the server.</td>
</tr>
<tr>
<td>SuiteScript 2.0 User Event Script Type</td>
<td>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. User event scripts execute on the server.</td>
</tr>
<tr>
<td>SuiteScript 2.0 Workflow Action Script Type</td>
<td>Workflow action scripts allow you to create custom actions that are defined on a record in a workflow. Workflow action scripts execute on the server.</td>
</tr>
<tr>
<td>SuiteScript 2.0 SDF Installation Script Type</td>
<td>SDF installation scripts are automatically executed when a SuiteApp project is deployed.</td>
</tr>
</tbody>
</table>
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:

```javascript
define(['N/runtime'], function (runtime) {
    function pageInit(context) {
        var strField = runtime.getCurrentScript().getParameter('SCRIPT', 'custscript_case_field');
        ...
    }
});
```

# 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-side 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-side 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).
SuiteScript 2.0 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 uninstall. 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, upgraded, or uninstalled. Each bundle installation script can contain triggers to be executed before install, after install, before update, after update, and after uninstall.

Bundle installation script failures terminate bundle installations, updates, or uninstalls. Bundle installation scripts can include their own error handling in addition to errors thrown by SuiteBundler and the SuiteScript engine. An error thrown by a bundle installation script returns an error code of Installation Error followed by the text defined by the script author. For information about scripting with bundle installation scripts, see SuiteScript 2.0 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.

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.

For help with writing scripts in SuiteScript 2.0, see SuiteScript 2.0 Hello World and SuiteScript 2.0 Entry Point Script Creation and Deployment.

```javascript
/**
 * @NApiVersion 2.0
 * @NScriptType BundleInstallationScript
 */
define([ 'N/runtime' ], function (runtime) {
  function checkPrerequisites() {
    if (!runtime.isFeatureInEffect({
      feature: 'TIMEOFFMANAGEMENT'
    })) {
      throw 'The TIMEOFFMANAGEMENT feature must be enabled. ' +
      'Please enable the feature and try again.';
    }
    return {
      beforeInstall: function beforeInstall(params) {
        checkPrerequisites();
      },
      beforeUpdate: function beforeUpdate(params) {
        checkPrerequisites();
      }
    }
  }

```
SuiteScript 2.0 Bundle Installation Script Entry Points

<table>
<thead>
<tr>
<th>Script Entry Point</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>afterInstall</td>
<td>Executes after a bundle is installed for the first time in a target account.</td>
<td></td>
</tr>
<tr>
<td>afterUpdate</td>
<td>Executes after a bundle in a target account is updated.</td>
<td></td>
</tr>
<tr>
<td>beforeInstall</td>
<td>Executes before a bundle is installed for the first time in a target account.</td>
<td></td>
</tr>
<tr>
<td>beforeUninstall</td>
<td>Executes before a bundle is uninstalled from a target account.</td>
<td></td>
</tr>
<tr>
<td>beforeUpdate</td>
<td>Executes before a bundle in a target account is updated.</td>
<td></td>
</tr>
</tbody>
</table>

afterInstall

Description | Executes after a bundle is installed for the first time in a target account.
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>params.version</td>
<td>number</td>
<td>The version of the bundle that is being installed in the target account.</td>
<td>Version 2016 Release 1</td>
</tr>
</tbody>
</table>

afterUpdate

Description | Executes after a bundle in a target account is updated.
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>params.fromVersion</td>
<td>number</td>
<td>The version of the bundle that is currently installed in the target account.</td>
<td>Version 2016 Release 1</td>
</tr>
</tbody>
</table>
### beforeInstall

**Description**
Executes before a bundle is installed for the first time in a target account.

Calls to scheduled scripts are not supported.

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>params.toVersion</td>
<td>number</td>
<td>The version of the bundle that is being installed in the target account.</td>
<td>Version 2016 Release 1</td>
</tr>
</tbody>
</table>

### beforeUninstall

**Description**
Executes before a bundle is uninstalled from a target account.

Calls to scheduled scripts are not supported.

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>params.version</td>
<td>number</td>
<td>The version of the bundle that is being uninstalled from the target account.</td>
<td>Version 2016 Release 1</td>
</tr>
</tbody>
</table>

### beforeUpdate

**Description**
Executes before a bundle in a target account is updated.

Calls to scheduled scripts are not supported.

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>params.fromVersion</td>
<td>number</td>
<td>The version of the bundle that is currently installed in the target account.</td>
<td>Version 2016 Release 1</td>
</tr>
<tr>
<td>params.toVersion</td>
<td>number</td>
<td>The version of the bundle that is being installed in the target account.</td>
<td>Version 2016 Release 1</td>
</tr>
</tbody>
</table>

SuiteScript 2.0 Client Script Type

Client scripts are scripts that are executed by predefined event triggers in the client-side or browser. They can validate user-entered data and auto-populate fields or sublists at various form events. For details, see SuiteScript 2.0 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 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.

For additional information about SuiteScript 2.0 client scripts, see the following:

- SuiteScript 2.0 Client Script Reference
  - SuiteScript 2.0 Client Scripts on currentRecord
  - SuiteScript 2.0 Client Script Role Restrictions
  - SuiteScript 2.0 Remote Objects in Client Scripts
  - SuiteScript 2.0 Client Script Best Practices
SuiteScript 2.0 Client Script Sample

For help with writing scripts in SuiteScript 2.0, see SuiteScript 2.0 Hello World and SuiteScript 2.0 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%.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType ClientScript
 */
define(['N/error'],
    function (error) {
        function pageInit(context) {
            if (context.mode !== 'create')
                return;
            var currentRecord = context.currentRecord;
            currentRecord.setValue({
                fieldId: 'entity',
                value: 107
            });
        }
        function saveRecord(context) {
            var currentRecord = context.currentRecord;
            if (!currentRecord.getValue({
                fieldId: 'entity'
            }) || currentRecord.getLineCount({
                fieldId: 'entity'
            }) === 0)
                currentRecord.setFieldValue('entity', 107);
        }
    });
```
SuiteScript 2.0 Client Script Type

```javascript
sublistId: 'item'
}) < 1 )
throw error.create(
    name: 'MISSING_REQ_ARG',
    message: 'Please enter all the necessary fields on the salesorder before saving'
); return true;
}

function validateField(context) {
    var currentRecord = context.currentRecord;
    var sublistName = context.sublistId;
    var sublistFieldName = context.fieldId;
    var line = context.line;
    if (sublistName === 'item') {
        if (sublistFieldName === 'quantity') {
            if (currentRecord.getCurrentSublistValue({
                sublistId: sublistName,
                fieldId: sublistFieldName
            }) < 3 )
            currentRecord.setValue({
                fieldId: 'otherrefnum',
                value: 'Quantity is less than 3'
            });
            else
            currentRecord.setValue({
                fieldId: 'otherrefnum',
                value: 'Quantity accepted'
            });
        }
    }
    return true;
}

function fieldChanged(context) {
    var currentRecord = context.currentRecord;
    var sublistName = context.sublistId;
    var sublistFieldName = context.fieldId;
    var line = context.line;
    if (sublistName === 'item' && sublistFieldName === 'item')
    currentRecord.setValue({
        fieldId: 'memo',
        value: 'Item: ' + currentRecord.getCurrentSublistValue({
            sublistId: 'item',
            fieldId: 'item'
        }) + ' is selected'
    });
}

function postSourcing(context) {
    var currentRecord = context.currentRecord;
    var sublistName = context.sublistId;
    var sublistFieldName = context.fieldId;
    var line = context.line;
    if (sublistName === 'item' && sublistFieldName === 'item')
    if (currentRecord.getCurrentSublistValue({
        sublistId: sublistName,
        fieldId: sublistFieldName
```
SuiteScript 2.0 Client Script Type

```javascript
if (currentRecord.getCurrentSublistValue(
    sublistId: sublistName,
    fieldId: 'pricelevels'
)) !== '1-1'
currentRecord.setCurrentSublistValue(
    sublistId: sublistName,
    fieldId: 'pricelevels',
    value: '1-1'
);

function lineInit(context) {
    var currentRecord = context.currentRecord;
    var sublistName = context.sublistId;
    if (sublistName === 'partners')
        currentRecord.setCurrentSublistValue(
            sublistId: sublistName,
            fieldId: 'partner',
            value: '55'
        );
}

function validateDelete(context) {
    var currentRecord = context.currentRecord;
    var sublistName = context.sublistId;
    if (sublistName === 'partners')
        if (currentRecord.getCurrentSublistValue(
            sublistId: sublistName,
            fieldId: 'partner'
        ) === '55')
            currentRecord.setValue(
                fieldId: 'memo',
                value: 'Removing partner sublist'
            );
        return true;
}

function validateInsert(context) {
    var currentRecord = context.currentRecord;
    var sublistName = context.sublistId;
    if (sublistName === 'partners')
        if (currentRecord.getCurrentSublistValue(
            sublistId: sublistName,
            fieldId: 'contribution'
        ) !== '100.0%')
            currentRecord.setCurrentSublistValue(
                sublistId: sublistName,
                fieldId: 'contribution',
                value: '100.0%'
            );
        return true;
}

function validateLine(context) {
    var currentRecord = context.currentRecord;
    var sublistName = context.sublistId;
    if (sublistName === 'partners')
        if (currentRecord.getCurrentSublistValue(
            sublistId: sublistName,
            fieldId: 'value'
SuiteScript 2.0 Client Script Reference

# SuiteScript 2.0 Client Scripts on currentRecord
# SuiteScript 2.0 Client Script Role Restrictions
# SuiteScript 2.0 Remote Objects in Client Scripts
# SuiteScript 2.0 Client Script Best Practices

SuiteScript 2.0 Client Scripts on currentRecord

You use the currentRecord module to access the record that is active in the current client-side 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.

```javascript
sublistId: sublistName,
fieldId: 'contribution'
}) !== '100.0%')
currentRecord.setCurrentSublistValue(
sublistId: sublistName,
fieldId: 'contribution',
value: '100.0%'
});
return true;
}
function sublistChanged(context) {
    var currentRecord = context.currentRecord;
    var sublistName = context.sublistId;
    var op = context.operation;
    if (sublistName === 'item')
        currentRecord.setValue(
fieldId: 'memo',
value: 'Total has changed to ' + currentRecord.getValue(
fieldId: 'total'
) + ' with operation: ' + op
);
}
return {
    pageInit: pageInit,
    fieldChanged: fieldChanged,
    postSourcing: postSourcing,
    sublistChanged: sublistChanged,
    lineInit: lineInit,
    validateField: validateField,
    validateLine: validateLine,
    validateInsert: validateInsert,
    validateDelete: validateDelete,
    saveRecord: saveRecord
};
```
SuiteScript 2.0 Client Script Type

SuiteScript 2.0 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:

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType ClientScript
 */
define(['N/ui/message'],
function (msg) {
    function showErrorMessage(msgText) {
        var myMsg = msg.create({
            title: 'Cannot Save Record',
            message: msgText,
            type: msg.Type.ERROR
        });
        myMsg.show({
            duration: 5000
        });
    }

    function saveRec(context) {
        var rec = context.currentRecord;
        var currentDate = new Date();
        var oneWeekAgo = new Date(currentDate - 1000 * 60 * 60 * 24 * 7);

        // Validate transaction date is not older than current time by one week
        if (rec.getValue({
            fieldId: 'trandate'
        }) < oneWeekAgo) {
            showErrorMessage("Cannot save sales order with trandate one week old.");
            return false;
        }

        // Validate total is greater than 0
        if (rec.getValue({
            fieldId: 'total'
        }) <= 0) {
            showErrorMessage("Cannot save sales order with negative total amount.");
            return false;
        }
        return true;
    }

    return {
        saveRecord: saveRec
    };
});
```
SuiteScript 2.0 Client Script Type

```javascript
* @NApiVersion 2.x
* @NScriptType ClientScript
*/
define(['N/search'], function (search) {

    function getSalesRepEmail(context) {
        var salesRep = context.currentRecord.getValue({
            fieldId: 'salesrep'
        });
        var salesRepEmail = search.lookupFields({
            type: 'employee',
            id: salesRep,
            columns: ['email']
        });
        alert(JSON.stringify(salesRepEmail));
    }

    return {
        fieldChanged: getSalesRepEmail
    }
});
```

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 server-side 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 2.0 User Event Script Type.

SuiteScript 2.0 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.

```javascript
/**
* @NApiVersion 2.0
* @NScriptType ClientScript
*/
define(['N/record'], function (record) {
    function saveRecord(context) {
        var journalEntry = record.load({
            id: 6,
            type: record.Type.JOURNAL_ENTRY,
            isDynamic: true
        });
    }
});
```
SuiteScript 2.0 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 and record.setCurrentSublistValue 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:

```javascript
require(["N/error", "N/record", "N/file"],
    function (error, record, file) {
        ...
    });
```

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

```javascript
function runMe() {
    var k = 1;
    k *= (k + 9);
    debugger;
    console.log(k);
}
```

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-side scripts. For more information about the SuiteScript Debugger, see the help topic SuiteScript Debugger.

When debugging client-side 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.

### SuiteScript 2.0 Client Script Entry Points and API

<table>
<thead>
<tr>
<th>Script Entry Point</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fieldChanged</td>
<td>Executed when a field is changed by a user or client side call.</td>
</tr>
<tr>
<td>lineInit</td>
<td>Executed when an existing line is selected.</td>
</tr>
<tr>
<td>pageInit</td>
<td>Executed when the page completes loading or when the form is reset.</td>
</tr>
<tr>
<td>postSourcing</td>
<td>Executed on transaction forms when a field that sources information from another field is modified.</td>
</tr>
<tr>
<td>saveRecord</td>
<td>Executed after the submit button is pressed but before the form is submitted.</td>
</tr>
<tr>
<td>Script Entry Point</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>sublistChanged</td>
<td>Executed after a sublist has been inserted, removed, or edited.</td>
</tr>
<tr>
<td>validateDelete</td>
<td>Executed when removing an existing line from an edit sublist.</td>
</tr>
<tr>
<td>validateField</td>
<td>Executes when a field is about to be changed by a user or client side call.</td>
</tr>
<tr>
<td>validateInsert</td>
<td>Executed when you insert a line into an edit sublist.</td>
</tr>
<tr>
<td>validateLine</td>
<td>Executed before a line is added to an inline editor sublist or editor sublist.</td>
</tr>
</tbody>
</table>

**fieldChanged**

**Description**
Executed when a field is changed by a user or client side call.

This event may also execute directly through beforeLoad user event scripts.

The following sample tasks can be performed:

1. Provide the user with additional information based on user input.
2. Disable or enable fields based on user input.

For an example, see SuiteScript Client Script Sample.

**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.

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.currentRecord</td>
<td>currentRecord.CurrentRecord</td>
<td>The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.sublistId</td>
<td>string</td>
<td>The sublist ID name.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.fieldId</td>
<td>string</td>
<td>The field ID name.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.line</td>
<td>string</td>
<td>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.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>
SuiteScript 2.0 Client Script Type

### Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.column</td>
<td>string</td>
<td>The column number (zero-based index) if the field is in a matrix.</td>
<td>Version 2015</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the field is not in a matrix, the default value is undefined.</td>
<td>Release 2</td>
</tr>
</tbody>
</table>

### lineInit

**Description**

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.currentRecord</td>
<td>currentRecord.CurrentRecord</td>
<td>The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.sublistId</td>
<td>string</td>
<td>The sublist ID name.</td>
<td>Version 2015</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Release 2</td>
</tr>
</tbody>
</table>

### pageInit

**Description**

Executed when the page completes loading or when the form is reset.

The following sample tasks can be performed:

- 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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.currentRecord</td>
<td>currentRecord.CurrentRecord</td>
<td>The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.mode</td>
<td>string</td>
<td>The mode in which the record is being accessed. The mode can be set to one of the following values: copy, create, edit</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

Note: For a complete script example, see SuiteScript Client Script Sample.

postSourcing

Description

Executed on transaction forms when a field that sources information from another field is modified. This event behaves like a fieldChanged event after all dependent field values have been set. The event waits for any slaved or cascaded field changes to complete before calling the user defined function.

For an example, see SuiteScript Client Script Sample.

Note: The event is not triggered by field changes for a field that does not have any slaved fields.

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.currentRecord</td>
<td>currentRecord.CurrentRecord</td>
<td>The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.sublistId</td>
<td>string</td>
<td>The sublist ID name.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>
### saveRecord

**Description**  
Executed after the submit button is pressed but before the form is submitted.

The following sample tasks can be performed:

- Provide alerts before committing the data.
- Enable fields that were disabled with other functions.
- Redirect the user to a specified URL.

For an example, see SuiteScript Client Script Sample.

**Returns**  
Boolean true if the record is valid. Boolean false to suppress form submission.

**Since**  
Version 2015 Release 2

#### Parameters

<i>Note: The scriptContext parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.</i>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.fieldId</td>
<td>string</td>
<td>The field ID name.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

### sublistChanged

<i>Note: The sublistChanged function replaces the SuiteScript 1.0 function recalc.</i>

**Description**  
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

<i>Note: The scriptContext parameter is a JavaScript object. It is automatically passed to the script entry point by NetSuite.</i>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.currentRecord</td>
<td>currentRecord.CurrentRecord</td>
<td>The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>
### Parameter: `validateDelete`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scriptContext.sublistId</code></td>
<td><code>string</code></td>
<td>The sublist ID name.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

#### Description
- Executed when removing an existing line from an edit sublist.
- For an example, see [SuiteScript Client Script Sample](https://www.netsuite.com/).  
- **Returns**: Boolean true if the sublist line is valid. Boolean false to block the removal.
- **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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scriptContext.currentRecord</code></td>
<td><code>currentRecord.CurrentRecord</code></td>
<td>The current form record.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td><code>scriptContext.sublistId</code></td>
<td><code>string</code></td>
<td>The sublist ID name.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

### Parameter: `validateField`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scriptContext.currentRecord</code></td>
<td><code>currentRecord.CurrentRecord</code></td>
<td>The current form record.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td><code>scriptContext.sublistId</code></td>
<td><code>string</code></td>
<td>The sublist ID name.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

#### Description
- Executes when a field is about to be changed by a user or client side call.
- This event executes on fields added in beforeLoad user event scripts.
- The following sample tasks can be performed:
  - Validate field lengths.
  - Restrict field entries to a predefined format.
  - Restrict submitted values to a specified range
  - Validate the submission against entries made in an associated field.
- For an example, see [SuiteScript Client Script Sample](https://www.netsuite.com/).  
- **Note**: This event does not apply to dropdown select or check box fields.

#### Returns
- Boolean true if the field is valid. Boolean false to prevent the field value from changing.
Parameters

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.currentRecord</td>
<td>currentRecord.CurrentRecord</td>
<td>The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.sublistId</td>
<td>string</td>
<td>The sublist ID name.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.fieldId</td>
<td>string</td>
<td>The field ID name.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.line</td>
<td>string</td>
<td>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.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.column</td>
<td>string</td>
<td>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.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

validateInsert

Description
Executed when you insert a line into an edit sublist.
For an example, see SuiteScript Client Script Sample.

Returns
Boolean true if the sublist line is valid. Boolean false to block the insert.

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.currentRecord</td>
<td>currentRecord.CurrentRecord</td>
<td>The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>
validateLine

Description
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
Boolean true if the sublist line is valid. Boolean false to reject the operation.

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.currentRecord</td>
<td>currentRecord.CurrentRecord</td>
<td>The current form record. For more information about CurrentRecord object members, see the help topic CurrentRecord Object Members.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.sublistId</td>
<td>string</td>
<td>The sublist ID name.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

SuiteScript 2.0 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 2.0 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.

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 2.0 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 2.0 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 groundbreaking – the system makes a context object available to most SuiteScript 2.0 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 2.0 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 2.0 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 2.0 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.

<table>
<thead>
<tr>
<th>Entry point</th>
<th>Purpose of corresponding function</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>getInputData(inputContext)</td>
<td>To identify data that needs processing. The system passes this data to the next stage.</td>
<td>yes</td>
</tr>
<tr>
<td>map(mapContext)</td>
<td>To apply processing to each key/value pair provided, and optionally pass data to the next stage.</td>
<td></td>
</tr>
<tr>
<td>reduce(reduceContext)</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>summarize(summaryContext)</td>
<td>To retrieve data about the script’s execution, and take any needed actions with the output of the reduce stage.</td>
<td>no</td>
</tr>
</tbody>
</table>

For full details on the map/reduce entry points and their corresponding context objects, see SuiteScript 2.0 Map/Reduce Script Entry Points and API.

Map/Reduce Script Samples

See the following samples:

# Example: Script that Counts Characters
# Example: Script that Processes Invoices

Example: Script that Counts Characters

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 hardcoded 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.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType MapReduceScript
 */
define(['N/file'],
function (file) {
    // Define characters that should not be counted when the script performs its
    // analysis of the text.

    const PUNCTUATION_REGEXP = /
    [\ua2000-\ua206F\ua200e-\ua206e\ua2200-\ua22ff\uf000-\uf0ff
    \uf010-\uf03f\uf040-\ufe00\ufe0f\ufe20-\ufe2f
    \ufef0-\ufef6\ufe00-\u206f\000-\006\010-\01f\020-\02f
    \030-\03f\040-\06f\070-\09f\0a0-\0ff\100-\19f
    \200-\26f\300-\36f\400-\50f\520-\58f\600-\63f
    \650-\68f\700-\78f\7bf-\7ff\800-\9ff\a00-\a1f
    \a20-\a8f\b00-\b1f\b20-\b8f\c00-\c8f\d00-\d8f
    \e00-\e8f\f00-\ff0\gif-\gif\037\039\03c\03d
    \03e-\03f\041\042\043\044\045\046\047\051
    \052\053\054\055\056\057\063\064\065\066]
    /g;

    // Use the getInputData function to return two strings.

    function getInputData() {
        return "the quick brown fox
        jumps over the lazy dog." .split(" ");
    }

    // After the getInputData function is executed, the system creates the following
    // key/value pairs:
    //
    // key: 0, value: 'the quick brown fox'
    // key: 1, value: 'jumps over the lazy dog.'

    function map(context) {
        // Create a loop that examines each character in the string. Exclude spaces
        // and punctuation marks.

        for (var i = 0; context.value && i < context.value.length; i++)
            if (context.value[i] !== ' ' && !PUNCTUATION_REGEXP.test(context.value[i])) {
                // For each character, invoke the context.write() method. This method saves
                // a new key/value pair. For the new key, save the character currently being
                // examined by the loop. For each value, save the number 1.

                context.write({
                    key: context.value[i],
                    value: 1
                });
            }
    }
});
```
// After the map function has been invoked for the last time, the shuffle stage
// begins. In this stage, the system sorts the 35 key/value pairs that were saved
// by the map function during its two invocations. From those pairs, the shuffle
// stage creates a new set of key/value pairs, where the each key is unique. In
// this way, the number of key/value pairs is reduced to 25. For example, the map
// stage saved three instances of {'e','1'}. In place of those pairs, the shuffle
// stage creates one pair: {'e', ['1','1','1']}. These pairs are made available to
// the reduce stage through the context.key and context.values properties.

// The reduce function is invoked one time for each of the 25 key/value pairs
// provided.

    function reduce(context) {

        // Use the context.write() method to save a new key/value pair, where the new key
        // equals the key currently being processed by the function. This key is a letter
        // in the alphabet. Make the value equal to the length of the context.values array.
        // This number represents the number of times the letter occurred in the original
        // string.

        context.write(
            {
                key: context.key,
                value: context.values.length
            }
        );
    }

// The summarize stage is a serial stage, so this function is invoked only one
// time.

    function summarize(context) {

        // Log details about the script's execution.

        log.audit(
            {
                title: 'Usage units consumed',
                details: context.usage
            }
        );

        log.audit(
            {
                title: 'Concurrency',
                details: context.concurrency
            }
        );

        log.audit(
            {
                title: 'Number of yields',
                details: context.yields
            }
        );

        // Use the context object's output iterator to gather the key/value pairs saved
        // at the end of the reduce stage. Also, tabulate the number of key/value pairs
        // that were saved. This number represents the total number of unique letters
        // used in the original string.

        var text = "";
        var totalKeysSaved = 0;
        var contextKeyIterator = context.keyiterator();
        while (contextKeyIterator.hasNext()) {
            var key = contextKeyIterator.next();
            var values = context.values(key);
            text += 'key: ' + key + ', values: ' + values + '

            totalKeysSaved += values.length;
        }

        // Print the total number of unique keys saved.
        log.audit(
            {
                title: 'Total keys saved',
                details: totalKeysSaved
            }
        );

    }

// The summarize stage is a serial stage, so this function is invoked only one
// time.

    function summarize(context) {

        // Log details about the script's execution.

        log.audit(
            {
                title: 'Usage units consumed',
                details: context.usage
            }
        );

        log.audit(
            {
                title: 'Concurrency',
                details: context.concurrency
            }
        );

        log.audit(
            {
                title: 'Number of yields',
                details: context.yields
            }
        );

        // Use the context object's output iterator to gather the key/value pairs saved
        // at the end of the reduce stage. Also, tabulate the number of key/value pairs
        // that were saved. This number represents the total number of unique letters
        // used in the original string.

        var text = "";
        var totalKeysSaved = 0;
        var contextKeyIterator = context.keyiterator();
        while (contextKeyIterator.hasNext()) {
            var key = contextKeyIterator.next();
            var values = context.values(key);
            text += 'key: ' + key + ', values: ' + values + '

            totalKeysSaved += values.length;
        }

        // Print the total number of unique keys saved.
        log.audit(
            {
                title: 'Total keys saved',
                details: totalKeysSaved
            }
        );

    }
context.output.iterator().each(function(key, value) {
    text += (key + ' ' + value + '\n');
    totalKeysSaved++;
    return true;
});

// Log details about the total number of pairs saved.
log.audit(
    title: 'Unique number of letters used in string',
    details: totalKeysSaved
);

// Use the Nfile module to create a file that stores the reduce stage output,
// which you gathered by using the output iterator.

var fileObj = file.create({
    name: 'letter_count_result.txt',
    fileType: file.Type.PLAINTEXT,
    contents: text
});

fileObj.folder = -15;
var fileId = fileObj.save();

log.audit(
    title: 'Id of new file record',
    details: fileId
);
}

// Link each entry point to the appropriate function.

return {
    getInputData: getInputData,
    map: map,
    reduce: reduce,
    summarize: summarize
};
});

Example: Script that Processes Invoices

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.

```plaintext
/**
 * @NApiVersion 2.0
 * @NScriptType MapReduceScript
 */
define([ 'N/search' , 'N/record' , 'N/email' , 'N/runtime' , 'N/error' ],
function (search, record, email, runtime, error)
{
    function  handleErrorAndSendNotification(e, stage)
    {
        log.error( 'Stage: ' + stage + ' failed', e);

        var  author = -5;
        var  recipients = 'notify@company.com' ;
        var  subject = 'Map/Reduce script ' + runtime.getCurrentScript().id + ' failed for stage: ' + stage;
        var  body = 'An error occurred with the following information:
' +
                    'Error code: ' + e.name + ' 
' +
                    'Error msg: ' + e.message;

        email.send({
            author: author,
            recipients: recipients,
            subject: subject,
            body: body
        });
    }

    function  handleErrorIfAny(summary)
    {
        var  inputSummary = summary.inputSummary;
        var  mapSummary = summary.mapSummary;
        var  reduceSummary = summary.reduceSummary;

        if  (inputSummary.error)
        {
            var  e = error.create(
                name: 'INPUT_STAGE_FAILED',
                message: inputSummary.error
            );
```
SuiteScript 2.0 Map/Reduce Script Type

};
handleErrorAndSendNotification(e, 'getInputData');

handleErrorInStage('map', mapSummary);
handleErrorInStage('reduce', reduceSummary);

function handleErrorInStage(stage, summary)
{
    var errorMsg = [];
    summary.errors.iterator().each(function(key, value){
        var msg = 'Failure to accept payment from customer id: ' + key + '. Error was: ' + JSON.parse(value).message + ' in:
        errorMsg.push(msg);
        return true;
    });
    if (errorMsg.length > 0)
    {
        var e = error.create({
            name: 'RECORD_TRANSFORM_FAILED',
            message: JSON.stringify(errorMsg)
        });
        handleErrorAndSendNotification(e, stage);
    }
}

function createSummaryRecord(summary)
{
    try
    {
        var seconds = summary.seconds;
        var usage = summary.usage;
        var yields = summary.yields;

        var rec = record.create({
            type: 'customrecord_summary',
        });

        rec.setValue({
            fieldId: 'name',
            value: 'Summary for M/R script: ' + runtime.getCurrentScript().id
        });

        rec.setValue({
            fieldId: 'custrecord_time',
            value: seconds
        });
        rec.setValue({
            fieldId: 'custrecord_usage',
            value: usage
        });
        rec.setValue({
            fieldId: 'custrecord_yields',
            value: yields
        });
    }
rec.save();
}
catch(e)
{
  handleErrorAndSendNotification(e, 'summarize');
}
}

function applyLocationDiscountToInvoice(recordId) {
  var invoice = record.load({
    type: record.Type.INVOICE,
    id: recordId,
    isDynamic: true
  });

  var location = invoice.getText({
    fieldId: 'location'
  });

  var discount;
  if (location === 'East Coast')
    discount = 'Eight Percent';
  else if (location === 'West Coast')
    discount = 'Five Percent';
  else if (location === 'United Kingdom')
    discount = 'Nine Percent';
  else
    discount = '';

  invoice.setText({
    fieldId: 'discountitem',
    text: discount,
    ignoreFieldChange : false
  });
  log.debug(recordId + ' has been updated with location-based discount.');
  invoice.save();
}

function getInputData() {
  return search.create({
    type: record.Type.INVOICE,
    filters: [[status, search.Operator.IS, 'open']],
    columns: ['entity'],
    title: 'Open Invoice Search'
  });
}

function map(context) {
  var searchResult = JSON.parse(context.value);
  var invoiceId = searchResult.id;
  var entityId = searchResult.values.entity.value;
SuiteScript 2.0 Map/Reduce Script Type

```
applyLocationDiscountToInvoice(invoiceId);

context.write(

  key: entityId,
  value: invoiceId

);

}

function reduce(context) {

  var customerId = context.key;

  var custPayment = record.transform({
    fromType: record.Type.CUSTOMER,
    fromId: customerId,
    toType: record.Type.CUSTOMER_PAYMENT,
    isDynamic: true
  });

  var lineCount = custPayment.getLineCount('apply');
  for (var j = 0; j < lineCount; j++)
  {
    custPayment.selectLine({'
      sublistId: 'apply',
      line: j
    });
    custPayment.setCurrentSublistValue({
      sublistId: 'apply',
      fieldId: 'apply',
      value: true
    });
  }

  var custPaymentId = custPayment.save();

  context.write(

    key: custPaymentId

  );

}

function summarize(summary) {

  handleErrorIfAny(summary);
  createSummaryRecord(summary);
}

return {
  getInputData: getInputData,
  map: map,
  reduce: reduce,
  summarize: summarize
};
```
SuiteScript 2.0 Map/Reduce Script Reference

SuiteScript 2.0 Map/Reduce Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer size</td>
<td>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.</td>
<td># Buffer Size</td>
</tr>
<tr>
<td>deployment instance</td>
<td>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.</td>
<td># SuiteScript 2.0 Map/Reduce Script Submission</td>
</tr>
<tr>
<td>exitOnError</td>
<td>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.</td>
<td># exitOnError</td>
</tr>
<tr>
<td>function invocation</td>
<td>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 the map function is invoked 20 times, one time for each pair.</td>
<td># SuiteScript 2.0 Map/Reduce Script Type</td>
</tr>
<tr>
<td>getInputData</td>
<td>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.</td>
<td># SuiteScript 2.0 Map/Reduce Script Stages</td>
</tr>
<tr>
<td>governance</td>
<td>A system of rules governing your usage of NetSuite. Specific limits exist for every script type, including map/reduce.</td>
<td># SuiteScript 2.0 Map/Reduce Governance</td>
</tr>
<tr>
<td>hard limit</td>
<td>A type of map/reduce governance limit that, when violated, causes an interruption to the current function invocation. Make sure you follow the SuiteScript 2.0 Map/Reduce Script Best Practices to avoid problems with hard limits.</td>
<td># SuiteScript 2.0 Map/Reduce Governance</td>
</tr>
<tr>
<td>job</td>
<td>A unit of execution managed by SuiteCloud Processors. The processing of a map/reduce script is always handled by multiple job</td>
<td># SuiteScript 2.0 Map/Reduce Stages</td>
</tr>
</tbody>
</table>

SuiteScript 2.0 Map/Reduce Script Stages

Adding Logic to Handle Map/Reduce Best Repeats
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>jobs</td>
<td>At least one job is created to handle the processing of each stage.</td>
<td></td>
</tr>
<tr>
<td>map</td>
<td>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.</td>
<td># SuiteScript 2.0 Map/Reduce Script Stages&lt;br&gt;# map(mapContext)</td>
</tr>
<tr>
<td>map/reduce</td>
<td>A computing paradigm designed for the processing of large data sets.</td>
<td># <a href="https://en.wikipedia.org/wiki/MapReduce">https://en.wikipedia.org/wiki/MapReduce</a></td>
</tr>
<tr>
<td>map/reduce script type</td>
<td>A SuiteScript 2.0 script type based on the map/reduce paradigm.</td>
<td># Map/Reduce Key Concepts</td>
</tr>
<tr>
<td>parallel stage</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>priority</td>
<td>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.</td>
<td># Map/Reduce Script Deployment Record&lt;br&gt;# SuiteCloud Processors – Priority Levels</td>
</tr>
<tr>
<td>processor</td>
<td>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.</td>
<td># Concurrency Limit&lt;br&gt;# SuiteCloud Processors</td>
</tr>
<tr>
<td>processor pool</td>
<td>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.</td>
<td># Concurrency Limit&lt;br&gt;# SuiteCloud Processors</td>
</tr>
<tr>
<td>reduce</td>
<td>The fourth stage in the processing of a map/reduce script. A reduce function is optional, but your script must use either 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.</td>
<td># SuiteScript 2.0 Map/Reduce Script Stages&lt;br&gt;# reduce(reduceContext)</td>
</tr>
<tr>
<td>retryCount</td>
<td>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.</td>
<td># retryCount</td>
</tr>
<tr>
<td>script deployment</td>
<td>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.</td>
<td># SuiteScript 2.0 Map/Reduce Script Submission</td>
</tr>
<tr>
<td>serial stage</td>
<td>A type of stage that is processed in its entirety by only one job. The getInputData, shuffle, and summarize stages are serial stages.</td>
<td># SuiteScript 2.0 Map/Reduce Script Stages</td>
</tr>
<tr>
<td>shuffle</td>
<td>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</td>
<td># SuiteScript 2.0 Map/Reduce Script Stages</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
<td>More Information</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>a map function is used</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>soft limit</td>
<td>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.</td>
<td># SuiteScript 2.0 Map/Reduce Governance</td>
</tr>
<tr>
<td>SuiteCloud Processors</td>
<td>The engine that processes map/reduce and scheduled scripts.</td>
<td># SuiteCloud Processors</td>
</tr>
<tr>
<td>summarize</td>
<td>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.</td>
<td># SuiteScript 2.0 Map/Reduce Script Stages</td>
</tr>
<tr>
<td></td>
<td># summarize(summaryContext)</td>
<td></td>
</tr>
<tr>
<td>task</td>
<td>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.</td>
<td># SuiteScript 2.0 Map/Reduce Script Submission</td>
</tr>
<tr>
<td>yielding</td>
<td>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 2.0 does not contain an API that lets you force a script to yield.</td>
<td># Yield After Minutes</td>
</tr>
</tbody>
</table>

**SuiteScript 2.0 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 reduce 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 Example: Script that Processes Invoices.

