OpenAir

User Scripting

April 14, 2018

ORACLE NETSUITE

Table of Contents

Introduction

User Scripting Overview

OpenAir user scripting is one component of the OpenAir platform, allowing you to customize OpenAir to better meet the unique needs of your business. OpenAir supports Form Scripts, Scheduled Scripts, Library Scripts, and Script Parameters.

User scripts are written in the industry standard JavaScript language. OpenAir is compliant with ECMAScript 5.

To ensure the security and stability of OpenAir, constraints and checks are placed on user scripting, see Scripting Governance. User scripting is prevented from accessing DOM methods, the file system, and sockets. Access to OpenAir is made available through NSOA Functions.

Scripts are stored in a Dedicated Scripting Workspace used exclusively for scripting and can only be altered through the Scripting Center. Scripts can be edited from the integrated Scripting Studio or by an external editor. To use the Scripting Center or Scripting Studio you need to be logged in as an administrator.

Before you begin writing scripts, we recommend you review Scripting Best Practices.

Tip: For a quick reference, see the OpenAir User Scripting Reference Card.

Scripting Switches

There are three switches used to control scripting:

- **Enable user scripts to be executed by forms** — enables the Scripting Center with the **Forms** tab and enables you to create Form Scripts. This switch also enables the **Script deployment** detail report section with the Form script deployment logs report, see Reporting.
- **Enable scheduled script deployments** — enables the Scripting Center with the **Scheduled** tab and enables you to create Scheduled Scripts. This switch also enables the **Script deployment** detail report section with the Scheduled script deployment logs report, see Reporting.
- **Enable user script support for Web Service API methods** — enables you to access **NSOA.wsapi** functions.

Note: Please contact OpenAir Support to enable these features.

There is one role used to control access to scripting reports:

- There is a **View the script deployment log report** role permission to enable non-administrators to view script deployment log reports, see Reporting.

Form Scripts

Form scripts are triggered to run by Events. When you create a form script it must be associated with a specific form.

Deploying a form script consists of specifying:

- **Event** — The event to trigger the script to run, see Events.

- **Entrance function** — The function defined in the script (attached to the form) you want called, see Entrance Function.

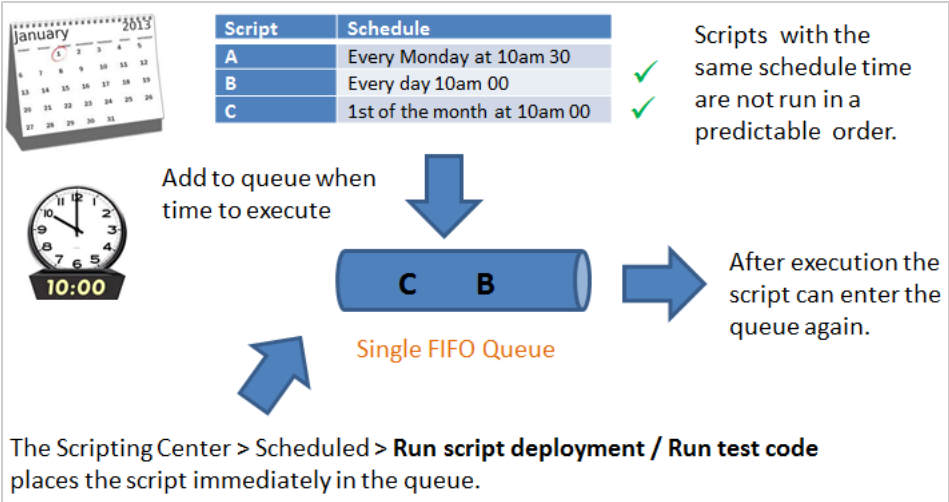
See Creating Form Scripts.

Note: Form scripts are executed within the context of the user who is logged in, see NSOA.wsapi.disableFilterSet([flag])

Scheduled Scripts

Scheduled scripts are created in a similar same way to form scripts and follow the same scripting workflow. The main differences are that scheduled scripts are not associated with a form, have higher Scripting Governance limits, and are executed according to a schedule defined when they are deployed.

Scripts are executed one at a time from a single first in first out (FIFO) queue.



See Creating Scheduled Scripts.

Tip: Two or more scripts with the same schedules times that need to run in a specific order should be merged into a single script, i.e. merge into one script with one Entrance Function calling each of the three functions in the desired order.

Note: Scheduled scripts are executed within the context of a user. You need to specify the user under which the script is to be executed when you deploy the script.

Tip: By default scheduled triggers are disabled on sandboxes. If you need to test scheduled triggers in your sandbox account, create a support case in SuiteAnswers and request the run_schedule_script trigger to be enabled for your sandbox account.

Library Scripts

Library scripts are created in a similar same way to form and scheduled scripts and follow the same scripting workflow.

Library scripts allow you to package the complexity of a scripted solution into calling scripts and supporting functions resulting in scripts that are easier to build and maintain. You can build libraries

of proven functions to reduce the cost of development and maintenance. Libraries are seamlessly integrated into the Scripting Studio to boost developer productivity.

See Creating Library Scripts.

Script Parameters

Script parameters allow developers to create scripts that can be configured without needing to change the script. Parameters are created and set in the same way as custom fields.

See Creating Parameters.

Script Terminology

Administrators can customize the terminology used in OpenAir to meet the unique needs of their company. For example, one company may use the word project to describe work to be accomplished. Another company may call it a case, job, or assignment. See **Interface: Terminology** in OpenAir Admin Guide Chapter 6 "Administration - Global Settings" for more information on customizing terminology in OpenAir.

The terminology set for an account can be directly accessed and used in scripts to create results that meet the unique needs of the company.

Scripts can be written to immediately reflect any terminology changes made by an administrator without the need to adjust the scripts in any way.

