ORACLE
NETSUITE

SuiteScript 2.0 for Experienced SuiteScript Developers

Student Guide

This printing: August 2021.

Oracle NetSuite
2300 Oracle Way
Austin, TX 78741

TABLE OF CONTENTS

BEFORE YOU BEGIN

Important Notes

⚠ IMPORTANT

This student guide uses functionality not covered in the videos.

If you need assistance with newly introduced syntax, please refer to the Help Center.

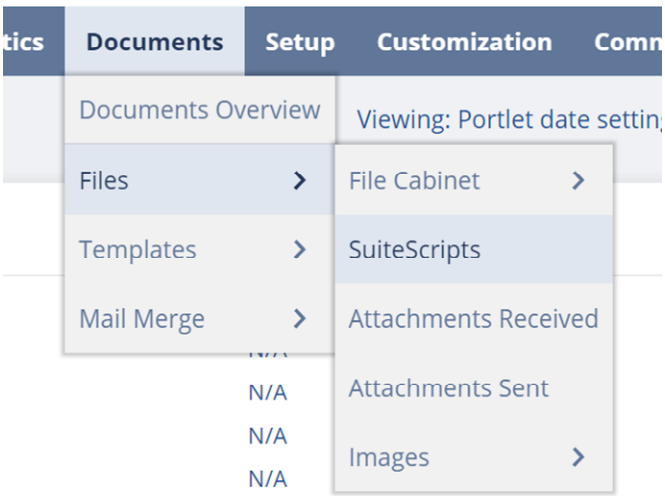
⚠ IMPORTANT

Eclipse IDE has been deprecated. Use a different IDE of your choosing.

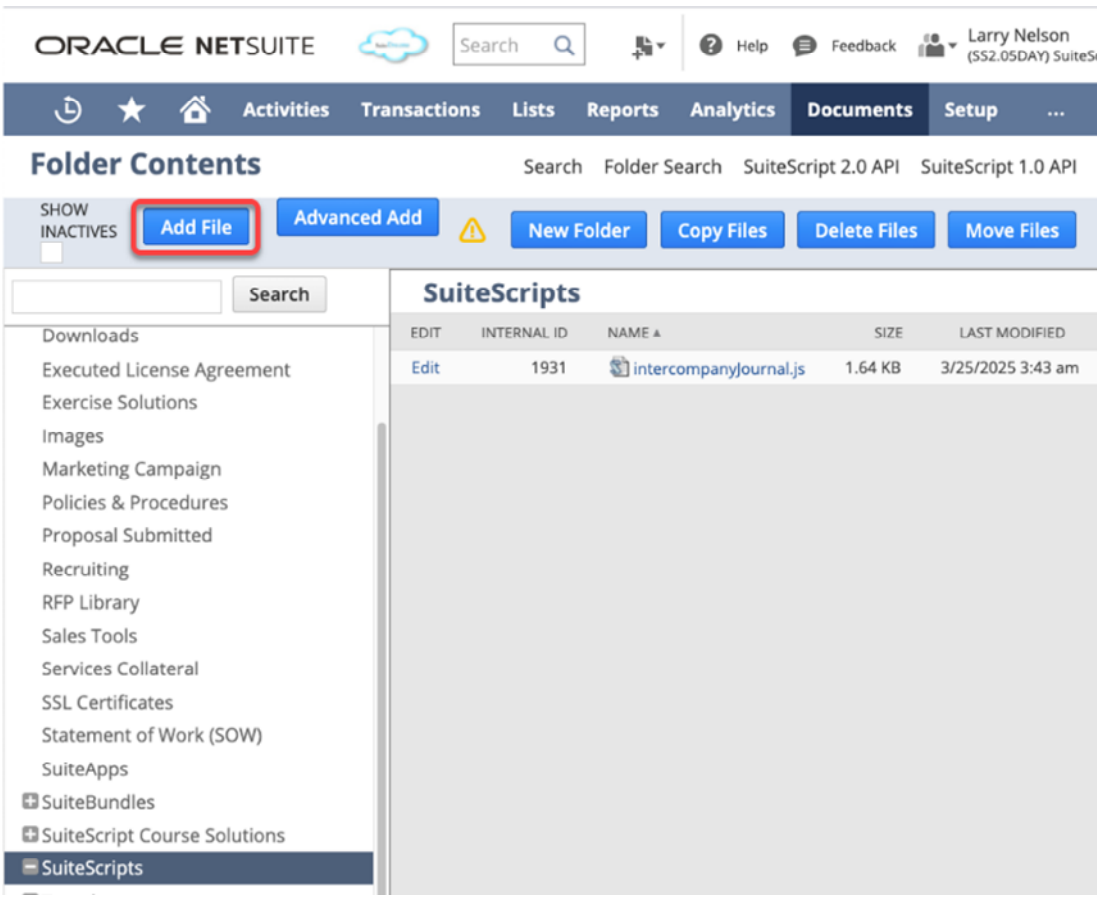
Refer to the below steps on how to upload files manually.

Uploading Files Manually

Go to **Documents > Files > SuiteScripts**.



Click on the **Add File** button on the top left and select the file to be uploaded.



Note: if there is a prompt for file replacement, click Yes.

INTRODUCING SUITESCRIPT 2.0

Module Exercises	
01	Welcome new customers through email
02	Debugging Server-side Scripts
03	Error handling and custom modules
04	Customer welcome reminder
05	Debugging Client-side SuiteScript 2.0 Scripts
Optional Exercises	
06	Custom UI Messages

01: Welcome New Customers Through e-mail

Scenario: New customers should automatically receive a welcome email as soon as the customer record is saved. After sending the email, a note should be added to the sales rep's record (which is the email sender) mentioning that an email was sent to the particular customer. Logs will also be added to the script to mention the same information.