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 2.0 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 2.0 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 2.0 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

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 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**.

**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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script</td>
<td>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.</td>
</tr>
<tr>
<td>Title</td>
<td>The user-defined name for the deployment.</td>
</tr>
<tr>
<td>ID</td>
<td>A unique identifier for the deployment. 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 delimiter. If you do not enter a value, the system automatically generates one. In both cases, the system automatically adds the prefix customdeploy to the ID created when the record is saved. Although not recommended, you can change the ID on an existing deployment by clicking the <strong>Change ID</strong> button.</td>
</tr>
</tbody>
</table>
| Deployed | 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:

- 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   | A value that determines how and when a script deployment can be submitted for processing. The primary options are:

- **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.

Note also that, regardless of the **Status**, the system submits the deployment for processing only if the Deployed box is checked.

For more details on this choice, see **Status**. |
| See Instances | 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. For details on working with the Map/Reduce Script Status page, see SuiteScript 2.0 Map/Reduce Script Status Page. |
| 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:

- **Debug** — suitable for scripts still being tested; this level shows all debug, audit, error and emergency messages. |
### SuiteScript 2.0 Map/Reduce Script Type

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td># Use Audit, Error, or Emergency — suitable for scripts in production mode. Of these three, Audit is the most inclusive and Emergency the most exclusive. For more details on each level, see Log Level.</td>
<td></td>
</tr>
<tr>
<td>Execute As Role</td>
<td>Indicates the role used to run the script. For map/reduce deployments, this value is automatically set to Administrator and cannot be edited.</td>
</tr>
<tr>
<td>Priority</td>
<td>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.</td>
</tr>
<tr>
<td>Important: You must understand SuiteCloud Processors before you change this setting. For details, see the help topic SuiteCloud Processors – Priority Levels.</td>
<td></td>
</tr>
<tr>
<td>Concurrency Limit</td>
<td>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.</td>
</tr>
<tr>
<td>Submit All Stages At Once</td>
<td>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.</td>
</tr>
<tr>
<td>Yield After Minutes</td>
<td>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.</td>
</tr>
<tr>
<td>Buffer Size</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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 through the deployment record’s Concurrency Limit field.

Note that 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 leave this field set to no number, then the maximum number of processors available to your account is used. The default value is 1.

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 use 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 2.0 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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Event</td>
<td>The map/reduce script deployment is submitted only once. Use this option if you want to schedule a future one-time submission.</td>
</tr>
<tr>
<td>Daily Event</td>
<td>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. 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.</td>
</tr>
</tbody>
</table>
### Field | Description
--- | ---
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

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**.

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 2.0 Map/Reduce Script Type**. If this script is your first, see **SuiteScript 2.0 Hello World** and **SuiteScript 2.0 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 timeframe. 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

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 2.0 Map/Reduce Script Type](#). If this script is your first, see [SuiteScript 2.0 Hello World](#) and [SuiteScript 2.0 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.

<i>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.</i>

### Submitting an On–Demand Map/Reduce Script Deployment from a Script

Map/reduce scripts can be submitted for processing on an on-demand basis from another server-side script. 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 2.0 Map/Reduce Script Type. If this script is your first, see SuiteScript 2.0 Hello World and SuiteScript 2.0 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-side script where you want to submit the map/reduce script, call task.create(options) to return a task.MapReduceScriptTask object:

```javascript
var scriptTask = task.create({ taskType: task.TaskType.MAP_REDUCE});
```

8. Set the MapReduceScriptTask.scriptId and MapReduceScriptTask.deploymentId properties:

```javascript
scriptTask.scriptId = 'customscript1';
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:
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

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 2.0 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.

## 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.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Usage limit per function invocation</th>
<th>Response to SSS_USAGE_LIMIT_EXCEEDED error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Input Data</td>
<td>10,000 units</td>
<td>The script ends the function invocation and exits the stage. It proceeds directly to the summarize stage.</td>
</tr>
<tr>
<td>Map</td>
<td>1,000 units</td>
<td>The response includes two parts. Note that you can configure the second part:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. The function invocation ends, even if its work on the current key/value pair is incomplete.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Reduce</td>
<td>5,000 units</td>
<td></td>
</tr>
<tr>
<td>Summarize</td>
<td>10,000 units</td>
<td>The script stops executing.</td>
</tr>
</tbody>
</table>

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 2.0 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 2.0 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 2.0 Map/Reduce Script Status Page

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 2.0 Map/Reduce Script Stages and SuiteScript 2.0 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 2.0 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 2.0 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.

Do not 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 server-side script debugging of a Map/Reduce script.

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 2.0 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 2.0 scripts, see the help topic Project Sample: SuiteScript 2.0 Unit Testing.
SuiteScript 2.0 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

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.

<table>
<thead>
<tr>
<th>Stage where interruption occurred</th>
<th>Script behavior</th>
<th>SSS_APP_SERVER_RESTART error code written to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Input Data</td>
<td>When the disruption occurs, the script stops executing. After the application server restarts, the system restarts the function.</td>
<td>n/a</td>
</tr>
<tr>
<td>Map</td>
<td>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: 1. The system evaluates the retryCount config setting. If retryCount is set to a value greater than 0 or if the retryCount 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 retryCount 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.</td>
<td># mapContext.errors — Contains the error codes recorded during previous attempts to process the current key/value pair. # mapSummary.errors — Contains all error codes recorded during the map stage.</td>
</tr>
<tr>
<td>Reduce</td>
<td>When the disruption occurs, the script stops executing. Any data that was saved during the invocation by using the context.write() method is discarded. Afterward, the system responds as follows: 1. The system evaluates the retryCount config setting. If retryCount is set to a value greater than 0, and the maximum number of retries has not yet</td>
<td># reduceContext.errors — Contains the error codes recorded during previous attempts to process the current key/value pair. # reduceSummary.errors — Contains all error codes recorded during the reduce stage.</td>
</tr>
<tr>
<td>Summarize</td>
<td>When the disruption occurs, the entire script stops executing. After the application server restarts, the system restarts the function.</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Stage where error occurred</th>
<th>Script behavior</th>
<th>Errors written to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Input Data</td>
<td>The script ends the function invocation and exits the stage. It proceeds directly to the summarize stage. This behavior cannot be configured.</td>
<td>inputSummary.error</td>
</tr>
<tr>
<td>Map</td>
<td>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 context.write() method is discarded. Afterward, the system responds as follows: 1. The system evaluates the retryCount config setting. If retryCount is set to a value greater than 0, and the maximum number of retries has not yet</td>
<td># mapContext.errors — Contains the error codes recorded during previous attempts to process the current key/value pair. # mapSummary.errors — Contains all error codes recorded during the map stage.</td>
</tr>
<tr>
<td>Stage where error occurred</td>
<td>Script behavior</td>
<td>Errors written to</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Reduce</td>
<td>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 exitOnError setting. If exitOnError 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. It does not do any further work on the pair it was processing when the error occurred.</td>
<td>reduceContext.errors — Contains the error codes recorded during previous attempts to process the current key/value pair. reduceSummary.errors — Contains all error codes recorded during the reduce stage.</td>
</tr>
<tr>
<td>Summarize</td>
<td>The script stops executing. This behavior cannot be configured.</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

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:

```javascript
// Add additional code.
...
return {
  retryCount: 2,
  exitOnError: true
}
```

SuiteScript 2.0
Logging Errors

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.

<table>
<thead>
<tr>
<th>Stage where error occurred</th>
<th>Property that contains data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Input Data</td>
<td><code>inputSummary.error</code></td>
</tr>
<tr>
<td>Map</td>
<td># <code>mapContext.errors</code> — Contains the error codes recorded during previous attempts to process the current key/value pair.</td>
</tr>
<tr>
<td></td>
<td># <code>mapSummary.errors</code> — Contains all error codes recorded during the map stage.</td>
</tr>
<tr>
<td>Reduce</td>
<td># <code>reduceContext.errors</code> — Contains the error codes recorded during previous attempts to process the current key/value pair.</td>
</tr>
<tr>
<td></td>
<td># <code>reduceSummary.errors</code> — Contains all error codes recorded during the reduce stage.</td>
</tr>
</tbody>
</table>

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.

```javascript
// Create a log entry showing each full serialized error thrown
// during previous attempts to process the current key/value pair.

mapContext.errors.iterator().each(function (key, error, executionNo) {
    log.error({
        title: 'Map error for key: ' + key + ', execution no ' + executionNo,
        details: error
    });

    return true;
});
```
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.

```javascript
// Create a log entry showing each full serialized error thrown // during previous attempts to process the current key/value pair.
reduceContext.errors.iterator().each(function (key, error, executionNo){
    log.error({
        title: 'Reduce error for key: ' + key + ', execution no ' + executionNo,
        details: error
    });
    return true;
});
```

Summarize Stage

You can use these snippets in a summarize function. They log data about errors thrown during previous stages.

```javascript
// If an error was thrown during the input stage, log the error.
if (summary.inputSummary.error)
    log.error({
        title: 'Input Error',
        details: summary.inputSummary.error
    });

// For each error thrown during the map stage, log the error, the corresponding key, // and the execution number. The execution number indicates whether the error was // thrown during the the first attempt to process the key, or during a // subsequent attempt.
summary.mapSummary.errors.iterator().each(function (key, error, executionNo){
    log.error({
        title: 'Map error for key: ' + key + ', execution no. ' + executionNo,
        details: error
    });
    return true;
});

// For each error thrown during the reduce stage, log the error, the corresponding // key, and the execution number. The execution number indicates whether the error was // thrown during the the first attempt to process the key, or during a // subsequent attempt.
summary.reduceSummary.errors.iterator().each(function (key, error, executionNo){
    log.error({
        title: 'Reduce error for key: ' + key + ', execution no. ' + executionNo,
        details: error
    });
});
```
Execution of Restarted Map/Reduce Stages

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 forcibly 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

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:

- Example 3: Design of a Robust Map/Reduce Script
- Example 4: A Problematic Map/Reduce Script
- Example 5: A Robust Map/Reduce Script
- Execution of Restarted Map/Reduce Stages
Example 3: Design of a Robust Map/Reduce Script

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 Execution of Restarted Map/Reduce Stages.

```javascript
define([], function () {
  return {
    getInputData: function (context) {
      if (context.isRestarted) {
        // I might do something differently
      }.
      return inputForNextStage; 
    },
    map: function (context) {
      if (context.isRestarted) {
        // I might do something differently
      }.
    },
    reduce: function (context) {
      if (context.isRestarted) {
        // I might do something differently
      }.
    },
    summarize: function (summary) {
      if (summary.isRestarted) {
```
Example 4: A Problematic Map/Reduce Script

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 Example 5: A Robust Map/Reduce Script 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.

```javascript
/*
 * @NApiVersion 2.0
 * @NScriptType mapreducescript
 */
define([ 'N/search', 'N/record' ],
    function (search, record) {
        return {
            getInputData: function (context) {
                var filter1 = search.createFilter({
                    name: 'mainline',
                    operator: search.Operator.IS,
                    values: true
                });
                var column1 = search.createColumn({ name: 'recordtype' });
                var srch = search.create({
                    type: search.Type.SALES_ORDER,
                    filters: [filter1],
                    columns: [column1]
                });
                return srch;
            },
            map: function (context) {
                var soEntry = JSON.parse(context.value);
                var so = record.load({
                    type: soEntry.values.recordtype,
                    id: context.key
                });
                // UPDATE so FIELDS
                so.save();
            }
        });
    });
```
context.write({
    key: soEntry.values.recordtype,
    value: context.key
  });

},

reduce: function (context)
{
  context.write({
    key: context.key,
    value: context.values.length
  });
},

summarize: function (summary)
{
  var totalRecordsUpdated = 0;
  summary.output.iterator().each(function (key, value)
  {
    log.audit({
      title: key + ' records updated',
      details: value
    });
    totalRecordsUpdated += parseInt(value);
    return true;
  });
  log.audit({
    title: 'Total records updated',
    details: totalRecordsUpdated
  });
}
};

Example 5: A Robust Map/Reduce Script

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.

```javascript
/**
 * @NApiVersion 2.0
 * @NScriptType mapreducescript
 */
define(['N/search', 'N/record'],
    function (search, record) {
        return {
            getInputData: function (context) {
                var filter1 = search.createFilter({
                    name: 'mainline',
                    operator: search.Operator.IS,
                    values: true
                });
                var filter2 = search.createFilter({
                    name: 'custbody_processed_flag',
                    operator: search.Operator.IS,
                    values: false
                });
                var column1 = search.createColumn({ name: 'recordtype' });
                var srch = search.create({
                    type: search.Type.SALES_ORDER,
                    filters: [filter1, filter2],
                    columns: [column1]
                });
                return srch;
            },
            map: function (context) {
                var soEntry = JSON.parse(context.value);
                var alreadyProcessed = false;
                if (context.isRestarted) {
                    var lookupResult = search.lookupFields({
                        type: soEntry.values.recordtype, id: context.key,
                        columns: ['custbody_processed_flag']
                    });
                    alreadyProcessed = lookupResult.custbody_processed_flag;
                }
                if (!alreadyProcessed) {
                    var so = record.load({
                        type: soEntry.values.recordtype, id: context.key
                    });
                    // UPDATE so FIELDS
                    so.setValue(
```
SuiteScript 2.0 Map/Reduce Script Type

SuiteScript 2.0 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:
SuiteScript 2.0 Map/Reduce Script Type

# 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:

```javascript
function getInputData() {
    return search.create({
        type: record.Type.INVOICE,
        filters: [['status', search.Operator.IS, 'open']],
        columns: ['entity'],
        title: 'Open Invoice Search'
    });
}
```

And the following snippet shows how to return a search object reference:

```javascript
... function getInputData () {
    // Reference a saved search with internal ID 1234.
    return {
        type: 'search',
        id: 1234
    };
}
...```

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 2.0 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 2.0 Map/Reduce Script Stages.

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Context Object</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getInputData(inputContext)</code></td>
<td>inputContext</td>
<td>Required</td>
<td>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.</td>
</tr>
<tr>
<td><code>map(mapContext)</code></td>
<td>mapContext</td>
<td>Optional, but if this entry point is skipped, the <code>reduce(reduceContext)</code> entry point is required.</td>
<td>Invokes the map stage.</td>
</tr>
</tbody>
</table>
SuiteScript 2.0 Map/Reduce Script Type

The following tables describe properties that are available through the map/reduce entry points.

**inputContext Object Members**

The following members are called on **inputContext**.

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Return Type / Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>inputContext.isRestarted</td>
<td>Boolean</td>
<td>Indicates whether the current invocation of the getInputData(inputContext) function represents a restart.</td>
</tr>
<tr>
<td>Object</td>
<td>inputContext.ObjectRef</td>
<td>object</td>
<td>An object that contains the input data.</td>
</tr>
</tbody>
</table>

The following members are called on the **inputContext.ObjectRef**.

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Return Type / Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>ObjectRef.id</td>
<td>string</td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>ObjectRef.type</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

**mapContext Object Members**

The following members are called on **mapContext**.

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Return Type / Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>mapContext.isRestarted</td>
<td>Boolean</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>mapContext.executionNo</td>
<td>property</td>
<td>Indicates whether the current invocation of the map(mapContext) function represents the first or a subsequent attempt to process the current key/value pair.</td>
</tr>
<tr>
<td>Member Type</td>
<td>Name</td>
<td>Return Type / Value Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------</td>
<td>--------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>mapContext.errors</td>
<td>iterator</td>
<td>Holds serialized errors that were thrown during previous attempts to execute the map(mapContext) function on the current key/value pair.</td>
</tr>
<tr>
<td></td>
<td>mapContext.key</td>
<td>string</td>
<td>The key to be processed during the current invocation of the map(mapContext) function.</td>
</tr>
<tr>
<td></td>
<td>mapContext.value</td>
<td>string</td>
<td>The value to be processed during the current invocation of the map(mapContext) function.</td>
</tr>
<tr>
<td>Method</td>
<td>mapContext.write(options)</td>
<td>void</td>
<td>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.</td>
</tr>
</tbody>
</table>

**reduceContext Object Members**

The following members are called on reduceContext.

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Return Type / Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>reduceContext.isRestarted</td>
<td>Boolean</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>reduceContext.executionNo</td>
<td>number</td>
<td>Indicates whether the current invocation of the reduce(reduceContext) function represents the first or a subsequent attempt to process the current key/value pair.</td>
</tr>
<tr>
<td></td>
<td>reduceContext.errors</td>
<td>iterator</td>
<td>Holds serialized errors that were thrown during previous attempts to execute the the reduce(reduceContext) function on the current key and its associated values.</td>
</tr>
<tr>
<td></td>
<td>reduceContext.key</td>
<td>string</td>
<td>The input key to process during the reduce stage.</td>
</tr>
<tr>
<td></td>
<td>reduceContext.values</td>
<td>string[]</td>
<td>The key to be processed during the current invocation of the reduce(reduceContext) function.</td>
</tr>
<tr>
<td>Method</td>
<td>reduceContext.write(options)</td>
<td>void</td>
<td>Writes the reduce(reduceContext) function as key/value pairs. This data is passed to the summarize stage.</td>
</tr>
</tbody>
</table>

**summaryContext Object Members**

The following members are called on the summaryContext.

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>summaryContext.isRestarted</td>
<td>read-only Boolean</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>summaryContext.concurrency</td>
<td>number</td>
<td>The maximum concurrency number when running the map/reduce script.</td>
</tr>
<tr>
<td></td>
<td>summaryContext.dateCreated</td>
<td>Date</td>
<td>The time and day when the script began running.</td>
</tr>
<tr>
<td>Member Type</td>
<td>Name</td>
<td>Value Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------</td>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>summaryContext.seconds</td>
<td>number</td>
<td>The total number of seconds that elapsed during the processing of the script.</td>
<td></td>
</tr>
<tr>
<td>summaryContext.usage</td>
<td>number</td>
<td>The total number of usage units consumed during the processing of the script.</td>
<td></td>
</tr>
<tr>
<td>summaryContext.yields</td>
<td>number</td>
<td>The total number of yields that occurred during the processing of the script.</td>
<td></td>
</tr>
<tr>
<td>summaryContext.inputSummary</td>
<td>object</td>
<td>Object that contains data about the input stage.</td>
<td></td>
</tr>
<tr>
<td>summaryContext.mapSummary</td>
<td>object</td>
<td>Object that contains data about the map stage.</td>
<td></td>
</tr>
<tr>
<td>summaryContext.reduceSummary</td>
<td>object</td>
<td>Object that contains data about the reduce stage.</td>
<td></td>
</tr>
<tr>
<td>summaryContext.output</td>
<td>iterator</td>
<td>Iterator that contains the keys and values saved as the output of the reduce stage.</td>
<td></td>
</tr>
</tbody>
</table>

**inputSummary Object members**

The following members are called on `summaryContext.inputSummary`.

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>inputSummary.dateCreated</td>
<td>Date</td>
<td>The time and day when the <code>getInputData(inputContext)</code> function began running.</td>
</tr>
<tr>
<td></td>
<td>inputSummary.error</td>
<td>string</td>
<td>Holds serialized errors thrown from the <code>getInputData(inputContext)</code> function.</td>
</tr>
<tr>
<td></td>
<td>inputSummary.seconds</td>
<td>number</td>
<td>The total number of seconds that elapsed during execution of the <code>getInputData(inputContext)</code> function. This tally does not include idle time.</td>
</tr>
<tr>
<td></td>
<td>inputSummary.usage</td>
<td>number</td>
<td>The total number of usage units consumed by processing of the <code>getInputData(inputContext)</code> function.</td>
</tr>
</tbody>
</table>

**mapSummary Members**

The following members are called on `summaryContext.mapSummary`.

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>mapSummary.concurrency</td>
<td>number</td>
<td>Maximum concurrency number when running <code>map(mapContext)</code>.</td>
</tr>
<tr>
<td></td>
<td>mapSummary.dateCreated</td>
<td>Date</td>
<td>The time and day when the first invocation of <code>map(mapContext)</code> function began.</td>
</tr>
<tr>
<td></td>
<td>mapSummary.errors</td>
<td>iterator</td>
<td>Holds serialized errors thrown during the map stage.</td>
</tr>
<tr>
<td></td>
<td>mapSummary.keys</td>
<td>iterator</td>
<td>Holds the keys passed to the map stage by the <code>getInputData</code> stage.</td>
</tr>
<tr>
<td></td>
<td>mapSummary.seconds</td>
<td>number</td>
<td>The total number of seconds that elapsed during the map stage.</td>
</tr>
<tr>
<td></td>
<td>mapSummary.usage</td>
<td>number</td>
<td>The total number of usage units consumed during the map stage.</td>
</tr>
</tbody>
</table>
SuiteScript 2.0 Map/Reduce Script Type

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mapSummary.yields</td>
<td>number</td>
<td>The total number of yields that occurred during the map stage.</td>
</tr>
</tbody>
</table>

reduceSummary Members

The following members are called on summaryContext.reduceSummary.

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>reduceSummary.concurrency</td>
<td>number</td>
<td>Maximum concurrency number when running reduce(reduceContext).</td>
</tr>
<tr>
<td></td>
<td>reduceSummary.dateCreated</td>
<td>Date</td>
<td>The time and day when the first invocation of the reduce(reduceContext) function began.</td>
</tr>
<tr>
<td></td>
<td>reduceSummary.errors</td>
<td>iterator</td>
<td>Holds serialized errors thrown during the reduce stage.</td>
</tr>
<tr>
<td></td>
<td>reduceSummary.keys</td>
<td>iterator</td>
<td>Holds the keys passed to the reduce stage.</td>
</tr>
<tr>
<td></td>
<td>reduceSummary.seconds</td>
<td>number</td>
<td>The total number of seconds that elapsed during the reduce stage.</td>
</tr>
<tr>
<td></td>
<td>reduceSummary.usage</td>
<td>number</td>
<td>The total number of usage units consumed during the reduce stage.</td>
</tr>
<tr>
<td></td>
<td>reduceSummary.yields</td>
<td>number</td>
<td>The total number of yields that occurred during the reduce stage.</td>
</tr>
</tbody>
</table>

getInputData(inputContext)

**Description**
Marks the beginning of the script’s execution. The purpose of the input stage is to generate the input data.

Executes when the getInputData entry point is triggered. This entry point is required.

For information about the context object provided to this entry point, see inputContext.

**Note:** When returning a inputContext.ObjectRef, the supported types are search and file.

**Note:** When getInputData() returns a data structure with a non-string value, before the value is stored, it is converted to a JSON string with JSON.stringify().

Also see Passing Search Data to the getInputData Stage.

**Returns**

**Examples:**

# Array

```javascript
[ { a: 'b', c: 'd' } ]
```

# Object

```javascript
{
  a: { ... },
  b: { ... }
}
```
# search.Search Object

```javascript
search.load({
  id: 1234
})
```

# search.Search Object Reference

```javascript
{
  type: 'search',
  id: 1234
}
```

# file.File Object

```javascript
file.load({
  id: 1234
})
```

# file.File Object Reference

```javascript
{
  type: 'file',
  path: '/SuiteScripts/data/names.txt'
}
```

Important: When returning a file.File Object or Object Reference, consider the following:

- 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.

# query.Query Object

```javascript
query.load({
  id: 'custworkbook237'
})
```

# query.Query Object Reference

```javascript
{
  type: 'query',
  id: 'custworkbook237'
}
```

Since Version 2015 Release 2

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.

```javascript
// Add additional code
```
function getInputData {

  // Reference a saved search that returns a list of NetSuite records that
  // require processing - for example, sales orders that are pending fulfillment.
  return {
    type: 'search',
    id: 1234
  };

  // Add additional code
}

inputContext

<table>
<thead>
<tr>
<th>Object Description</th>
<th>This object passed to the <code>getInputData(inputContext)</code> entry point function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Since</td>
<td>Version 2016 Release 1</td>
</tr>
</tbody>
</table>

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.

```
// Add additional code
...
function getInputData {

  // Reference a saved search that returns a list of NetSuite records that
  // require processing - for example, sales orders that are pending fulfillment.
  return {
    type: 'search',
    id: 1234
  };

  // Add additional code
}
```

inputContext.isRestarted

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Indicates whether the current invocation of the <code>getInputData(inputContext)</code> function is the first.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Typically, the <code>getInputData</code> function is invoked one time only. However, if the function is interrupted by an application server restart, the system restarts the <code>getInputData</code> function.</td>
</tr>
<tr>
<td></td>
<td>If the value of this property is true, the <code>getInputData(inputContext)</code> has been restarted.</td>
</tr>
<tr>
<td></td>
<td>You can use this property to help make your script more robust.</td>
</tr>
<tr>
<td>Type</td>
<td>read-only Boolean</td>
</tr>
</tbody>
</table>

Since Version 2016 Release 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.

```javascript
... func getInputData(context) {
    if (context.isRestarted) {
        log.debug('GET_INPUT isRestarted', 'YES');
    } else {
        log.debug('GET_INPUT isRestarted', 'NO');
    }
    var extractSearch = search.load({ id: 'customsearch35' });
    return extractSearch;
}
...
```

`inputContext.ObjectRef`

**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 Version 2015 Release 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.

```javascript
...
{
    type: 'search',
    id: 1234 //search internal id
}
...
```

`ObjectRef.id`

**Property Description** The internal ID or script ID of the object. For example, the saved search ID.

**Type** string | number

**Since** Version 2015 Release 2
SuiteScript 2.0 Map/Reduce Script Type

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.

```json
...
{
  type: 'search',
  id: 1234 //search internal id
}
...
```

**ObjectRef.type**

<table>
<thead>
<tr>
<th>Property Description</th>
<th>The object's type.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>string</td>
</tr>
<tr>
<td>Values</td>
<td>'search'</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.

```json
...
{
  type: 'search',
  id: 1234 //search internal id
}
...
```

**map(mapContext)**

<table>
<thead>
<tr>
<th>Description</th>
<th>Executes when the map entry point is triggered.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>For information about the context object provided to this entry point, see mapContext.</td>
</tr>
<tr>
<td>Returns</td>
<td>Void</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapContext</td>
<td>Object</td>
<td>Required</td>
<td>Object that contains:</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Required / Optional</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>---------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
|           |      |                     | # 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. |

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 the `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.

```javascript
// Add additional code
...
function map(context)
{
  for (var i = 0; context.value && i < context.value.length; i++)
    if (context.value[i] !== ' ' && !PUNCTUATION_REGEXP.test(context.value[i]))
      {
        context.write(
          {
            key: context.value[i],
            value: 1
          });
      }
}
...
// Add additional code
```

mapContext

Object Description

Object that contains:

# 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.

For a complete list of this object’s methods and properties, see [mapContext Object Members](#).

Since

Version 2015 Release 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](#).
... function map(context) {
    for (var i = 0; context.value && i < context.value.length; i++)
        if (context.value[i] !== ' ' && !PUNCTUATION_REGEXP.test(context.value[i]))
            context.write({
                key: context.value[i],
                value: 1
            });
}
...

mapContext.isRestarted

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Property</strong></td>
<td>Indicates whether the <code>map(mapContext)</code> function has been invoked previously for the current key/value pair.</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>For an overview of events that can cause a restart, see System Response After a Map/Reduce Interruption.</td>
</tr>
<tr>
<td></td>
<td>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 mapContext.isRestarted value is true, your script should take the necessary actions to avoid duplicate processing. For examples, see Adding Logic to Handle Map/Reduce Restarts.</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Boolean</td>
</tr>
<tr>
<td><strong>Since</strong></td>
<td>Version 2016 Release 1</td>
</tr>
</tbody>
</table>

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.

```
// Add additional code.
...
function map(context) {
    if (context.isRestarted)
        {
            // Add logic designed to assess how much processing was completed for this key/value pair and react accordingly.
        }
    else
        {
            // Let full processing continue for the key/value pair.
        }
}
// Add additional code.
```

mapContext.executionNo

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Property</strong></td>
<td>Indicates whether the current invocation of the <code>map(mapContext)</code> function is the first or a subsequent invocation for the current key/value pair.</td>
</tr>
</tbody>
</table>
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 mapContext.executionNo 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.

### 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.

```javascript
// Add additional code
...

if (context.executionNo === 1) {
    // Permit full processing of the key/value pair.
}
else if (context.executionNo === 2) {
    // Take steps to check whether any processing was previously completed.
}
else {
    // Take other steps that might be necessary in the case of more than one previous attempt.
    ...

    // Add additional code
```

### mapContext.errors

**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**

Version 2018 Release 1

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator().each(parameters)</td>
<td>function</td>
<td>required</td>
<td>Executes one time for each error.</td>
</tr>
</tbody>
</table>
parameters

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteratorFunction(key, error, executionNo)</td>
<td>function</td>
<td>required</td>
<td>Provides logic to be executed during each iteration.</td>
</tr>
</tbody>
</table>

See also functionParameters.

functionParameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>string</td>
<td>optional</td>
<td>Represents the key/value pair that the map function was attempting to process when the error occurred.</td>
</tr>
<tr>
<td>error</td>
<td>string</td>
<td>optional</td>
<td>A serialization of the error thrown.</td>
</tr>
<tr>
<td>executionNo</td>
<td>number</td>
<td>optional</td>
<td>Indicates whether the error occurred during the first or a subsequent attempt to process the key/value pair.</td>
</tr>
</tbody>
</table>

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.

```javascript
// Add additional code
...

// Create a log entry showing each full serialized error, and the corresponding key.
context.errors.iterator().each(function (key, error, executionNo){
  log.error({
    title: 'Map error for key: ' + key + ', execution no ' + executionNo,
    details: error
  });
  return true;
});

// Log only the name and description of each error thrown.
context.errors.iterator().each(function (key, error, executionNo){
  var errorObject = JSON.parse(error);
  log.error({
    title: 'Reduce error for key ' + key + ', execution no ' + executionNo,
    details: errorObject.name + ': ' + errorObject.message
  });
  return true;
});

// Calculate and log the number of errors encountered.
var errorCount = 0;
context.errors.iterator().each(function() {
  errorCount ++;
});
```
```javascript
return true;
});

log.audit({
    title: 'Errors for map key: ' + context.key,
    details: 'Total number of errors: ' + errorCount
});
```

---

**mapContext.key**

<table>
<thead>
<tr>
<th>Property Description</th>
<th>The key to be processed during the map stage.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># If the input type is an array, the key is the index of the element.</td>
</tr>
<tr>
<td></td>
<td># If the input type is an object, the key is the key in the object.</td>
</tr>
<tr>
<td></td>
<td># 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.</td>
</tr>
<tr>
<td>Note:</td>
<td>Each key cannot exceed 4000 bytes.</td>
</tr>
<tr>
<td>Type</td>
<td>string</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

**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.

```javascript
// Add additional code
...

context.write({
    key: context.value[i],
    value: 1
});
...
// Add additional code
```

---

**mapContext.value**

<table>
<thead>
<tr>
<th>Property Description</th>
<th>The value to be processed during the map stage.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># If the input type is an array, the mapContext.value is the value in the element.</td>
</tr>
<tr>
<td></td>
<td># If the input type is an object, the mapContext.value is the value in the object.</td>
</tr>
<tr>
<td></td>
<td># If the input type is a result set, the the mapContext.value is a search.Result object converted to a JSON string by using JSON.stringify().</td>
</tr>
<tr>
<td>Note:</td>
<td>Each value cannot exceed 1 megabyte.</td>
</tr>
<tr>
<td>Type</td>
<td>string</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>
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.

```javascript
// Add additional code.
...

// Assume that the search result is a list of phone call records. This snippet parses the results
// and uses the values of the title and the message fields from each record.

var searchResult = JSON.parse(context.value);

var title = searchResult.values.title;
var message = searchResult.values.message;
...

// Add additional code.
```

### mapContext.write(options)

<table>
<thead>
<tr>
<th>Method Description</th>
<th>Writes the key/value pairs to be passed to the shuffle and then the reduce stage.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If your script includes both a map and reduce function, you must use this method in order for the reduce function to be invoked.</td>
</tr>
</tbody>
</table>

| Returns | Void |

| Since | Version 2015 Release 2 |

#### Parameters

- **options**:
  - **Type**: The options parameter is a JavaScript object.