See Accessing Terminology.

Platform Solutions

You can create scripts and store them with all their dependent libraries and parameters in a single solution (XML) file. You can then apply the solution directly to another account. Solutions are stored in XML files to make them easy to read, transfer, archive, and compare.

Tip: All of the examples described in Real World Use Cases are provided as solutions, see Creating Solutions.

Getting Started

With scripting enabled the Scripting Center section is available in Administration, see Scripting Switches.

Note: This also enables the **Scripts** section in **Modify the form permissions** forms and in **Administration > Customization**.

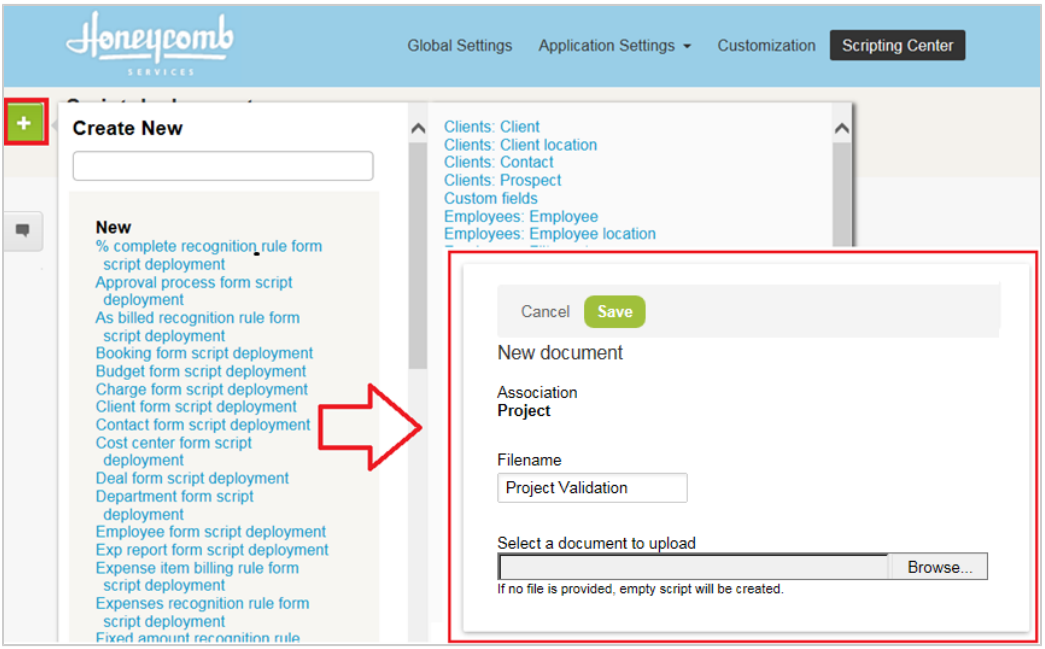
Quick Start

1. Log in as an Administrator and navigate to the **Scripting Center** section.

   Note: Make sure you have the necessary switches enabled, see Scripting Switches.

2. Create a new script from the **Create Button**.

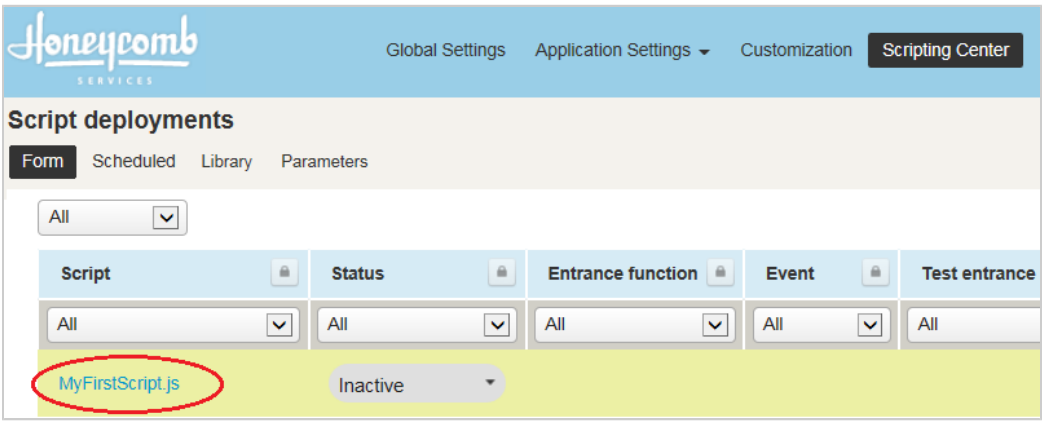
Note: This example shows how to create a form script, see also Creating Scheduled Scripts.



You need to specify a unique filename for the script in the Dedicated Scripting Workspace. You can optionally select a document that already has the script you need otherwise an empty script file will be created. If you specify a document to upload then a new script file is created from the specified file and the original file left untouched.

Note: An individual script can only be associated with one form. The same script cannot be triggered by two different forms or even form events. An individual form may trigger as many scripts as necessary.