To make development easier, here's the copy of the requirement done in SuiteScript 1.0

Note: For this exercise, adding the `type == 'create'` filter is optional. This should speed up the testing of your script by allowing you to use existing records.

```
function afterSubmit(type){
   if (type == 'create') {
      var customer = nlapiGetNewRecord();
      var customerId = customer.getId();
      var salesRepId = customer.getFieldValue('salesrep');
      var customerName = customer.getFieldValue('entityid');
      var salesRepName = customer.getFieldText('salesrep');

      nlapiSendEmail(salesRepId, customerId, 'Welcome to SuiteDreams',
                'Welcome. We are glad for you to be a customer of SuiteDreams');

      var salesRep = nlapiLoadRecord('employee', salesRepId);
      var salesRepComments = salesRep.getFieldValue('comments');
      salesRep.setFieldValue('comments', salesRepComments + '\n' +
                           'Welcome email sent to' + customerName);
      nlapiSubmitRecord(salesRep);

      nlapiLogExecution('AUDIT', 'Welcome email sent', 'Sent To : ' + customerName +
                        ' By : ' + salesRepName);
   }
}
```

Build the script

1. Create a new **SuiteScript File** and name it sdr_ue_customer.js.

Note: For the **Script Type**, choose **Blank Script** for now so you can familiarize yourself with the SS 2.0 syntax.

2. Start building the script by adding a `define` statement and the required annotations.

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

   };
});
```

3. Add an `afterSubmit` entry point to your module.

```
return {
   afterSubmit : function (context) {

   }
}
```

4. Get the current customer record from the `newRecord` property of the context object.

5. Extract the following values from your customer object:

- Internal ID of the customer record
- Internal ID of the Sales Rep
- Customer ID (Entity ID)
- Sales Rep name

6. Add the SS 2.0 email and record modules as a dependency to your module.

```
define([
   'N/record',
   'N/email'
], function (record, email) {
```

7 Send an email to the customer using the following settings:

author	Sales Rep Internal ID
recipients	Customer Internal ID
subject	Welcome to SuiteDreams
body	Welcome. We are glad for you to be a customer of SuiteDreams.

8 Load the sales rep record using the record module.

TIPS AND TRICKS

SuiteScript 2.0 uses predefined enum values to set values such as the record type. Since JavaScript doesn't really have an enum type, you can still use string values to set the type. This is not recommended though. If the type values change, your script should still work if you're using the predefined enum values.

9 Modify the employee's `notes` field to mention that the welcome email was sent to the customer.

Note: Don't forget to get the existing notes first so the values are not overwritten.

10 Save your changes to the record.

Note: The `save` method is in your customer record object, not in the record module.

11 Finally, add some audit logs to mention the sales rep who sent the email and the customer who received it.

12 Upload the file and create a script record. Use the following configuration:

Name	SuiteDreams UE Customer
ID	_sdr_ue_customer
Description	<Add a meaningful description>

Deployments (subtab)

Applies to	Customer
ID	_sdr_ue_customer

Test

DID YOU KNOW?

The Script Debugger in your NetSuite account can be used to debug server-side scripts similar to SS 1.0 scripts.

IMPORTANT

As of this writing, there's a known issue regarding the use of the `getText` method in records from the `newRecord` and `oldRecord` property. If you get the error, "Invalid API usage", you must use `getValue` to return the value set with `setValue`," just comment out your `getText` method call.

As a workaround, you can load the record using the `record.load` method instead.

13 Edit an existing record, and then **Save** to trigger the `afterSubmit` entry point.

14 Check the **Execution Log** to verify if the script was executed.

15 Open the customer and sales rep records. Confirm that the email was sent by looking for the email in the **Communication** subtab.

16 Optionally, add a condition to restrict execution to record creation. You can use the `type` property of your `afterSubmit`'s context object for this purpose.

02: Debugging Server-side Scripts

Scenario: This exercise takes a look at how debugging works in SuiteScript 2.0.

Debug Existing Scripts

Note: Debugging SuiteScript 2.0 server-side scripts is exactly the same as SuiteScript 1.0 scripts, as you will see in the following steps.

1. Open the NetSuite Script Debugger (**Customization > Scripting > Script Debugger**).
2. Click the **Debug Existing** button in the **Script Debugger** page.
3. Choose the script that you've just created, and then click the **Select and Close** button.
4. Invoke the script, and then click on the **Step Over** button to go through a few lines of your code.
5. Go to the **Local Variables** subtab. Notice that the variables are displayed the same way as in SS 1.0 scripts.
6. Complete the execution of your script.

Debug Ad-hoc Scripts

7. Go to your script and copy the entire `define` statement including the `afterSubmit` function.
8. Paste the copied code into the **Script Debugger**'s editor window.
9. Change the **API Version** option to **2.0**.
10. Replace the `define` keyword with `require`

    Note: Remember that the syntax for both `define` and `require` are almost identical.

11. Move your `afterSubmit`'s implementation outside of your function so that it's executed immediately. Your code should look something like this.

```
require([
   'N/record',
   'N/email',
], function(record, email) {
   // Implementation here
   return {
      afterSubmit : function(context) {
         // Empty
```