#### Note:

The **options** parameter is a JavaScript object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.key</td>
<td>String or object. However, note that if you provide an object, the system calls JSON.stringify() on your input.</td>
<td>required</td>
<td>The key to write</td>
</tr>
<tr>
<td>options.value</td>
<td>String or object. However, note that if you provide an object, the system calls JSON.stringify() on your input.</td>
<td>required</td>
<td>The value to write</td>
</tr>
</tbody>
</table>

### 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.

```javascript
...  function map(context)
  {
      for (var i = 0; context.value &amp;&amp; i &lt; context.value.length; i++)
          if (context.value[i] !== ' ' &amp;&amp; '/PUNCTUATION_REGEXP.test(context.value[i]))
              {  context.write(
                  key: context.value[i],
                  value: 1
                );
              }
  }
...
reduce(reduceContext)

**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 values. 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:

- 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**

Version 2015 Release 2

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduceContext</td>
<td>Object</td>
<td>Required</td>
<td>Object that contains:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># The data to process during the reduce stage.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># Logic that lets you save data to pass to the summarize stage.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># Other properties you can use within the reduce function.</td>
</tr>
</tbody>
</table>

**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 the `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.

```javascript
function reduce(context)
{
    context.write({
        key: context.key,
        value: context.values.length
    });
}
```

---
reduceContext

<table>
<thead>
<tr>
<th>Object Description</th>
<th>Contains the key/value pairs to process during the reduce stage.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.

```javascript
... function reduce(context)
{
    context.write({
        key: context.key,
        value: context.values.length
    });
}
...
```

reduceContext.isRestarted

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Indicates whether the <code>reduce(reduceContext)</code> function has been invoked previously for the current key and values.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For an overview of events that can cause a restart, see System Response After a Map/Reduce Interruption.</td>
</tr>
<tr>
<td></td>
<td>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 reduceContext.isRestarted property is true, your script should take the necessary actions to avoid duplicate processing. For examples, see Adding Logic to Handle Map/Reduce Restarts.</td>
</tr>
<tr>
<td></td>
<td>Related properties include reduceContext.executionNo and reduceContext.errors.</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2016 Release 1</td>
</tr>
</tbody>
</table>

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.

```javascript
// Add additional code.
...
function reduce (context) {
    if (context.isRestarted) {
        // Add logic designed to assess how much processing was completed for this key
        // and react accordingly.
    } else {
        // Let full processing continue for the current key and its values.
    }
}
```
reduceContext.executionNo

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Indicates whether the current invocation of the <code>reduce(reduceContext)</code> function is the first, second, third, or fourth for the current key and its values.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For an overview of events that can cause a restart, see System Response After a Map/Reduce Interruption.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>For examples of how to write a robust map/reduce script, see Adding Logic to Handle Map/Reduce Restarts.</td>
</tr>
<tr>
<td></td>
<td>Related properties include <code>reduceContext.isRestarted</code> and <code>reduceContext.errors</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Since</td>
<td>Version 2018 Release 1</td>
</tr>
</tbody>
</table>

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.

```javascript
// Add additional code
...

// Add additional code

if (context.executionNo === 1) {
    // Permit full processing of the key/value pair.
}

else if (context.executionNo === 2) {
    // Take steps to check whether any processing was previously completed.
}

else {
    // Take other steps that might be necessary in the case of more than
    // one previous attempt.
}
...
// Add additional code
```

reduceContext.errors

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Holds serialized errors that were thrown during previous attempts to execute the reduce function on the current key and its values.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
SuiteScript 2.0 Map/Reduce Script Type

Members

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator().each(Parameters)</td>
<td>function</td>
<td>required</td>
<td>EXECUTES ONE TIME FOR EACH ERROR.</td>
</tr>
</tbody>
</table>

parameters

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteratorFunction(key, error, executionNo)</td>
<td>function</td>
<td>required</td>
<td>PROVIDES LOGIC TO BE EXECUTED DURING EACH ITERATION.</td>
</tr>
</tbody>
</table>

See also functionParameters.

functionParameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>string</td>
<td>optional</td>
<td>REPRESENTS THE KEY/VALUE PAIR THAT THE REDUCE FUNCTION WAS ATTEMPTING TO PROCESS WHEN THE ERROR OCCURRED.</td>
</tr>
<tr>
<td>error</td>
<td>string</td>
<td>optional</td>
<td>A SERIALIZATION OF THE ERROR THROWN.</td>
</tr>
<tr>
<td>executionNo</td>
<td>number</td>
<td>optional</td>
<td>INDICATES WHETHER THE ERROR OCCURRED DURING THE FIRST OR A SUBSEQUENT ATTEMPT TO PROCESS THE KEY.</td>
</tr>
</tbody>
</table>

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.

```
// Add additional code
...

// Create a log entry showing each full serialized error, and the corresponding key.
context.errors.iterator().each(function (key, error, executionNo){
    log.error({
        title: 'Reduce error for key: ' + key + ', execution no ' + executionNo,
        details: error
    });
    return true;
});

// Log only the name and description of each error thrown.
context.errors.iterator().each(function (key, error, executionNo){
    var errorObject = JSON.parse(error);
    log.error({
        title: 'Reduce error for key ' + key + ', execution no ' + executionNo,
        details: errorObject.name + ': ' + errorObject.message
    });
```

SuiteScript 2.0
// Calculate and log the number of errors encountered.

var errorCount = 0;
context.errors.iterator().each(function () {
    errorCount ++;
    return true;
});

log.audit (
    title: 'Errors for reduce key: ' + context.key,
    details: 'Total number of errors: ' + errorCount
);

reduceContext.key

<table>
<thead>
<tr>
<th>Property Description</th>
<th>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.</th>
</tr>
</thead>
</table>

Note: Each key cannot exceed 4000 bytes.
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:

- 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 `JSON.stringify()`.

**Note:** Each value cannot exceed 1 megabyte.

---

<table>
<thead>
<tr>
<th>Type</th>
<th>string[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

### 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](#).

```javascript
... 
function reduce(context)
{
    context.write(
        {
            key: context.key ,
            value: context.values.length
        });
}
... 
```

### reduceContext.write(options)

<table>
<thead>
<tr>
<th>Method Description</th>
<th>Writes key/value pairs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>Void</td>
</tr>
</tbody>
</table>

**Since** Version 2015 Release 2

### Parameters

**Note:** The options parameter is a JavaScript object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.key</td>
<td>String or object. However, note that if you provide an object, the system calls <code>JSON.stringify()</code> on your input.</td>
<td>required</td>
<td>The key to write.</td>
</tr>
<tr>
<td>options.value</td>
<td></td>
<td>required</td>
<td>The value to write.</td>
</tr>
</tbody>
</table>

### 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](#).

```javascript
... 
```
function reduce(context)
{
    context.write({
        key: context.key,
        value: context.values.length
    });
}

summarize(summaryContext)

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
Version 2015 Release 2

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>summary</td>
<td>summaryContext</td>
<td>Required</td>
<td>Holds statistics regarding the execution of a map/reduce script.</td>
</tr>
</tbody>
</table>

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.

... function summarize(summary)
{
    var type = summary.toString();
    log.audit(type + ' Usage Consumed', summary.usage);
    log.audit(type + ' Concurrency Number', summary.concurrency);
    log.audit(type + ' Number of Yields', summary.yields);
    var contents = '';
    summary.output.iterator().each(function (key, value)
    {
        contents += (key + ' ' + value + '
' );
        return true;
    });
    ...

summaryContext

Object Description
Holds statistics regarding execution of a map/reduce script.

Since
Version 2015 Release 2
SuiteScript 2.0 Map/Reduce Script Type

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.

```javascript
function summarize(summary) {
    var type = summary.toString();
    log.audit(type + ' Usage Consumed', summary.usage);
    log.audit(type + ' Concurrency Number', summary.concurrency);
    log.audit(type + ' Number of Yields', summary.yields);
    var contents = '';
    summary.output.iterator().each(function (key, value) {
        contents += (key + ' ' + value + '
');
        return true;
    });
}
```

summaryContext.isRestarted

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Indicates whether the summarize(summaryContext) function was invoked again.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>If the value is true, the current process invoked by summarize(summaryContext) 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.</td>
</tr>
<tr>
<td>Type</td>
<td>read-only Boolean</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2016 Release 1</td>
</tr>
</tbody>
</table>

Syntax

```javascript
... function summarize(summary) {
    if (summary.isRestarted) {
        log.debug('SUMMARY isRestarted', 'YES');
    } else {
        log.debug('SUMMARY isRestarted', 'NO');
    }
    log.debug('summarize', JSON.stringify(summary));
    ...}
```

summaryContext.concurrency

| 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

**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.

```javascript
... 
log.audit(type + ' Concurrency Number', summary.concurrency);
... 
```

**summaryContext.dateCreated**

<table>
<thead>
<tr>
<th>Property Description</th>
<th>The time and day when the map/reduce script began running.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Date</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

**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.

```javascript
... 
log.audit(type + ' Creation Date', summary.dateCreated);
... 
```

**summaryContext.seconds**

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Total seconds elapsed when running the map/reduce script.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>number</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

**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.

```javascript
... 
log.audit(type + ' Total seconds elapsed', summary.seconds);
... 
```

**summaryContext.usage**

| Property Description | Total number of usage units consumed when running the map/reduce script. |
### SummaryContext.yields

**Property Description**
Total number of yields when running the map/reduce script.

**Type**
number

**Since**
Version 2015 Release 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.

```javascript
... 
  log.audit(type + ' Usage Consumed', summary.usage);
  ...
```

### SummaryContext.inputSummary

**Object Description**
Holds statistics regarding the input stage.

**Since**
Version 2015 Release 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.

```javascript
... 
  log.audit(type + ' Number of Yields', summary.yields);
  ...
```

### InputSummary.dateCreated

**Property Description**
The time and day when `getInputData(inputContext)` began running.

**Type**
Date

**Since**
Version 2015 Release 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.

```javascript
... 
  logMetrics(summary.inputSummary);
  log.error('Input Error', summary.inputSummary.error);
  ...
```
Since Version 2015 Release 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.

```javascript
... 
log.audit('Creation Date', summary.inputSummary.dateCreated);
...
```

### inputSummary.seconds

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Total seconds elapsed when running <code>getInputData(inputContext)</code> (does not include idle time).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>number</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.

```javascript
... 
log.audit('Time Elapsed', summary.inputSummary.seconds);
...
```

### inputSummary.usage

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Total number of usage units consumed when running <code>getInputData(inputContext)</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>number</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.

```javascript
... 
log.audit('Usage', summary.inputSummary.usage);
...
```

### inputSummary.error

<table>
<thead>
<tr>
<th>Property Description</th>
<th>If applicable, holds a serialized error that is thrown from <code>getInputData(inputContext)</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>string</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>
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.

```plaintext
...
logMetrics(summary.inputSummary);
log.error('Input Error', summary.inputSummary.error);
...

summaryContext.mapSummary

<table>
<thead>
<tr>
<th>Object Description</th>
<th>Holds statistics regarding the map stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.

```plaintext
...
logMetrics(summary.mapSummary);
var mapKeys = [];
summary.mapSummary.keys.iterator().each(function (key)
{
    mapKeys.push(key);
    return true;
});
log.audit('MAP keys processed', mapKeys);
summary.mapSummary.errors.iterator().each(function (key, error)
{
    log.error('Map Error for key: ' + key, error);
    return true;
});
...

mapSummary.concurrentn

<table>
<thead>
<tr>
<th>Property Description</th>
<th>The maximum concurrency number for executing parallel tasks during the map stage.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note:</td>
<td>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.</td>
</tr>
<tr>
<td>Type</td>
<td>number</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.
mapSummary.dateCreated

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The time and day when <code>map(mapContext)</code> began running.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.

```
... 
log.audit('Creation Date', summary.mapSummary.dateCreated);
... 
```

mapSummary.keys

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holds the keys that were passed to the map stage by the getInputData stage.</td>
<td></td>
</tr>
</tbody>
</table>

Note that keys are sorted in the following way:

- If all values are numeric, values are listed in numeric order.
- If one or more values are strings, values are listed in lexicographical order.

<table>
<thead>
<tr>
<th>Type</th>
<th>iterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

Members

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iterator().each(parameters)</code></td>
<td>function</td>
<td>required</td>
<td>Executes one time for each key.</td>
</tr>
</tbody>
</table>

parameters

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iteratorFunction(key, executionCount, completionState)</code></td>
<td>function</td>
<td>required</td>
<td>Provides logic to be executed during each iteration.</td>
</tr>
</tbody>
</table>

See also `functionParameters`.

functionParameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>string</td>
<td>optional</td>
<td>Represents a key that was passed to the map stage.</td>
</tr>
<tr>
<td>executionCount</td>
<td>number</td>
<td>optional</td>
<td>The number of times the map function was invoked for the key.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Required/Optional</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------</td>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>completionState</td>
<td>string</td>
<td>optional</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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](#).

```javascript
// Add additional code.
...

// Log all keys from the map stage.

var mapKeys = [];
summary.mapSummary.keys.iterator().each(function (key) {
    mapKeys.push(key);
    return true;
});

log.debug(
    {
        title: 'Map stage keys',
        details: mapKeys
    });

// Create a log entry showing whether the map function was executed successfully for each key.

summary.mapSummary.keys.iterator().each(function (key, executionCount, completionState) {
    log.debug(
        {
            title: 'Map key ' + key + ',
            details: 'Outcome for map key ' + key + ': ' + completionState + ' // Number of attempts used: ' + executionCount
        });
    return true;
});

// Create a log entry showing the total number of keys for which the map function executed successfully.

var mapKeysProcessed = 0;
summary.mapSummary.keys.iterator().each(function (key, executionCount, completionState) {
    if (completionState === 'COMPLETE') {
        mapKeysProcessed++;
    }
    return true;
});
```
SuiteScript 2.0 Map/Reduce Script Type

mapSummary.seconds

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Total seconds elapsed when running map(mapContext).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>number</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.

```javascript
... 
log.audit(' Time Elapsed', summary.mapSummary.seconds); 
... 
```

mapSummary.usage

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Total number of usage units consumed when running map(mapContext).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>number</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.

```javascript
... 
log.audit(' Usage', summary.mapSummary.usage); 
... 
```

mapSummary.yields

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Total number of times yields when running map(mapContext).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>number</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.
mapSummary.errors

**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**

Version 2015 Release 2

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator().each(parameters)</td>
<td>function</td>
<td>required</td>
<td>Executes one time for each error.</td>
</tr>
</tbody>
</table>

**parameters**

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteratorFunction(key, error, executionNo)</td>
<td>function</td>
<td>required</td>
<td>Provides logic to be executed during each iteration.</td>
</tr>
</tbody>
</table>

**functionParameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>string</td>
<td>optional</td>
<td>Represents a key that was passed to the map stage for processing.</td>
</tr>
<tr>
<td>error</td>
<td>string</td>
<td>optional</td>
<td>The error thrown during processing of the corresponding key.</td>
</tr>
<tr>
<td>executionNo</td>
<td>number</td>
<td>optional</td>
<td>Indicates whether the error occurred during the first, second, third, or fourth attempt to process the key.</td>
</tr>
</tbody>
</table>

**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.

```javascript
// Add additional code
...
// Create a log entry showing each full serialized error
summary.mapSummary.errors.iterator().each(
  function(key, error, executionNo) {
    log.error({
      title: 'Map error for key: ' + key + ', execution no. ' + executionNo,
      details: error
    });
    return true;
  }
);```

SuiteScript 2.0
SuiteScript 2.0 Map/Reduce Script Type

Summary

// Log only the name and description of each error thrown
summary.mapSummary.errors.iterator().each(
    function (key, error, executionNo) {
        var errorObject = JSON.parse(error);
        log.error({
            title: 'Map error for key: ' + key + ', execution no. ' + executionNo,
            details: errorObject.name + ': ' + errorObject.message
        });
        return true;
    });

// Calculate and log the total number of errors encountered during the map stage
var errorCount = 0;
summary.mapSummary.errors.iterator().each(
    function() {
        errorCount++;
        return true;
    });

log.audit(
    title: 'Map stage errors',
    details: 'Total number of errors: ' + errorCount
);

// Add additional code

summaryContext.reduceSummary

<table>
<thead>
<tr>
<th>Object Description</th>
<th>Holds statistics regarding the reduce stage.</th>
</tr>
</thead>
</table>

**Since**

Version 2015 Release 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.

```plaintext
... summary.reduceSummary.errors.iterator().each(function (key, error) {
    log.error('Reduce Error for key: ' + key, error);
    return true;
});
```

**reduceSummary.concurrency**

| 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

**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.

```javascript
... log.audit('Concurrency', summary.reduceSummary.concurrency);
... reduceSummary.dateCreated
```

**reduceSummary.dateCreated**

<table>
<thead>
<tr>
<th>Property Description</th>
<th>The time and day when reduce(reduceContext) began running.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Date</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

**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.

```javascript
... log.audit('Creation Date', summary.reduceSummary.dateCreated);
... reduceSummary.keys
```

**reduceSummary.keys**

<table>
<thead>
<tr>
<th>Property Description</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>iterator</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator().each(parameters)</td>
<td>function</td>
<td>required</td>
<td>Executes one time for each key.</td>
</tr>
</tbody>
</table>

**parameters**

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteratorFunction(key, executionCount, completionState)</td>
<td>function</td>
<td>required</td>
<td>Provides logic to be executed during each iteration.</td>
</tr>
</tbody>
</table>
SuiteScript 2.0 Map/Reduce Script Type

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>See also functionParameters.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

functionParameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>string</td>
<td>optional</td>
<td>Represents a key that was passed to the reduce stage.</td>
</tr>
<tr>
<td>executionCount</td>
<td>number</td>
<td>optional</td>
<td>The number of times the reduce function was invoked for the key.</td>
</tr>
<tr>
<td>completionState</td>
<td>string</td>
<td>optional</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

```javascript
//Add additional code.
...

// Create a single log entry listing all of the keys that were passed to the reduce stage.
var reduceKeys = [];

summary.reduceSummary.keys.iterator().each(function (key){
    reduceKeys.push(key);
    return true;
});

log.debug({
    title: 'Reduce stage keys',
    details: reduceKeys
});

// Create a log entry for each key. The entry shows whether the reduce function was executed successfully for that key.
summary.reduceSummary.keys.iterator().each(function (key, executionCount, completionState){

    log.debug({
        title: 'Reduce key ' + key,
        details: 'Outcome for reduce key ' + key + ': ' + completionState + ' // Number of attempts used: ' + executionCount
    });

    return true;
});
```
// creates a single log entry showing the total number of keys for which the reduce function was invoked successfully.

    var reduceKeysProcessed = 0;
    summary.reduceSummary.keys.iterator().each(function (key, executionCount, completionState){
        if (completionState === 'COMPLETE'){
            reduceKeysProcessed++;
        }
    return true;
    });

    log.debug({
        title : 'Reduce key statistics',
        details: 'Total number of reduce keys processed successfully: ' + reduceKeysProcessed
    });

    //Add additional code.

reduceSummary.seconds

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Total seconds elapsed when running reduce(reduceContext).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>number</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.

    ...
    log.audit(' Time Elapsed', summary.reduceSummary.seconds);
    ...

reduceSummary.usage

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Total number of usage units consumed when running reduce(reduceContext).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>number</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.

    ...
    log.audit(' Usage', summary.mapSummary.usage);
    ...
reduceSummary.yields

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Type</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of times yields when running reduce(reduceContext).</td>
<td>number</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.

```javascript
... 
log.audit(' Total Yields', summary.reduceSummary.yields); 
... 
```

reduceSummary.errors

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Type</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holds all serialized errors thrown from the reduce(reduceContext) function. May also hold the SSS_APP_SERVER_RESTART error code, which is recorded in the event of an application server restart.</td>
<td>iterator</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

Members

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator().each(parameters)</td>
<td>function</td>
<td>required</td>
<td>Executes one time for each error.</td>
</tr>
</tbody>
</table>

parameters

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteratorFunction(key, error, executionNo)</td>
<td>function</td>
<td>required</td>
<td>Provides logic to be executed during each iteration.</td>
</tr>
</tbody>
</table>

See also functionParameters.

functionParameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>string</td>
<td>optional</td>
<td>Represents a key that was passed to the reduce stage for processing.</td>
</tr>
<tr>
<td>error</td>
<td>string</td>
<td>optional</td>
<td>A serialization of the error thrown.</td>
</tr>
<tr>
<td>executionNo</td>
<td>number</td>
<td>optional</td>
<td>Indicates whether the error occurred during the first, second, third, or fourth attempt to process the key.</td>
</tr>
</tbody>
</table>

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.
// Add additional code
...

// Create a log entry showing each full serialized error.

summary.reduceSummary.errors.iterator().each(function(key, error, executionNo) {
    log.error({
        title: 'Reduce error for key: ' + key + ', execution no. ' + executionNo,
        details: error
    });
    return true;
});

// Log only the name and description of each error thrown.

summary.reduceSummary.errors.iterator().each(function(key, error) {
    var errorObject = JSON.parse(error);
    log.error({
        title: 'Reduce error for key: ' + key + ', execution no. ' + executionNo,
        details: errorObject.name + ': ' + errorObject.message
    });
    return true;
});

// Calculate and log the total number of errors encountered during the reduce stage.

var errorCount = 0;
summary.reduceSummary.errors.iterator().each(function() {
    errorCount ++;
    return true;
});

log.audit({
    title: 'Reduce stage errors',
    details: 'Total number of errors: ' + errorCount
});

summaryContext.output

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Iterator that provides keys and values that are saved as output during the reduce stage.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note</td>
<td>Keys are listed in lexicographical order.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>iterator</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Since</th>
<th>Version 2015 Release 2</th>
</tr>
</thead>
</table>

Members

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator().each(parameters)</td>
<td>function</td>
<td>required</td>
<td>Executes one time for each key/value pair.</td>
</tr>
</tbody>
</table>
SuiteScript 2.0 Map/Reduce Script Type

**parameters**

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteratorFunction(key, value).</td>
<td>function</td>
<td>required</td>
<td>Provides logic to be executed during each iteration.</td>
</tr>
</tbody>
</table>

See `functionParameters`.

**functionParameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>string</td>
<td>optional</td>
<td>The key saved at the end of the reduce stage by using the <code>reduceContext.write(options)</code> method.</td>
</tr>
<tr>
<td>value</td>
<td>string</td>
<td>optional</td>
<td>The value saved at the end of the reduce stage by using the <code>reduceContext.write(options)</code> method.</td>
</tr>
</tbody>
</table>

**Syntax**

```javascript
// Add additional code
...
// Create a variable to track how many key/value pairs were written.
vvar totalRecordsUpdated = 0;

// If the number of key/value pairs is expected to be manageable, log
// each one.
summary.output.iterator().each(function (key, value){
  log.audit({
    title: 'summary.output.iterator',
    details: 'key: ' + key + ' / value: ' + value
  });
  totalRecordsUpdated++;
  return true;
});

// Create a log entry showing the number of
log.debug({
  title: 'Total records updated',
  details: totalRecordsUpdated
});
...
// Add additional code.
```

SuiteScript 2.0 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.
update script. For more information, see SuiteScript 2.0 Scheduled Script Type and SuiteScript 2.0 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. See the help topic Running a Mass Update Script in NetSuite.

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

For information about scripting with mass update scripts, see SuiteScript 2.0 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.

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 2.0, see SuiteScript 2.0 Hello World and SuiteScript 2.0 Entry Point Script Creation and Deployment.

```
/**
 * @NApiVersion 2.0
 * @NScriptType MassUpdateScript
 */
define([ 'N/record' ],
    function (record) {
        function each(params) {
            // Set the probability to 61%
            var recOpportunity = record.load({
                type: params.type,
                id: params.id
            });
            recOpportunity.setValue('probability', 61);
            recOpportunity.save();
        }
        return {
            each: each
        };
    });
```

SuiteScript 2.0 Mass Update Script Entry Points

<table>
<thead>
<tr>
<th>Script Entry Point</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>each</td>
<td>Iterates through all applicable records so that you can apply logic to each record. See each for parameters and syntax.</td>
</tr>
</tbody>
</table>
each

<table>
<thead>
<tr>
<th>Description</th>
<th>Iterates through each applicable record.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2016 Release 1</td>
</tr>
</tbody>
</table>

Parameters

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>params.id</td>
<td>number</td>
<td>The ID of the record being processed by the mass update.</td>
<td>Version 2016 Release 1</td>
</tr>
<tr>
<td>params.type</td>
<td>string</td>
<td>The record type of the record being processed by the mass update.</td>
<td>Version 2016 Release 1</td>
</tr>
</tbody>
</table>

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.

```javascript
// Add additional code
...
function each(params) {
    // Set the probability to 61%
    var recOpportunity = record.load({
        type: params.type,
        id: params.id
    });
    ...
    // Add additional code
}
```

SuiteScript 2.0 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-side 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.

![Portlet Image]

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.

See the following for more information about the Portlet Script type:

- SuiteScript 2.0 Portlet Script Reference
- Creating and Deploying a Portlet Script
- Guidelines for Creating a Dashboard SuiteApp Icon
- SuiteScript 2.0 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.clientScriptFileId
SuiteScript 2.0 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 2.0, see SuiteScript 2.0 Hello World and SuiteScript 2.0 Entry Point Script Creation and Deployment.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType Portlet
 */
define([ 'N/search' ],
function (search) {
    function render(params) {
        var portlet = params.portlet;
        portlet.title = 'Simple Form Portlet';
        var fld = portlet.addField({
            id: 'text',
            type: 'text',
            label: 'Text'
        });
        fld.updateLayoutType({
            layoutType: 'normal'
        });
        fld.updateBreakType({
            breakType: 'startcol'
        });
        portlet.setSubmitButton({
            url: 'http://httpbin.org/post',
            label: 'Submit',
            target: '_top'
        });
        return {
            render: render
        };
    }
    return {
        render: render
    };
});
```

Inline HTML Portlet Script Sample

```javascript
/**
*/
```
SuiteScript 2.0 Portlet Script Type

Links and Indents Portlet Script Sample

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType Portlet
 */

define(['N/search'],
    function (search) {
        function render(params) {
            var portlet = params.portlet;
            portlet.title = 'Search Engines';
            portlet.addElement(
                text: 'NetSuite',
                url: 'http://www.netsuite.com'
            );
            portlet.addElement(
                text: 'Oracle',
                url: 'http://www.oracle.com'
            );
        }
        return {render: render}
    });
```

Simple List Portlet Script Sample

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType Portlet
 */

define(['N/search'],
    function (search) {
        function render(params) {
            params.portlet.title = 'My Portlet';
            var content = '<td><span><b>Hello!!!</b></span></td>';
            params.portlet.html = content;
        }
        return {render: render}
    });
```

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.
function render(params) {
    var isDetail = (params.column == 2);
    var portlet = params.portlet;
    portlet.title = isDetail ? "My Detailed List" : "My List";
    portlet.addColumn({
        id: 'internalid',
        type: 'text',
        label: 'Number',
        align: 'LEFT'
    });
    portlet.addColumn({
        id: 'entity',
        type: 'text',
        label: 'ID',
        align: 'LEFT'
    });
    if (isDetail) {
        portlet.addColumn({
            id: 'email',
            type: 'text',
            label: 'E-mail',
            align: 'LEFT'
        });
        portlet.addColumn({
            id: 'custentity_multiselect',
            type: 'text',
            label: 'Multiselect',
            align: 'LEFT'
        });
    }
    var filter = search.createFilter({
        name: 'email',
        operator: search.Operator.ISNOTEMPTY
    });
    var customerSearch = search.create({
        type: 'customer',
        filters: filter,
        columns: ['internalid', 'entity', 'email', 'custentity_multiselect']
    });
    var count = isDetail ? 15 : 5;
    customerSearch.run().each(function(result) {
        portlet.addRow(result.getAllValues());
        return --count > 0;
    });
}
return {
    render: render
};
Creating and Deploying a Portlet Script

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 2.0 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 Guidelines for Creating a Dashboard SuiteApp Icon.
   The following image indicates where the Dashboard fields are located:

   For more information about the script deployment process, see Methods of Deploying a Script.

Guidelines for Creating a Dashboard SuiteApp Icon

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 2.0 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.clientScriptFileId`
  - `Portlet.clientScriptModulePath`
  - `Portlet.html`
# Portlet.title

render(params)

<table>
<thead>
<tr>
<th>Description</th>
<th>Definition of the portlet script trigger point.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>params.portlet</td>
<td>Portlet Object</td>
<td>The Portlet object used for rendering.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>params.column</td>
<td>string</td>
<td>The column index for the portlet on the dashboard. This parameter is read-only.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>params.entity</td>
<td>string</td>
<td>The customer ID for the selected customer. This parameter is read-only.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

Portlet Object

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.

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Return Type / Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>Portlet.addField(options)</td>
<td>serverWidget.Field</td>
<td>Adds a field to the form.</td>
</tr>
<tr>
<td></td>
<td>Portlet.setSubmitButton(options)</td>
<td>serverWidget.Button</td>
<td>Adds a submit button to the form.</td>
</tr>
</tbody>
</table>
SuiteScript 2.0 Portlet Script Type

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Return Type / Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Portlet.clientScriptFileId</td>
<td>number or string</td>
<td>The script file ID to be used in the portlet. Can be in either numerical or string format.</td>
</tr>
<tr>
<td>Property</td>
<td>Portlet.clientScriptModulePath</td>
<td>string</td>
<td>The script path to be used in the portlet.</td>
</tr>
<tr>
<td>Property</td>
<td>Portlet.title</td>
<td>string</td>
<td>The title of the portlet.</td>
</tr>
</tbody>
</table>

**Inline HTML Portlet Object Members**

These members are only available to HTML Portlet objects.

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Return Type / Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Portlet.html</td>
<td>string</td>
<td>The complete HTML contents of the portlet.</td>
</tr>
<tr>
<td>Property</td>
<td>Portlet.title</td>
<td>string</td>
<td>The title of the portlet.</td>
</tr>
</tbody>
</table>

**Links and Indents Portlet Object Members**

These members are only available to Links Portlet objects.

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Return Type / Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>Portlet.addLine(options)</td>
<td>Object</td>
<td>Adds a line to the portlet.</td>
</tr>
<tr>
<td>Property</td>
<td>Portlet.title</td>
<td>string</td>
<td>The title of the portlet.</td>
</tr>
</tbody>
</table>

**Simple List Portlet Object Members**

These members are only available to List Portlet objects.

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Name</th>
<th>Return Type / Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>Portlet.addColumn(options)</td>
<td>serverWidget.ListColumn</td>
<td>Adds a column to the portlet.</td>
</tr>
<tr>
<td>Method</td>
<td>Portlet.addEditColumn(options)</td>
<td>serverWidget.ListColumn</td>
<td>Adds an Edit or Edit/View column to the portlet.</td>
</tr>
<tr>
<td>Method</td>
<td>Portlet.addRow(options)</td>
<td>Object</td>
<td>Adds a row to the portlet.</td>
</tr>
<tr>
<td>Method</td>
<td>Portlet.addRow(options)</td>
<td>Object</td>
<td>Adds multiple rows to the portlet.</td>
</tr>
<tr>
<td>Property</td>
<td>Portlet.title</td>
<td>string</td>
<td>The title of the portlet.</td>
</tr>
</tbody>
</table>

**Portlet.addColumn(options)**

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.id</td>
<td>string</td>
<td>required</td>
<td>The internal ID of this column. The internal ID must be in lowercase, contain no spaces.</td>
<td>2016.2</td>
</tr>
<tr>
<td>options.label</td>
<td>string</td>
<td>required</td>
<td>The label of this column.</td>
<td>2016.2</td>
</tr>
<tr>
<td>options.type</td>
<td>string</td>
<td>required</td>
<td>The field type for this column. For more information about possible values, see the help topic serverWidget.FieldType.</td>
<td>2016.2</td>
</tr>
<tr>
<td>options.align</td>
<td>string</td>
<td>optional</td>
<td>The layout justification for this column. For more information about possible values, see the help topic serverWidget.LayoutJustification.</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see SuiteScript 2.0 Portlet Script Type.

```javascript
var newColumn = params.portlet.addColumn({
    id: 'column1',
    label: 'Text',
    type: serverWidget.FieldType.TEXT,
    align: serverWidget.LayoutJustification.RIGHT
});
```

Portlet.addEditColumn(options)

Method Description

Adds an Edit or Edit/View column to the portlet.

Returns

serverWidget.ListColumn

Entry Point

render(params)

Since

2016.2

Parameters

Note: The options parameter is a JavaScript object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.column</td>
<td>string</td>
<td>required</td>
<td>The internal ID of the column to the left of which the Edit/View column is added.</td>
<td>2016.2</td>
</tr>
</tbody>
</table>
### Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.showHrefCol</td>
<td>string</td>
<td>optional</td>
<td>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).</td>
<td>2016.2</td>
</tr>
<tr>
<td>options.showView</td>
<td>boolean</td>
<td>optional</td>
<td>If true, then an Edit/View column is added. Otherwise, only an Edit column is added. The default setting is false.</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

### Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see SuiteScript 2.0 Portlet Script Type.