3. Click on the **Script** link in the Scripting Center to open the script in the Scripting Studio.



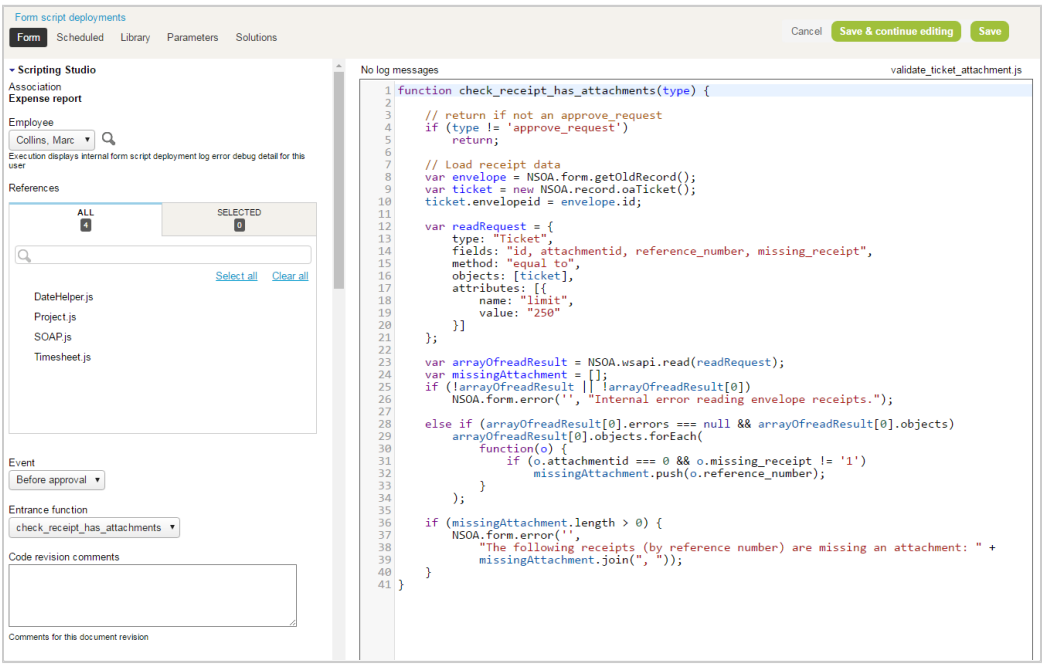
4. Type the script into the editor and then fill out the fields in the Scripting Studio Tools and Settings:
   a. Select the user that the script will run for 'In testing' state, see Testing & debugging.
   b. Select any libraries referenced by this script.

c. Select the **Event** to trigger the script, see Events.

d. Select the **Entrance function**, the name of your function to run in the editor, see Entrance Function.

e. Use the **Code revision comments** to comment the script changes made.

f. Click **SAVE**.

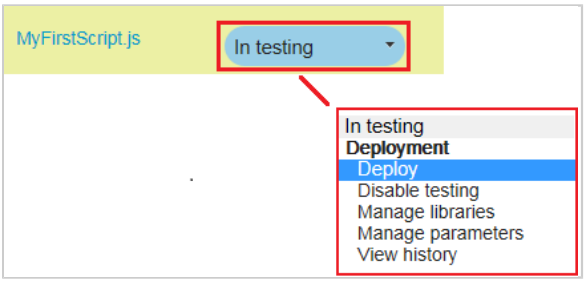


Note: The act of saving a script in the "Inactive" state will move the script to the "In testing" state, see Scripting Workflow.

5. The script will now run when the **SAVE** button is pressed on the form to which it has been deployed.

Important: Test your scripts in a sandbox account before deploying to a production account.

6. To deploy the script, select the **Deploy** option from the **Status** menu, see Scripting Workflow.



For more details see:

- Scripting Center — How to build, test, and deploy your scripts.
- Scripting Studio — Details on the OpenAir IDE.
- NSOA Functions — Details on the functions provided to access OpenAir.

- JavaScript — How to use the JavaScript language.
- Code Samples — OpenAir user script examples.
- Real World Use Cases — Larger examples provided to assist you in developing your own scripts.

Logs

Script logs are the primary means for Testing & debugging a script and for monitoring the health of a deployed script. Any errors that occur during run time are written to the script log.

Scripts can write to the log using the NSOA.meta.log(severity, message) and NSOA.meta.alert(message) functions. Detailed SOAP API request and response messages can also be logged by calling the NSOA.wsapi.enableLog([flag]) function from within a script.

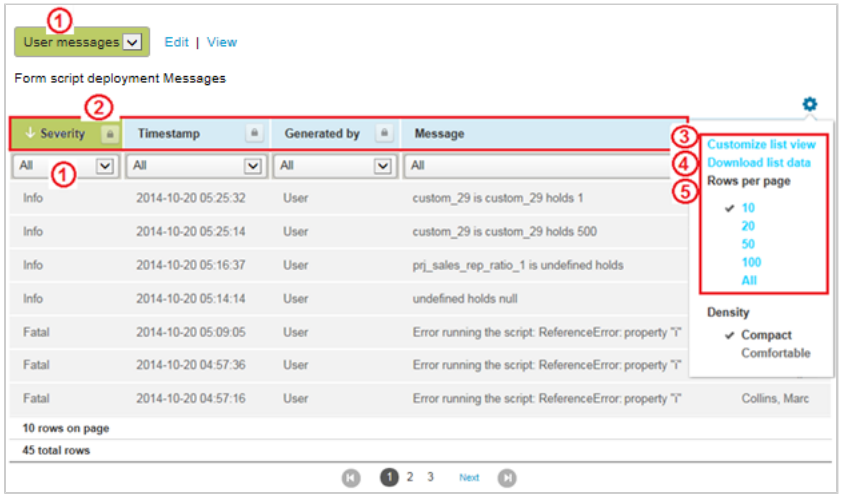
Each log entry contains the following information:

- **Severity** — The supplied severity: "Fatal", "Error", "Warning", "Info", "Debug", or "Trace".
- **Timestamp** — The time the message was logged.
- **Generated by** — For example, whether the message was generated by your script or the system.
- **Message** — The full message text.

Tip: If you load the script into an Editor you can quickly find the line number reported in the log message, see Testing Form Scripts.

View Log

You can view any log messages a script has generated via the "View Log" link from the Scripting Center and Scripting Studio, see also Reporting.