```
}}});
```

Note: Since the script is getting executed ad-hoc, it shouldn't be defined in an entry point function.

12 Delete the whole `return` statement including the empty `afterSubmit` function.

13 Modify the script so that it doesn't have any context specific information, such as the need for the `newRecord` property. You'll need to load the customer record since that is currently coming from the context.

Note: Ad-hoc debugging sessions do not have any record context. Context-sensitive functions and methods will not work. This behavior is the same as in SS 1.0.

Feel free to hardcode the internal id or other values for the debugging session to work.

14 Click the **Debug Script** button to start the debugging session.

15 Add a breakpoint beside the callback function or on the first line of your code within the function by clicking in the space to the right of the line number.

IMPORTANT

If you don't add a breakpoint, the debugger will execute `require` as one statement.

16 Execute to the breakpoint, and then step through your code to check that the script executes properly.

03: Error Handling and Custom Modules

Scenario: This extends the previously created User Event script by creating a custom module to handle potential errors.

Handle potentials errors

1. Go to the User Event script you previously created.
2. Wrap a `try/catch` block around the code from the `record.load` call to the end of the script.
3. Go to the `catch` block and evaluate the exception. The exception is an object, and you can use the `instanceof` operator to determine whether the exception is a native JavaScript error or a SuiteScript error. Assuming there is an exception object named `e`; the statement "`e instanceof Error`" returns true when it is a JavaScript error.

   If the error is a JavaScript error, log the following object properties:

   - name, message, stack, fileName, and lineNumber

   Note: `fileName` and `lineNumber` may not always have values.

   If the error is a SuiteScript error, log the following object properties:

   - name, message, stack, id, cause

   Note: `id` may not always have a value. `cause` may need to be wrapped in `JSON.stringify` in order to get it to show correctly in the log.
4. Confirm that your code is working by testing for the two different kinds of exceptions:
   - A JavaScript error can be thrown in various ways. One way is to change a `getValue` or `setValue` method to one that does not exist, e.g. `getValue2`. This will generate a JavaScript `TypeError`, which is `instanceof Error`.
   - A SuiteScript error is thrown if your `record.load` function fails. You can make it fail either by entering an invalid enum for the `type` property, or a non-existent internal id for the `id` property.

Create custom module

5. Create a “lib” folder inside of your SuiteCloud IDE project.

6 Create a new blank script named “sdr_error_handler.js” and put it inside of the “lib” folder.

7 Add a `define` statement and create a function named `customLog`. Place the `customLog` function inside the `define` statement's callback function. This function should accept an error object as a parameter.

IMPORTANT

Do not add any JSDoc annotations since this is a library module, not an entry point module. Adding JSDoc tags will trigger the validation asking you to add an entry point function to your script.

TIPS AND TRICKS

The syntax used to define scripts will affect how functions are called in your library. Any functions implemented before the `return` statement can be called anywhere in the module. Functions implemented within the `return` statement are local to that section of code.

```
define(function () {
   function myMethod1() {
      myMethod2(); // Works fine
   }

   function myMethod2() {
      myMethod1(); // Works fine
   }

   return {
      myMethod1 : myMethod1,
      myMethod2 : myMethod2
   }
});
```

In the previous code, we can see that calling `myMethod2` from `myMethod1` will work fine. Let's look at another example:

```
define(function () {
   function myMethod2() {
      myMethod1(); // Will not work
   }

   return {
      myMethod1 : function () {
         myMethod2(); // Works fine
      },

      myMethod2 : myMethod2
   }
});
```

In this example, we can see that calling `myMethod1` from `myMethod2` won't work because `myMethod1` is not available to the function. On the other hand, `myMethod2` can be called from `myMethod1` because that has already been defined in the module.

Unlike entry point functions where you would not call an `afterSubmit` function from a `beforeSubmit` function, calling library functions from another library function is common.

8 Go back to your User Event script and move all the code inside your `catch` block to the `customLog` function of your library script.

Call the module

9 In the User Event script, add the new library module as a dependency. Use `errorHandler` as a variable name.

DID YOU KNOW?

Custom modules are referred to by adding the path to the files in the File Cabinet. Absolute or dynamic paths can be used.

Absolute paths must contain the whole path from the root (with or without the file extension), for example: "**/**SuiteScripts/globalLib/libraryFile" or "**/**SuiteScripts/myProject/lib/libraryFile".

For dynamic paths, it depends on the location of the file within the project folder. If the file is in the same folder as your calling script, entering just the filename without the extension is fine (e.g., "libraryFile"). To refer to a file from the root of the project folder, use a period. For example, to refer to a file inside the lib folder that's in the root of the project folder, use "./lib/libraryFile".

10 Go to the `catch` block and execute the `customLog` function in your `errorHandler` module, passing in the exception.

Test

11 Save a customer record to trigger the script.

12 The exercise is complete if the error logging was made from the custom module.

04: Customer Welcome Reminder

Scenario: Before creating customer records, sales reps should take the time to call the customer first. To remind the sales rep, a pop-up reminder should appear just before the record is saved. If the sales rep finds the notification disruptive, they have the option to disable it through their preference setting.

Note: For this exercise, adding the "create" filter is optional. Instead of triggering the script only when creating new customer records, you should initially test this with existing customers. This should speed up the testing of your script by allowing you to use existing records.

Create a new Client-Side script

1. Create a new Client-Side script in the SuiteCloud IDE, naming it "sdr_cs_customer.js". Do not click **Finish**.
2. Click the **Add/Remove Module** button to add your dependencies.
3. Choose **N/runtime** from the list of **Available Modules**, and then click the **>** button.

   Note: We'll be using this module to get the value of a script parameter that controls whether a pop-up reminder message is displayed.

4. Click **OK**, and then **Finish** to save your script.

Build the script

Note: This section of the exercise assumes that you have a script parameter created. You'll create the script record along with the script parameter in the following section.

5. Using the runtime module's `getCurrentScript` method, get the object that references the currently running script.
6. From the script object, get the script parameter value with the assumed name: "custscript_sdr_display_notification". This will determine if the notification reminder is displayed or not. Also, assume the script parameter value will return a boolean.

7 Use a JavaScript **confirm** statement to pop up the notification reminder message. Wrap this in a condition to filter execution based on the script parameter value. See the next step for further details.