```javascript
var newColumn = params.portlet.addEditColumn({
  column: 'column1',
  showHrefCol: true,
  showView: true
});
```

### Portlet.addField(options)

**Method Description**

Adds a field to the form.

**Returns**

serverWidget.Field

**Entry Point**

render(params)

**Since**

2016.2

### Parameters

**Note:** The options parameter is a JavaScript object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.id</td>
<td>string</td>
<td>required</td>
<td>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.</td>
<td>2016.2</td>
</tr>
<tr>
<td>options.label</td>
<td>string</td>
<td>required</td>
<td>The label for this field.</td>
<td>2016.2</td>
</tr>
<tr>
<td>options.type</td>
<td>string</td>
<td>required</td>
<td>The field type for the field.</td>
<td>2016.2</td>
</tr>
</tbody>
</table>
### Parameter Table

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.source</td>
<td>string</td>
<td>optional</td>
<td>The internal ID or script ID of the source list for this field if it is a select (List/Record) or multi-select field.</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

#### Notes:
- For radio fields only, the `source` parameter must contain the internal ID for the field.

#### Important:
- After you create a select or multi-select field that is sourced from a record or list, you cannot add additional values with `Field.addSelectOption(options)`. The select values are determined by the source record or list.

For more information about working with radio buttons, see the help topic [Working with Radio Buttons](#).

### Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see [SuiteScript 2.0 Portlet Script Type](#).

```javascript
... var newField = params.portlet.addField(
  id: 'textfield',
  type: serverWidget.FieldType.TEXT,
  label: 'text'
);... 
```

### Portlet.addLine(options)

**Method Description**: Adds a line to the portlet.

**Returns**: Object

**Entry Point**: `render(params)`

**Since**: 2016.2

### Parameters

#### Note:
- The `options` parameter is a JavaScript object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.text</td>
<td>string</td>
<td>required</td>
<td>The text for the line.</td>
<td>2016.2</td>
</tr>
<tr>
<td>options.url</td>
<td>string</td>
<td>optional</td>
<td>The URL link.</td>
<td>2016.2</td>
</tr>
</tbody>
</table>
### Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see SuiteScript 2.0 Portlet Script Type.

```javascript
...  
params.portlet.addLine({
    text: 'NetSuite',
    url: 'http://www.netsuite.com',
    align: 4
  });
...  
```

### Portlet.addRow(options)

<table>
<thead>
<tr>
<th>Method Description</th>
<th>Adds a row to the portlet.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>Object</td>
</tr>
<tr>
<td>Entry Point</td>
<td>render(params)</td>
</tr>
<tr>
<td>Since</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

### Parameters

**Note:** The options parameter is a JavaScript object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.row</td>
<td>search.Result</td>
<td>required</td>
<td>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.</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

### Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see SuiteScript 2.0 Portlet Script Type.

```javascript
...  
params.portlet.addRow({
  row: {
    columnid1: 'value1',
    columnid2: 'value2'
  }
});
...  
```
Portlet.addRows(options)

<table>
<thead>
<tr>
<th>Method Description</th>
<th>Adds multiple rows to the portlet.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>Object</td>
</tr>
<tr>
<td>Entry Point</td>
<td>render(params)</td>
</tr>
<tr>
<td>Since</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

Parameters

Note: The options parameter is a JavaScript object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.rows</td>
<td>search.Result[]</td>
<td>required</td>
<td>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.</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see SuiteScript 2.0 Portlet Script Type.

```javascript
... 
params.portlet.addRows({
  rows:
    [{
      columnid1: 'value1',
      columnid2: 'value2'
    }, {
      columnid1: 'value2',
      columnid2: 'value3'
    }]
});
...
```

Portlet.setSubmitButton(options)

<table>
<thead>
<tr>
<th>Method Description</th>
<th>Adds a submit button to the form.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>serverWidget.Button</td>
</tr>
<tr>
<td>Entry Point</td>
<td>render(params)</td>
</tr>
</tbody>
</table>
Since 2016.2

Parameters

Note: The options parameter is a JavaScript object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.url</td>
<td>string</td>
<td>required</td>
<td>The URL that the form posts data to.</td>
<td>2016.2</td>
</tr>
<tr>
<td>options.label</td>
<td>string</td>
<td>optional</td>
<td>The button label.</td>
<td>2016.2</td>
</tr>
<tr>
<td>options.target</td>
<td>string</td>
<td>optional</td>
<td>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. Setting this value to _hidden allows submission to a backend that returns results to a hidden child iframe within the portlet’s embedded iframe.</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see SuiteScript 2.0 Portlet Script Type.

```javascript
...  
params.portlet.setSubmitButton( {  
  url: 'http://httpbin.org/post',  
  label: 'Submit',  
  target: '_top'  
});  
...  
```

Portlet.clientScriptFileId

<table>
<thead>
<tr>
<th>Property Description</th>
<th>Description</th>
<th>Type</th>
<th>Entry Point</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The script file ID to be used in the portlet.</td>
<td>number or string</td>
<td>render(params)</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Thrown If</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROPERTY_VALUE_CONFLICT</td>
<td>You attempted to set this value when the Portlet.clientScriptModulePath property value has already been specified. For more information, see Portlet.clientScriptFileId.</td>
</tr>
</tbody>
</table>
Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see SuiteScript 2.0 Portlet Script Type.

```javascript
... 
params.portlet.clientScriptFileId = 32;
... 
```

Portlet.clientScriptModulePath

<table>
<thead>
<tr>
<th>Property Description</th>
<th>The script path to be used in the portlet.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>string</td>
</tr>
<tr>
<td>Entry Point</td>
<td>render(params)</td>
</tr>
<tr>
<td>Since</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Thrown If</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROPERTY_VALUE_CONFLICT</td>
<td>You attempted to set this value when the Portlet.clientScriptFileId property value has already been specified. For more information, see Portlet.clientScriptFileId.</td>
</tr>
</tbody>
</table>

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see SuiteScript 2.0 Portlet Script Type.

```javascript
... 
params.portlet.clientScriptModulePath = '/SuiteScripts/clientScript.js';
... 
```

Portlet.html

<table>
<thead>
<tr>
<th>Property Description</th>
<th>The complete HTML contents of the portlet.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>string</td>
</tr>
<tr>
<td>Entry Point</td>
<td>render(params)</td>
</tr>
<tr>
<td>Since</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see SuiteScript 2.0 Portlet Script Type.

```javascript
... 
params.portlet.html = htmlcontents;
... 
```
Portlet.title

<table>
<thead>
<tr>
<th>Property Description</th>
<th>The title of the portlet.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>string</td>
</tr>
<tr>
<td>Entry Point</td>
<td>render(params)</td>
</tr>
<tr>
<td>Since</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

Syntax

The following code snippet shows the syntax for this member. It is not a functional example. For a complete script example, see SuiteScript 2.0 Portlet Script Type.

```
...
params.portlet.title = 'My Portlet';
...
```

SuiteScript 2.0 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 2.0 Getting Started with RESTlets
# RESTlet Authentication
# SuiteScript 2.0 RESTlet Reference
# SuiteScript 2.0 RESTlet Script and Request Samples
# SuiteScript 2.0 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.

SuiteScript 2.0 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
RESTlet Key Concepts

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 an alternative to other script types, such as server-side 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

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 2.0. 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 2.0 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 2.0 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 2.0 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.

Identifying a RESTlet in a Call

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.

<table>
<thead>
<tr>
<th>Source of the Call</th>
<th>Identify the RESTlet with:</th>
</tr>
</thead>
<tbody>
<tr>
<td>An external client</td>
<td>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.</td>
</tr>
<tr>
<td>A client SuiteScript with an active session in the same NetSuite account where the RESTlet is deployed</td>
<td>Either of the following:</td>
</tr>
<tr>
<td></td>
<td># The partial URL shown on the script deployment record, in the URL field (3). For an example, see Example: Client Script that Calls a RESTlet.</td>
</tr>
<tr>
<td></td>
<td># 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: Suitelet that Calls a RESTlet.</td>
</tr>
<tr>
<td>A server SuiteScript in the same NetSuite account where the RESTlet is deployed</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td># By using the REST roles service. For details, see the help topic The REST roles Service.</td>
</tr>
<tr>
<td></td>
<td># 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: Suitelet that Calls a RESTlet.</td>
</tr>
<tr>
<td>A server SuiteScript in a different NetSuite account from where the RESTlet is deployed</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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: Suitelet that Calls a RESTlet.
Selecting an HTTP Method for Calling a RESTlet

When you call a RESTlet, you can use one of four supported HTTP methods: delete, get, post, or put.

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

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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Placement of required input</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete</td>
<td>Arguments must be embedded in the URL used to make the call.</td>
</tr>
<tr>
<td>get</td>
<td></td>
</tr>
<tr>
<td>post</td>
<td>Arguments must be included in a request body written in JSON (JavaScript Object Notation) or plain text.</td>
</tr>
<tr>
<td>put</td>
<td></td>
</tr>
</tbody>
</table>

Note: For examples of requests, see SuiteScript 2.0 RESTlet Script and Request Samples.

Creating a Content–Type Header

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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>application/json</td>
<td>JSON is the appropriate choice for most RESTlets that require a request body, because it lets you map values to fields.</td>
</tr>
<tr>
<td>application/xml</td>
<td>XML is supported only for the get method.</td>
</tr>
<tr>
<td>text/plain</td>
<td>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.</td>
</tr>
</tbody>
</table>

Structuring the Header

Format your content-type header as follows:

```
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: 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 2.0 RESTlet Reference

See the following topics for more details about working with RESTlets:

# SuiteScript 2.0 RESTlet Error Handling
# SuiteScript 2.0 RESTlet Governance
# SuiteScript 2.0 RESTlet Security

SuiteScript 2.0 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

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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>206 Partial Content</td>
<td>You receive this error if you attempt to use a content-type that is unsupported for the HTTP method you are using.</td>
</tr>
<tr>
<td>302 Moved Temporarily</td>
<td>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.</td>
</tr>
<tr>
<td>400 Bad Request</td>
<td>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.)</td>
</tr>
<tr>
<td>401 Unauthorized</td>
<td>There is no valid NetSuite login session for the RESTlet call.</td>
</tr>
<tr>
<td>403 Forbidden</td>
<td>The RESTlet request was sent to an invalid domain.</td>
</tr>
<tr>
<td>404 Not Found</td>
<td>A RESTlet script is not defined in the RESTlet request.</td>
</tr>
<tr>
<td>405 Method Not Allowed</td>
<td>The request method used is not valid.</td>
</tr>
</tbody>
</table>
SuiteScript Errors Returned by RESTlets

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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID_LOGIN_ATTEMPT</td>
<td>Invalid login attempt.</td>
<td>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.</td>
</tr>
<tr>
<td>INVALID_LOGIN_CREDENTIALS</td>
<td>You have entered an invalid email address or account number. Please try again.</td>
<td>This error indicates a problem in an NLAuth header.</td>
</tr>
<tr>
<td>INVALID_REQUEST</td>
<td>The request could not be understood by the server due to malformed syntax.</td>
<td>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.</td>
</tr>
<tr>
<td>INVALID_RETURN_DATA_FORMAT</td>
<td>You should return {1}.</td>
<td>This error is used if the response data does not match the expected format, as specified by the content-type header.</td>
</tr>
<tr>
<td>SSS_INVALID_SCRIPTLET_ID</td>
<td>That Suitelet is invalid, disabled, or no longer exists.</td>
<td>If you receive this error, make sure that the URL points to the correct RESTlet script deployment ID.</td>
</tr>
<tr>
<td>unexpected_error</td>
<td>An unexpected error occurred. Error ID: [1]</td>
<td></td>
</tr>
<tr>
<td>TWO_FA_REQD</td>
<td>Two-Factor Authentication required.</td>
<td>This error indicated that two-factor authentication is required for the role, but it is missing.</td>
</tr>
</tbody>
</table>

SuiteScript Error Message Formatting for RESTlets

The following examples illustrate SuiteScript errors formatting for each supported content-type.

**JSON**

```json
{
  "error":
    {
```

---

**SuiteScript 2.0**
SuiteScript 2.0 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.

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

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

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

SuiteScript 2.0 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.

SuiteScript 2.0 RESTlet Script and Request Samples

For sample RESTlet scripts and requests, see the following sections:

# Example: Hello World RESTlet
# Example: RESTlet that Can Retrieve, Delete or Create
# Example: RESTlet that Adds Multiple Records
# Example: RESTlet that Manipulates Scheduled Script
# Example: Client Script that Calls a RESTlet
# Example: Suitelet that Calls a RESTlet
# Example: Shell Script that Calls a RESTlet

For help with writing other types of scripts in SuiteScript 2.0, see SuiteScript 2.0 Hello World and SuiteScript 2.0 Entry Point Script Creation and Deployment.

## Example: Hello World RESTlet

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**

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType restlet
 */
define([], function () {
  return {
    get : function() {
      return "Hello World!"
    }
  }
});
```

### 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:

```
https://rest.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.

```
GET /app/site/hosting/restlet.nl?script=482&deploy=1 HTTP/1.1
HOST: rest.netsuite.com
authorization: NLAuth nlauth_account=12345, nlauth_email=john@smith.com, nlauth_signature=Welcome123
content-type: application/json
cookie: ...
```

The RESTlet would return the following response:

```
Hello World!
```
Example: RESTlet that Can Retrieve, Delete or Create

The following RESTlet includes logic for all supported entry points: get, delete, post, and put.

RESTlet

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType Restlet
 */
define([ 'N/record' , 'N/error' ],
  function (record, error) {

    function doValidation(args, argNames, methodName) {
      for (var i = 0; i < args.length; i++)
        if (!args[i] && args[i] !== 0)
          throw error.create({
            name: 'MISSING_REQ_ARG',
            message: 'Missing a required argument: [' + argNames[i] + '] for method: ' + methodName
          });
    }

    // Get a standard NetSuite record
    function _get(context) {
      doValidation([context.recordtype, context.id], [ 'recordtype' , 'id' ], 'GET');
      return JSON.stringify(record.load({
        type: context.recordtype,
        id: context.id
      }));
    }

    // Delete a standard NetSuite record
    function _delete(context) {
      doValidation([context.recordtype, context.id], [ 'recordtype' , 'id' ], 'DELETE');
      record.delete({
        type: context.recordtype,
        id: context.id
      });
      return String(context.id);
    }

    // Create a NetSuite record from request params
    function post(context) {
      doValidation([context.recordtype], [ 'recordtype' ], 'POST');
      var rec = record.create({
        type: context.recordtype
      });
      for (var fldName in context)
        if (context.hasOwnProperty(fldName))
          if (fldName !== 'recordtype')
            rec.setValue(fldName, context[fldName]);
      var recordId = rec.save();
      return String(recordId);
    }

    // Upsert a NetSuite record from request param
    function put(context) {
      doValidation([context.recordtype, context.id], [ 'recordtype' , 'id' ], 'PUT');
      var rec = record.load({
        type: context.recordtype,
```
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:

&[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:

https://rest.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:

GET /app/site/hosting/restlet.nl?script=474&deploy=1&recordtype=phonecall&id=9363 HTTP/1.1
HOST: rest.netsuite.com
authorization: NLAuth nlauth_account=12345, nlauth_email=john@smith.com, nlauth_signature=Welcome123
content-type: application/json
cookie: ...

In response, the system would return data like the following:

{  
  "id": "9363"  
  "type": "phonecall"  
  "isNewDynamic": false  
  "fields": {  
    "wfinstances": ""  
    "nlloc": "0"  
    "nlsub": "1"  
    "createddate": "5/18/2016 1:01 am"  
}
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:

```
{ "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:

```
https://rest.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:

```
POST /app/site/hosting/restlet.nl?script=474&deploy=1 HTTP/1.1
HOST: rest.netsuite.com
authorization: NLAuth nlauth_account=12345, nlauth_email=john@smith.com, nlauth_signature=Welcome123
content-type: application/json
cookie: ...
```

In response, the system returns the internal ID of the newly created record. For example:

```
9564
```

Example: RESTlet that Adds Multiple Records

The following RESTlet example can create several contact records in one call.
RESTlet

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType restlet
 */
define(['N/record'], function (record) {
    return {
        post: function (restletBody) {
            return {
                post: function (restletBody) {
                    var restletData = restletBody.data;
                    for (var contact in restletData) {
                        var objRecord = record.create({
                            type: record.Type.CONTACT,
                            isDynamic: true
                        });
                        var contactData = restletData[contact];
                        for (var key in contactData) {
                            if (contactData.hasOwnProperty(key)) {
                                objRecord.setValue({
                                    fieldId: key,
                                    value: contactData[key]
                                });
                            }
                        }
                        var recordId = objRecord.save({
                            enableSourcing: false,
                            ignoreMandatoryFields: false
                        });
                        log.debug(recordId);
                    }
                }
            }
        }
    }
});
```

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.

```json
```

Example: RESTlet that Manipulates Scheduled Script

The following RESTlet passes a value to a scheduled script.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType restlet
 */
```
Example: Client Script that Calls a RESTlet

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.

Example: Suitelet that Calls a RESTlet

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: The scriptId, deploymentId, and authorization data in this sample are placeholders. Before using this script, replace these values with valid data from your NetSuite account.
Example: Shell Script that Calls a RESTlet

The following shell script shows how you 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.

```bash
#!/bin/sh

#Put your RESTlet url here.
my_url= "https://123456.restlets.api.netsuite.com/app/site/hosting/restlet.nl?script=126&deploy=1"

#Put the body of your request in a file named data.json.
DATA_FILE="@data.json"

#Update the following line with valid values from your NetSuite account.
AUTH_STRING="Authorization: NLAuth nlauth_account=123456, nlauth_email=jsmith%40ABC.com, nlauth_signature=xxxx"

#Capture the response from your RESTlet.
/usr/bin/curl -H "${CONTENT_FLAG}" -H "${AUTH_STRING}" -d "${DATA_FILE}" $my_url > /tmp/restlet_response.txt
```

SuiteScript 2.0 RESTlet Script Entry Points

<table>
<thead>
<tr>
<th>Script Entry Point</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>All RESTlet entry points return the HTTP response body.</td>
</tr>
<tr>
<td>delete</td>
<td></td>
</tr>
<tr>
<td>Script Entry Point</td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td></td>
</tr>
<tr>
<td>put</td>
<td></td>
</tr>
<tr>
<td>post</td>
<td></td>
</tr>
</tbody>
</table>

### delete

**Description**

Returns the HTTP response body.

- Returns a string when request Content-Type is 'text/plain'.
- Returns an Object when request Content-Type is 'application/json'.

**Returns**

string | object

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestParams</td>
<td>Object</td>
<td>required</td>
<td>The parameters from the HTTP request URL. For all content types, parameters are passed into the function as a JavaScript Object.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

### get

**Description**

Returns the HTTP response body.

- Returns a string when request Content-Type is 'text/plain'.
- Returns an Object when request Content-Type is 'application/json' or 'application/xml'.

**Returns**

string | object

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestParams</td>
<td>Object</td>
<td>The parameters from the HTTP request URL. For all content types, parameters are passed into the function as a JavaScript Object.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>
post

**Description**  Returns the HTTP response body.

- Returns a string when request Content-Type is 'text/plain'.
- Returns an Object when request Content-Type is 'application/json'.

**Returns**  string | object

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestBody</td>
<td>string</td>
<td>Object</td>
<td>required  The HTTP request body.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># Pass the request body as a string when the request Content-Type is 'text/plain'</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># Pass the request body as a JavaScript Object when the request Content-Type is 'application/json'.</td>
<td></td>
</tr>
</tbody>
</table>

put

**Description**  Returns the HTTP response body.

- Returns a string when request Content-Type is 'text/plain'.
- Returns an Object when request Content-Type is 'application/json'.

**Returns**  string | object

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestBody</td>
<td>string</td>
<td>Object</td>
<td>required  The HTTP request body.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># Pass the request body as a string when the request Content-Type is 'text/plain'</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td># Pass the request body as a JavaScript Object when the request Content-Type is 'application/json'.</td>
<td></td>
</tr>
</tbody>
</table>
SuiteScript 2.0 Scheduled Script Type

Scheduled scripts are server-side 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 2.0 scheduled scripts, see the following:

# SuiteScript 2.0 Scheduled Script Reference
# SuiteScript 2.0 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 2.0 Scheduled Script Execution
# SuiteScript 2.0 Scheduled Script Debugging
# SuiteScript 2.0 Scheduled Script Status Page
# SuiteScript 2.0 Scheduled Script Handling of Server Restarts
# SuiteScript 2.0 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.

Scheduled Script Use Cases

Use this script type for basis scheduled or on demand tasks. Your SuiteScript 2.0 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-side script

Note: If you previously used SuiteScript 1.0 scheduled scripts, many of the use cases for those scripts now apply to the SuiteScript 2.0 SuiteScript 2.0 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.
With SuiteScript 2.0 scheduled scripts, you cannot set recovery points and you do not have the ability to yield. There is no SuiteScript 2.0 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 2.0 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

<table>
<thead>
<tr>
<th>Script Entry Point</th>
<th>Definition of the scheduled script trigger point</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute</td>
<td>Definition of the scheduled script trigger point</td>
</tr>
</tbody>
</table>

Scheduled Script API

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>context.InvocationType</td>
<td>Enumeration that holds the string values for scheduled script execution contexts.</td>
</tr>
</tbody>
</table>

Scheduled Script Sample

This sample script 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,

For help with writing scripts in SuiteScript 2.0, see SuiteScript 2.0 Hello World and SuiteScript 2.0 Entry Point Script Creation and Deployment.

```javascript
/**
*NApiVersion 2.x
*NScriptType ScheduledScript
*/
define(['N/search', 'N/record', 'N/email', 'N/runtime'],
    function (search, record, email, runtime) {
        function execute(context) {
            if (context.type !== context.InvocationType.ON_DEMAND)
                return;
            var searchId = runtime.getCurrentScript().getParameter("custscript_searchid");
            try {
                search.load({
                    id: searchId
                }).run().each(function(result) {
                    log.debug({
                        details: 'transforming so :' + result.id + ' to item fulfillment'
                    });
                });
            } catch (e) {
                log.error({
                    details: 'Error loading search: ' + searchId
                });
            }
        }
    });
```
SuiteScript 2.0 Scheduled Script Type

SuiteScript 2.0 Scheduled Script Reference

# SuiteScript 2.0 Scheduled Script Submission
# SuiteScript 2.0 Scheduled Script Execution
# SuiteScript 2.0 Scheduled Script Debugging
# SuiteScript 2.0 Scheduled Script Status Page
# SuiteScript 2.0 Scheduled Script Handling of Server Restarts

SuiteScript 2.0 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

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script</td>
<td>A link to the script record associated with the deployment</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>The user-defined name for the deployment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>A unique identifier for the deployment:</td>
</tr>
<tr>
<td>#</td>
<td>If this field is left blank when the deployment is created, the system generates the ID.</td>
</tr>
<tr>
<td>#</td>
<td>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 &quot;_&quot; as a delineator.</td>
</tr>
</tbody>
</table>

**Note:** For both custom and system-defined IDs, the system appends the string “customdeploy” to the ID.

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**.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployed</td>
<td>Indicates whether the deployment is active. This box must be enabled if you want your script to execute.</td>
</tr>
<tr>
<td>If you disable this option:</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>The script does not execute, regardless of the status. This applies even if a submission schedule is configured on the Schedule subtab.</td>
</tr>
<tr>
<td>#</td>
<td>The <strong>Save and Execute</strong> option no longer displays on the <strong>Save</strong> dropdown when the deployment record is in edit mode.</td>
</tr>
<tr>
<td>#</td>
<td>The script cannot be submitted programmatically. For more information, see Submitting an On Demand Scheduled Script Instance from a Script.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>This value determines how and when a script can be submitted to SuiteCloud Processors for processing. Possible options are:</td>
</tr>
<tr>
<td>#</td>
<td><strong>Testing:</strong> You can test the script in the SuiteScript Debugger by using Deployed Debugging. Deployed debugging is restricted to the script owner.</td>
</tr>
<tr>
<td>#</td>
<td><strong>Scheduled:</strong> 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.</td>
</tr>
</tbody>
</table>
### Field | Description
--- | ---
| **Note:** If you schedule a recurring submission of the script, the script's deployment status on the SuiteScript 2.0 Scheduled Script Status Page remains as Scheduled, even after the script completes its execution. | 
| **# Not Scheduled:** The script is not currently scheduled for automatic submission. You can submit an on demand instance of the script with either: | 
| # The Save and Execute button on the deployment record | 
| # The task.ScheduledScriptTask API | 
| You can submit an on demand instance of the script only if there is no other unfinished instance of the same script. | 
| **See Instances** | A link to the SuiteScript 2.0 Scheduled Script Status Page, filtered for all deployment instances of the script associated with this particular deployment. |
| **Log Level** | This value determines the information logged for this deployment. Entries are displayed on the Execution Log subtab. Possible options are: |
| # **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** | 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 |
| **Note:** Each SuiteScript 2.0 scheduled script instance is handled by a single job within SuiteCloud Processors. | |
| **Important:** You must understand SuiteCloud Processors before you change this setting. | 
| Possible options are: | |
| # **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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Event</td>
<td>The scheduled script task is submitted only once. Use this option if you want to schedule a future one time submission.</td>
</tr>
<tr>
<td>Daily Event</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Weekly Event</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Monthly Event</td>
<td>The scheduled script instance is submitted at least once per scheduled month.</td>
</tr>
<tr>
<td>Yearly Event</td>
<td>The scheduled script instance is submitted at least once per year.</td>
</tr>
<tr>
<td>Start Date</td>
<td>The first submission occurs on this date. This field is mandatory if a one time or recurring schedule is set.</td>
</tr>
<tr>
<td>Start Time</td>
<td>If a value is selected, the first submission occurs at this time.</td>
</tr>
<tr>
<td>Repeat</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>End By</td>
<td>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.</td>
</tr>
<tr>
<td>No End Date</td>
<td>The schedule does not have a set end date.</td>
</tr>
</tbody>
</table>

Scheduled Script Deployments that Continue to Use Queues

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

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 2.0 Hello World and SuiteScript 2.0 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

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**. You cannot step through an entry point script (or any other script that uses the `define` statement) with On Demand Debugging.

**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 2.0 Hello World** and **SuiteScript 2.0 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

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 2.0 Hello World and SuiteScript 2.0 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-side script where you want to submit the scheduled script instance, call task.create(options) to return a task.ScheduledScriptTask object:
   ```javascript
   var scriptTask = task.create({
     taskType: task.TaskType.SCHEDULED_SCRIPT,
   });
   ```
9. Set the ScheduledScriptTask.scriptId and ScheduledScriptTask.deploymentId properties:
   ```javascript
   scriptTask.scriptId = 1234;
   scriptTask.deploymentId = 'customdeploy1';
   ```
10. Call ScheduledScriptTask.submit() to submit the scheduled script instance to **SuiteCloud Processors**.

### SuiteScript 2.0 Scheduled Script Execution

All scheduled script instances are executed (processed) by **SuiteCloud Processors**. For additional information on how scheduled scripts are submitted, see SuiteScript 2.0 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 2.0 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**. You cannot step through an entry point script (or any other script that uses the define statement) with **On Demand Debugging**.
SuiteScript 2.0 Scheduled Script Status Page

The Scheduled Script Status page shows the current and past runtime statuses of all scheduled script instances in your account that have submitted in the last 30 days. 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.

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pending</td>
<td>The script is submitted and is waiting to be processed.</td>
</tr>
<tr>
<td>Deferred</td>
<td>The script is eligible for processing but has not been processed due to constraints. For example, deferred status occurs when one job must wait for another to finish.</td>
</tr>
<tr>
<td>Processing</td>
<td>The script is running.</td>
</tr>
<tr>
<td>Retry</td>
<td>The script entered the processing state and failed to complete normally. It is eligible to be retried.</td>
</tr>
<tr>
<td>Complete</td>
<td>The script completed normally.</td>
</tr>
<tr>
<td>Cancelled</td>
<td>Due to a NetSuite server error, the script was canceled during or before script execution.</td>
</tr>
<tr>
<td>Failed</td>
<td>The script began processing but failed to complete normally. Examine your script for possible errors.</td>
</tr>
</tbody>
</table>

SuiteScript 2.0 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 2.0, 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 2.0, 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:

```javascript
// 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`.

```
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 Example 5: A Robust Map/Reduce Script.
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.

```javascript
/**
 * @NApiVersion 2.0
 * @NScriptType scheduledscript
 */
define([ 'N/search', 'N/record' ],
    function (search, record){
        return {
            execute: function (context)
            {
                var filter1 = search.createFilter(
                    name: 'mainline',
                    operator: search.Operator.IS,
                    values: true);
                var filter2 = search.createFilter(
                    name: 'custbody_processed_flag',
                    operator: search.Operator.IS,
                    values: false);
                var srch = search.create(
                    type: search.Type.SALES_ORDER,
                    filters: [filter1, filter2],
                    columns: []);
                var pagedResults = srch.runPaged();

                pagedResults.pageRanges.forEach(function(pageRange){
                    var currentPage = pagedResults.fetch({ index: pageRange.index });
                    currentPage.data.forEach(function(result){
                        var so = record.load(
                            type: record.Type.SALES_ORDER,
                            id: result.id);
                    });
                });
            }
        }
    })
```
SuiteScript 2.0 Script Entry Points and API

Entry Points:

# execute

API:

# context.InvocationType

execute

<table>
<thead>
<tr>
<th>Description</th>
<th>Definition of the scheduled script trigger point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

Parameters

Note: The scriptContext parameter is a JavaScript object. NetSuite automatically passes this object to the script entry point.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.type</td>
<td>string</td>
<td>required</td>
<td>The context in which the script is executed. Values are reflected in the context.InvocationType enum.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

context.InvocationType

<table>
<thead>
<tr>
<th>Enum Description</th>
<th>Enumeration that holds the string values for scheduled script execution contexts.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>SuiteScript 2.0 Scheduled Script Type</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>SCHEDULED</th>
<th>The normal execution according to the deployment options specified in the UI.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON_DEMAND</td>
<td>The script is executed via a call from a script (using ScheduledScriptTask.submit()).</td>
</tr>
</tbody>
</table>
SuiteScript 2.0 Suitelet Script Type

Suitelets are extensions of the SuiteScript API that allow you to build custom NetSuite pages and backend logic. Suitelets are server-side 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, 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, or to other external hosts outside of NetSuite.
   
   # Offloading server logic from client scripts to a backend Suitelet shipped without source code to protect sensitive intellectual property.

   **Important:** RESTlets can be used as an alternative to backend Suitelets.

**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.

See the following for more information about the Suitelet script type:

- SuiteScript 2.0 Suitelet Script Reference
- SuiteScript 2.0 How Suitelet Scripts are Executed
- SuiteScript 2.0 Reserved Parameter Names in Suitelet URLs
- SuiteScript 2.0 Suitelet Script Best Practices
- SuiteScript 2.0 Suitelet Script Deployment Page
- SuiteScript 2.0 Embedding HTML in Suitelets
- SuiteScript 2.0 Suitelet Script Entry Points and API
SuiteScript 2.0 Suitelet Script Type Sample

For help with writing scripts in SuiteScript 2.0, see SuiteScript 2.0 Hello World and SuiteScript 2.0 Entry Point Script Creation and Deployment.

The following sample shows how to create a Suitelet form that lets you write and send an email:

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

// this creates a Suitelet form that lets you write and send an email

define([ 'N/ui/serverWidget', 'N/email', 'N/runtime' ],
    function (ui, email, runtime) {
        function onRequest(context) {
            if (context.request.method === 'GET') {
                var form = ui.createForm({
                    title: 'Demo Suitelet Form'
                });
                var subject = form.addField({
                    id: 'subject',
                    type: ui.FieldType.TEXT,
                    label: 'Subject'
                });
                subject.layoutType = ui.FieldLayoutType.NORMAL;
                subject.breakType = ui.FieldBreakType.STARTCOL;
                subject.isMandatory = true;
                var recipient = form.addField({
                    id: 'recipient',
                    type: ui.FieldType.EMAIL,
                    label: 'Recipient email'
                });
                recipient.isMandatory = true;
                var message = form.addField({
                    id: 'message',
                    type: ui.FieldType.TEXTAREA,
                    label: 'Message'
                });
                message.displaySize = {
                    width: 60,
                    height: 10
                }; form.addSubmitButton({
                    label: 'Send Email'
                });
                context.response.writePage(form);
            } else {
                var request = context.request;
                email.send({
                    author: runtime.getCurrentUser().id,
                    recipients: request.parameters.recipient,
                    message: request.parameters.message
                });
            }
        }
    });
```
SuiteScript 2.0 Suitelet Script Reference

SuiteScript 2.0 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
SuiteScript 2.0 Suitelet Script Type

# A browser redirect to another page on the Internet

⚠️ Important: You can only redirect to external URLs from Suitelets that are accessed externally (in other words, the Suitelet has been designated as “Available Without Login” and is accessed from its external URL).

# A browser redirect to an internal NetSuite page. The NetSuite page can be either a standard page or custom page that has been dynamically generated using UI objects.

4. The data renders in the user's browser.

SuiteScript 2.0 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:

<table>
<thead>
<tr>
<th>Reserved Suitelet URL Parameter Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
</tr>
<tr>
<td>id</td>
</tr>
<tr>
<td>cp</td>
</tr>
<tr>
<td>l</td>
</tr>
<tr>
<td>popup</td>
</tr>
<tr>
<td>s</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>_nodrop</td>
</tr>
<tr>
<td>sc</td>
</tr>
<tr>
<td>sticky</td>
</tr>
</tbody>
</table>

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 2.0 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 Web Site 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.