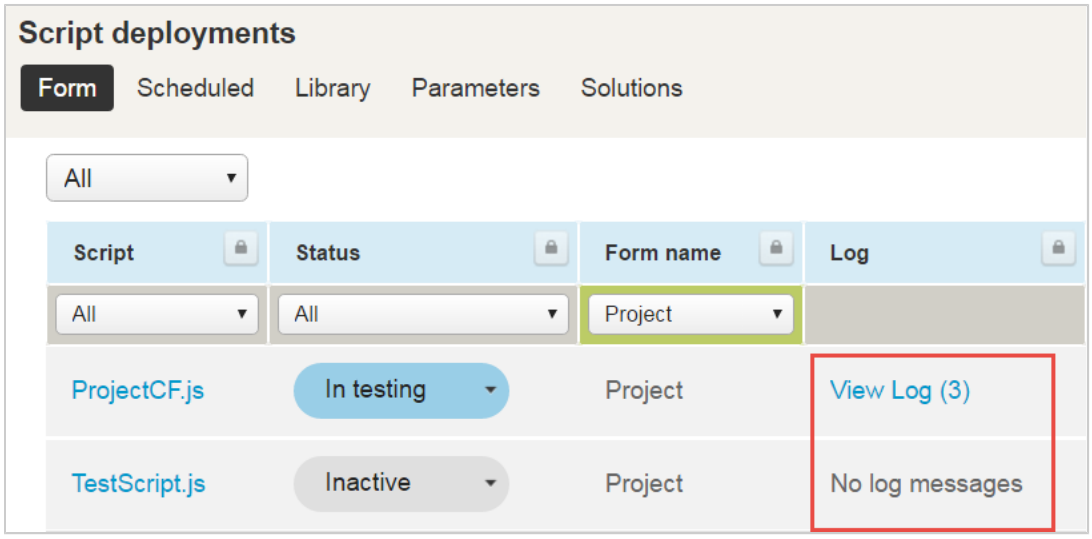
The log view has the following standard OpenAir features:

1. Filter log entries
2. Sort log entries
3. Customize list view
4. Download list data as a CSV, HTML, and PDF formatted file
5. Set the number of rows displayed on a page

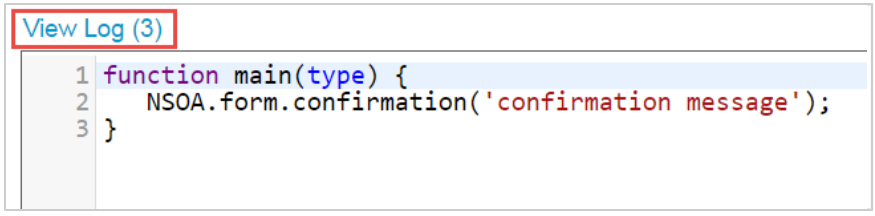
Note: Errors generated by a library are reported into the calling form or scheduled script. Libraries do not have separate logs.

Administrators can control the messages that are written to deployed scripts by setting the Log Severity for the script.

You can see how many log entries are part of a log without having to open each log with the “Display the number of logs at 'View logs' link” feature. This feature shows a count of log entries as part of the "View Log" link for Form and Scheduled Script Deployments.



The number of logs also appears next to the "View Log" link in the Scripting Editor.



To use this feature, go to the User Center > Personal Settings > Scripting Studio Options and select the "Display the number of logs at 'View logs' link" option.

Log Severity

Script logs recognize the following severities: "Fatal", "Error", "Warning", "Info", "Debug", or "Trace".

Note: If a severity is used that the log system does not recognize then it is written as an "Info" severity.

The NSOA.meta.log(severity, message) function takes two parameters, the first is severity and the second is the message to log. The NSOA.meta.alert(message) function takes a message parameter and writes "Info" severity message.

Severity is case insensitive so the following calls are all treated as the same:

```
NSOA.meta.log('debug',"message");
```

```
NSOA.meta.log('Debug',"message");
NSOA.meta.log('DEBUG',"message");
```

The following are also treated as the same:

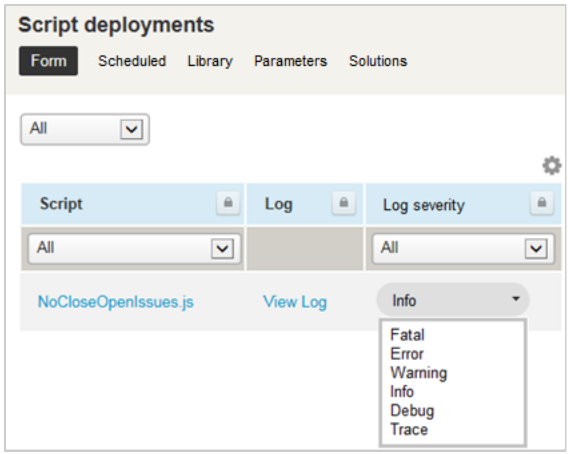
```
NSOA.meta.log('myseverity',"message");
NSOA.meta.log('Info',"message");
```

This is the same as calling:

```
NSOA.meta.alert("message");
```

If you trigger a script that is either "In testing" (or "Active revising" and you are logged in as the test user) then ALL log messages are logged.

If you trigger a script that is "Active" (or "Active revising" and you are not logged in as the test user) then the log messages written are controlled by the **Log severity** set for the script in the Scripting Center.



Non-deployed scripts log all messages but deployed scripts log messages according to the Log severity setting.

Calls to NSOA.meta.log(severity, message) with the severity parameter set to "Debug" or "Trace" do not consume units but are limited to a maximum of 1000 per script.

The default Log severity level for deployed scripts is "Error". This means that only "Error" and "Fatal" severities are written to log. In this case "Trace", "Debug", "Info", and "Warning" messages are simply ignored.