```
var myVar = confirm('message');
```

8 Continue the code with the following:

If the value of the script parameter is true,

- Get the current record from your context object
- Get the customer's name from the current record
- Using the `confirm` statement, display the message **"**Please call <customer name> to welcome them as a customer before saving the record. Click Cancel to go back to the record."
- Return the `confirm` statement value from the function

> **Note**: A boolean must be returned from a `saveRecord` function just as in SuiteScript 1.0.

Create the script record

9 Upload the script to the file cabinet and create the script record. Use the following configuration

Name	SuiteDreams CS Customer
ID	_sdr_cs_customer
Description	<Add a meaningful description>

10 Deploy the script to the customer record, and then save.

11 Go back to the currently saved script record and go to the **Parameters** subtab.

12 Click the **New Parameter** button, and create a script parameter with the following settings:

Label	Display Call Notification
ID	_sdr_display_notification
Type	Check Box
Preference	User

Note: Do not save the script parameter/field yet.

13 Go to the **Validation & Defaulting** subtab and enable the default checked field.

14 Save the script parameter.

CAUTION

Make sure that the ID you used when you created the script parameter is the same as the ID in your code.

Test

15 Edit an existing customer record, and then save. Make sure that the message appears.

16 Disable the notification by going to the **Custom Preferences** subtab at **Home > Set Preferences**.

17 Check that the notification was properly disabled.

05: Debugging Client-side SuiteScript 2.0 Scripts

Scenario: Debugging client-side scripts in SuiteScript 2.0 is different from SuiteScript 1.0. It's important that you are familiar with the new process. Two different ways to debug are covered in this exercise.

Using the debugger statement

1. Open the script that you've previously created.
2. Inside your `saveRecord` function, **add** `debugger;` at the start of the function. Upload your changes.

DID YOU KNOW?

Adding a **debugger** statement to your Client-Side script is similar to adding a breakpoint. This stops the execution of the script at the point where the **debugger** statement is executed.

3. Edit an existing customer record and open your browser's debugger/developer tool.

IMPORTANT

Client-Side scripts are debugged locally using your browser's debugger/developer tool. The common keyboard shortcut for opening the debugger is <F12> (on Windows). While this is common, your browser might have assigned a different keyboard shortcut for it.

4. Save the customer record to trigger the debugging session.

Note: The browser should stop the execution of your script at the point where you've added the `debugger` statement.

DID YOU KNOW?

SuiteScript 2.0 entry point functions are rendered separately during execution. This is why the debugger is not linking the function to a particular JavaScript file.

5. Continue the execution of your script by stepping through your code until it finishes.

Loading the entry point modules

Note: This section shows another way of debugging client-side SuiteScript 2.0 scripts. It allows you to add a breakpoint to an already running script without having to add a `debugger` statement.

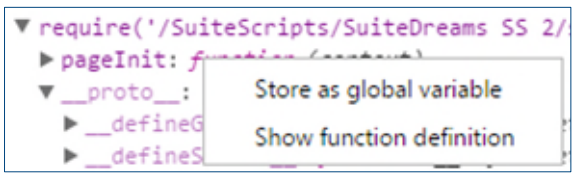
6. Go back to your script and remove the `debugger` statement.
7. Save it and upload your changes.
8. Go to your customer page and refresh it to clear cache.
9. In your browser's debugger, go to the **Sources** tab (or **Script** tab, depending on your browser).
10. In the **Watch** pane, click the **+** or add button and load the script using `require`. Use the following format:

    ```
    require('<file cabinet path>/<filename without extension>')
    ```

    For example:

    ```
    require('/SuiteScipts/SS 2.0 Project/sdr_cs_customer')
    ```

11. At this point, you should be able to inspect the module object. If it did not load, try refreshing the watch list or the customer form.
12. Right-click on the function you want to inspect and click on **Show function definition**.



IMPORTANT

You need to right-click on the part where it says “function”. Clicking on the name of the function, like `saveRecord`, will give you a different context menu.

13. Add a breakpoint by clicking on the line number of the first statement in your `saveRecord` function.
14. Click the **Save** button to trigger your script.
15. Step through your code until the execution of your script completes.

IMPORTANT

Because of the nature of this troubleshooting process, it will not work on troubleshooting `pageInit` functions.

06: Custom UI Messages (Optional)

Scenario: Replace the current notification message with a less intrusive message when the page is loaded.

Modify Script

1. Open your client-side customer script, **sdr_cs_customer.js**.
2. Include the `N/ui/message` module in the script, and then go to the `pageInit` function
3. Create a message object using the `N/ui/message` module's `create` method. Use the following configuration:

type	message.Type.WARNING
title	Customer contact reminder
message	Please call the customer to welcome them to the company.

4. To display the message, use the `show` method on your created message object. Display the message for 10 seconds.

   Note: The duration is timed in milliseconds. To display the message for 5 seconds, use 5000 as a duration value.

Test

5. Open a new customer form. The exercise is complete if you see the message appear on the form.

THE PROMISE API

Module Exercises	
01	Investigating the Promise API
02	Leveraging the Promise-based Messages

01: Investigating the Promise API

Scenario: Before using the Promise API, we'll be investigating the performance when processing multiple synchronous calls to the server from the client-side. The exercise will have you compare the performance difference between promise (asynchronous) and non-promise (synchronous) calls. Multiple sales order records will be loaded upon `pageInit`, and you will be inspecting the performance using your browser's developer tools.

Create the function calls

1. Create a Client-Side script that will trigger a `pageInit` entry point. This script will be triggered on a sales order record.
2. Add the record module to your script.
3. Create two functions, one for loading records using the promise API and the other for non-promise calls. Have both functions accept a parameter for the internal id of a record.

BEST PRACTICES

Make sure to put the functions outside of the returned entry point functions so that you can call them from your `pageInit` function.

4. Go to your non-promise function and load a sales order record using the internal id passed to the function.
5. After loading the sales order, get the total amount value and log the information to the browser console.

TIPS AND TRICKS

Aside from the `alert` statement, another way of displaying information on the client-side is by using the `console.log` method.

> For this to work, you need to open the browser's developer tools and have the console open.

6 Go to your promise function and do the same thing as your non-promise function except use the promise version of `record.load`.

```
function promiseCall(id) {
   record.load.promise({
       // Set the required properties
   }).then(
       // The salesOrder value is returned by record.load.promise
       function(salesOrder){
           // Get the total field and log that in the console
       }
   );
}
```

7 Add the previously created error handling library to the script.

8 Go back to your promise function and add a `catch` method. Call the error handlers log method inside of your `catch` method.