SuiteScript 2.0 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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script</td>
<td>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.</td>
</tr>
<tr>
<td>Title</td>
<td>The user-defined name for the deployment.</td>
</tr>
<tr>
<td>ID</td>
<td>A unique identifier for the deployment. 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. If you do not enter a value, the system automatically generates one. In both cases, the system automatically adds the prefix customdeploy to the ID created when the record is saved. Although not recommended, you can change the ID on an existing deployment by clicking the Change ID button.</td>
</tr>
<tr>
<td>Deployed</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>When the Deployed box is cleared, and the status is Scheduled, any times configured on the Schedule subtab are ignored and the script deployment is not submitted.</td>
</tr>
<tr>
<td>Status</td>
<td>A value that determines how and when a script deployment can be submitted for processing. The primary options are:</td>
</tr>
<tr>
<td></td>
<td>Scheduled — The script deployment is submitted for processing at the times indicated on the Schedule subtab.</td>
</tr>
<tr>
<td></td>
<td>Not Scheduled — The script deployment is submitted for processing only when invoked manually, either through the UI or programmatically. Note also that, regardless of the Status, the system submits the deployment for processing only if the Deployed box is checked.</td>
</tr>
</tbody>
</table>
## Event Type
Use the Event Type list to specify a script execution context at the time of script deployment.

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.

**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.

## 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:

- # **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.)

**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, which is supported for 2019.1 and future releases.

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.

**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:

<table>
<thead>
<tr>
<th>Suitelet URL Type</th>
<th>Deployment Status</th>
<th>Select All check boxes</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal</td>
<td>Testing</td>
<td>not checked</td>
<td>Suitelet deploys successfully</td>
</tr>
<tr>
<td>internal</td>
<td>Testing</td>
<td>checked</td>
<td>Suitelet deploys successfully</td>
</tr>
<tr>
<td>internal</td>
<td>Released</td>
<td>not checked</td>
<td>Error message: You do not have privileges to view this page.</td>
</tr>
<tr>
<td>internal</td>
<td>Released</td>
<td>checked</td>
<td>Suitelet deploys successfully</td>
</tr>
<tr>
<td>external</td>
<td>Testing</td>
<td>not checked</td>
<td>Error message: You are not allowed to navigate directly to this page.</td>
</tr>
<tr>
<td>external</td>
<td>Released</td>
<td>checked</td>
<td>Suitelet deploys successfully</td>
</tr>
<tr>
<td>external</td>
<td>Released</td>
<td>not checked</td>
<td>Error message: You do not have privileges to view this page.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roles</td>
<td>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.</td>
</tr>
<tr>
<td>Groups</td>
<td>Select the Groups that can access your Suitelet. If you do not select any Groups, all Groups have access.</td>
</tr>
<tr>
<td>Partners</td>
<td>Select the Partners that can access your Suitelet. To give access to all Partners, check the All Partners box.</td>
</tr>
<tr>
<td>Departments</td>
<td>Select the Departments that can access your Suitelet. If you do not select any Departments, all Departments have access.</td>
</tr>
<tr>
<td>Employees</td>
<td>Select the Employees that can access your Suitelet. To give access to all Employees, check the All Employees box.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
SuiteScript 2.0 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.

```javascript
var htmlImage = form.addField({
  id: 'custpage_htmlfield',
  type: serverWidget.FieldType.INLINEHTML,
  label: 'HTML Image'
});
htmlImage.defaultValue = "<img src='https://system.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.

![Contact Information](image.png)

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.

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

define([ 'N/https', 'N/record', 'N/email', 'N/search'],
 function  callbackFunction(https, record, email, search) {

  function  getFunction(context) {
    var  contentRequest = https.get({
      url : "https://LinkToFormPage.html" // see the next snippet
    });
    var  contentDocument = contentRequest.body;
    var  sponsorid = context.request.parameters.sponsorid;
    if  (sponsorid && sponsorid != "" && sponsorid != null ) {
      contentDocument = contentDocument.replace("{{sponsorid}}", sponsorid);
      log.debug("Setting Sponsor", sponsorid)
    }
    var  projectid = context.request.parameters.projectid;
    if  (projectid && projectid != "" && projectid != null ) {
      contentDocument = contentDocument.replace("{{projectid}}", projectid);
      log.debug("Setting Project", projectid);
    }
    context.response.write(contentDocument);
  }

  function  postFunction(context) {
    var  params = context.request.parameters;
    var  emailString = "First Name: {{firstname}}
    Last Name: {{lastname}}
    Email: {{email}}
    Facebook URL: {{custentity_fb_url}}"
    var  contactRecord = record.create({
      type : "contact",
      isDynamic: true
    });
    for  (param in  params) {
      if  (param === "company") {
        if  (params[param] !== "{{sponsorid}}") {
          contactRecord.setValue({
            fieldId: param,
            value: params[param]
          });
          var  lkpfld = search.lookupFields({
            type: "vendor",
            id: params["company"],
            columns: ["entityid"]
          });
        }
      }
    }
  }
});
```
SuiteScript 2.0 Suitelet Script Type

```javascript
var contactId = contactRecord.save({
  ignoreMandatoryFields: true,
  enableSourcing: true
});

log.debug( "Record ID", contactId);

if (params["project"] && params["project"] !== "" && params["project"] != null && params["project"] !== {{projectid}}) {
  var lkpfld = search.lookupFields({
    type: "job",
    id: params["project"],
    columns: ["companyname"]
  });

  emailString += "Project Name: " + lkpfld.companyname;

  var participationRec = record.create({
    type: "customrecord_project_participants",
    isDynamic: true
  });

  participationRec.setValue({
    fieldId: "custrecord_participants_volunteer",
    value: contactId
  })

  participationRec.setValue({
    fieldId: "custrecord_participants_project",
    value: params["project"]
  })

  var participationId = participationRec.save(
```
```javascript

email.send({
    author: -5,
    recipients: params["email"],
    subject: "Thank you!",
    body: "Thank you for volunteering:

" + emailString
});

var contentRequest = https.get({
    url: "https://LinkToFormCompletePage.html"
});

var contentDocument = contentRequest.body;
context.response.write(contentDocument);

function onRequestFxn(context) {
    if (context.request.method === "GET") {
        getFunction(context)
    } else {
        postFunction(context)
    }
}

return {
    onRequest: onRequestFxn
};
```
The following screenshot illustrates the form that is created by the linked HTML page.

**SuiteScript 2.0 Suitelet Script Entry Points and API**

```
# onRequest(params)

onRequest(params)
```

**Description**
Definition of the Suitelet script trigger point.

**Returns**
void

**Since**
Version 2015 Release 2

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
</table>
SuiteScript 2.0 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.

**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.

For additional information about SuiteScript 2.0 User Event Scripts, see the following:

- SuiteScript 2.0 User Event Script Reference
- How User Events are Executed
- User Event Script Best Practices
- SuiteScript 2.0 User Event Script Tutorial
- SuiteScript 2.0 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.

SuiteScript 2.0 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 2.0, see SuiteScript 2.0 Hello World and SuiteScript 2.0 Entry Point Script Creation and Deployment.

```javascript
/**
 * @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); // replace '46' with one specific to your account
    }
    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 (error) {
                log.error(error.name);
            }
        }
    }
    return {
        beforeLoad: beforeLoad,
        beforeSubmit: beforeSubmit,
        afterSubmit: afterSubmit
    };
```
SuiteScript 2.0 User Event Script Reference

# How User Events are Executed
# User Event Script Best Practices
# SuiteScript 2.0 User Event Script Tutorial

How User Events are Executed

⚠️ 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.

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

1. The client sends a read operation request for record data. The client request can come from the user interface, SOAP web services, CSV import, or server–side 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.

The following steps and diagram provide 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–side SuiteScript calls, CSV imports, or XML. The application server:
   - performs basic permission checks on the client
   - 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.

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

User Event Script Best Practices

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.

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.
# Limit the amount of script execution in user event scripts (< 5 seconds) 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.
# Mission critical business logic implemented using user events should be accompanied by a 'Clean up' scheduled script to account for any unexpected errors or mis-fires.
# 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-side SuiteScripts. In addition to Sales Orders, scripts on Case and Customer records also execute because of 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. Another approach is to use a Suitelet as an intermediary. The first user event script invokes a Suitelet that creates or edits another record, and the second user event script is triggered on that record.

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.

SuiteScript 2.0 User Event Script Tutorial

A user event script is a server-side script that is triggered by actions taken on a record. This script type can include logic that is executed after a user has started the process of opening the record but before the record loads, after the user clicks Save, and 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 2.0 Script Basics for an explanation of terms used in this tutorial. For an overview of the basic structural elements required in any SuiteScript 2.0 entry point script, see SuiteScript 2.0 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 2.0 User Event Script Type. For details about these entry points, see beforeLoad(scriptContext), beforeSubmit(scriptContext), and afterSubmit(scriptContext).

Step One: 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.

Step Two: 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.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */

// Load two standard modules.
define ( [ 'N/record', 'N/ui/serverWidget' ] ,

  // Add the callback function.
  function (record, serverWidget) {

    // In the beforeLoad function, disable the Notes field.
    function myBeforeLoad (context) {
      if (context.type !== context.UserEventType.CREATE)
        return;
      var form = context.form;
      var notesField = form.getField({
        id : 'comments'
      });
      notesField.updateDisplayType({
        displayType : serverWidget.FieldDisplayType.DISABLED
      });
    }

    // In the beforeSubmit function, add test to the Notes field.
    function myBeforeSubmit(context) {
      if (context.type !== context.UserEventType.CREATE)
        return;
      var newEmployeeRecord = context.newRecord;
      newEmployeeRecord.setValue({
        fieldId : 'comments',
        value : 'Orientation date TBD.'
      });
    }

    // In the afterSubmit function, begin creating a task record.
    function myAfterSubmit(context) {
      if (context.type !== context.UserEventType.CREATE)
        return;
    }
  }
```

---

SuiteScript 2.0

[Oracle NetSuite logo]
var newEmployeeRecord = context.newRecord;
var newEmployeeFirstName = newEmployeeRecord.getValue({
    fieldId: 'firstname'
});
var newEmployeeLastName = newEmployeeRecord.getValue({
    fieldId: 'lastname'
});
var newEmployeeSupervisor = newEmployeeRecord.getValue({
    fieldId: 'supervisor'
});
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
        });
    }
}

// Add the return statement that identifies the entry point functions.
return {
    beforeLoad: myBeforeLoad,
    beforeSubmit: myBeforeSubmit,
    afterSubmit: myAfterSubmit
};

Step Three: 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.

```javascript
function myBeforeSubmit(context) {
    if (context.type !== context.UserEvent.Type.CREATE)
        return;

    var newEmployeeRecord = context.newRecord;
    newEmployeeRecord.setValue({
        fieldId: 'comments',
        value: 'Orientation date TBD.'
    });
}
```

<table>
<thead>
<tr>
<th>Callout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>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(context). This function uses two of those properties: type and newRecord. 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.</td>
</tr>
<tr>
<td>2</td>
<td>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.UserEvent.Type.</td>
</tr>
<tr>
<td>3</td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

```javascript
function myAfterSubmit(context) {
    if (context.type !== context.UserEvent.Type.CREATE)
        return;

    var newEmployeeRecord = context.newRecord;
    var newEmployeeFirstName = newEmployeeRecord.getValue({
        fieldId: 'firstname'
    });
    var newEmployeeLastName = newEmployeeRecord.getValue({
        fieldId: 'lastname'
    });
    var newEmployeeSupervisor = newEmployeeRecord.getValue({
        fieldId: 'supervisor'
    });
}
```
<table>
<thead>
<tr>
<th>Callout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>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 <code>type</code> property to identify the type of action the user is taking. To see a list of the possible actions, review <code>context.UserEventType</code>.</td>
</tr>
<tr>
<td>2</td>
<td>These statements retrieve several pieces of data from the record. They do so by using the context object’s <code>newRecord</code> property, which gives the script access to the record that has just been submitted. This code also uses the <code>Record.getValue(options)</code> method to retrieve the data.</td>
</tr>
</tbody>
</table>

The rest of the function creates the task record:

```javascript
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('Task record created successfully',
            title: 'Task record created successfully',
            details: 'New task record ID: ' + newTaskId
        );
    } catch (e) {
        log.error(
            title: e.name,
            details: e.message
        );
    }
}
```
return Statement

The following image shows the callback function’s return statement.

```
return {
    beforeLoad: myBeforeLoad,
    beforeSubmit: myBeforeSubmit,
    afterSubmit: myAfterSubmit
}
```

<table>
<thead>
<tr>
<th>Callout</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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. |

Step Four: 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. 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.

Step Five: 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.

---

**Step Six: 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.

   ![Confirmation](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 2.0 Custom Module Tutorial.

SuiteScript 2.0 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.

<table>
<thead>
<tr>
<th>Script Entry Point</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>afterSubmit(scriptContext)</td>
<td>Executes immediately after a write operation on a record.</td>
</tr>
<tr>
<td>beforeLoad(scriptContext)</td>
<td>Executes whenever a read operation occurs on a record, and prior to returning the record or page.</td>
</tr>
<tr>
<td>beforeSubmit(scriptContext)</td>
<td>Executes prior to any write operation on the record.</td>
</tr>
<tr>
<td>context.UserEventType</td>
<td>Holds the string values for user event execution contexts.</td>
</tr>
</tbody>
</table>

afterSubmit(scriptContext)

Description

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:

- 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.
- 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.

This event can be used with the following context.UserEventType:
SuiteScript 2.0 User Event Script Type

---

# 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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.newRecord</td>
<td>record.Record</td>
<td>required</td>
<td>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.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.oldRecord</td>
<td>record.Record</td>
<td>required</td>
<td>The old record in read only mode.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.type</td>
<td>string</td>
<td>required</td>
<td>The trigger type. Use the context.UserEventType enum to set this value.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

For an example of the afterSubmit entry point, see SuiteScript 2.0 User Event Script Sample.

**beforeLoad(scriptContext)**

**Description**

Executes whenever a read operation occurs on a record, and prior to returning the record or page.

These operations include navigating to a record in the UI, reading a record in SOAP web services, and loading a record.

The beforeLoad event cannot be used to source standard records. Use the pageInit client script for this purpose. See pageInit.

**Notes:**

# beforeLoad user events cannot be triggered when you load/access an online form.
Data cannot be manipulated for records that are loaded in beforeLoad scripts. If you attempt to update a record loaded in beforeLoad, the logic is ignored.

Data can be manipulated for records created in beforeLoad user events.

Attaching a child custom record to its parent or detaching a child custom record from its parent triggers an edit event.

This event can be used with the following `context.UserEventType`:

- `create`
- `edit`
- `view`
- `copy`
- `print`
- `email`
- `quick view`

## Returns

`void`

## Since

Version 2015 Release 2

### Parameters

**Note:** The `scriptContext` parameter is a JavaScript object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scriptContext.type</code></td>
<td><code>string</code></td>
<td>The type of operation invoked by the event.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td><code>scriptContext.request</code></td>
<td><code>http.ServerRequest</code></td>
<td>The HTTP request information sent by the browser. If the event was triggered by a server action, this value is not present.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

For an example of the `beforeLoad` entry point, see SuiteScript 2.0 User Event Script Sample.

### beforeSubmit(scriptContext)

**Description**

Executes prior to any write operation on the record.

Changes made to the current record in this script persist after the write operation.
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.

**Notes:**

# The approve, cancel, and reject argument types are only available for record types such as sales orders, expense reports, timebills, purchase orders, and return authorizations.
# Only beforeLoad and afterSubmit user event 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.

This event can be used with the following context.UserEventType:

# 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 the Mark Complete link on call and task records)
# reassign (specify this type for a beforeSubmit script to execute when users click the Grab link on case records)
# editforecast (specify this type for a beforeSubmit script to execute when users update opportunity and estimate records using the Forecast Editor)

**Returns**

void

**Since**

Version 2015 Release 2

**Parameters**

Note: The scriptContext parameter is a JavaScript object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.oldRecord</td>
<td>record.Record</td>
<td>required</td>
<td>The old record.</td>
<td>Version 2015 Release 2</td>
</tr>
<tr>
<td>scriptContext.type</td>
<td>string</td>
<td>required</td>
<td>The trigger type. Use the context.UserEventType enum to set this value.</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

For an example of the beforeSubmit entry point, see SuiteScript 2.0 User Event Script Sample.
context.UserEventType

<table>
<thead>
<tr>
<th>Enum Description</th>
<th>Holds the string values for user event execution contexts.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The following user events types are supported for user event entry points:</td>
</tr>
<tr>
<td></td>
<td># afterSubmit(scriptContext)</td>
</tr>
<tr>
<td></td>
<td># beforeLoad(scriptContext)</td>
</tr>
<tr>
<td></td>
<td># beforeSubmit(scriptContext)</td>
</tr>
</tbody>
</table>

**Note:** JavaScript does not include an enumeration type. The SuiteScript 2.0 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.

<table>
<thead>
<tr>
<th>Module</th>
<th>SuiteScript 2.0 User Event Script Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Since</td>
<td>Version 2015 Release 2</td>
</tr>
</tbody>
</table>

**Values**

# 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.

# COPY
# 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 2.0 User Event Script Sample.
SuiteScript 2.0 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 sublists 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 implemented 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 SuiteScript 2.0 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.

Workflow Action Script Sample

This sample shows how to store a return value from a custom action script into a workflow field. This example can be useful to satisfy the following use cases:

# You want to get a value from the Item sublist and use this value as a condition in the workflow. You use obj.getSublistValue in the script and return this in the workflow.
# You want to check if a certain item is existing 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 #2).

This script sample assumes the following set-up in the NetSuite account:

# Make sure 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 the 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.

For help with writing scripts in SuiteScript 2.0, see SuiteScript 2.0 Hello World and SuiteScript 2.0 Entry Point Script Creation and Deployment.

```/**
 * @NApiVersion 2.x
 */```
SuiteScript 2.0 Workflow Action Script Entry Points and API

onAction(scriptContext)

<table>
<thead>
<tr>
<th>Description</th>
<th>Defines a Workflow Action script trigger point.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>Since</td>
<td>Version 2016 Release 1</td>
</tr>
</tbody>
</table>
Parameters

i Note: The scriptContext parameter is a JavaScript object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required / Optional</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptContext.newRecord</td>
<td>record.Record</td>
<td>required</td>
<td>The new record. record.Record.save() is not permitted.</td>
<td>2016.1</td>
</tr>
<tr>
<td>scriptContext.oldRecord</td>
<td>record.Record</td>
<td>required</td>
<td>The old record. record.Record.save() is not permitted.</td>
<td>2016.1</td>
</tr>
<tr>
<td>scriptContext.form</td>
<td>serverWidget.Form</td>
<td>optional</td>
<td>The form through which the script interacts with the record. This parameter is available only in the beforeLoad context.</td>
<td>2016.2</td>
</tr>
<tr>
<td>scriptContext.type</td>
<td>string</td>
<td>optional</td>
<td>An event type, such as create, edit, view, or delete.</td>
<td>2016.2</td>
</tr>
<tr>
<td>scriptContext.workflowId</td>
<td>integer</td>
<td>optional</td>
<td>The internal ID of the workflow that calls the script.</td>
<td>2016.2</td>
</tr>
</tbody>
</table>

SuiteScript 2.0 SDF Installation Script Type

Installation scripts for SDF are available as a SuiteScript 2.0 script type. SuiteApp projects can include SDF installation scripts that are automatically executed when the project is deployed.

Use this script type when you want to write code that perform tasks during an SDF deployment to your target account. Tasks can include setup, configuration, and data management tasks that would otherwise have to be completed by account administrators. For example:

# Trigger code to verify the account
# Customize the installation
# Set up the target account
# Prevent installation if proper setup has not occurred
# Migrate existing data

For more information about changing the SuiteApp deploy file, see the help topic Customizing SuiteApp Project Deployment using SDF Installation Scripts.

You can use SuiteCloud Development Framework (SDF) to manage SDF 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 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 about Copy to Account, see the help topic Copy to Account Overview.

Using an SDF Installation Script

You can create script records and script deployments for instances of this script type the same ways you can for other SuiteScript 2.0 scripts.
Like all SuiteScript 2.0 scripts, an SDF installation script must include the required JSDoc tags, can use the define() function to load modules, and needs only one entry point function run. This script type also has its own parameters and supports custom script parameters.

SDF installation scripts are governed by a maximum of 10,000 units per execution. For more information, see the help topic Script Usage Unit Limits.

Parameters

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fromVersion</td>
<td>string</td>
<td>The version of the SuiteApp currently installed on the account.</td>
</tr>
<tr>
<td>toVersion</td>
<td>string</td>
<td>The version of the SuiteApp that will be installed on the account.</td>
</tr>
</tbody>
</table>

SDF installation script failures terminate SuiteApp deployments and can include their own error handling in addition to errors thrown by SDF and the SuiteScript engine. An error thrown by a SDF installation script returns an error followed by the text defined by the script author.

The following sample is a script that migrates a SuiteApp from version 1.0.0 to version 1.1.0.

Sample

```javascript
/**
 * @NApiVersion 2.0
 * @NScriptType SDFInstallationScript
 */
define([ 'N/record', 'N/search', 'N/runtime' ], function (record, search, runtime) {
    function migrate(context) {
        if (context.fromVersion === '1.0.0' && context.toVersion === '1.1.0') {
            var tSearch = search.create({
                type: search.Type.TRANSACTION,
                columns: [{
                    name: 'internalid'
                }],
                filters: [{
                    name: 'recordtype',
                    operator: search.Operator.IS,
                    values: ['customtransaction_install_script']
                }]
            });
            tSearch.run().each(function(result) {
                var tranInternalId = result.getValue({ name: 'internalid' });
                var tran = record.load({
                    type: 'customtransaction_install_script',
                    id: tranInternalId
                });
                var fieldValue = runtime.getCurrentScript().getParameter({ name: 'custscript1' });
                tran.setValue({
                    fieldId: 'custbody_install_script',
                    value: fieldValue
                });
            });
        }
    }
});
```
After you have created a .js file with your installation script code, you need to add this file to the file cabinet. If you are using the SuiteCloud IDE, this process is automated.

To execute the script during your SuiteApp project deployment, the script must be associated with a script record and deployment record. For more information, see Creating a Script Record and Running SuiteScript 1.0 in NetSuite Overview.

Script records and deployments can be created in your project as XML representations, using SuiteScript, and in your NetSuite account. For more information, see the help topics SDF Installation Scripts as XML Definitions and SuiteCloud Development Framework XML Reference.
SuiteScript 2.0 Record Actions and Macros

Overview of Record Action and Macro APIs

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 is triggered by the click of a UI button. 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 can click a button in the UI, or a script can run that calls the API that corresponds 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 script types, both client and server-side. 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.

View a video about executing Actions and Macros.

Supported Record Actions

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 ID.
Supported Record Macros

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 ID.

<table>
<thead>
<tr>
<th>Record Type</th>
<th>Macro ID</th>
<th>UI Button Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Billing Rate Card</td>
<td>modifyPriceByPercent</td>
<td>Recalculate</td>
</tr>
<tr>
<td>Project Charge Rule</td>
<td>copyResources</td>
<td>Copy Resources from Tasks</td>
</tr>
<tr>
<td>Weekly Timesheet</td>
<td>checkTimeLimits</td>
<td>Check Time Limits</td>
</tr>
<tr>
<td>Weekly Timesheet</td>
<td>copyFromWeek</td>
<td>Copy From Week</td>
</tr>
<tr>
<td>Transaction Record Macros</td>
<td>getSummaryTaxTotals</td>
<td>Get Summary Tax Totals</td>
</tr>
<tr>
<td>Sales Order</td>
<td>autoAssignLocations</td>
<td>Auto Assign Locations</td>
</tr>
<tr>
<td>Transaction Record Macros</td>
<td>calculateTax</td>
<td>Preview Tax</td>
</tr>
</tbody>
</table>
SuiteScript 2.0 JSDoc Validation

JSDoc 3 is a documentation generator for JavaScript source code. Users typically employ this tool to generate an HTML API reference. Developers insert specific comment blocks into their source code. 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).

```javascript
/**
 * Creates a file.
 * @param {string} name [required] - file name
 * @param {string} fileType [required] - file type
 * @param {string} contents [required] - file content
 * @returns {object} file.File
 */
```

JSDoc parses the source code for each comment block. The HTML output is then generated based on the content of the comment blocks and an evaluation of the code. JSDoc 3 also lets users create custom JSDoc tags. These tags can be defined to trigger events (for example, displaying a certain page).

SuiteScript 2.0 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 2.0 JSDoc Tags.

When a SuiteScript 2.0 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 2.0 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/](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.

<table>
<thead>
<tr>
<th>Valid Examples</th>
<th>Invalid Examples</th>
</tr>
</thead>
</table>
| /**
 * @NApiVersion 2.x
 */ | /*
 * @NApiVersion 2.x
 */ |

The JSDoc tag comment block does not start with /**.

// @NApiVersion 2.x
SuiteScript 2.0 JSDoc Validation

The JSDoc tag comment is a single-line comment and not a block.

```javascript
/**
 * @NApiVersion 2.x
 */
```

The JSDoc tag comment block does not include a carriage return after the value.

SuiteScript 2.0 JSDoc Tags

The following table describes the available SuiteScript 2.0 JSDoc tags. SuiteScript 2.0 entry point scripts must include the following two tags:

- `@NApiVersion`
- `@NScriptType`

For more information about entry point validation, including possible errors, see SuiteScript 2.0 Entry Point Script Validation.

<table>
<thead>
<tr>
<th>JSDoc Tag</th>
<th>Possible Values</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@NApiVersion</td>
<td>2.0</td>
<td>Required for entry point scripts</td>
<td>This tag is used in two ways: # For SuiteScript 2.0 entry point scripts, this tag is a required declaration. It indicates to NetSuite the SuiteScript version to use. # For SuiteScript 2.0 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. The 2.x value always represents the latest version of SuiteScript that is generally available. It does not represent any versions that are released as beta features. For more information about SuiteScript versioning, see SuiteScript Versioning Guidelines.</td>
</tr>
<tr>
<td></td>
<td>2.1</td>
<td>Optional for custom modules</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@NScriptType</td>
<td>BundleInstallationScript</td>
<td>Required for entry point scripts</td>
<td>This tag identifies the type of script defined in the file.</td>
</tr>
<tr>
<td></td>
<td>ConsolidatedRateAdjustorPlugin</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CustomGLPlugin</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ClientScript</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EmailCapturePlugin</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MapReduceScript</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MassUpdateScript</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PaymentGatewayPlugin</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: In NetSuite 2019.2, the 2.1 value indicates that a script uses SuiteScript 2.1, which is being released as a beta feature. For more information about SuiteScript 2.1, see the help topic SuiteScript 2.1 Beta.
## Controlling Access to Scripts and Custom Modules

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.

### JSDoc Tag

<table>
<thead>
<tr>
<th>Possible Values</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PluginTypeImpl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Portlet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PromotionsPlugin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Restlet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ScheduledScript</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShippingPartnersPlugin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suitelet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaxCalculationPlugin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UserEventScript</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WorkflowActionScript</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### @NModuleScope

<table>
<thead>
<tr>
<th>Possible Values</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SameAccount</td>
<td>Optional</td>
<td>This tag is used to control access to scripts and custom modules.</td>
</tr>
<tr>
<td>TargetAccount</td>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td>Public</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- # 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.
- The default value is SameAccount.

For more information, see [Controlling Access to Scripts and Custom Modules](#).
If you do not set the scope, SameAccount applies as the default. For information about adding a JSDoc tag, see SuiteScript 2.0 JSDoc Validation.

The following table lists and describes the access control modifiers (supported module scopes) that are available in SuiteScript 2.0:

<table>
<thead>
<tr>
<th>ModuleScope Value</th>
<th>Description</th>
<th>Access Disallowed When:</th>
<th>Visibility of Source Code</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>SameAccount</td>
<td>Limits script access to modules native to the same environment in which the script was created and uploaded.</td>
<td>A bundled module from a different source account attempts to import the module.</td>
<td>Visible during runtime</td>
<td># 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).</td>
</tr>
<tr>
<td></td>
<td>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.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A script file that is 'native' to the environment is added to an account using an authenticated user session.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TargetAccount</td>
<td>Limits script access to modules native to the same source or target environment as the script.</td>
<td>A bundled module that is not native to the target or source account attempts to import the module.</td>
<td>Hidden during runtime</td>
<td># 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.</td>
</tr>
<tr>
<td></td>
<td>A source environment includes the NetSuite account (and any associated sandboxes and Release Preview accounts) from which a bundle is installed into another account.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A target environment includes the NetSuite account (and any associated sandboxes and Release Preview accounts) into which a bundle is installed (whether via Bundle Copy or Bundle Install).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Public</td>
<td>Any bundle (native and third party) that is installed in the account can run the script.</td>
<td>The script is not installed in the account.</td>
<td>Hidden during runtime</td>
<td># 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.</td>
</tr>
</tbody>
</table>

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 2.0 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. Make sure that the file has the required script elements, as described in SuiteScript 2.0 Entry Point Script Validation, and upload the file to your File Cabinet.

2. Do one of the following:
   - Deploy your script on one or more record types, as described in SuiteScript 2.0 Record-Level Scripts.
   - If your script is a client script that should be deployed only at the form level, follow the steps described in SuiteScript 2.0 Form-Level Scripts.

For help understanding 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 scripts can be deployed at the record level only.

Record-Level and Form-Level Scripts

A client script can be deployed in one of two ways: 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 the 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 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 run 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 form 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 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.

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 full details about deploying a script at the record level, see SuiteScript 2.0 Record-Level Scripts. For details about deploying a client script at the form level, see SuiteScript 2.0 Form-Level Scripts.

Note: Record-level client scripts can also be used on forms and lists that have been generated through UI Objects during Suitelets development. Form-based client scripts cannot be used by Suitelets.

SuiteScript 2.0 Entry Point Script Validation

For a SuiteScript 2.0 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. This data is used later when you create a script record based on the script.
If NetSuite detects that an element within the 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. If you are attaching a client script to a custom form, errors can also be returned at that time.

For more details, see the following sections:

* Entry Point Script Validation Guidelines
* Entry Point Script Validation Examples
* Entry Point Script Validation Error Reference

**Entry Point Script Validation Guidelines**

For an entry point script to be properly formatted, its structure must meet certain guidelines. Additionally, the script must include two required JSDoc tags.

If the script does not meet all requirements, you cannot create a script record based on the script, nor can you attach it to a form. In some cases, you are not permitted to upload the file. Similarly, if a correctly formatted script has been attached to a script record or a custom form, you are not permitted to edit the file and save changes that would introduce errors.

For details, see the following sections:

* 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.
At a high level, the following rules apply:

# Each script must have a structure that meets certain criteria, as described in the next section, Correct Structure for Entry Point Scripts.

# Each script file must include two JSDoc tags: @NApiVersion and @NScriptType. 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 criteria. That is, 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. Put another way, the return statement must include only one script type interface.
# The interface 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.
# JavaScript syntax errors are not permitted.

Required JSDoc Tags for Entry Point Scripts

Every entry point script must contain the JSDoc tags described in the following table. Note that this table includes details about some errors, but for full details about validation errors, see Entry Point Script Validation Error Reference.

<table>
<thead>
<tr>
<th>JSDoc Tag</th>
<th>Possible Values</th>
<th>Purpose</th>
<th>Possible Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>@NApiVersion</td>
<td>2.0, 2.1, 2.x, 2.X</td>
<td>Identifies the SuiteScript version to use.</td>
<td>If you omit this tag within a file that is properly formatted in all other ways for SuiteScript 2.0, the upload of the script fails. For details, see FAILED_TO_VALIDATE_SCRIPT_FILE.</td>
</tr>
<tr>
<td>@NScriptType</td>
<td>BundleInstallationScript, ConsolidatedRateAdjustorPlugin, CustomGLPlugin, ClientScript, EmailCapturePlugin, MapReduceScript, MassUpdateScript, PaymentGatewayPlugin, PluginTypeImpl, Portlet, PromotionsPlugin, Restlet, ScheduledScript, ShippingPartnersPlugin</td>
<td>Identifies the type of script defined in the file.</td>
<td>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.</td>
</tr>
</tbody>
</table>

Note: In NetSuite 2019.2, the 2.1 value indicates that a script uses SuiteScript 2.1, which is being released as a beta feature. For more information about SuiteScript 2.1, see the help topic SuiteScript 2.1 Beta.
### JSDoc Tag

<table>
<thead>
<tr>
<th>JSDoc Tag</th>
<th>Possible Values</th>
<th>Purpose</th>
<th>Possible Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Suitelet</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TaxCalculationPlugin</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UserEventScript</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WorkflowActionScript</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For full details about working with JSDoc tags in SuiteScript 2.0, including details on formatting them properly, see SuiteScript 2.0 JSDoc Validation.

## Entry Point Script Validation Examples

The following examples show correct and incorrect code snippets.