Administrators can set the **Log severity** level for deployed scripts.

Note: "Fatal" and system generated messages are ALWAYS logged! A system Info message is written to the log when the log severity is changed.

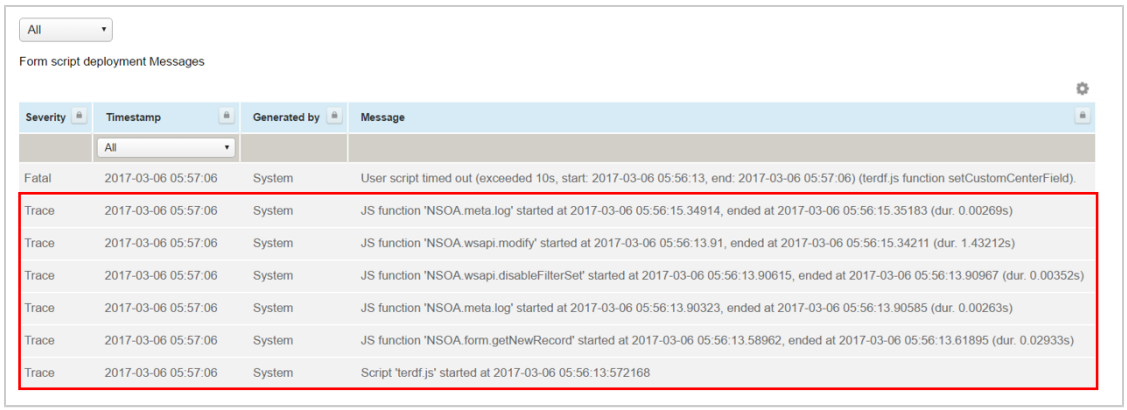
Tip: You can set the log severity to "Warning" or "Error" to save space and improve system performance for scripts that are operating correctly and generating log information that you are sure you don't need.

Tip: You can set the log severity of a deployed script to "Debug" to track down errors that only occur for a deployed script.

See Scripting Return Codes for more details.

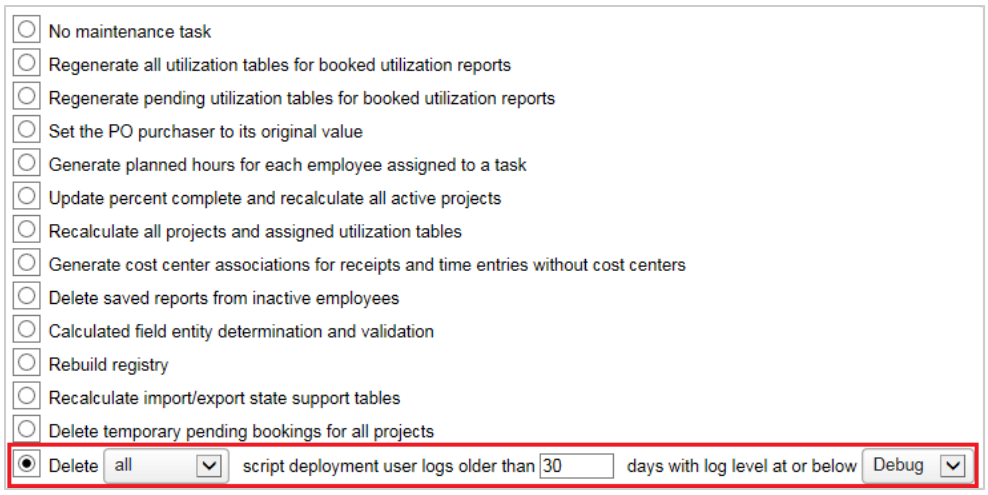
Trace Level Logs

Fatal "User script timed out" log messages are followed by "Trace" log messages which break down the time used in the script to assist you in identifying the root cause of the time out. The log messages indicate the time taken by each function call in the script.



Delete Log Entries

The delete log entries maintenance task is available to allow administrators to delete log entries that are no longer needed. This can be useful to save space and create smaller backup files.

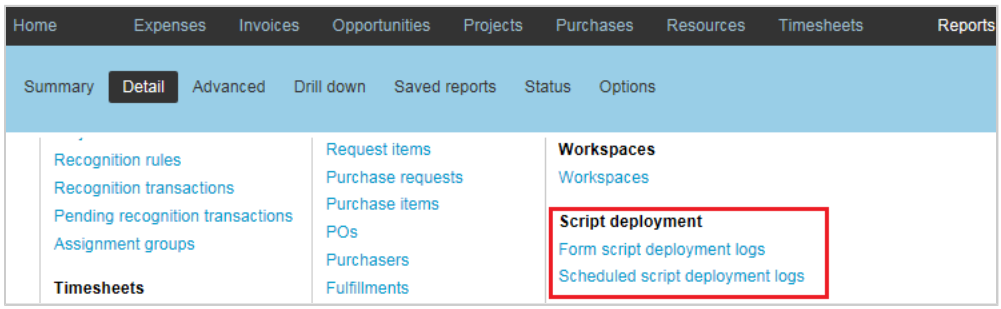


The delete logs task is available from Administration > Global Settings > Maintenance settings.

Tip: Use this maintenance task when your system is not busy and be careful not to delete log entries that you may need.

Important: You are recommended to keep at least the last 30 days of log.