```
function promiseCall(id) {
   record.load.promise({
       // Set the required properties
   }).then(
       // The salesOrder value is returned by record.load.promise
       function(salesOrder){
           // Get the total field and log that in the console
       }
   ).catch(
      function(ex){
          // call the error handlers log method
      }
   );
}
```

Note: The promise's `catch` method can be mistakenly identified as the `catch` block definition. If you get an error message notification for this line, it is safe to ignore it.

Prepare the script for testing

9 Go to your `pageInit` function and call your non-promise function about 15-20 times, loading different records on each call.

10 Add the same amount of calls to your promise function. Load the same records as in the calls to the non-promise function. For now, comment out the promise API calls.

11 Create a script record and deploy it to the sales order record.

Test

12 Go to a new sales order form (**Transaction > Sales > Enter Orders**).

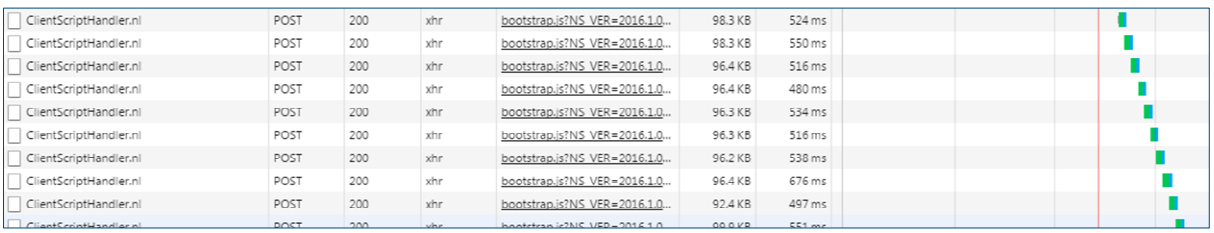
13 Open your browser's developer tools and go to the **Network** tab.

> **Note**: Monitoring network traffic in Chrome is done on the **Network** tab. If you're using a different browser, the tab might be labeled differently.

14 Click the **XHR** button (may vary across browsers) to filter the **Network** tab to XMLHttpRequests. This allows you to monitor the performance of calls to the server.

15 Refresh the sales order form and monitor the performance of the requests sent by your non-promise function.

Your network calls should look something like this:



> **Note**: Looking at this kind of request, you can see the requests are called one after the other. Multiple requests like this will take a long time to do.

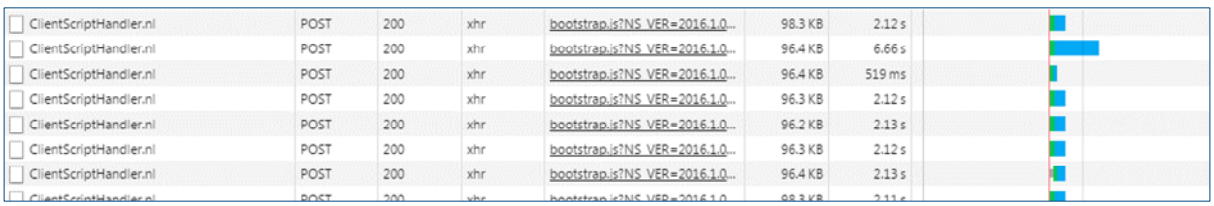
16 Take note of the total amount of time it took for the requests to complete.

Time spent on non-promise calls:

17 Comment out the non-promise calls and uncomment the promise calls.

18 Refresh the page again and notice how the network logs have changed.

The network calls should have changed to this:



Note: Comparing this to the previous request, each promise call runs independently from the main thread. This allows the script to make multiple server calls without waiting for the previous server call to finish; making this a more efficient approach.

19 Take note of the total amount of time it took for the promise requests to complete:

Time spent on promise calls:

BEST PRACTICES

Promise calls are extremely effective when used properly. Make sure to think about processing multiple threads at the same to get the most out of promises.

02: Leveraging the Promise-based Messages

Scenario: SuiteScript 2.0 includes new promise-based dialog boxes that developers can use to display messages. This has the advantage of displaying messages using the same NetSuite look and feel as the rest of the application. Additionally, since it's promise-based, we can have the script continue executing in the background while the user is reading the message, making the script appear to execute faster.

This exercise will have you call a suitelet that logs the current user when editing an existing expense report.

Inspect Logging Suitelet

1. Go to the list of scripts and open the **SuiteDreams SL Log User** script.
2. Click the **preview sdr_sl_log_user.js** link to view the script's source code.
3. Notice that the script simply creates a log that an expense report is edited.
4. Close the script's source code, but keep the script record open for now.

Build the script

5. Create a Client-Side script for the expense report record. Include the `N/https`, `N/ui/dialog`, and `N/url` module dependencies in the script.
6. Go to the `pageInit` function definition.
7. Add a condition that the script will continue only when the user is editing an existing record.

   Note: You can use the `mode` property of the context object to check if the record is being created or edited.

8. Get the expense report record from the context.
9. Extract the transaction id from the expense report object. This will be passed to the suitelet later.
10. Using the `alert` method of the `dialog` module, display the following:

title	Edit log notification
message	Please note that the user information is logged when editing an expense report.

11 Go back to the suitelet's script record and copy the script and deployment ids.

Script ID	
Deployment ID	

12 Using the information from the previous step, get the suitelet url using the **url** module's `resolveScript` method. The method takes `scriptId` and `deploymentId` parameters.

13 Call the suitelet using the `post` method of the `https` module. Assign the suitelet url to the **url** parameter. Pass the transaction id to the request using the `body` parameter.

Note: The `body` parameter takes an object. The key/value pairs are extracted within the suitelet as request parameters. Reopen the suitelet code to determine the name of the object key for transaction id.

14 Upload the Client-Side script to the File Cabinet and create a script record for it.

Test

15 Edit an existing expense report to trigger the script. The dialog box will display if the script properly triggers.

16 Without clicking **OK** or closing the dialog box, go back to the suitelet's script record.

17 Go to the **Execution Log** subtab, and you should see the log entry generated by the suitelet if the script has been triggered successfully.

TIPS AND TRICKS

In a standard JavaScript `alert` statement, the rest of the script will execute only after the user clicks on the **OK** button. Since we're using a promise-based dialog box, the `alert` will run on a separate thread, allowing the rest of the script to continue executing.

This is a very good strategy to use on slow running client scripts. While the user is reading the message, the client script can continue processing in the background.

HANDLING BIG DATA WITH MAP/REDUCE

Module Exercises	
01	Determine payment amounts per customer
02	Calling Map/Reduce scripts

01: Determine payment amounts per customer

Scenario: As part of the reporting process, the company wants to get a report on all deposited and undeposited funds per customer.

> **Note**: To keep the exercise simple, we'll be logging the values instead of creating an actual report.

Create a Payment Search

1. Go to the NetSuite UI and create a **Transaction** search (**List > Search > Saved Searches > New**).
2. Use the following settings for the search:

Search Title	Customer Payments
ID	_sdr_payments

Filter	Description
Type	is Payment
Main Line	is true

Columns : Field
Name
Status
Amount

Note: We're creating a saved search instead of a scripted search in order to use the Object Reference format for the `getInputData` stage of `Map/Reduce`.

3. **Save & Run** the search to check if you're getting the right results.

Create the script

4. Create a Map/Reduce script and name it **sdr_mr_payment_report.js**. Include the `N/search` module in the script.
5. In the `getInputData` function, `return` an Object Reference by pointing to the saved search you created above.

Note: An Object Reference is simply a payload object that has `type` and `id`. For example:

```
return {
      type : 'search',
      id   : 1234   // use internal id
};
```

6. Go to the `map` function of your script.
7. Extract the search result from the `map` stage's context object. This is stored in the `value` property, i.e. `value` represents a `search.Result` object from the result set returned by the search that is specified in the `getInputData` function. `value` has been automatically converted to a JSON string representation of the search result.

Note: The `map` stage processes a single search result value per invocation. You need not loop through the results as you would in other script types.

8. Log the **value** property of the context object to inspect the JSON string.

Initial Test

9. Upload the script to the File Cabinet.
10. Create a script record and click **Save and Deploy** to create a deployment record.
11. On the Script Deployment page, specify an ID, and then **Save** the record.

Note: Feel free to initially modify the saved search to get fewer results. This will make the execution faster.

12. Edit the deployment record again and execute the Map/Reduce script.

13 Once the execution is complete, go to the **Execution Log** and take note of the JSON structure.

TIPS AND TRICKS

There are several online JSON editors/formatters you can use to make it easy to look at the structure of a JSON string.

Total Customer Payments

14 Go back to the script and convert the JSON string to a JavaScript object using `JSON.parse`.

Note: Remember that the system will be returning a JSON string. Using the `JSON.parse` function makes the value easier to handle.

15 Write a key/value pair back into the context using the `write` method of the `context` object. The `write` method has a method signature that takes a key as the first parameter and a value as the second parameter. Use the customer name as a `key`. Set `value` to be an object containing status and amount.

16 Go to the `reduce` function of your script.

17 Extract the array of values that is associated to the key written from the `map` stage. You can get this from the `values` parameter of the `reduce`'s context object.

Note: Similar to the `map` function, the `reduce` function processes each individual key/value pair so there's no need to iterate through a set of results.

18 Create a variable that will hold the deposited and undeposited values. Initialize those values to 0.

19 Process each entry in the `values` array:

- Since you're using an object as a value for your key/value pair, you need to convert the individual array entry from a JSON string to a JavaScript object using `JSON.parse`.
- Total up deposited and undeposited amounts based upon the payment status (Deposited versus Not Deposited).
- Log both the customer name and the combined totals for each customer.

Display summary statistics

20 Extract the following values from the `summary` object in the `summarize` stage:

- Number of Queues used (`concurrency`)
- Number of Yields done (`yields`)

21 Add some error handling code to handle errors from all the previous stages.

> **Note**: It's best practice to always add the error handling code in all map/reduce scripts. Also, make sure to add an if statement to check for the error objects before extracting it from the summary object.

Test

22 Go back to the script deployment and execute the script again.

23 Click the **Details** link to view the status of execution of all stages.

24 Go back to the script or deployment record's **Execution Log** subtab. The exercise is complete if the invoice data is properly logged.

02: Calling Map/Reduce scripts (Optional)

Scenario: To automate the reporting process, the Map/Reduce script needs to automatically execute after saving a customer record. The Map/Reduce script should be triggered from your previously created User Event script.

Modify the User Event script

1. Edit the **sdr_ue_customer.js** script.
2. Add the `N/task` module to your script.
3. Create a `task` object to execute your Map/Reduce script. Details are in the following steps.
4. Configure the Map/Reduce's Script ID and Script Deployment ID as property values on the `task` object.
5. Set up a property on the `task` object that identifies the type of task, which is that a Map/Reduce script is being called.
6. Go to the Map/Reduce's script record and create a script parameter to hold the internal id of the customer. Take note of the script parameter id.

   Script Parameter ID:
7. In your User Event script, pass the customer id as a parameter in the `task` object. The parameter payload is a key/value pair with the script parameter ID as the key.

IMPORTANT

Make sure to use the script parameter ID exactly or it will not work.

8. Use the `task` object's `submit` method to execute the Map/Reduce script.
9. Upload your changes to the File Cabinet.

Modify the Map/Reduce script

10 Load the `N/runtime` module into your Map/Reduce script.

11 In the `getInputData` function:

- Using the `runtime`'s `getCurrentScript` method, load the Map/Reduce's `Script` object and get the value of the script parameter.
- Add the customer id from the script parameter to the search, so only payments from that particular customer will be processed. Previously, you used the Object Reference approach toward identifying the search. This approach can no longer be used because it does not except a dynamic filter. You have two options that are centered around using a `search.Search` object:
  - Load the existing saved search, and then add a dynamic filter.
  - Create a fully coded search. The saved search is not used, so all existing filters will need to be added inside of your script.

12 Upload your changes to the File Cabinet.

Test

13 Test by saving a customer record to trigger the script.

14 The exercise is complete if the payment information was logged for the saved customer.

USING 3RD PARTY APIS

Module Exercises	
01	Automatically set the due date on Task records
02	Display a chart of a Sales Rep's Monthly Sales

01: Automatically set the due date on Task records

Scenario: When creating new task records, users want to automatically set the due date to 30 days after the start date.

Prepare the script

1. Create a new Client-Side script file and name it **sdr_cs_task.js**.
2. Between the JSDoc tags and the `define` statement, add a `require` statement to load the Moment JS library.

   Note: Moment JS is a non-AMD library so you need to use the `require` statement. The syntax is:

   ```
   require.config({
      paths : {
         <moduleName> : <jsFilePath(string)>
      }
   });
   ```

   The library file is already preloaded into the **/SuiteScripts - Globals/lib** folder.
3. In your `define` statement, call the module you configured in your `require` statement.

   Note: Make sure that you use the same module name as in the **paths** configuration.

Implement the Requirement (initially without use of Moment JS)

4. In your `pageInit` function, add a line to get the current `task` record.
5. Add a condition so that the rest of the script will only trigger when the user is creating a new record.
6. Get the START DATE field value. This returns a JavaScript `Date` object.

7 Add 30 days to the START DATE and set the result into the DUE DATE field. You can add days to a JavaScript Date object using its `setDate` method. Following is a code snippet of the approach:

```
var dateValue1 = record.getValue({
    fieldId: 'dateValue1'
});
var dateValue2 = new Date();
dateValue2.setDate(dateValue1.getDate() + 30);
record.setValue({
    fieldId: 'datevalue2',
    value: dateValue2
});
```

8 Upload the file to the File Cabinet and create a new Script record. Deploy it to the task record.

Initial Testing

9 Create a new task record (**Activities > Scheduling > Tasks > New**) and check if the values were properly set. To troubleshoot, you can log to the browser console and/or the script execution log.

Modify Script (use functionality from Moment JS)

10 Go back to the script and change the Due Date calculation to use the `add` method from Moment JS.

Note: The Moment JS `add` method can be used to add any number of date-related values to a JavaScript `Date` object. The syntax for this is:

```
var newDate = moment(<baseDate>).add(<number>,
              <unitString>).toDate();
```

For example:

```
var newDate = moment(startDate).add(3, 'months').toDate();
```

IMPORTANT

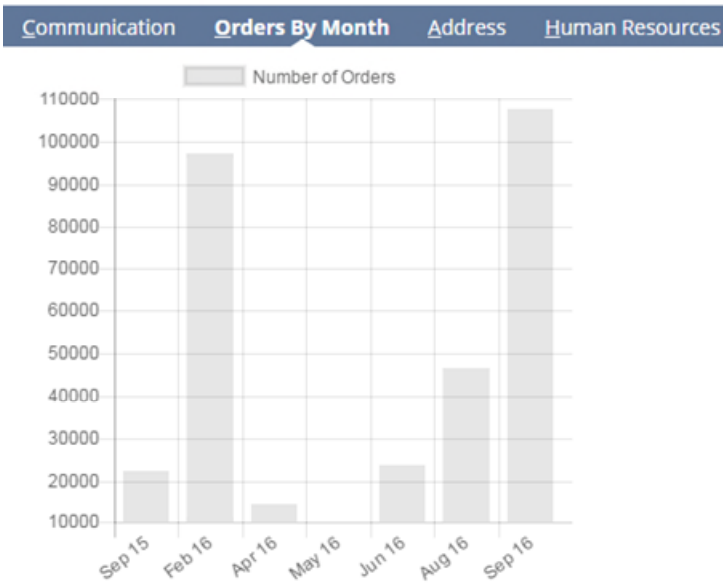
The Moment JS `add` method returns a Moment JS wrapper object, so make sure you use its `toDate` method to convert it to a JavaScript `Date`. Otherwise, the system will not recognize it as a valid `Date` object.

Final Testing

11 Go back to a new Task record and verify if the script is working. This concludes the exercise.

02: Display a chart of a Sales Rep's Monthly Sales

Scenario: To track a sales rep's performance, SuiteDreams wants to add a bar chart on the employee record when an existing employee is opened. Two third party JavaScript libraries are incorporated into the solution. Here is an example of how the bar chart will look by the end of this exercise (the specific data points will vary based on the employee record you use, and the position of the new subtab may vary too):



Overview of the business requirements:

- The bar chart shows related sales order totals, broken down by month.
- The bar chart is not editable, but should display when users view and edit employee records. You'll want to make sure the chart doesn't display in other instances, for example, when opening a new employee form.
- The bar chart should only display to Sales Reps, not all employees.
- The bar chart should not be rendered when triggered from outside the user interface, e.g. when triggered from SuiteTalk

Overview of the technical components:

- A User Event script deployed to an employee record coordinates the overall process:
  - Execute a search of sales orders for an employee being viewed or edited
  - Combine the search results with an html template (stored in the file cabinet), the result of which is placed in an Inline HTML field on a custom tab

- The bar chart is generated in the browser by using the Chart JavaScript library.
- The bar chart is generated into the html5 **canvas** element.
- The charting logic is placed inside of an Inline HTML field. The field content is a combination of html (e.g. the **canvas** element) and client-side JavaScript interacting with the Chart library upon page load.
- The client-side JavaScript in the Inline HTML field contains sales order data. Since the sales order data varies from employee to employee, part of the Inline HTML field needs to be dynamically generated. One option is to dynamically generate the content by manually concatenating strings of html and client-side JavaScript inside of a User Event script. Another option is to use a templating engine. The latter is chosen for this exercise. The templating engine used in this exercise is the Handlebars JavaScript library.
- The template is stored in the file cabinet, loaded into a User Event script, compiled using Handlebars, with the output placed into an Inline HTML field.

All components are pre-built with exception of the User Event script:

- The Chart and Handlebars JavaScript libraries are preloaded in the File Cabinet (**/SuiteScripts – Globals/lib**)
- The chart template is in the File Cabinet (**/SuiteScripts – Globals/html**)
- A summary saved search named “Order Summary (chart data)” returns sales orders for all sales reps, grouped by month.

Create the Script

1. Create a User Event script with the `N/runtime`, `N/search`, `N/ui/serverWidget`, and `N/file` modules loaded. Name it “sdr_ue_employee_chart.js”.
2. Using the `require.config` method, configure the Handlebars JS library.
3. Add Handlebars to the `define` statement.

Initial Testing

4. In the `beforeLoad` function, log the version of the Handlebars library.

Note: This is done in order to check if the library was properly loaded. Doing so helps check all the "moving parts" of the script for easier debugging.

The version information can be extracted from the library's **VERSION** property, e.g. **handlebars.VERSION**.

5. Upload the script to the File Cabinet and create a Script record. Deploy the script to the employee record.
6. Open an employee record and verify that a value for the version is properly logged, and then continue to the next step.

Restrict Generation of Bar Chart

7. Get the employee record from the context object.
8. Add a condition so the script continues only when the user is a sales rep:
   - Option #1: Inspect the Sales Rep checkbox on the employee record (**Human Resources** subtab).
   - Option #2: Edit the “Order Summary (chart data)” Saved Search. Add a filter that joins to the sales rep's record, evaluating the Sales Rep checkbox.
9. Add a condition so the script continues only when the user is viewing or editing a record.
10. Add a condition so the script continues only when the record is being loaded from the user interface context