### Correct: Includes All Required Elements

The following script is correctly formatted. It includes a define statement, a return statement with an entry point, and a function that corresponds with the entry point. Additionally, the value of the @NScriptType tag matches the script type interface used.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */

define([ 'N/record' ],

function (record) {
    function myBeforeSubmitFunction(context) {
        if (context.type !== context.UserEventTypes.CREATE) return;
        var customerRecord = context.newRecord;
        customerRecord.setValue({
            fieldId: 'comments',
            value: 'Please follow up with this customer!
        });
    }
    return {
        beforeSubmit: myBeforeSubmitFunction
    };
});
```

### Incorrect: Missing Return Statement

The following script does not have a return statement. If you try to upload a script structured this way, the system returns an error reading “SuiteScript 2.0 entry point scripts must implement one script type function.” For details, see SCRIPT_OF_API_VERSION_20 MUST ...
Incorrect: Missing Define Statement

The following script uses a require statement instead of a define statement. If you try to upload a script structured like this, the system returns an error reading “SuiteScript 2.0 entry point scripts must implement one script type function.” For details, see SCRIPT_OF_API_VERSION_20 MUST ....

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */

require(['N/record'], function (record) {
  function myBeforeSubmitFunction(context) {
    if (context.type !== context.UserEventTypes.CREATE) return;
    var customerRecord = context.newRecord;
    customerRecord.setValue('comments', 'Please follow up with this customer!');
  }
});
```

Incorrect: Missing JSDoc tag

The following script is incorrectly formatted because it does not have an @NScriptType tag. The system would permit you to upload a file with this script. However, if you try to create a script record with this script, the system returns an error reading “@NScriptType is mandatory for 2.0 entry point script.” For details, see MISSING_SCRIPT_TYPE.

```javascript
/**
 * @NApiVersion 2.0
 */
define(['N/record'], function (record) {
  function myBeforeSubmitFunction(context) {
    if (context.type !== context.UserEventTypes.CREATE) return;
    var customerRecord = context.newRecord;
    customerRecord.setField({
      fieldId: 'comments',
      value: 'Please follow up with this customer!
    });
  }
});
```
Incorrect: Missing JSDoc tag

The following script is incorrectly formatted, because it does not have an @NApiVersion tag. If you try to upload a script structured like this, the system returns an error reading “Failed to validate script file.” For details, see FAILED_TO_VALIDATE_SCRIPT_FILE.

```javascript
/**
 * @NScriptType userevents
 */

define([ 'N/record' ], function(record) {
    function myBeforeSubmitFunction(context) {
        if (context.type !== context.UserEventTypes.CREATE)
            return;
        var customerRecord = context.newRecord;
        customerRecord.setValue('comments', 'Please follow up with this customer!');
    }

    return {
        beforeSubmit : myBeforeSubmitFunction
    };
});
```

Entry Point Script Validation Error Reference

The following table describes errors that can be returned when working with entry point scripts. These errors can be thrown during the process of uploading entry point scripts, when creating script records, or when attaching scripts to forms. Some errors can also be returned when editing script files that have already been uploaded to NetSuite, attached to script records, or attached to forms.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Long Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANNOT_CHANGE_API_VERSION</td>
<td>This file is used by a SuiteScript {1} script; you cannot change the API version of the file.</td>
</tr>
<tr>
<td>FAILED_TO_VALIDATE_SCRIPT_FILE</td>
<td>Failed to validate script file: {1}</td>
</tr>
<tr>
<td>INVALID_API_VERSION</td>
<td>Invalid JSDoc tag value; valid values are: @NApiVersion [2.X, 2.x, 2.0]</td>
</tr>
<tr>
<td>INVALID_JSDOC_TAG_VALUE</td>
<td>Invalid JSDoc tag value; valid values are: {1}</td>
</tr>
<tr>
<td>MISSING_SCRIPT_TYPE</td>
<td>@NScriptType is mandatory for 2.0 entry point script.</td>
</tr>
<tr>
<td>MULTIPLE_DEFINE_CALLS</td>
<td>Invalid define call, define should only be called one time per module. Define calls found at the following line numbers: {1}</td>
</tr>
<tr>
<td>SCRIPT_OF_API_VERSION_20_CANNOT_IMPLEMENT_MORE_THAN_ONE_SCRIPT_TYPE_INTERFACES</td>
<td>SuiteScript 2.0 entry point scripts cannot implement functions for more than one script type.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Long Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------------</td>
</tr>
<tr>
<td>SCRIPT_OF_API_VERSION_20_MUST_IMPLEMENT_A_SCRIPT_TYPE_INTERFACE</td>
<td>SuiteScript 2.0 entry point scripts must implement one script type function.</td>
</tr>
<tr>
<td>SS_V2_FILE_USED_FOR_FORM_SCRIPTS_MUST_IMPLEMENT_CLIENT_SCRIPT_TYPE</td>
<td>SuiteScript version 2 file used for form scripts must implement client script type.</td>
</tr>
<tr>
<td>SYNTAX_ERROR</td>
<td>Syntax error: {1}</td>
</tr>
<tr>
<td>THE_FILE_IS_USED_AS_SCRIPT_TYPE_1_CANNOT_BE_CHANGED_TO_2</td>
<td>The file is used as script type {1}, cannot be changed to {2}.</td>
</tr>
<tr>
<td>WRONG_SCRIPT_TYPE</td>
<td>Script file includes @NScriptType {1}; this script type cannot be used to {2}.</td>
</tr>
</tbody>
</table>

**CANNOT_CHANGE_API_VERSION**

The long description for this error code is: This file is used by a SuiteScript {1} script; you cannot change the API version of the file.

This error can be displayed when you attempt to edit a script file associated with a script record or a custom form. Specifically, this error is displayed if you try to edit and save changes to the @NApiVersion value. For example, if you try to remove the expression @NApiVersion 2.0 from a version 2.0 script, the system displays this error. Similarly, if you try to add @NApiVersion 2.0 expression to a version 1.0 script, the system displays this error.

In some cases, this error is preceded by the INVALID_API_VERSION error. The INVALID_API_VERSION error is included if the new @NApiVersion value you are trying to use is invalid.

Note that SuiteScript 1.0 files do not use the @NApiVersion tag. 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**

The long description for this error code is: Failed to validate script file: [1]

The system displays this error in a few cases. For example, you see this message if your script is formatted correctly for SuiteScript 2.0 in all ways except that it lacks the @NApiVersion tag.

**INVALID_API_VERSION**

The long description for this error code is: Invalid JSDoc tag value; valid values are: @NApiVersion [2.X, 2.x, 2.0]

This error is displayed when you attempt to upload a file with an invalid value for the @NApiVersion tag. It can also be displayed if you edit a script file that was already uploaded and try to modify the value of @NApiVersion tag.

If you see this error, check your file to make sure the @NApiVersion tag has a valid value. Valid values include the following:

```
#  2.0
#  2.x
#  2.X
```
INVALID_JSDOC_TAG_VALUE

The long description for this error code is: Invalid JSDoc tag value; valid values are: {1}

This error can be displayed if your script file uses an invalid value for any JSDoc tag. For example, if you use an invalid value for @NScriptType, the system displays this error.

To resolve the problem, check your file to make sure the value you used for the tag does not include typos or other errors.

MULTIPLE_DEFINE_CALLS

The long description for this error code is: Invalid define call, define should only be called one time per module. Define calls found at the following line numbers: {1}

This error is displayed if your script includes more than one define call. If you see this error, review the script and edit it appropriately.

MISSING_SCRIPT_TYPE

The long description for this error code is: @NScriptType is mandatory for 2.0 entry point script.

The system displays this error if you attempt to create a script record based on a SuiteScript 2.0 script that does not include the @NScriptType tag. To resolve the problem, edit the file and add the @NScriptType tag with the appropriate value.

The system does not display this error when you are uploading your file to the File Cabinet. It displays this error only when you try to create a script record, or if you edit a script file attached to a script record.

SCRIPT_OF_API_VERSION_20 CANNOT ...

The full code for this error is:
SCRIPT_OF_API_VERSION_20_CANNOT_IMPLEMENT_MORE_THAN_ONE_SCRIPT_TYPE_INTERFACES

The corresponding long description is: SuiteScript 2.0 entry point scripts cannot implement functions for more than one script type.

The system displays this error if your interface includes entry points from more than one script type. If you see this error, review your script’s interface. Check that all entry points belong to a single script type. For details on the available entry points for each script type, see SuiteScript 2.0 Script Types.

SCRIPT_OF_API_VERSION_20 MUST ...

The full code for this error is:
SCRIPT_OF_API_VERSION_20_MUST_IMPLEMENT_A_SCRIPT_TYPE_INTERFACE.

The corresponding long description is: SuiteScript 2.0 entry point scripts must implement one script type function.

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.

SS_V2_FILE_USED_FOR_FORM_SCRIPT

The full code for this error is:
SS_V2_FILE_USED_FOR_FORM_SCRIPTS_MUST_IMPLEMENT_CLIENT_SCRIPT_TYPE.

The corresponding long description is: SuiteScript version 2 file used for form scripts must implement client script type.

This error is displayed if you try to attach a script other than a client script to a custom form. Only a client script can be deployed at the form level. Other types of entry point scripts can be deployed only at the record level. For full details
about working with form scripts, see SuiteScript 2.0 Form-Level Scripts. For details about deploying other types of
entry point scripts, see SuiteScript 2.0 Record-Level Scripts.

SYNTAX_ERROR

The long description for this error code is: Syntax error: {1}

THE_FILE_IS_USED_AS_SCRIPT_TYPE_1 ...

The full error code for this error is: THE_FILE_IS_USED_AS_SCRIPT_TYPE_1_CANNOT_BE_CHANGED_TO_2.
The corresponding long description is: The file is used as script type {1}, cannot be changed to {2}
You may see this error when editing a script file that is attached to an existing script record. If you edit the file so that it
uses a different script type interface from what it previously used, and a different @NScriptType value, the system
generates this error.
This error occurs because the Type field on the script record cannot be changed. To avoid this problem, make your changes in a file that is not already associated with a script record. Then create a new script record based on that file.

WRONG_SCRIPT_TYPE

The long description for this error code is: Script file includes @NScriptType {1}; this script type cannot be used to
{2}.
The system displays this error if the file @NScriptType value is not compatible with the entry points in the script’s return statement. If you see this error, double-check the @NScriptType value used in your script. Make sure that it corresponds with the entry points used by your script. If the tag’s value does not match the script type interface, you cannot upload the file to the File Cabinet. For details on the available entry points for each script types, see SuiteScript 2.0 Script Types.

SuiteScript 2.0 Record-Level Scripts

After you have written a valid entry point script, as described in SuiteScript 2.0 Entry Point Script Validation, you must take the following steps if you want to deploy the script on records in your NetSuite account:

1. Use the script file to create a script record, 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.

Important: These steps are specific to entry point scripts deployed at the record level.

Script Record Creation

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. By contrast, the system uses the data in the file to automatically set several key fields. These fields are described in the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value Taken From</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Version</td>
<td>The @NApiVersion tag in your script file</td>
</tr>
<tr>
<td>Script File</td>
<td>The name of your script file.</td>
</tr>
</tbody>
</table>
At a high level, you can create a script record by using the following approach: Go to Customization > Scripting > Scripts > New. Select the appropriate file in the Script File dropdown list, then click **Create Script Record**.

In response, the system opens a new script record. The Type, API Version, and Script File fields are already populated. To complete the process, enter a name for the script record, and set any optional fields as needed. Then click **Save**.

After you save, the system shows the new script record in view mode. On the Scripts subtab, the system lists all possible entry point functions that could exist for this script type, with checks beside the ones used in your script.

The check boxes on the Scripts subtab cannot be edited directly. To check or clear the boxes, edit the script to add or remove the appropriate functions. When you save your changes, the script record’s check boxes are updated to match the contents of the script file.

For more detailed help on creating a script record, see [Creating a Script Record](#).

### Creating a Script Record

The process of creating a script record varies depending on the script’s type. The following procedure uses general steps that apply to all types. For certain script types, additional steps may be required.

**To create a script record:**

1. Go to Customization > Scripting > Scripts > New.
   The system displays the **Upload Script File** page.

2. In the **Script File** dropdown list, select the appropriate SuiteScript 2.0 file. The dropdown list populates automatically with SuiteScript files that you have uploaded to your File Cabinet. This list includes version 1.0 and version 2.0 files.
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 dropdown list to display a Plus icon.

Click the icon to display a dialog box for uploading a file from your local environment.

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

The system displays the Script page, with your .js file listed on the Scripts subtab. The read-only Type field is automatically populated based on the content of your file. The read-only API Version field is automatically populated with the version number: 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. In these cases, 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.

After the script record is saved, only the owner of the record or a system administrator can modify the record.

9. If appropriate, check the **Inactive** box. When a script is set to Inactive, any deployments associated with the script are also inactive.

10. On the **Parameters** subtab, define any parameters (custom fields) that are used by functions in the script.

11. On the **Unhandled Errors** subtab, optionally define the people to be notified if script errors occur.

Three types of error notifications are sent:

- An initial email is sent about the first occurrence of an error within an hour.
- An email with aggregated error information for every hour is sent. (The error counter is reset when this email is sent.)
- An email is sent about the first 100 errors that have occurred after the error counter is set to 0.

For example, an error is thrown 130 times within an hour. An initial email is sent. After the 100th occurrence, another email is sent. Since there are an additional 30 occurrences within the same hour, a final summary email is sent at the end of the hour. During the second hour, if there are only 50 occurrences of the error, only one summary email is sent at the end of that hour.

```plaintext
Note: By default, the Notify Script Owner box is checked. Alternatively or in addition to the script owner, you can identify other people, as follows:
```

- Check the Notify All Admins box, if all administrators should be notified.
- Select one or more groups from the **Groups** dropdown 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.

12. 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.

13. 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.

14. If the **Scripts** subtab includes the **Custom Plug-in Types** sublist, optionally add lines to this list.

15. 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 a page that lets you create a new script deployment record.

---

**Script Deployment**

Before an entry point script will run in your NetSuite account, it must be deployed. You can deploy a script at the time you create a script record, or you can deploy it later. The deployment settings available vary depending on the script’s type and on how you deploy the script.

When you deploy a script, the system 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.

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

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. In most cases, you can also use these methods to edit a script deployment:

# Work with the Deployments Sublist
# Use the Script Deployment Record
# Use N/record Module Methods

Work with the Deployments Sublist

You can deploy a script at the same time as you create a script record. This process is described in Deploying a Script by using the Deployments Sublist. This approach lets you define values for many deployment fields, although not all. You can use this technique at the following times:

# When you are creating a script record.
# For some script types, when you are editing an existing script record. However, for some script types, the Deployments sublist is read-only when you are editing the script record.

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 a script deployment:

1. When you save your Script record, you can immediately create a Script Deployment record by selecting Save and Deploy from the Script record Save button.
   
   If you want to update a deployment that already exists, go to Customization > Scripting > Script Deployments and select Edit next to the deployment.

2. On the Script Deployment page:
   # For Suitelet, Scheduled, and Portlet scripts, in the Title field, provide a name for the deployment.
   # For User Event and Client scripts, in the Applies To field, select the record the script will run against. In the Applies To field you can also select All Records to deploy the script to all records that officially support SuiteScript. (For a list of these records, see the help topic SuiteScript Supported Records.)

3. In the ID field, if desired, enter a custom scriptId for the deployment. If you do not create a scriptId, a system-generated internalId is created for you.
For information on whether to create a custom ID, see the help topic Creating a Custom Script Deployment ID.

4. (Optional) Clear the Deployed check box if you do not want to deploy the script. Otherwise, accept the default. A script will not run in NetSuite until the Deployed check box is selected.

5. If the Execute in Commerce Context box is displayed, optionally check the box. This option, available only for user event scripts, determines whether the script will be triggered by activity in the web store. When this box is checked, the script executes in response to triggering events from SuiteCommerce Advanced, SuiteCommerce Site Builder, or SuiteCommerce InStore. When the box is cleared, the script does not execute in response to activity in those areas. Clearing the box can result in performance improvements in the web store.

6. In the Status field, set the script deployment status. See the help topic Setting Script Deployment Status.

7. (Optional) In the Event Type dropdown list, specify an event type for the script execution. See the help topic Setting Script Execution Event Type from the UI.

8. (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.

9. 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.

10. If this deployment is for a Suitelet, you can check the Available Without Login box to generate an External URL on save of this deployment. See the help topic Setting Available Without Login.

Note: As of NetSuite 2019.1, External URLs use account-specific domains. The External URL (Deprecated) field will no longer show in 2019.2. Only URLs with the account-specific domain format will be supported in 2020.1. As a best practice, use url.resolveScript(options) to discover the Suitelet URL instead of hard-coding when possible.

11. On the Audience tab, specify the audiences for the script. See the help topic Defining Script Audience.

12. On the Links tab (for Suitelets only), if you want to launch your Suitelet from the UI, create a menu link for the Suitelet. See the help topic Running a Suitelet in NetSuite.

13. (Optional) On the Execution Log tab, create custom views for all script logging details. See the help topic Creating Script Execution Logs.

14. Click Save.

   Note that for portlet scripts, you must enable the portlet to display on your dashboard (see the help topic Displaying Portlet Scripts on the Dashboard).

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 can want to modify a script deployment record programmatically, you can load it by using record.load(options). You can modify the 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

If you want an entry point script to execute in NetSuite, you must deploy it.

This topic describes how to deploy a script by using the Deployments sublist. You can use this approach when you are in the process of creating a script record that you want to deploy. Depending on the script type, you can also use this approach when you are editing an existing script record.
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 dropdown 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.
   # If the Execute in Commerce Context column is displayed, either leave the field cleared or click in the column to display a check box. This option is available for user event scripts only. It determines whether the script will be triggered by activity in the web store. For more details, see Managing Web Store Performance Impact.
   # In the Status column, select the appropriate deployment status. Note that the available values vary depending on the script type.
   # If the Event Type dropdown list is displayed, optionally select a value from the dropdown 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 just been created. In some cases, this page lets you configure additional fields. For details, see Updating a Script Deployment.

Updating a Script Deployment

If appropriate, you can edit an existing script deployment and make changes to it.

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, or by using the script deployment record entry form. This topic describes the latter method.

When you open the full script deployment record for editing, you have access to a greater number of fields than are available on the Deployments sublist. For example, the script deployment record includes the following subtabs:
# Audience subtab – 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.

# Links subtab – For a Suitelet, lets you specify the menu paths that permit users to access the Suitelet.

# Schedule – for a map/reduce or scheduled script, lets you configure the schedule that determines when the script runs.

⚠️ 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

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 using the script deployment record’s Execute in Ecommerce Context option. This option lets you control whether the deployment is triggered by an event in SuiteCommerce Advanced, SuiteCommerce Site Builder, or SuiteCommerce InStore.

Scripts can significantly slow web store performance. By disassociating scripts from web store activity, you can improve web store response times.

You can set this option at the time you deploy a script. You can also go back and configure the option later, as described in the following procedure.

To enable or disable web store triggers for a user event script deployment:

1. Open the deployment for editing. For example, navigate to Customization > Scripting > Script Deployments. Locate the appropriate deployment and click the corresponding edit link.
2. On the script deployment record, locate the Execute in Ecommerce Context. Check or clear the box as appropriate.
3. Save the record.

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

If appropriate, 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 the corresponding View link.
3. Click the System Notes subtab.

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 2.0 Form-Level Scripts

In some cases, you might want a client script to deploy on one form only. To configure this behavior, 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.

With both custom entry forms and custom transaction forms, you can also use logic in the script to create a button or a menu item. These controls are sometimes referred to as custom actions. For custom address forms, you can deploy the script on the form, but you cannot configure custom actions.

The alternative to deploying your script on a form is to deploy it on a record type. In the latter case, the script is executed globally on that record type, regardless of which form is being used. By contrast, when you attach a client script to a form, it runs on that form only. For more details about the differences between these two approaches, see Record-Level and Form-Level Scripts.

For details on deploying a client script at the form level, see the following sections:

# Attaching a Client Script to a Form
# Configuring a Custom Action

Important: Only client scripts can be attached to forms. Other types of entry point scripts can be deployed only at the record level. A script deployed at the record level is used on all forms associated with that record type. For help with record deployment, see SuiteScript 2.0 Record-Level Scripts.

Attaching a Client Script to a Form

You can attach a client script to a form after you have made sure that it is a valid script, as described in SuiteScript 2.0 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 of the 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 of the 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. Location the appropriate form and click the corresponding Edit link.
2. Navigate to the **Custom Code** subtab.

3. In the **Script File** dropdown list, select the SuiteScript 2.0 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 dropdown 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 populate the Script File field, the system updates the page to populate the **SuiteScript API Version** field. 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.

The check 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 check boxes are updated to match the contents of the script file.

4. Click **Save**.

### Configuring a Custom Action

In some cases, you might want to configure a custom action that uses logic contained in your client script. A custom action can be either of the following:

- A custom button that appears at the top of the page.

- A custom menu item that appears as an option when the user points to the Actions label.
You configure these elements by using logic contained in the client script attached to your form.

**Note:** It is not possible to configure a custom action for a custom address form. You can configure custom actions for custom entry and custom transaction forms.

To configure a custom action on a custom form:

1. If the 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. Navigate to the **Actions** subtab.
3. Navigate 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**. For help understanding the validation terms used in this topic, see **Entry Point Script Validation Guidelines**.
SuiteScript 2.0 Custom Modules

# Custom Modules Overview

With SuiteScript 2.0, 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 2.0 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/](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 2.0 Anatomy of a Custom Module Script

All SuiteScript 2.0 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.

<table>
<thead>
<tr>
<th>General Area</th>
<th>Callout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 — JSDoc tags and other annotations</td>
<td>2</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>3 and 4</td>
<td>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.</td>
</tr>
</tbody>
</table>
SuiteScript 2.0 Anatomy of a Custom Module Script

<table>
<thead>
<tr>
<th>General Area</th>
<th>Callout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 and 6</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>1 — define statement</td>
<td>7</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>The define function’s second argument, which is a callback function.</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>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 2.0 Custom Module Tutorial.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>The callback function’s return statement.</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>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 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.</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>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.</td>
</tr>
</tbody>
</table>

Note: For details on the standard structure for an entry point script, see SuiteScript 2.0 Anatomy of a Script.

SuiteScript 2.0 Custom Module Tutorial

A custom module script holds logic that can be used by other scripts. With this feature, if you have a piece of logic that is required by 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.

See the following sections:

# 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 topic can be used to automatically create 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 topic, 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.

Step One: Deploy the Prerequisite Script

If you have not deployed the user event script described in SuiteScript 2.0 User Event Script Tutorial, deploy it now.

Step Two: 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.

```javascript
/**
 * phoneCall.js
 * @NApiVersion 2.x
 * @NModuleScope Public
 */

// This script must create a record, so it loads the
// N/record module.

define ( [ 'N/record' ] ,

// The next line marks the beginning of the callback
// function. The 'record' argument is an object that
// represents the record module.

function (record) {

// The next line marks the beginning of the entry point
// function.

function scheduleCall (context) {

  var newPhoneCall = record.create({
    type: record.Type.PHONE_CALL,
    isDynamic: true
  });

  newPhoneCall.setValue({
    fieldId: 'title',
    value: context.phoneCallTitle
  });

  newPhoneCall.setValue({
```
SuiteScript 2.0 Custom Module Tutorial

```javascript
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
  );

}

// Add the return statement that identifies the entry point function.

return {
  schedule: scheduleCall,
}
```

Step Three: 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 Step Two: 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.
SuiteScript 2.0 Custom Module Tutorial

<table>
<thead>
<tr>
<th>Callout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>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.</td>
</tr>
<tr>
<td>2 and 3</td>
<td>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.</td>
</tr>
<tr>
<td>4 and 5</td>
<td>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.</td>
</tr>
</tbody>
</table>

## 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.

```javascript
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
        });
    }
}
```
SuiteScript 2.0 Custom Module Tutorial

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.

3. | 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.
   
   The values of the context object — `phoneCallTitle`, `phoneCallOwner`, and `phoneCallNumber` — must be defined in the script that calls the custom module.

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.

```
return {
  schedule : scheduleCall,
}
```

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. | 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.
   
   Whatever object is referenced, it must be defined within the same script file.

Step Four: 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. 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*.

Step Five: 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 2.0 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.

```javascript
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.

```javascript
function(record, serverWidget, phone ) {
```

4. In the script’s afterSubmit function, add logic to retrieve the employee’s phone number.

```javascript
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.

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

define ( [ 'N/record', 'N/ui/serverWidget', './phoneCall' ] ,
        function (record, serverWidget, phone ) {

    // In the beforeLoad function, disable the
    // Notes field on the record.

    function myBeforeLoad (context) {

        if (context.type !== context.UserEventType.CREATE) return;

        var form = context.form;

        var notesField = form.getField({
            id: 'comments'
        });

        var phoneData = {
            phoneNumber : newEmployeePhoneNumber,
            phoneCallOwner : newEmployeeSupervisor,
            phoneCallTitle : 'Welcome ' + newEmployeeFirstName + '

       );

        phone.schedule(phoneData);
    }
```
notesField.updateDisplayType({
    displayType: serverWidget.FieldDisplayType.DISABLED
});
}

// In the beforeSubmit function, add a message to the
// Notes field.

function myBeforeSubmit(context) {
    if (context.type !== context.UserEventType.CREATE)
        return;

    var newEmployeeRecord = context.newRecord;
    newEmployeeRecord.setValue('comments', 'Orientation date TBD.');
}

// In the afterSubmit function, take several actions culminating
// in creating both a task record and a phone call record.

function myAfterSubmit(context) {
    // If the user is not creating a new record, then stop
    // executing.
    if (context.type !== context.UserEventType.CREATE)
        return;

    // Use the context object's newRecord property to retrieve values
    // from the new record.
    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'
});

// If the user entered a value for the supervisor field,
// create a task record for the supervisor.

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
        });
    }
}

// If the user entered values for both the supervisor and
// phone number fields, use the phoneCall module to
// schedule a phone call.

if (newEmployeeSupervisor && newEmployeePhoneNumber) {
    var phoneData =

{  
    phoneNumber: newEmployeePhoneNumber,  
    phoneCallOwner: newEmployeeSupervisor,  
    phoneCallTitle: 'Welcome' + newEmployeeFirstName +  
    ' ' + newEmployeeLastName  
};  

phone.schedule(phoneData);  

return {  
    beforeLoad: myBeforeLoad,  
    beforeSubmit: myBeforeSubmit,  
    afterSubmit: myAfterSubmit  
};  

};

Step Six: 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. 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.

Step Seven: 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 succesfully:
   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

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:

<table>
<thead>
<tr>
<th>Path Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute Paths</td>
<td>Specifies the path from the root folder in the file cabinet. Absolute paths start with a forward slash (/). For example, &quot;/SuiteScripts/MyApp/Util&quot;.</td>
</tr>
</tbody>
</table>
| Relative Paths      | 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. Relative Paths start with a period (.).

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".

Note: Do not use relative paths in global contexts. For example, when using the require function or SuiteScript Debugger. |
| Bundle Virtual Paths| 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. Bundle virtual paths start with a forward slash followed by a period (/.). For example:

```
# /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.

```
// myModule.js
define(['/SuiteScripts/lib1', '/SuiteScripts/lib2'], // myModule has a dependency on modules lib1 and lib2
    ...
);
```

**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.

```
// myModule.js
define(['/lib/lib1', '/lib/lib2'], // myModule has a dependency on modules lib1 and lib2
    ...
);
```

**Bundle Virtual Paths**

Use a bundle virtual path to point to a bundle file cabinet location. Bundle virtual paths start with a forward slash (/). 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.

```
// myModule.js
require(['/bundle/101'], // myModule has a dependency on modules with the bundle id 100 and 101
    ...
);
```

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 2.0 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**

**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.

The following steps demonstrate the steps to load a custom module by name:

1. Create your module file. For example `math.js`.
2. Author your custom module name and its contents:
3. 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.

   ```json
   ...
   {
      "paths": {
         "math": "/SuiteScripts/Example/math"
      }
   }
   ...
   ``

4. Load the custom module in your SuiteScript 2.0 script by including the `NAmdConfig` tag and passing the name to the `define` Object.

   ```javascript
   /**
    * @NApiVersion 2.x
    * @NScriptType uereventscript
    * @NAmdConfig ./myconfig.json
    */
    define(['math'],
    function (math) {
      return {
        beforeLoad: function beforeLoad() {
          log.debug('test', math.add(1,2));
        }
      }
    });
   ...  
   ```

Custom Module Examples

The following examples demonstrate using the define and require functions when working with custom modules.

# Example: Define a custom utility module
# Example: Import the custom utility module
# Example: Define a custom module for a SuiteApp
# Example: Import a third-party JavaScript library

Example: 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.

```javascript
/**
 * counter.js
 * @NApiVersion 2.x
 */

define(function(){
    var counter = 0;

    function incrementValue() {
        counter++;
    }
    function getValue() {
        return counter;
    }

    return {
        increment: incrementValue,
        value: getValue
    }
});
```

**Example: 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.

```javascript
/**
 * customRecord.js
 * @NApiVersion 2.x
 */

define(['N/record', 'counter'],
    function(record, counter){
        function createCustomRec() {
            record.create(...);
            counter.increment();
        }

        return {
            createCustomRecord: createCustomRec
        }
});
```

**Example: 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.

```javascript
/**
 * @NApiVersion 2.0
 * @NModuleScope public
 */
```
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:

```javascript
define(['/bundle/<bundle ID>/<module path>'],
    function (<module name>){
        <logic goes here>
    }
);
```

Example: Import a third-party JavaScript library

```javascript
# Import a third-party library
# Example: 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, create a JSON configuration file (`JsLibraryConfig.json`) and save it in the file cabinet:

```json
...
{
    "paths": {
        "coolthing": "/SuiteScripts/myFavoriteJsLibrary"
    }
}
...
```

Then, you could use the `NAmdConfig` JSdoc tag to provide a relative path from the script that needs the `coolthing` module:

```javascript
/**
 * @NApiVersion 2.0
 * @NScriptType usereventscript
 * @NAmdConfig ./JsLibraryConfig.json
 */

define(['coolthing'],
    function (coolthing)
    {
        return {
            beforeLoad: function beforeLoad(ctx)
            {
                coolthing.times(2, function () {
                    log.debug({
                        title: 'log',
                        details: 'log'
                    });
                });
            }
        }
    });
```
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:

```javascript
...
{
  "paths": {
    "jquery": "/SuiteScripts/myjQueryLib"
  }
}
...
```

From the script using the jQuery module, reference the path to your JSON configuration file in the NAmdConfig tag. For example, `* @NAmdConfig /SuiteScripts/myjQueryLib.json`.

```javascript
...
* @NAmdConfig  ./jQueryConfig.json
*/
```

```
define(["jquery"], function(jquery) {
});
```...

**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.

**Example: Add a non-AMD library**

In this example, the following file would be uploaded to the file cabinet (as a JavaScript file that does not register as AMD compatible).

```javascript
var myMath = {
  add: function(num1, num2) {
    return num1 + num2;
  },
  subtract: function(num1, num2) {
    return num1 - num2;
  },
  multiply: function(num1, num2) {
    return num1 * num2;
  },
  divide: function(num1, num2) {
    return num1 / num2;
  },
};
```
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:

```json
...
{
  "paths": {
    "math": "SuiteScripts/Shim Example/math"
  },
  "math": {
    "exports": "myMath"
  }
}
...
```

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 tag with a path to the JSON configuration file is included.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType usereventscript
 * @NAmConfig ./MathConfig.json
 */

define(['math'],
  function (math) {
    return {
      beforeLoad: function beforeLoad() {
        log.debug({
          title: 'test',
          details: myMath.add(1,2)
        });
      }
    }
  });
```

### Troubleshooting Errors

#### 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](#).
Frequently Asked Questions: Custom Modules

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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Example</th>
<th>Caveat</th>
</tr>
</thead>
</table>
| 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: |

```
define(['require'], function (require) {  
    require([`/ParentFolder/customFile`], function (customFile) {  
        //your code  
    });  
});
```

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 [Example: Import a third-party JavaScript library](#).
SuiteScript 2.0 Scripting Records and Subrecords

For information on using SuiteScript 2.0 to work with records and subrecords, see the following topics:

- SuiteScript 2.0 Scripting Records
- SuiteScript 2.0 Scripting Subrecords

SuiteScript 2.0 Scripting Records

For details on using SuiteScript 2.0 to work with records, see the following sections:

- SuiteScript 2.0 – Standard and Dynamic Modes
- SuiteScript 2.0 – Record Modules

SuiteScript 2.0 – 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 2.0 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 (submitted) with `Record.save(options)`. Standard mode is also called deferred dynamic mode and you see both terms in the SuiteScript help.

**Dynamic mode:**

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 2.0 Record Modes.

When a SuiteScript 2.0 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.

**How you can tell if a record is in dynamic mode:**

There are two SuiteScript 2.0 properties that indicate if a record is in dynamic mode:

- `Record.isDynamic`
- `CurrentRecord.isDynamic`

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 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 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 2.0 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 pass in a specific argument. See the help topic for the applicable method for more information.

### SuiteScript 2.0 – Record Modules

With SuiteScript 2.0, you use the `N/record Module` and `N/currentRecord Module` to script with records.

In server-side scripts, use the `N/record Module`. See the following topics for examples of working with records in server-side scripts:

- SuiteScript 2.0 User Event Script Sample
- SuiteScript 2.0 User Event Script Tutorial
- SuiteScript 2.0 Custom Module Tutorial

In client scripts:

- Use `N/currentRecord Module` methods to interact with the record that is active in the current client-side 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 2.0 Client Scripts on currentRecord
- SuiteScript 2.0 Remote Objects in Client Scripts
- Disable Field on Client PageInit using SuiteScript 2.0

### SuiteScript 2.0 Scripting Subrecords

For details on using SuiteScript 2.0 to work with subrecords, see the following sections:

- Understanding Subrecords
- Subrecord Scripting in SuiteScript 2.0 Compared With 1.0
- Scripting Subrecords that Occur on Sublist Lines
- Scripting Subrecords that Occur in Body Fields
Understanding Subrecords

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.

What Is a Subrecord?

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:

```...
```
// Load the purchase order.

var rec = record.load(
    {type : record.Type.PURCHASE_ORDER,
     id : 7,
     isDynamic : false});

// Retrieve the subrecord. For sublistId, use the ID of the relevant sublist on
// the purchase order record type. For fieldId, use the ID of the summary field
// on the sublist that holds the subrecord.

var subrec = rec.getSublistSubrecord(
    {sublistId : 'item',
     line : 0,
     fieldId : 'inventorydetail'});

...

For a full script sample, see Example: Creating an Inventory Detail Subrecord.

Subrecord Scripting Overview

Use the following guidelines when scripting with subrecords:

# Understand 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

Understand When Subrecords Are Read-Only

You can work with subrecords in both client and server-side 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-side 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.

**Example: Creating an Inventory Detail Subrecord**

The following example shows a typical approach to creating a subrecord.

This script creates a purchase order with one inventory item in its sublist. The sample also creates an inventory detail subrecord to store a lot number for the item.

Before using this sample 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 sample 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 2.0 to Create a Subrecord in a Sublist Field.
Note: For help deploying a user event script, see SuiteScript 2.0 Entry Point Script Creation and Deployment.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */

define([ 'N/record' ],

    function(record) {

        function myAfterSubmit(context) {

            // Create the purchase order.

            var rec = record.create(
                {
                    type: record.Type.PURCHASE_ORDER,
                    isDynamic: false
                }
            );

            // Set body fields on the purchase order. Replace both of these
            // hardcoded values with valid values from your NetSuite account.

            rec.setValue(
                {
                    fieldId: 'entity',
                    value: '2'
                }
            );

            rec.setValue(
                {
                    fieldId: 'location',
                    value: '2'
                }
            );

            // Insert a line in the item sublist.

            rec.insertLine(
                {
                    sublistId: 'item',
                    line: 0
                }
            );

            // Set the required fields on the line. Replace the hardcoded value
            // for the item field with a valid value from your NetSuite account.

            rec.setSublistValue(
                {
                    sublistId: 'item',
                    fieldId: 'item',
                    line: 0,
                    value: '7'
                }
            );
        }
    }
)
```
rec.setSublistValue({
    sublistId: 'item',
    fieldId: 'quantity',
    line: 0,
    value: 1
});

// Instantiate the subrecord. To use this method, you must
// provide the ID of the sublist, the number of the line you want
// to interact with, and the ID of the summary field.

var subrec = rec.getSublistSubrecord({
    sublistId: 'item',
    line: 0,
    fieldId: 'inventorydetail'
});

// Insert a line in the subrecord's inventory assignment sublist.
subrec.insertLine({
    sublistId: 'inventoryassignment',
    line: 0
});

subrec.setSublistValue({
    sublistId: 'inventoryassignment',
    fieldId: 'quantity',
    line: 0,
    value: 1
});

// Set the lot number for the item. Although this value is
// hardcoded, you do not have to change it, because it doesn't
// reference a record in your account. For the purpose of this
// sample, the value can be any string.

subrec.setSublistValue({
    sublistId: 'inventoryassignment',
    fieldId: 'receiptinventorynumber',
    line: 0,
    value: '01234'
});

// Save the record. Note that the subrecord object does
// not have to be explicitly saved.
try {
    var recId = rec.save();

    log.debug({

title: 'Record created successfully',
  details: 'Id: '  + recId
});

} catch (e) {
  log.error(
    title: e.name,
    details: e.message
  );
}
}

return {
  afterSubmit: myAfterSubmit
};

For additional samples, see Scripting Subrecords that Occur on Sublist Lines and Scripting Subrecords that Occur in Body Fields.