Reporting



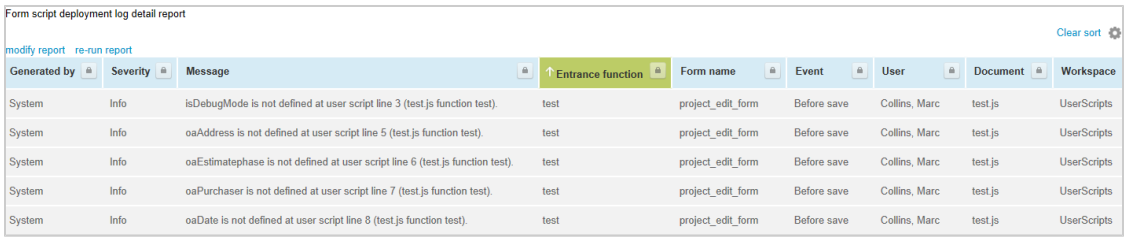
This section contains the Form script deployment logs report and the Scheduled script deployment logs report.

To view the **Form script deployment logs** detail report you need the **Enable user scripts to be executed by forms** switch enabled.

To view the **Scheduled script deployment logs** details report you need the **Enable scheduled script deployments** switch enabled.

Non-administrators can see the reports if they have been assigned the **View the script deployment log report** role permission.

Form script deployment logs



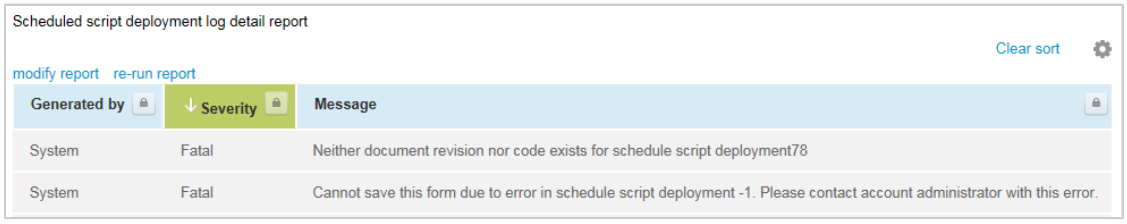
This report allows you to view all the log messages for all form script deployments. See NSOA.meta.log(severity, message) for more details.

You can also see the SOAP request and response messages if NSOA.wsapi.enableLog([flag]) is used in a script.

To view this report, you need the **Enable user scripts to be executed by forms** switch enabled.

There is a **View the script deployment log report** role permission for non-administrators to view this report.

Scheduled script deployment logs



This report allows you to view all the log messages generated by all scheduled script deployments. See NSOA.meta.log(severity, message) for more details.

To view this report, you need the **Enable scheduled script deployments** switches enabled.

There is a **View the script deployment log report** role permission for non-administrators to view this report.

Scripting Return Codes

The following return codes may appear in scheduled script or form script deployment logs.

Return Code	Description
0	OK/Success
100	Unknown error
101	Compilation error
102	Script timed out
103	Script used all units
104	Uncaught JavaScript exception
105	Uncaught Perl exception

Platform Role Permissions

As of the April 16, 2016 release, Administrators can assign Platform Roles to users to control access to critical features of the Scripting Center and Scripting Studio. You can create Platform Roles by navigating to Administration > Roles. We recommend creating the following roles:

- Script Administrator
- Script Developer
- Script QA
- Script Deploy

Roles can be assigned a number of role permissions:

- View Scripting Center — allows you to access and view the Scripting Center by navigating to Administration > Scripting Center.
- Create script — allows you to create a new script.
- Change script log level — allows you to set what types of information to log.

- View script in Scripting Studio — allows you to view a script in the Scripting Studio.
- View and modify script in Scripting Studio — allows you to view a script and make changes to it in the Scripting Studio.
- Enable script testing — allows you to move a script to “In testing” status.
- Upload script revision code — allows you to upload new code revisions after a script has been deployed.
- Disable script testing — allows you to move an “In testing” script to “Inactive” status.
- Discard script changes — allows you to discard any script changes made since the last save.
- Deploy new script — allows you to save a new script and move it to “Active” status.
- Deploy script changes — allows you to save changes to an “In testing” script and move it to “Active” status.
- Undeploy script — allows you to move an “Active” script to “In testing” status.
- Delete script — allows you to delete a script.
- Set form script “Execute As Employee” — set an employee for script deployment when running a script under another user.
- Run schedule script test code — allows you to run schedule script test code in either “In testing” or “Active: revising” states.
- Run schedule script code — allows you to run currently deployed script code.
- Cancel schedule script queued runs — allows you to cancel any previously-scheduled runs waiting for processing in the queue.
- View script parameters — allows you to view, create, and modify script parameters.
- View and modify script parameters — allows you to view, create, and modify script parameters.
- Set script parameter value — allows you to use the “Set” link for the script parameter value.
- View solutions — allows you to view solutions, but not edit them.
- View and modify solutions — allows you to view, create, and modify solutions.
- Export solution — allows you to export a solution based on a particular script deployment.
- Upload solution — allows you to upload a solution XML file.
- Apply solution — allows you to create all objects specified in a solution and create a log file.
- Delete solution — allows you to delete a solution, all of its history, and logs.

We suggest creating the following roles and assigning them these permissions:

Permissions	Script Administrator	Script Developer	Script QA	Script Deploy
View Scripting Center	✓			
Create script	✓	✓		
Change script log level	✓	✓	✓	
View script in Scripting Studio	✓	✓	✓	
View and modify script in Scripting Studio	✓	✓		
Enable script testing	✓	✓	✓	