Hint: Use a method in the `N/runtime` module.

Prepare the Order Data

11. Go to the NetSuite UI and edit the “Order Summary (chart data)” Saved Search.
12. Inspect the search **Criteria**.

Note: This search will look for all sales orders in the system and will return only mainline information. The search will be modified later so that it's filtered based on the employee record.

13. Inspect the search columns (**Results**).

Note: There are four pieces of information returned in the search: the Sales Rep, transaction date (displayed in YYMM format), transaction date (displayed in Mon YY format), and the sum of the amount. The 3rd column, displaying month and year, will be used as labels in the bar graph. The 4th column, displaying the amount, will be used as the data points on the bar graph.

14 Take note of the Saved Search id.

ID:

15 Go back to the script and load the Saved Search using the `search.load` method.

16 Create a new search filter using the `search.createFilter` method. The search results need to be filtered so that the search results will be limited to information about the sales of the currently loaded employee.

17 Add the new search filter to the `filters` property of the search object that was returned when loading the Saved Search via `search.load`. `filters` is an array, and you can use `array.push` to add the new filter to it.

18 Execute the search and process the results using the following pseudocode:

- Use the `Search.run` method to execute the search and either the `ResultSet.each` or `ResultSet.getRange` methods to extract the results. Loop through the results and extract the 3rd (month and year) and 4th (amount) search columns:
  - Store all the 3rd search column values in an array. Use “labels” as the name of the array.
  - Store all 4th search column values in another array. Use “data” as the name of the array.

19 It's suggested that you do some more testing before continuing. Log the `labels` and `data` arrays to verify the correct set of data.

Render the chart template

20 Since the bar chart should only appear if there's sales information, add a condition to continue only if sales data is available.

Note: You can check against the length of your `labels` or `data` array. Additionally, you can add an audit log that will indicate that the chart was not

generated for a particular user. This can help distinguish if the script isn't working or if the chart was not generated as expected.

21 Load the **chart_template.html** file from the File Cabinet using the `file.load` method.

Note: The chart template file is in **SuiteScripts – Globals/html**. The chart template file is a Handlebars template. If specifying the file using the file path, do not begin with a leading forward slash.

22 Compile the content of the chart template using the Handlebars `compile(<templateContent>)` method.

Note: You need to pass the content of the chart template file to the `compile` method as a parameter. The file content can be extracted using the `File.getContents` method.

23 The compiled Handlebars template that is returned from the above step is a JavaScript function. Call this function, passing the `labels` and `data` arrays as an object payload. Store the rendered chart in a separate variable.

Note: The compiled Handlebars template accepts an object payload as a parameter. Use `labels` and `data` as the property names for the object payload. These are the names used in the chart template, so Handlebars will not be able to extract the data if other names are used.

24 It's suggested that you do some more testing before continuing. Log the rendered chart, verifying that the html content has been dynamically interspersed with information from the `labels` and `data` arrays.

Set Up the User Interface

25 Get the form object from the script's context object and add a subtab using the **Form.addTab** method. Use the following configuration:

id	custpage_subtab_orders
label	Orders By Month

Note: Store the returned subtab object in a variable, as it will be used in the next step. Also, the subtab will not appear unless there's at least one UI element in it.

TIPS AND TRICKS

By default, the subtab is added to the end of the list of subtabs. You can insert your new subtab before another subtab using the **Form.insertTab** method.

For example:

```
form.insertTab({
    // Specify variable returned from executing Form.addTab
    tab     : ordersByMonthTab,

    // Id for Address subtab; you may need to inspect the
    // generated html to find the id of a subtab
    nexttab: 'address'
});
```

26 Add an Inline HTML field to the subtab using `Form.addField`. Use the following configuration:

id	custpage_field_orders
label	Orders By Month
type	serverWidget.FieldType.INLINEHTML
container	custpage_subtab_orders

Note: The field is properly added if you see the **Orders By Month** subtab on the form, even if the subtab is empty.

27 Set the rendered chart as a value for the Orders By Month inline HTML field. This is done using the field object's `defaultValue` property.

Final Test

28 Open the employee record that you're using to log in. It should display the rendered chart data on the **Orders By Month** subtab.

Note: You can use other employees that are sales reps for this test.

29 Open an employee record that is not a sales rep such as Aubrey Pober. The chart information should not be displayed.

DID YOU KNOW?

Handlebars, Lodash, and other 3rd party JavaScript libraries can be used for templating. It's up to the developer to choose the library that best meets project requirements.