SupportedDeployments for Subrecord Scripting

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 Understanding Subrecords.

The following types of deployments are supported:

# Server–side Scripts Deployed on Parent Records
# Client Scripts Deployed on Parent Records (with Limitations)
# Client Scripts Deployed on Custom Address Forms

Server–side Scripts Deployed on Parent Records

If you want to create a server-side 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-side 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**

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:

# Example: Subrecord that Occurs in a Sublist Field

# Example: Subrecord that Occurs in a Body Field

**Example: Subrecord that Occurs in a Sublist Field**

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 2.0 to work with subrecords that exist on sublist lines, see Scripting Subrecords that Occur on Sublist Lines.

Example: Subrecord that Occurs in a Body Field

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 2.0 to work with subrecords that exist in body fields, see [Scripting Subrecords that Occur in Body Fields](#).

**Structure of a Subrecord**

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:

- **Example: Writable Body Fields**
- **Example: Writable Sublist**
- **Example: Writable Body Fields and Sublist**

**Example: Writable Body Fields**

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 Example: Creating an Address Sublist Subrecord and Example: Creating an Address on a Subsidiary Record.

**Example: Writable Sublist**

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 Example: Creating a Landed Cost Sublist Subrecord.

Example: Writable Body Fields and Sublist

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 Example: Creating an Order Schedule Sublist Subrecord.

Finding Subrecord Details in the Records Browser

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.
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 **Example: Creating an Inventory Detail Subrecord on a Body Field**.

- **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 **Example: Creating a Landed Cost Sublist Subrecord**.

- **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 sample, see **Example: Using SuiteScript 2.0 to Create a New Shipping Address**.

### 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.
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.

Understanding the Address Subrecord

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.

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.

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:

1. Use one of the following methods to retrieve the subrecord: getCurrentSublistSubrecord(), getSublistSubrecord(), or getSubrecord(). For an example, see Example: Retrieving an Address Subrecord.
2. 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 side in addition to the client side. Be aware that this is expected behavior. For logic in a client script attached to an address subrecord to execute only once, wrap the logic in an if statement that immediately exits the script on the server side. For example:

```javascript
if (typeof document !== 'undefined') {
  // client script logic
}
```

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:
SuiteScript 2.0 Scripting Subrecords

```javascript
define([],
    function() {
        function saveRecord(context) {
            var currentRecord = context.currentRecord;
            return true;
        }
        return {
            saveRecord: saveRecord
        }
    }
);
```

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 2.0 Client Script Type.

Subrecord Scripting in SuiteScript 2.0 Compared With 1.0

Compared with SuiteScript 1.0, SuiteScript 2.0 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. You could use `nlapiEditSubrecord` to edit a subrecord.

By contrast, in SuiteScript 2.0, 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 2.0, 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 2.0, 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 2.0, you must instantiate the address subrecord by referencing the appropriate summary field.

Scripting Subrecords that Occur on Sublist Lines

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 2.0 to Create a Subrecord in a Sublist Field
- Using SuiteScript 2.0 to Edit a Subrecord that Occurs in a Sublist Field
- Using SuiteScript 2.0 to Retrieve a Sublist Subrecord

Using SuiteScript 2.0 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 2.0 – 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:

```javascript
var orderScheduleSubrecord = blanketPurchaseOrder getCurrentSublistSubrecord(
  sublistId : 'item',
  fieldId : 'orderschedule'
);
```

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.

```javascript
orderScheduleSubrecord setValue(
  fieldId : 'createpurchaseorder',
  value : LEAD
);
```

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:

```javascript
orderScheduleSubrecord.selectNewLine(
  sublistId : 'schedule',
);
orderScheduleSubrecord.setCurrentSublistValue(
  sublistId : 'schedule',
  fieldId : 'quantity',
);```
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 sample, see Example: Creating an Inventory Detail Sublist Subrecord and Example: Creating an Address Sublist Subrecord.

**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 2.0 – 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:

```javascript
inventoryDetailSubrecord = rec.getSublistSubrecord({
    sublistId: 'item',
    fieldId: 'inventorydetail',
    line: 0
});
```

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:

```javascript
... subrecordInvDetail.setSublistValue({
  sublistId: 'inventoryassignment',
  fieldId: 'receiptinventorynumber',
  line: 0,
  value: '012345'
});
...```

6. Save the record with the `Record.save(options)` method.

Note: For a full script sample of creating a sublist subrecord in standard mode, see Example: Creating a Landed Cost Sublist Subrecord.

Example: Creating an Inventory Detail Sublist Subrecord

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 sample, 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 2.0 to Create a Subrecord in a Sublist Field.

To learn about SuiteScript scripting modes, see SuiteScript 2.0 – Standard and Dynamic Modes

```javascript
/** *
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */

define([ 'N/record' ],

function(record) {

  function afterSubmit(context) {

    // Create the purchase order.

    var rec = record.create({
      type: record.Type.PURCHASE_ORDER,
      isDynamic: true
    });

    // Set body fields on the purchase order.

    ...```

```
rec.setValue({
    fieldId: 'entity',
    value: '1663'
});

rec.setValue({
    fieldId: 'location',
    value: '6'
});

// Create one line in the item sublist.

rec.selectNewLine({
    sublistId: 'item'
});

rec.setCurrentSublistValue({
    sublistId: 'item',
    fieldId: 'item',
    value: '299'
});

rec.setCurrentSublistValue({
    sublistId: 'item',
    fieldId: 'quantity',
    value: 1
});

// Create the subrecord for that line.

var subrec = rec.getCurrentSublistSubrecord({
    sublistId: 'item',
    fieldId: 'inventorydetail'
});

// Add a line to the subrecord's inventory assignment sublist.

subrec.selectNewLine({
    sublistId: 'inventoryassignment'
});

subrec.setCurrentSublistValue({
    sublistId: 'inventoryassignment',
    fieldId: 'quantity',
    value: 2
});

subrec.setCurrentSublistValue({
    sublistId: 'inventoryassignment',
    fieldId: 'receiptinventorynumber',
    value: '01234'
});
// Save the line in the subrecord's sublist.
subrec.commitLine({
    sublistId: 'inventoryassignment'
});

// Save the line in the record's sublist
rec.commitLine({
    sublistId: 'item'
});

// Save the record.
try {
    var recId = rec.save();
    log.debug({
        title: 'Record created successfully',
        details: 'Id: ' + recId
    });
} catch (e) {
    log.error({
        title: e.name,
        details: e.message
    });
}

return {
    afterSubmit: afterSubmit
};

Example: Creating an Address Sublist Subrecord

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 2.0 to Create a Subrecord in a Sublist Field.

To learn about SuiteScript scripting modes, see SuiteScript 2.0 – Standard and Dynamic Modes

/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */
SuiteScript 2.0 Scripting Subrecords

define([ 'N/record' ],

    function (record) {

        function afterSubmit(context) {

            // Create the record.
            var rec = record.create(
                type: record.Type.EMPLOYEE,
                isDynamic: true
            );

            // Set the required body fields.
            rec.setValue(
                fieldId: 'firstname',
                value: 'John'
            );

            rec.setValue(
                fieldId: 'lastname',
                value: 'Smith'
            );

            rec.setValue(
                fieldId: 'subsidiary',
                value: '1'
            );

            // Create a line in the Address sublist.
            rec.selectNewLine(
                sublistId: 'addressbook'
            );

            // Set an optional field on the sublist line.
            rec.setCurrentSublistValue(
                sublistId: 'addressbook',
                fieldId: 'label',
                value: 'Primary Address'
            );

            // Create an address subrecord for the line.
            var subrec = rec.getCurrentSublistSubrecord(
                sublistId: 'addressbook',
                fieldId: 'addressbookaddress'
// Set body fields on the subrecord. Because the script uses
dynamic mode, you should set the country value first. The country
value determines which address form is to be used, so by setting
this value first, you ensure that the values for the rest
of the form’s fields will be set properly.

subrec.setValue({
    fieldId: 'country',
    value: 'US'
});

subrec.setValue({
    fieldId: 'city',
    value: 'San Mateo'
});

subrec.setValue({
    fieldId: 'state',
    value: 'CA'
});

subrec.setValue({
    fieldId: 'zip',
    value: '94403'
});

subrec.setValue({
    fieldId: 'addr1',
    value: '2955 Campus Drive'
});

subrec.setValue({
    fieldId: 'addr2',
    value: 'Suite 100'
});

// Save the sublist line.

rec.commitLine({
    sublistId: 'addressbook'
});

// Save the record.

try {
    var recId = rec.save();

    log.debug({
        title: 'Record created successfully',
        details: 'Id: ' + recId
    });
}
Example: Creating an Order Schedule Sublist Subrecord

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 2.0 to Create a Subrecord in a Sublist Field.

To learn about SuiteScript scripting modes, see SuiteScript 2.0 – Standard and Dynamic Modes

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */
define([ 'N/record' ],
    function(record) {

        function afterSubmit(context) {

            // Create the record.
            var rec = record.create(
                {
                    type: record.Type.BLANKET_PURCHASE_ORDER,
                    isDynamic: true
                });

            // Set body fields on the record.
            rec.setValue(
                {
                    fieldId: 'entity',
                    value: '1663'
                });
            rec.setValue(
```
SuiteScript 2.0 Scripting Subrecords

```javascript
rec.setValue({
    fieldId: 'location',
    value: '6'
});

rec.setValue({
    fieldId: 'memo',
    value: '456789'
});

// Create one line in the item sublist.

rec.selectNewLine({
    sublistId: 'item',
    line: 0
});

rec.setCurrentSublistValue({
    sublistId: 'item',
    fieldId: 'item',
    value: '500'
});

rec.setCurrentSublistValue({
    sublistId: 'item',
    fieldId: 'quantity',
    value: '1'
});

// Create the subrecord for that line.

var subrec = rec.getCurrentSublistSubrecord({
    sublistId: 'item',
    fieldId: 'orderschedule'
});

// Set a field on the body of the subrecord.

subrec.setValue({
    fieldId: 'createpurchaseorder',
    value: 'LEAD'
});

// Create a line in the subrecord's sublist.

subrec.selectNewLine({
    sublistId: 'schedule',
    line: 0
});

subrec.setCurrentSublistValue({
    sublistId: 'schedule',
    fieldId: 'quantity',
    value: '1'
});
```
value: 1
});

var nextQuarter = new Date();
nextQuarter.setDate(nextQuarter.getDate() + 90);

subrec.setCurrentSublistValue({
  sublistId: 'schedule',
  fieldId: 'trandate',
  value: nextQuarter
});

// Save the line in the subrecord's sublist.
subrec.commitLine({
  sublistId: 'schedule'
});

// Save the line in the record's sublist
rec.commitLine({
  sublistId: 'item'
});

// Save the record.
try {
  var recId = rec.save();

  log.debug({
    title: 'Record created successfully',
    details: 'Id: ' + recId
  });
} catch (e) {

  log.error({
    title: e.name,
    details: e.message
  });
}

return {
  afterSubmit: afterSubmit
};
Example: Creating a Landed Cost Sublist Subrecord

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 sample, 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 2.0 to Create a Subrecord in a Sublist Field.

To learn about SuiteScript scripting modes, see SuiteScript 2.0 – Standard and Dynamic Modes

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */

define([ 'N/record' ],

    function(record) {

        function afterSubmit(context) {

            // Create the record.
            var rec = record.create({
                type: record.Type.VENDOR_BILL,
                isDynamic: false
            });

            // Set body fields on the record.
            rec.setValue({
                fieldId: 'entity',
                value: '1663'
            });

            rec.setValue({
                fieldId: 'location',
                value: '6'
            });

            rec.setValue({
                fieldId: 'tranid',
                value: '101A'
            });

            // Set the Landed Cost per Line field to true.
            rec.setValue({
                fieldId: 'landedcostperline',
                value: true
            });

        }
    }

    function init() {

    }

    function validate() {

    }

    function beforeSubmit(context) {

    }

});
```
// Add an item to the Item sublist.

rec.insertLine({
    sublistId: 'item',
    line: 0
});

// Set values on the sublist line.

rec.setSublistValue({
    sublistId: 'item',
    fieldId: 'item',
    line: 0,
    value: '599'
});

rec.setSublistValue({
    sublistId: 'item',
    fieldId: 'quantity',
    line: 0,
    value: 1
});

rec.setSublistValue({
    sublistId: 'item',
    fieldId: 'location',
    line: 0,
    value: '6'
});

// Create the subrecord.

var subrec = rec.getSublistSubrecord({
    sublistId: 'item',
    fieldId: 'landedcost',
    line: 0
});

// Add a line to the subrecord's Landed Cost Data sublist.

subrec.insertLine({
    sublistId: 'landedcostdata',
    line: 0
});

// Set values on the subrecord's sublist line.


Using SuiteScript 2.0 to Edit a Subrecord that Occurs in a Sublist Field

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 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:
   ```javascript
   # 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:
   ```javascript
   var addressSubrecord = rec.getCurrentSublistSubrecord({
       sublistId: 'addressbook',
       fieldId: 'addressbookaddress'
   });
   ```

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:
   ```javascript
   addressSubrecord.setValue({
       fieldId: 'city',
       value: 'St. Petersburg'
   });
   ```
   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 sample that shows editing a sublist subrecord in dynamic mode, see Example: Updating an Order Schedule Sublist Subrecord.

### 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 2.0 – 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:

```javascript
... 
inventoryDetailSubrecord = rec.getSublistSubrecord({ 
    sublistId: 'item', 
    fieldId: 'inventorydetail', 
    line: 0 
}); 
... 
```

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:

```javascript
... 
inventoryDetailSubrecord.setSublistValue({ 
    sublistId: 'inventoryassignment', 
    fieldId: 'receiptinventorynumber', 
    line: 0, 
    value: '56789' 
}); 
... 
```

5. Save the record with the `save()` method.

**Note:** For a full script sample showing how to modify a subrecord in standard mode, see [Example: Updating an Address Subrecord](#).

**Example: Updating an Order Schedule Sublist Subrecord**

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 sample, 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 2.0 to Edit a Subrecord that Occurs in a Sublist Field. To learn about SuiteScript scripting modes, see SuiteScript 2.0 – Standard and Dynamic Modes

```javascript
define([ 'N/record' ],

  function (record) {

    function afterSubmit(context) {

      // Load the blanket purchase order record.
      var rec = record.load(
        { 
          type: record.Type.BLANKET_PURCHASE_ORDER,
          id: 3319,
          isDynamic: true
        });

      // Select the sublist and line.
      rec.selectLine(
        { 
          sublistId: 'item',
          line: 0
        });

      // Retrieve the subrecord.
      var subrec = rec.getCurrentSublistSubrecord(
        { 
          sublistId: 'item',
          fieldId: 'orderschedule'
        });

      // Select the appropriate line in the subrecord's sublist.
      subrec.selectLine(
        { 
          sublistId: 'schedule',
          line: 0
        });

      // Identify the field to be modified, and set new value.
      var nextQuarter = new Date();
      nextQuarter.setDate(nextQuarter.getDate() + 90);
      subrec.setCurrentSublistValue(
        { 
          sublistId: 'schedule',
          fieldId: 'trandate',
          value: nextQuarter
        });
    }

    afterSubmit()
  }

);
Example: Updating an Address Subrecord

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 2.0 to Edit a Subrecord that Occurs in a Sublist Field.

To learn about SuiteScript scripting modes, see SuiteScript 2.0 – Standard and Dynamic Modes
function(record) {

    function afterSubmit(context) {

        // Load the record.
        var rec = record.load({
            type: record.Type.EMPLOYEE,
            id: 1863,
            isDynamic: false
        });

        // Retrieve the subrecord to be modified.
        var subrec = rec.getSublistSubrecord({
            sublistId: 'addressbook',
            fieldId: 'addressbookaddress',
            line: 0
        });

        // Change a field on the subrecord.
        subrec.setValue({
            fieldId: 'addr1',
            value: '15 Main Street'
        });

        // Save the record.
        try {
            var recId = rec.save();

            log.debug({
                title: 'Record updated successfully',
                details: 'Id: ' + recId
            });
        } catch (e) {

            log.error({
                title: e.name,
                details: e.message
            });
        }
    }

    return {
        afterSubmit: afterSubmit
    }
}
Using SuiteScript 2.0 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 2.0 – 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:

   ```javascript
   ...
   var addressSubrecord = rec.getCurrentSublistSubrecord({
       sublistId: 'addressbook',
       fieldId: 'addressbookaddress'
   });
   ...
   ```

4. If you want to retrieve a value from the body of the subrecord, use the `Record.getValue(options)` method. For example, you could use an expression like the following to retrieve a detail from the body of an address subrecord:

   ```javascript
   ...
   var cityValue = addressSubrecord.getValue({
       fieldId: 'city'
   });
   ...
   ```

5. If you want to retrieve a value from the subrecord’s sublist, use the `Record.getSublistValue(options)` method.
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 2.0 – 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:

   ```javascript
   var orderScheduleSubrecord = rec.getSublistSubrecord({
     sublistId: 'item',
     fieldId: 'orderschedule',
     line: 0
   });
   ...
   ```

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:

   ```javascript
   var dateValue = orderScheduleSubrecord.getSublistValue({
     sublistId: 'schedule',
     fieldId: 'trandate',
     line: 1
   });
   ...
   ```
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 2.0 to Retrieve a Sublist Subrecord.

To learn about SuiteScript scripting modes, see SuiteScript 2.0 – Standard and Dynamic Modes.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType userEventScript
 */

define([ 'N/record' ],

    function(record) {

        function afterSubmit(context) {

            // Load the record.

            var rec = record.load({
                type: record.Type.BLANKET_PURCHASE_ORDER,
                id: 3120,
                isDynamic: true
            });

            // Retrieve the subrecord.

            rec.selectLine({
                sublistId: 'item',
                line: 0
            });

            var subrec = rec.getCurrentSublistSubrecord({
                sublistId: 'item',
                fieldId: 'orderschedule'
            });

            // Create a variable and initialize it to the value of the trandate field.

            var dateValue = subrec.getSublistValue({
                sublistId: 'schedule',
                fieldId: 'trandate',
                line: 0
            });

            // Create a variable and initialize it to the value of the memo field.

            var memoValue = subrec.getSublistValue({
                sublistId: 'item',
                fieldId: 'memo',
                line: 0
            });
        }
    }

    afterSubmit;

    // End of script.
```
var memoValue = subrec.getSublistValue({
    sublistId: 'schedule',
    fieldId: 'memo',
    line: 0
});

// Print the retrieved values to the execution log.

try {
    log.debug({
        title: 'date value',
        details: 'date value: ' + dateValue
    });
    log.debug({
        title: 'memo value',
        details: 'memo value: ' + memoValue
    });
} catch (e) {
    log.error({
        title: e.name,
        details: e.message
    });
}

return {
    afterSubmit: afterSubmit
};

Example: Retrieving an Address Subrecord

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 2.0 to Retrieve a Sublist Subrecord.

To learn about SuiteScript scripting modes, see SuiteScript 2.0 – Standard and Dynamic Modes

/**
 * @NApiVersion 2.x
 * @NScriptType usereventscript
 */
define(['N/record'],
    function(record) {

    })
function afterSubmit(context) {
    // Load the record.
    var rec = record.load({
        type: record.Type.EMPLOYEE,
        id: 1863,
        isDynamic: false
    });

    // Retrieve a value from the sublist line
    var labelValue = rec.getSublistValue({
        sublistId: 'addressbook',
        fieldId: 'label',
        line: 0
    });

    // Retrieve the subrecord associated with that same line.
    var subrec = rec.getSublistSubrecord({
        sublistId: 'addressbook',
        fieldId: 'addressbookaddress',
        line: 0
    });

    // Create a variable to initialize it to the value of the subrecord's city field.
    var cityValue = subrec.getValue({
        fieldId: 'city'
    });

    // Print the retrieved values to the execution log.
    try {
        log.debug({
            title: 'label value',
            details: 'label value: ' + labelValue
        });

        log.debug({
            title: 'city value',
            details: 'city value: ' + cityValue
        });
    } catch (e) {
        log.error({
            title: 'Error retrieving values',
            details: 'Error: ' + e
        });
    }
}
Scripting Subrecords that Occur in Body Fields

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 2.0 to Create a Body Field Subrecord
# Using SuiteScript 2.0 to Update a Body Field Subrecord
# Using SuiteScript 2.0 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 2.0 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 2.0 – 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)` method. 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:

   ```javascript
   var inventoryDetailSubrecord = rec.getSubrecord({
     fieldId: 'inventorydetail'
   });
   ...
   ```

3. Set 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, 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 were creating an inventory detail subrecord, you could use the following expression to create a line on the subrecord’s inventory assignment sublist:

   ```javascript
   inventoryDetailSubrecord.selectNewLine({
     sublistId: 'inventoryassignment',
   });
   inventoryDetailSubrecord.setCurrentSublistValue({
     sublistId: 'inventoryassignment',
     fieldId: 'receiptinventorynumber',
     value: '012345'
   });
   inventoryDetailSubrecord.commitLine({
     sublistId: 'inventoryassignment'
   })
   ```
5. Save the record.

**Note:** For full script samples showing how to create a body field subrecord using dynamic mode, see Example: Creating an Address on a Subsidiary Record and Example: Creating an Inventory Detail Subrecord on a Body Field.

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 2.0 – 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:

   ```javascript
   var addressSubrecord = rec.getSubrecord({
   fieldId: 'mainaddress'
});
   ...
   ```

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:

   ```javascript
   inventoryDetailSubrecord.insertLine({
   sublistId: 'inventoryassignment',
   line: 0
 });
   
   inventoryDetailSubrecord.setSublistValue({
   sublistId: 'inventoryassignment',
   fieldId: 'receiptinventorynumber',
   line: 0,
   value: '12345'
 });
   ...
   ```

5. Save the record.
Example: Creating an Address on a Subsidiary Record

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 sample, 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 2.0 to Create a Body Field Subrecord.

To learn about SuiteScript scripting modes, see SuiteScript 2.0 – Standard and Dynamic Modes

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */

define(['N/record'],

function(record) {

    function afterSubmit(context) {

        var rec = record.creator({
            type: record.Type.SUBSIDIARY,
            isDynamic: true
        });

        // Set body fields on the record.

        rec.setValue({
            fieldId: 'name',
            value: 'US Subsidiary'
        });

        rec.setValue({
            fieldId: 'state',
            value: 'CA'
        });

        // Create the address subrecord.

        var subrec = rec.getSubrecord({
            fieldId: 'mainaddress',
        });

        subrec.setValue({
            fieldId: 'city',
            value: 'San Mateo'
        });
    }

    // Use this function when the record is submitted
    afterSubmit(null); // null argument is optional

});
```
Example: Creating an Inventory Detail Subrecord on a Body Field

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 sample, 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 2.0 to Create a Body Field Subrecord.

To learn about SuiteScript scripting modes, see SuiteScript 2.0 – Standard and Dynamic Modes

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */

define([ 'N/record' ],
    function (record) {

        function afterSubmit(context) {

            // Create the record.
            var rec = record.create({
                type: record.Type.ASSEMBLY_BUILD,
                isDynamic: true
            });

            // Set body fields.
            rec.setValue({
                fieldId: 'subsidiary',
                value: '1'
            });

            rec.setValue({
                fieldId: 'location',
                value: '6'
            });

            rec.setValue({
                fieldId: 'item',
                value: '699'
            });

            rec.setValue({
                fieldId: 'quantity',
                value: 1
            });

            // Create the subrecord.
            var subrec = rec.getSubrecord({
                fieldId: 'inventorydetail'
            });
        }
    });
```
// Create a line on the subrecord's inventory assignment sublist.

subrec.selectNewLine({
    sublistId: 'inventoryassignment',
});

subrec.setCurrentSublistValue({
    sublistId: 'inventoryassignment',
    fieldId: 'receiptinventorynumber',
    value: '012345'
});

subrec.commitLine({
    sublistId: 'inventoryassignment'
});

// Save the record.

try {
    var recId = rec.save();

    log.debug({
        title: 'Record created successfully',
        details: 'Id: ' + recId
    });
} catch (e) {

    log.error({
        title: e.name,
        details: e.message
    });
}

return {
    afterSubmit: afterSubmit
};

---

Using SuiteScript 2.0 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 2.0 – 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:

```javascript
var subrec = call.getSubrecord({
    fieldId: 'inventorydetail'
});
```

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 samples showing how to edit a body field subrecord using dynamic mode, see Example: Editing a Body Field Address Subrecord and Example: Editing a Body Field Inventory Detail Subrecord.

---

**Example: Editing a Body Field Address Subrecord**

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 2.0 to Update a Body Field Subrecord.

```javascript
/** *
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */
define([
    'N/record'
],
function (record) {

    function afterSubmit(context) {

        // Load the record.

        var rec = record.load({
            objectId: context.itemId,
            isDynamic: true
        });

        rec.selectLine({
            lineId: 'line_1',
            fieldId: 'field_1'
        });

        rec.setCurrentSublistValue({
            sublistId: 'sublist_1',
            fieldId: 'field_2',
            lineId: 'line_1',
            newValue: 'new value'
        });

        rec.commitLine({
            lineId: 'line_1'
        });

        rec.save();
    }

    // Call the function after the record is submitted.

    record.afterSubmit.after = afterSubmit;
});
```
SuiteScript 2.0 Scripting Subrecords

```javascript
  type: record.Type.SUBSIDIARY,
  id: 1,
  isDynamic: true
});

// Load the subrecord.

var subrec = rec.getSubrecord({
  fieldId: 'mainaddress'
});

// Make changes to one field.

subrec.setValue({
  fieldId: 'addr1',
  value: '12331-A Riata Trace Parkway'
});

// Save the record.

try {
  var recId = rec.save();

  log.debug({
    title: 'Record updated successfully',
    details: 'Id: ' + recId
  });

} catch (e) {

  log.error({
    title: e.name,
    details: e.message
  });
}

return {
  afterSubmit : afterSubmit
};
```

Example: Editing a Body Field Inventory Detail Subrecord

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 2.0 to Update a Body Field Subrecord.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */
define([ 'N/record' ],
    function (record) {

        function afterSubmit(context) {

            // Load the record.
            var rec = record.load({
                type: record.Type.ASSEMBLY_BUILD,
                id: 4918,
                isDynamic: true
            });

            // Load the subrecord.
            var subrec = rec.getSubrecord({
                fieldId: 'inventorydetail'
            });

            // Identify a line on the subrecord's sublist.
            subrec.selectLine({
                sublistId: 'inventoryassignment',
                line: 0
            });

            // Make changes to one sublist field.
            subrec.setCurrentSublistValue({
                sublistId: 'inventoryassignment',
                fieldId: 'receiptinventorynumber',
                value: '890123'
            });

            // Save the record.
            try {
                var recId = rec.save();

                log.debug({
                    title: 'Record updated successfully',
                    details: 'Id: ' + recId
                });
            } catch (e) {
                log.error({
                    title: 'Record update unsuccessful',
                    details: e.message
                });
            }
        }
    });
```
Using SuiteScript 2.0 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 2.0 – Standard and Dynamic Modes

To use SuiteScript 2.0 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:

   ```javascript
   var subrec = call.getSubrecord({
      fieldId: 'mainaddress'
   });
   ```

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:

   ```javascript
   var cityValue = subrec.getValue({
      fieldId: 'city'
   });
   ```

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:

   ```javascript
   var cityValue = subrec.getValue({
      sublistId: 'inventoryassignment',
      fieldId: 'receiptinventorynumber',
   });
   ```
Example: Retrieving a Body Field Address Subrecord

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 if 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 2.0 to Update a Body Field Subrecord](http://www.example.com).

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */
define (['N/config', 'N/record'],
function (config, record) {

function afterSubmit(context) {

// The Company Information page is not a standard record. You access it by
// using the config.load method.

var companyInfo = config.load(
    {type: config.Type.COMPANY_INFORMATION}
);

// Retrieve some of the preferences stored on the main part of the
// Company Information page. You interact with these preferences as if
// they were body fields.

var companyName = companyInfo.getValue(
    {fieldId: 'companyname'}
);

var employerId = companyInfo.getValue(
    {fieldId: 'employerid'}
);
}
```
// The Company Information page also includes some address fields on
// the body of the record. Retrieve these fields.

var state = companyInfo.getValue({
    fieldId: 'state'
});

var country = companyInfo.getValue({
    fieldId: 'country'
});

// Retrieve details from the subrecord that represents the main address.
// As a first step, instantiate the subrecord.

var mainAddress = companyInfo.getSubrecord({
    fieldId: 'mainaddress'
});

// Retrieve values from the main address subrecord.

var mainAddressCity = mainAddress.getValue({
    fieldId: 'city'
});

var mainAddressState = mainAddress.getValue({
    fieldId: 'state'
});

// Retrieve details from the subrecord that represents the shipping address.
// To start, instantiate the subrecord.

var shippingAddress = companyInfo.getSubrecord('shippingaddress');

// Retrieve values from the subrecord.

var shippingAddressCity = shippingAddress.getValue({
    fieldId: 'city'
});

var shippingAddressState = shippingAddress.getValue({
    fieldId: 'state'
});

// Retrieve details from the subrecord that represents the return address.
// To start, instantiate the shipping address subrecord.
var returnAddress = companyInfo.getSubrecord('returnaddress');

// Retrieve values from the subrecord.

var returnAddressCity = returnAddress.getValue({
    fieldId: 'city'
});

var returnAddressState = returnAddress.getValue({
    fieldId: 'state'
});

// Write selected details to the log.

log.debug ({
    title: 'mainAddressState',
    details: mainAddressState
});

log.debug ({
    title: 'shippingAddressState',
    details: shippingAddressState
});

log.debug ({
    title: 'returnAddressState',
    details: returnAddressState
});

return {
    afterSubmit: afterSubmit
};

---

Scripting Transaction Shipping and Billing Addresses

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
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 Example: Using SuiteScript 2.0 to Select an Existing Shipping Address.

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 Example: Using SuiteScript 2.0 to Create a New Shipping Address.

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.

Example: Using SuiteScript 2.0 to Create a New Shipping Address

The following script 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.

To use this sample, 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 2.0 – Standard and Dynamic Modes). For more information, see Understanding the Address Subrecord.
define([ 'N/record' ],

function(record) {

    function afterSubmit(context) {

        // Create the record.
        var rec = record.create(
            type: record.Type.SALES_ORDER,
            isDynamic: true
        );

        // Set body fields on the record.
        rec.setValue(
            fieldId: 'entity',
            value: '2163'
        );

        rec.setValue(
            fieldId: 'memo',
            value: '102A'
        );

        // Create the subrecord.
        var subrec = rec.getSubrecord(
            fieldId: 'shippingaddress'
        );

        // Set values on the subrecord.
        // Set country field first when script uses dynamic mode
        subrec.setValue(
            fieldId: 'country',
            value: 'US'
        );

        subrec.setValue(
            fieldId: 'city',
            value: 'New York'
        );

        subrec.setValue(
            fieldId: 'state',
            value: 'New York'
        );
    }
})
subrec.setValue(
    {
      fieldId: 'zip',
      value: '10018'
    });

subrec.setValue(
    {
      fieldId: 'addr1',
      value: '8 W 40th St.'
    });

// Create a line in the item sublist.

rec.selectNewLine(
    {
      sublistId: 'item'
    });

rec.setCurrentSublistValue(
    {
      sublistId: 'item',
      fieldId: 'item',
      value: '100'
    });

rec.setCurrentSublistValue(
    {
      sublistId: 'item',
      fieldId: 'quantity',
      value: '11'
    });

rec.commitLine(
    {
      sublistId: 'item'
    });

// Save the record.

try {
  var recId = rec.save();

  log.debug(
    {
      title: 'Record created successfully',
      details: 'Id: ' + recId
    });
}
catch (e) {

  log.error(
    {
      title: e.name,
      details: e.message
    });
}
}

return {
  afterSubmit: afterSubmit
}
Example: Using SuiteScript 2.0 to Select an Existing Shipping Address

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.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */

define(['N/record'],
    function(record) {
        function afterSubmit(context) {
            var rec = record.create({
                type: record.Type.SALES_ORDER,
                isDynamic: true
            });

            // Set body fields on the record.
            rec.setValue({
                fieldId: 'entity',
                value: '110'
            });

            // Create one line in the item sublist.
            rec.selectNewLine({
                sublistId: 'item'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'item',
                value: '100'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'quantity',
                value: '3'
            });

            // Set subrecord methods.
            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'firstName',
                value: 'John'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'lastName',
                value: 'Doe'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'street',
                value: '123 Main St'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'city',
                value: 'Anytown'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'state',
                value: 'CA'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'zip',
                value: '12345'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'country',
                value: 'US'
            });

            // Set subrecord methods.
            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'firstName',
                value: 'Jane'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'lastName',
                value: 'Smith'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'street',
                value: '456 Oak St'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'city',
                value: 'Nowhere'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'state',
                value: 'NC'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'zip',
                value: '123456'
            });

            rec.setCurrentSublistValue({
                sublistId: 'item',
                fieldId: 'country',
                value: 'US'
            });
        }
    });
```

Note: For details about working with a transaction’s shipping or billing address, see Scripting Transaction Shipping and Billing Addresses.
Example: Using SuiteScript 2.0 to Retrieve a Shipping Address

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.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType UserEventScript
 */
define([ 'N/record' ],
function (record) {
  function beforeSubmit(context) {
    if (context.type==context.UserEventType.CREATE) {

      var salesOrder = context.newRecord;

      var shipToAddress = salesOrder.getSubrecord({
        fieldId: 'shippingaddress'
      });

      log.debug({
```
Example: Using SuiteScript 2.0 to Retrieve one Value from a Shipping Address

The following script loads a sales order record. It retrieves a value from the shipping address and prints it to the execution log.