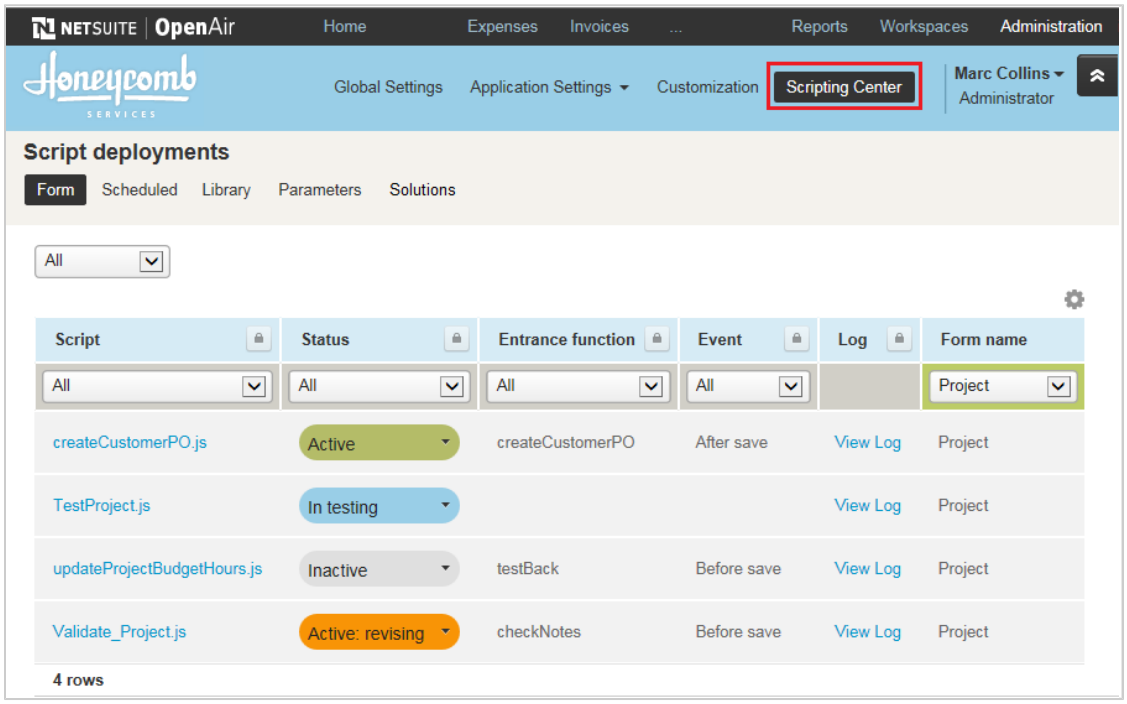
Permissions	Script Administrator	Script Developer	Script QA	Script Deploy
Upload script revision code	✓	✓		
Disable script testing	✓	✓	✓	
Discard script changes	✓	✓		
Deploy new script	✓			✓
Deploy script changes	✓			✓
Undeploy script	✓			✓
Delete script	✓	✓		
Set form script Execute As User	✓			
Run schedule script test code	✓	✓		
Run schedule script code	✓			
Cancel schedule script queued runs	✓	✓		
View script parameters	✓	✓	✓	
View and modify script parameters	✓	✓		
Set script parameter value	✓	✓		✓
View solutions	✓	✓	✓	
Create solution	✓	✓	✓	
Upload solution	✓	✓	✓	✓
Download solution	✓	✓	✓	✓
Apply solution	✓	✓	✓	✓
Delete solution	✓			

Scripting for Mobile Devices

Mobile devices support scripts which are triggered by “Before approval” and “After approval” events. Scripts triggered by “On submit,” “Before save,” or “After save” are not supported for mobile devices. For an example of a script which is compatible with mobile devices, see Ensure resource time entry matches booking planning and project worked hours.

User Scripting

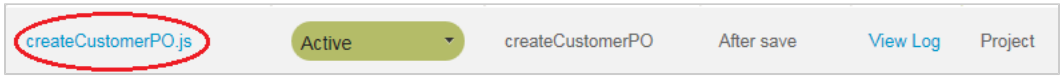
Scripting Center



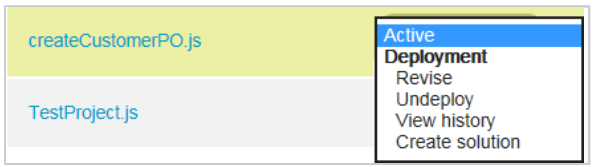
The Scripting Center is accessed from **Administration** > **Scripting Center** and gives administrators complete control over all script deployments and development activities from a central location.

The Scripting Center has five tabs:

- **Form** — See Creating Form Scripts.
- **Scheduled** — See Creating Scheduled Scripts.
- **Library** — See Creating Library Scripts.
- **Parameters** — See Creating Parameters.
- **Solutions** — See Creating Solutions.



From the Scripting Center you can launch the Scripting Studio by clicking on a script link.



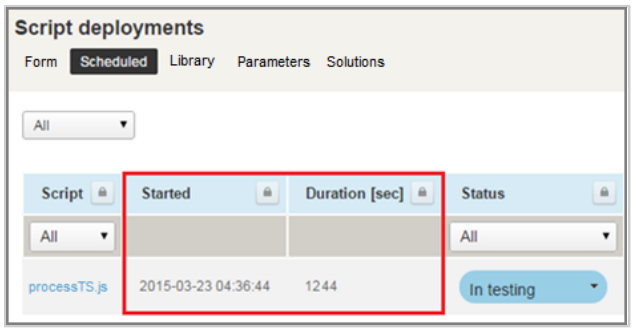
Scripts are moved through the Scripting Workflow from the **Status** menu.

Note: Customers that choose not to use the Scripting Studio in favor or another editor are still fully supported from the Scripting Center.

You can view any log messages the script has generated via the "View Log" link, see Logs.

- **Script** — This is the script to run on the event, click to edit the script in the Scripting Studio.
- **Status** — Indicates the state of the script in the Scripting Workflow.
- **Entrance Function** — This is the entrance function to call in the script, see Entrance Function.
- **Event** — This is the event that will trigger the script to run, see Events.
- **Form name** — This is the form that will trigger the script, see Creating Form Scripts.