```javascript
// Load the record.
var rec = record.load(
    { type: record.Type.SALES_ORDER,
      id: 5025,
      isDynamic: true
    });

// Retrieve the subrecord.
var subrec = rec.getSubrecord(
    { fieldId: 'shippingaddress'
  });

// Create a variable and initialize it to the value of the subrecord's city field.
var cityValue = subrec.getValue(
    { fieldId: 'city'
  });
```

Note: For details about working with a transaction’s shipping or billing address, see Scripting Transaction Shipping and Billing Addresses.
try {

    log.debug({
        title: 'city value',
        details: 'city value: ' + cityValue
    });

} catch (e) {

    log.error({
        title: e.name,
        details: e.message
    });
}

return {
    pageInit: pageInit
};
}
SuiteScript 2.0 Custom Pages

You can use SuiteScript 2.0 methods to create custom forms and list pages. See the following details:

# SuiteScript 2.0 Custom Forms
# SuiteScript 2.0 Custom List Pages
# SuiteScript 2.0 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 2.0 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 2.0 Suitelet Script Type.
# Portlet: Portlet scripts are rendered within dashboard portlets. For information about portlet scripts, see SuiteScript 2.0 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 2.0 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, such as standard buttons for Reset or Submit. The Reset button removes edits from a form, and 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 reset button to a form, see the help topic `Form.addResetButton(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.
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 OUTSIDEBELOW 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.

```javascript
//Add additional code
...
var outsideAbove = form.addField({
  id: 'outsideabovefield',
  type: serverWidget.FieldType.TEXT,
  label: 'Outside Above Field'
});
outsideAbove.updateLayoutType({
  layoutType: serverWidget.FieldLayoutType.OUTSIDEABOVE
});
... //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.
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.

```javascript
//Add additional code
...

var email = form.addField({
  id: 'emailfield',
  type: serverWidget.FieldType.EMAIL,
  label: 'Email',
  container: 'usergroup'
});

email.updateBreakType({
  breakType : serverWidget.FieldBreakType.STARTCOL
});
...

//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.
Sample Custom Form Script

The following screenshot displays a form created by a Suitelet. You can use the sample code provided below to create this form.

Steps for Creating a Custom Form with SuiteScript 2.0

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.

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

2. Add the define function to load the N/ui/serverWidget Module module.

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

define(['N/ui/serverWidget'], function(serverWidget) {
    // Section One - Forms - See "Steps for Creating a Form"
    // Section Two - Tabs - See "Steps for Adding a Tab to a Form"
    // Section Three - Sublist - See "Steps for Adding a Sublist to a Form"
    return {
    }
});
```

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.

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

define(['N/ui/serverWidget'], function(serverWidget) {
    function onRequest(context) {
        if (context.request.method === 'GET') {
            context.response.writePage(form);
        // Section One - Forms - See "Steps for Creating a Form"
        // Section Two - Tabs - See "Steps for Adding a Tab to a Form"
        // Section Three - Sublist - See "Steps for Adding a Sublist to a Form"
        } else {
            // Section Four - Output - Used in all sections
        }
    }
    return {
    }
});
```

4. In section one, call the createForm method using serverWidget.createForm(options).
SuiteScript 2.0 Custom Forms

5. In section one, create Field Groups to organize the fields using \texttt{Form.addFieldGroup(options)}. Use the properties defined in \texttt{serverWidget.FieldGroup} to control the appearance and behavior of the field groups.

\begin{verbatim}
  var usergroup = form.addFieldGroup({
    id: 'usergroup',
    label: 'User Information'
  });
  usergroup.isSingleColumn = true;

  var companygroup = form.addFieldGroup({
    id: 'companygroup',
    label: 'Company Information'
  });
\end{verbatim}

6. In section one, use \texttt{Form.addResetButton(options)} and \texttt{Form.addSubmitButton(options)} to add the Reset and Submit buttons to a form. Change the text displayed on the buttons by modifying the label property.

\begin{verbatim}
  form.addSubmitButton({
    label: 'Submit'
  });

  form.addResetButton({
    label: 'Clear Fields'
  });
\end{verbatim}

7. In section one, use \texttt{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 \texttt{serverWidget.Field} to change the behavior and properties of the fields. This sample uses \texttt{Field.isMandatory} to define several required fields, and \texttt{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 \texttt{serverWidget.FieldType}.

\begin{verbatim}
  var select = form.addField({
    id: 'titlefield',
    type: serverWidget.FieldType.SELECT,
    label: 'Title',
    container: 'usergroup'
  });

  select.addSelectOption({
    value: 'Mr.',
    text: 'Mr.'
  });

  select.addSelectOption({
    value: 'MS.',
  ...
\end{verbatim}

\textbf{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.
8. In section four, define a variable for each field in this sample.
var delimiter = /\u0001/;
var titleField = context.request.parameters.titlefield;
var fnameField = context.request.parameters.fnamefield;
var lnameField = context.request.parameters.lnamefield;
var emailField = context.request.parameters.emailfield;
var companyField = context.request.parameters.companyfield;
var phoneField = context.request.parameters.phonefield;
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.

context.response.write('You have entered:
+ '<br/>' Name: ' + titleField + ' ' + fnameField + ' ' + lnameField
+ '<br/>' Email: ' + emailField
+ '<br/>' Company: ' + companyField
+ '<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.

   // Section Two - Tabs

   var tab1 = form.addTab({
       id: 'tab1id',
       label: 'Payment'
   });
   tab1.helpText = 'Help Text Goes Here';

   var tab2 = form.addTab({
       id: 'tab2id',
       label: 'Inventory'
   });

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.
3. 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).

```javascript
//Subtab One Fields
var ccselect = form.addField(
    {id: 'cctypefield',
     type: serverWidget.FieldType.SELECT,
     label: 'Credit Card',
     container: 'subtab1id'});

ccselect.addSelectOption(
    {value: 'PayCard0',
     text: 'Payment Card 0'});

ccselect.addSelectOption(
    {value: 'PayCard1',
     text: 'Payment Card 1'});

ccselect.addSelectOption(
    {value: 'PayCard2',
     text: 'Payment Card 2'});

var expmonth = form.addField(
    {id: 'expmonth',
     type: serverWidget.FieldType.SELECT,
     label: 'Expiry Month',
     container: 'subtab1id'});

expmonth.updateLayoutType(
    {layoutType: serverWidget.FieldLayoutType.STARTROW});

expmonth.addSelectOption(
    {value: 'Month0'});
```

```javascript
form.addSubtab(
    {id: 'subtab1id',
     label: 'Payment Information',
     tab: 'tab1id'});

form.addSubtab(
    {id: 'subtab2id',
     label: 'Transaction Record',
     tab: 'tab1id'});
```
expmonth.addSelectOption({
  value: '01',
  text: 'Jan'
});

expmonth.addSelectOption({
  value: '02',
  text: 'Feb'
});

expmonth.addSelectOption({
  value: '03',
  text: 'Mar'
});

expmonth.addSelectOption({
  value: '04',
  text: 'Apr'
});

expmonth.addSelectOption({
  value: '05',
  text: 'May'
});

expmonth.addSelectOption({
  value: '06',
  text: 'Jun'
});

expmonth.addSelectOption({
  value: '07',
  text: 'Jul'
});

expmonth.addSelectOption({
  value: '08',
  text: 'Aug'
});

expmonth.addSelectOption({
  value: '09',
  text: 'Sep'
});

expmonth.addSelectOption({
  value: '10',
  text: 'Oct'
});

expmonth.addSelectOption({
  value: '11',
  text: 'Nov'
});

expmonth.addSelectOption({
  value: '12',
  text: 'Dec'
});
value: '12',
    text: 'Dec'
});

var expyear = form.addField({
    id: 'expyear',
    type: serverWidget.FieldType.SELECT,
    label: 'Expiry Year',
    container: 'subtab1id'
});

expyear.updateLayoutType({
    layoutType: serverWidget.FieldLayoutType.ENDROW});

expyear.addSelectOption({
    value: '2020',
    text: '2020'
});

expyear.addSelectOption({
    value: '2019',
    text: '2019'
});

expyear.addSelectOption({
    value: '2018',
    text: '2018'
});

var credfield = form.addCredentialField({
    id: 'credfield',
    label: 'Credit Card Number',
    restrictToDomains: 'www.mysite.com',
    restrictToScriptIds: 'customscript_my_script',
    restrictToCurrentUser: false,
    container: 'subtab1id'
});
credfield.maxLength = 32;

// Subtab Two fields
form.addField({
    id: 'transactionfield',
    type: serverWidget.FieldType.LABEL,
    label: 'Transaction History - Coming Soon',
    container: 'subtab2id'
});

// Tab Two fields
form.addField({
    id: 'inventoryfield',
    type: serverWidget.FieldType.LABEL,
    label: 'Inventory - Coming Soon',
    container: 'tab2id'
});
4. 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.

```javascript
var ccField = context.request.parameters.cctypefield;
var ccNumber = context.request.parameters.ccredfield;
var expMonth = context.request.parameters.expmonth;
var expYear = context.request.parameters.expyear;
```

5. In section four, update the output statement to display the variables created when the Submit button is clicked.

```javascript
context.response.write('You have entered:
   Name : ' + titleField + ' ' + fnameField + ' ' + lnameField
   Email : ' + emailField
   Company : ' + companyField
   Phone : ' + phoneField + ' Website : ' + urlField
   Credit Card : ' + ccField
   Number : ' + ccNumber
   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.

1. 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`.

```javascript
var sublist = form.addSublist({
   id : 'sublistid',
   type : serverWidget.SublistType.INLINEEDITOR,
   label : 'Inventory',
   tab : 'tab2id'
});
```

2. 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.

```javascript
sublist.addButton({
   id : 'buttonId',
   label : 'Print ',
   functionName : '' // Add the function triggered on button click
});
```

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`.

```javascript
sublist.addField({
   id : 'datefieldid',
   type : serverWidget.FieldType.DATE,
   label : 'Date'
});
```

---

**Note:** `Sublist.addField(options)` does not support the inline HTML field type.
Complete Custom Form Script Sample

The following code creates a Suitelet that generates a custom form containing several field types, reset and submit buttons, tabs, and a sublist. For steps to create this script, see Sample Custom Form Script.

```javascript
function onRequest(context) {
  if (context.request.method === 'GET') {
    // Section One - Forms - See "Steps for Creating a Form"
    var form = serverWidget.createForm({
      title: 'Customer Information'
    });

    var usergroup = form.addFieldGroup({
      id: 'usergroup',
      label: 'User Information'
    });
    usergroup.isSingleColumn = true;

    var companygroup = form.addFieldGroup({
      id: 'companygroup',
      label: 'Company Information'
    });

    var select = form.addField({
      id: 'titlefield',
      type: serverWidget.FieldType.SELECT,
      label: 'Title',
      container: 'usergroup'
    });

    sublist.addField({
      id: 'productfieldid',
      type: serverWidget.FieldType.TEXT,
      label: 'Product'
    });

    sublist.addField({
      id: 'qtyfieldid',
      type: serverWidget.FieldType.INTEGER,
      label: 'Quantity'
    });

    sublist.addField({
      id: 'upfieldid',
      type: serverWidget.FieldType.CURRENCY,
      label: 'Unit Cost'
    });
  }
}
```
SuiteScript 2.0 Custom Forms

```javascript
});
select.addSelectOption(
    value: 'Mr.',
    text: 'Mr.'
);
select.addSelectOption(
    value: 'Ms.',
    text: 'Ms.'
);
select.addSelectOption(
    value: 'Dr.',
    text: 'Dr.'
);

var fname = form.addField(
    id: 'fnamefield',
    type: serverWidget.FieldType.TEXT,
    label: 'First Name',
    container: 'usergroup'
);
fname.isMandatory = true;

var lname = form.addField(
    id: 'lnamefield',
    type: serverWidget.FieldType.TEXT,
    label: 'Last Name',
    container: 'usergroup'
);
lname.isMandatory = true;

form.addField(
    id: 'emailfield',
    type: serverWidget.FieldType.EMAIL,
    label: 'Email',
    container: 'usergroup'
);

var companyname = form.addField(
    id: 'companyfield',
    type: serverWidget.FieldType.TEXT,
    label: 'Company',
    container: 'companygroup'
);
companyname.defaultValue = 'Company Name';

form.addField(
    id: 'phonefield',
    type: serverWidget.FieldType.PHONE,
    label: 'Phone Number',
    container: 'companygroup'
);

form.addField(
    id: 'urlfield',
    type: serverWidget.FieldType.URL,
    label: 'URL',
    container: 'companygroup'
)
```
SuiteScript 2.0 Custom Forms

```javascript
type: serverWidget.FieldType.URL,
label: 'Website',
container: 'companygroup'
});

form.addSubmitButton({
  label: 'Submit'
});

form.addResetButton({
  label: 'Reset'
});

// Section Two - Tabs - See "Steps for Adding a Tab to a Form"
var tab1 = form.addTab({
  id: 'tab1id',
  label: 'Payment'
});
tab1.helpText = 'Help Text Goes Here';

var tab2 = form.addTab({
  id: 'tab2id',
  label: 'Inventory'
});

form.addSubtab({
  id: 'subtab1id',
  label: 'Payment Information',
  tab: 'tab1id'
});

form.addSubtab({
  id: 'subtab2id',
  label: 'Transaction Record',
  tab: 'tab1id'
});

// Subtab One Fields
var ccselect = form.addField({
  id: 'cctypefield',
  type: serverWidget.FieldType.SELECT,
  label: 'Credit Card',
  container: 'subtab1id'
});

ccselect.addSelectOption({
  value: 'PayCard0',
  text: 'Payment Card 0'
});

ccselect.addSelectOption({
  value: 'PayCard1',
  text: 'Payment Card 1'
});
```
ccselect.addSelectOption({
    value: 'PayCard2',
    text: 'Payment Card 2'
});

var expmonth = form.addField({
    id: 'expmonth',
    type: serverWidget.FieldType.SELECT,
    label: 'Expiry Date:',
    container: 'subtab1id'
});
expmonth.updateLayoutType({
    layoutType: serverWidget.FieldLayoutType.STARTROW
});

expmonth.addSelectOption({
    value: '01',
    text: 'Jan'
});

expmonth.addSelectOption({
    value: '02',
    text: 'Feb'
});

expmonth.addSelectOption({
    value: '03',
    text: 'Mar'
});

expmonth.addSelectOption({
    value: '04',
    text: 'Apr'
});

expmonth.addSelectOption({
    value: '05',
    text: 'May'
});

expmonth.addSelectOption({
    value: '06',
    text: 'Jun'
});

expmonth.addSelectOption({
    value: '07',
    text: 'Jul'
});

expmonth.addSelectOption({
    value: '08',
    text: 'Aug'
});
expmonth.addSelectOption(
    value: '09',
    text: 'Sep'  
);

expmonth.addSelectOption(
    value: '10',
    text: 'Oct'  
);

expmonth.addSelectOption(
    value: '11',
    text: 'Nov'  
);

expmonth.addSelectOption(
    value: '12',
    text: 'Dec'  
);

var expyear = form.addField(
    id: 'expyear',
    type: serverWidget.FieldType.SELECT,
    label: 'Expiry Year',
    container: 'subtab1id'
);

expyear.updateLayoutType(
    layoutType: serverWidget.FieldLayoutType.ENDROW);

expyear.addSelectOption(
    value: '2020',
    text: '2020'  
);

expyear.addSelectOption(
    value: '2019',
    text: '2019'  
);

expyear.addSelectOption(
    value: '2018',
    text: '2018'  
);

var credfield = form.addCredentialField(
    id: 'credfield',
    label: 'Credit Card Number',
    restrictToDomains: 'www.mysite.com',
    restrictToScriptIds: 'customscript_my_script',
    restrictToCurrentUser: false,
    container: 'subtab1id'
);
credfield.maxLength = 32;

// Subtab two Fields
form.addField({
  id: 'transactionfield',
  type: serverWidget.FieldType.LABEL,
  label: 'Transaction History - Coming Soon',
  container: 'subtab2id'
});

// Tab Two Fields
form.addField({
  id: 'inventoryfield',
  type: serverWidget.FieldType.LABEL,
  label: 'Inventory - Coming Soon',
  container: 'tab2id'
});

// Section Three - Sublist - See "Steps for Adding a Sublist to a Form"

var sublist = form.addSublist({
  id: 'sublistid',
  type: serverWidget.SublistType.INLINEEDITOR,
  label: 'Inline Sublist',
  tab: 'tab2id'
});

sublist.addButton({
  id: 'buttonId',
  label: 'Print',
  functionName: '' // Add the function triggered on button click
});

// Sublist Fields
sublist.addField({
  id: 'datefieldid',
  type: serverWidget.FieldType.DATE,
  label: 'Date'
});

sublist.addField({
  id: 'productfieldid',
  type: serverWidget.FieldType.TEXT,
  label: 'Product'
});

sublist.addField({
  id: 'qtyfieldid',
  type: serverWidget.FieldType.INTEGER,
  label: 'Quantity'
});

sublist.addField({
  id: 'upfieldid',
  type: serverWidget.FieldType.CURRENCY,
  label: 'Unit Cost'
});
SuiteScript 2.0 Custom Forms

```javascript
context.response.writePage(form);

} else {
    // Section Four - Output - Used in all sections
    var delimiter = '\u0001';
    var titleField = context.request.parameters.titlefield;
    var fnameField = context.request.parameters.fnamefield;
    var lnameField = context.request.parameters.lnamefield;
    var emailField = context.request.parameters.emailfield;
    var companyField = context.request.parameters.companyfield;
    var phoneField = context.request.parameters.phonefield;
    var urlField = context.request.parameters.urlfield;

    var ccField = context.request.parameters.cctypefield;
    var ccNumber = context.request.parameters.credfield;
    var expMonth = context.request.parameters.expmonth;
    var expYear = context.request.parameters.expyear;

    context.response.write('You have entered:
' + titleField + ' ' + fnameField + ' ' + lnameField
    + ' Email: ' + emailField
    + ' Company: ' + companyField
    + ' Phone: ' + phoneField + ' Website: ' + urlField
    + ' Credit Card: ' + ccField
    + ' Number: ' + ccNumber
    + ' Expiry Date: ' + expMonth + '/' + expYear);
}

onRequest: onRequest
```

SuiteScript 2.0 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 2.0 Suitelet 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 Custom List Page Script

The following screenshot displays a list page created by a Suitelet. You can use the sample code provided below to create this list yourself.
Steps for Creating a Custom List Page with SuiteScript 2.0

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 to walk through the steps for creating the script, complete the following procedure.

To create a list page, complete the following steps:

1. Create a Suitelet, and add the required JSDoc tags.

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

2. Add the `define` function to load the `N/ui/serverWidget Module` module.

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

define([ 'N/ui/serverWidget'],
    function (serverWidget) {
    
});
```

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.

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

define([ 'N/ui/serverWidget'],
    function (serverWidget) {
        if (context.request.method === 'GET') {
            return;
        }
    
});
```

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.
function (serverWidget) {
    function onRequest(context) {
        if (context.request.method === 'GET') {
            // Section One - List - See 'Steps for Creating a List', Step Five
            // Section Two - Columns - See 'Steps for Creating a List', Step Seven
            context.response.writePage(list);
        } else {
        }
    }
    return {
        onRequest: onRequest
    }
});

4. In section one, create a list page using serverWidget.createList(options). You can use serverWidget.ListStyle to define the style of the list page.

```javascript
var list = serverWidget.createList({
    title: 'Purchase History'
});
list.style = serverWidget.ListStyle.REPORT;
```

5. 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.

```javascript
list.addButton({
    id: 'buttonId',
    label: 'Test',
    functionName: /* the function called when the button is pressed */
});
```

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.

```javascript
var datecol = list.addColumn({
    id: 'column1',
    type: serverWidget.FieldType.DATE,
    label: 'Date',
    align: serverWidget.LayoutJustification.RIGHT
});
list.addColumn({
    id: 'column2',
    type: serverWidget.FieldType.TEXT,
    label: 'Product',
    align: serverWidget.LayoutJustification.RIGHT
});
list.addColumn({
    id: 'column3',
});
```

⚠️ Important: The CHECKBOX field type is not supported by List.addColumn(options), and the MULTISELECT field is not supported by SuiteScript 2.0 Suitelets.
SuiteScript 2.0 Custom List Pages

7. In section two, use `List.addColumn(options)` to define a single row on the list. Or use `List.addColumn(options)` to define multiple rows at once.

```javascript
list.addRow({
row : { column1 : '02/12/2018' , column2 : 'Widget' , column3 : '4' , column4 : '4.50'  }
});

list.addRows([
{column1 : '02/12/2018' , column2 : 'Widget' , column3 : '4' , column4 : '4.50' },
{column1 : '02/14/2018' , column2 : 'Sprocket' , column3 : '6' , column4 : '11.50' },
{column1 : '02/16/2018' , column2 : 'Gizmo' , column3 : '9' , column4 : '1.25' }
]);
```

Complete Custom List Page Sample Script

The following code creates a Suitelet that generates a custom list page. For steps to create this script, see Sample Custom List Page Script.

```javascript
/**
 * @NApiVersion 2.x
 * @NScriptType Suitelet
 */
define([ 'N/ui/serverWidget' ],
function (serverWidget) {
function onRequest(context){
if (context.request.method === 'GET'){
    // Section One - List - See 'Steps for Creating a List', Step Five
    var list = serverWidget.createList({
        title : 'Purchase History'
    });

    list.style = serverWidget.ListStyle.REPORT;

    list.addButton({
        id : 'buttonId',
        label : 'Test',
        functionName : '' // the function called when the button is pressed
    });

    // Section Two - Columns - See 'Steps for Creating a List', Step Seven
```
SuiteScript 2.0 Custom List Pages

```javascript
var datecol = list.addColumn(
    id : 'column1',
    type : serverWidget.FieldType.DATE,
    label : 'Date',
    align : serverWidget.LayoutJustification.RIGHT
);

list.addColumn(
    id : 'column2',
    type : serverWidget.FieldType.TEXT,
    label : 'Product',
    align : serverWidget.LayoutJustification.RIGHT
);

list.addColumn(
    id : 'column3',
    type : serverWidget.FieldType.INTEGER,
    label : 'Quantity',
    align : serverWidget.LayoutJustification.RIGHT
);

list.addColumn(
    id : 'column4',
    type : serverWidget.FieldType.CURRENCY,
    label : 'Unit Cost',
    align : serverWidget.LayoutJustification.RIGHT
);

list.addRows(
    rows : [{column1 : '05/30/2018', column2 : 'Widget', column3 : '4', column4 : '4.50'},
            {column1 : '05/30/2018', column2 : 'Sprocket', column3 : '6', column4 : '11.50'},
            {column1 : '05/30/2018', column2 : 'Gizmo', column3 : '9', column4 : '1.25'}]
); context.response.writePage(list);
```

SuiteScript 2.0 Working with UI Objects

For information about using SuiteScript 2.0 UI Objects, see:

# Custom UI development
# UI Component Overview
# SuiteScript 2.0 UI Modules
# Using HTML
# Creating Custom Assistants
Custom UI development

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 2.0 UI Components

SuiteScript 2.0 UI components are a collection of objects and methods contained in the serverWidget API module. Using SuiteScript 2.0 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 2.0 UI Components, see Custom UI development.

**Note:** While SuiteScript 2.0 is recommended, SuiteScript 1.0 contains a set of UI Objects that can be used to build a custom UI. For more information on SuiteScript 1.0, see the help topic UI Objects 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 suitlet 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 Inline HTML.

UI Component Overview

UI Components are a collection of objects that can be used to customize your NetSuite experience using SuiteScript 2.0. 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 DOM. NetSuite UI should only be accessed using SuiteScript APIs.
**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 2.0 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 2.0 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 2.0 Portlet Script Type](#).

**User Event** scripts can also be used to manipulate custom forms. For more information on User Event Scripts, see [SuiteScript 2.0 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 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 2.0 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 2.0 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 2.0 UI Modules

Several API modules are available for building a custom UI using SuiteScript 2.0. This guide focuses primarily on building a UI using the serverWidget module, but these three modules are commonly used in the development of a custom UI.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/ui/serverWidget</td>
<td>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.</td>
</tr>
<tr>
<td>n/ui/message</td>
<td>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.</td>
</tr>
<tr>
<td>n/ui/dialog</td>
<td>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.</td>
</tr>
</tbody>
</table>

Using HTML

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

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.

```javascript
var htmlImage = form.addField({
  id: 'custpage_htmlfield',
  type: serverWidget.FieldType.INLINEHTML,
  label: 'HTML Image',
});
htmlImage.defaultValue = '<img src='https://system.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.

```html
<form method="post" class="form-horizontal" action="https://LinkToSuitelet.js">
  <table>
    <tbody>
      <tr>
        <td>First Name</td>
        <td class="col-md-8">
          <input class="form-control" id="firstname" placeholder="First Name" name="firstname" required="" type="text">
        </td>
      </tr>
    </tbody>
  </table>
</form>
```
Form Processing Managed by the Suitelet

The Suitelet manages the information collected from the HTML form. This Suitelet uses the \texttt{https.get(options)} method from the \texttt{N/https Module} to access the content from the HTML page, and uses methods from the \texttt{N/record Module}, \texttt{N/email Module}, and \texttt{N/search Module} to collect, process, and respond to the data the user submits.

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

define([ 'N/https', 'N/record', 'N/email', 'N/search'],

    function  callbackFunction(https, record, email, search) {

        function getFunction(context) {

            var  contentRequest = https.get({
                url : "https://LinkToFormPage.html"
            });
            var  contentDocument = contentRequest.body;
            var  sponsorid = context.request.parameters.sponsorid;

            if  (sponsorid) {
                contentDocument = contentDocument.replace("{{sponsorid}}", sponsorid);
                log.debug("Setting Sponsor", sponsorid)
            }

            var  projectid = context.request.parameters.projectid;

            if  (projectid) {
                contentDocument = contentDocument.replace("{{projectid}}", projectid);
            }

            return  contentDocument;
        }

```
```javascript
log.debug("Setting Project", projectid);

context.response.write(contentDocument);

function postFunction(context) {

var params = context.request.parameters;

var emailString = "First Name: {{firstname}}
Last Name: {{lastname}}
Email: {{email}}
Facebook URL: {{custentity_fb_url}}"

var contactRecord = record.create(
    type: "contact",
    isDynamic: true
);

for (param in params) {
    if (param === "company") {
        if (params[param] !== {{sponsorid}}) {
            contactRecord.setValue(
                fieldId: param,
                value: params[param]
            );
            var lkpfld = search.lookupFields(
                type: "vendor",
                id: params["company"],
                columns: ["entityid"]
            );
            emailString += " Sponsors: " + lkpfld.entityid;
        }
    } else {
        contactRecord.setValue(
            fieldId: "custentity_sv_shn_isindi",
            value: true
        );
    }
    if (param !== "project") {
        contactRecord.setValue(
            fieldId: param,
            value: params[param]
        );
        var replacer = "{{" + param + "}}";
        emailString = emailString.replace(replacer, params[param]);
    }
}

var contactId = contactRecord.save(
    ignoreMandatoryFields: true,
    enableSourcing: true
);
```
log.debug("Record ID", contactId);

if (params["project"]
["project"] !== "{{projectid}}") {

  var lkpfld = search.lookupFields(
    type: "job",
    id: params["project"],
    columns: ["companyname"]
  );

  emailString += "Project Name: " + lkpfld.companyname;

  var participationRec = record.create(
    type: "customrecord_project_participants",
    isDynamic: true
  );

  participationRec.setValue(
    fieldId: "custrecord_participants_volunteer",
    value: contactId
  )

  participationRec.setValue(
    fieldId: "custrecord_participants_project",
    value: params["project"]
  )

  var participationId = participationRec.save(
    enableSourcing: true,
    ignoreMandatoryFields: true
  )

  log.debug("Email String", emailString);

  email.send(
    author: -5,
    recipients: 256,
    subject: "New Volunteer Signed Up",
    body: "A new volunteer has joined:

    " + emailString
  );

  email.send(
    author: -5,
    recipients: params["email"],
    subject: "Thank you!",
    body: "Thank you for volunteering:

    " + emailString
  );

  var contentRequest = https.get(
    url: "https://LinkToFormCompletePage.html"
  );

  var contentDocument = contentRequest.body;
context.response.write(contentDocument);

}

function onRequestFxn(context) {

    if (context.request.method === "GET") {
        getFunction(context)
    } else {
        postFunction(context)
    }

}

return {
    onRequest: onRequestFxn
};

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

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.

Assistants are built with Suitelets and by using the N/ui/serverWidget Module.

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

To develop an assistant, complete the following steps:

1. **Create a new assistant and define steps.**
   Create a new assistant in the Suitelet and define each step for the assistant.

2. **Build assistant pages.**
   Use fields, field groups, and sublists to build out the assistant pages for each step.

3. **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.

For more information, see the help topic Understanding the Assistant Workflow.

Supported UI Components for Assistants

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

```javascript
/**
 * @NApiVersion 2.0
 * @NScriptType suitelet
 */
define([ 'N/ui/serverWidget' ], function (serverWidget) {  
  return {
    onRequest: function(context) {
      var assistant = serverWidget.createAssistant({ title : "New Supervisor", hideNavBar : true });
      var assignment = assistant.addStep({ id : 'assignment', label : 'Select new supervisor' });
      var review = assistant.addStep({ id : 'review', label : 'Review and Submit' });

      var writeAssignment = function() {
        assistant.addField({ id : "newsupervisor", type : "select", label : "Name", source : "employee" });
      }
    }
  }
});
```
assistant.addField({ id: "assignedemployee", type: "select", label: "Employee", source: "employee" });

var writeReview = function() {
    var supervisor = assistant.addField({ id: "newsupervisor", type: "text", label: "Name" });
    supervisor.defaultValue = context.request.parameters.inpt_newsupervisor;
    var employee = assistant.addField({ id: "assignedemployee", type: "text", label: "Employee" });
    employee.defaultValue = context.request.parameters.inpt_assignedemployee;
}

var writeResult = function() {
    var supervisor = context.request.parameters.newsupervisor;
    var employee = context.request.parameters.assignedemployee;
    context.response.write( "Supervisor: " + supervisor + "\nEmployee: " + employee);
}

var writeCancel = function() {
    context.response.write("Assistant was cancelled");
}

if (context.request.method == 'GET') //GET method means starting the assistant
{
    writeAssignment();
    assistant.currentStep = assignment;
    context.response.writePage(assistant)
}
else //POST method - process step of the assistant
{
    if (context.request.parameters.next == 'Finish') //Finish was clicked
        writeResult();
    else if (context.request.parameters.cancel) //Cancel was clicked
        writeCancel();
    else if (assistant.currentStep.stepNumber == 1) //transition from step 1 to step 2
        writeReview();
        assistant.currentStep = assistant.getNextStep();
        context.response.writePage(assistant);
    else //transition from step 2 back to step 1
        writeAssignment();
        assistant.currentStep = assistant.getNextStep();
        context.response.writePage(assistant);
}
};