Scheduled Queue Status

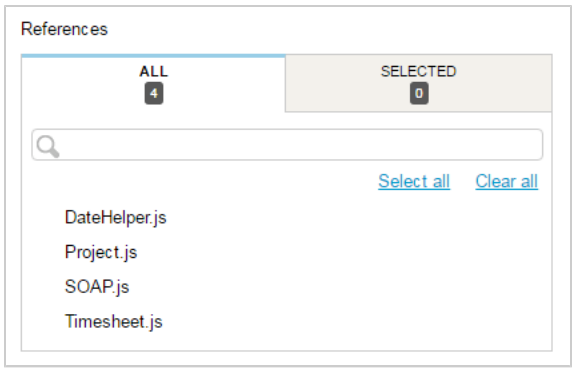


The **Started** and **Duration [sec]** columns on the Scripting Center > Scheduled tab allows administrators to monitor the processing of scheduled scripts in the queue. Refresh your screen to see the progress. The **Started** and **Duration [sec]** columns are cleared when the script completes.

Dedicated Scripting Workspace

OpenAir incorporates a dedicated scripting workspace used exclusively for scripting. The dedicated scripting workspace is hidden in OpenAir. Files in the dedicated scripting workspace can only be altered through the **Scripting Center**. This feature provides additional security for the maintenance of scripts. It is not possible to accidentally delete an active script or to create scripts with the same name. This feature also simplifies the user interface as you do not need to specify a workspace to store the script.

Manage libraries

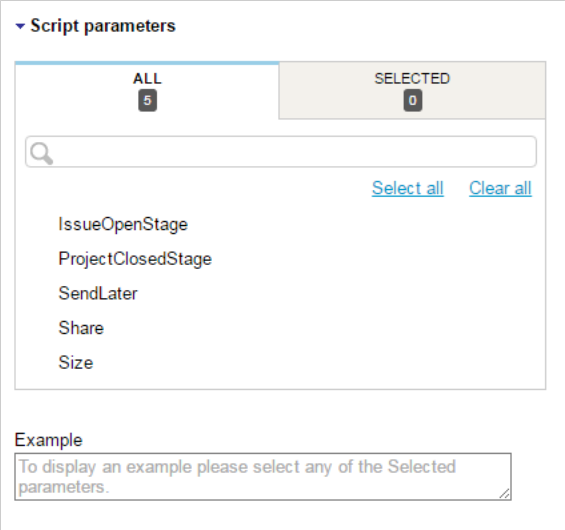


You can specify the libraries a script references by selecting **Manage libraries** from the Scripting Center **Status** menu. This performs the same function as selecting libraries in the Scripting Studio Tools and Settings and is provided for developer using an external editor.

Note: You can only manage the libraries of "In testing" and "Active: revising" scripts.

Important: You cannot select an "Inactive" library and you cannot deploy a script that is referencing a library that has not been deployed.

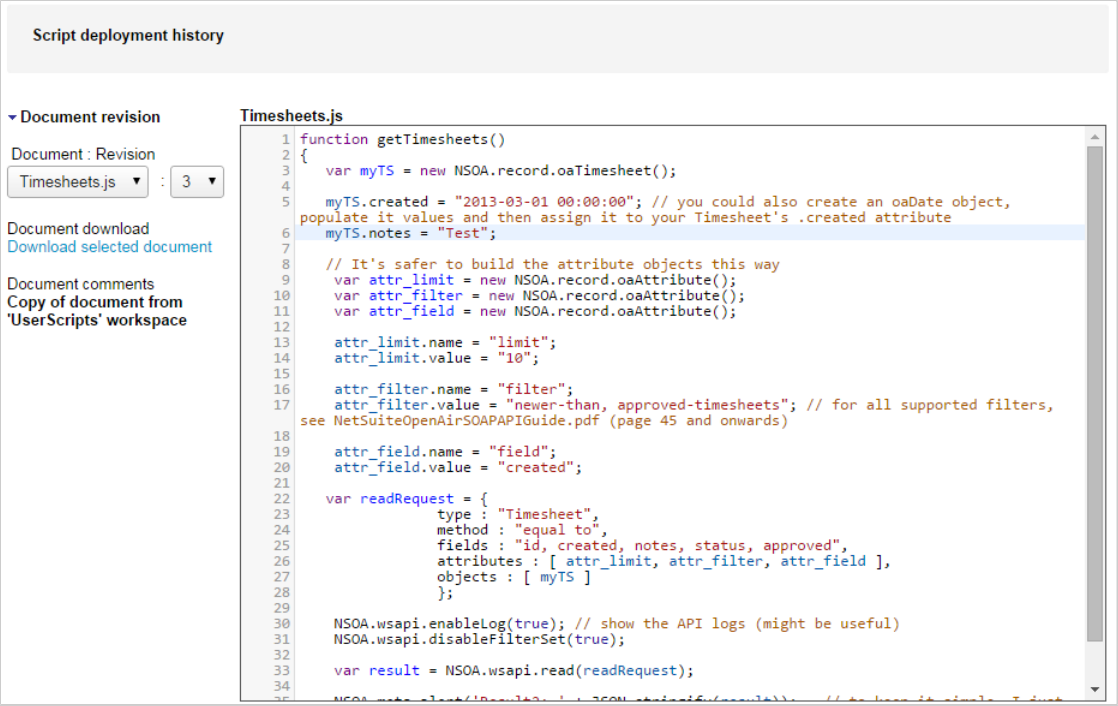
Manage parameters



You can specify the parameters a script uses by selecting **Manage parameters** from the Scripting Center **Status** menu. This performs the same function as selecting parameters in the Script parameters section of the Scripting Studio and is provided for developer using an external editor.

Note: You can only manage the parameters of ”In testing” and “Active: revising” scripts.

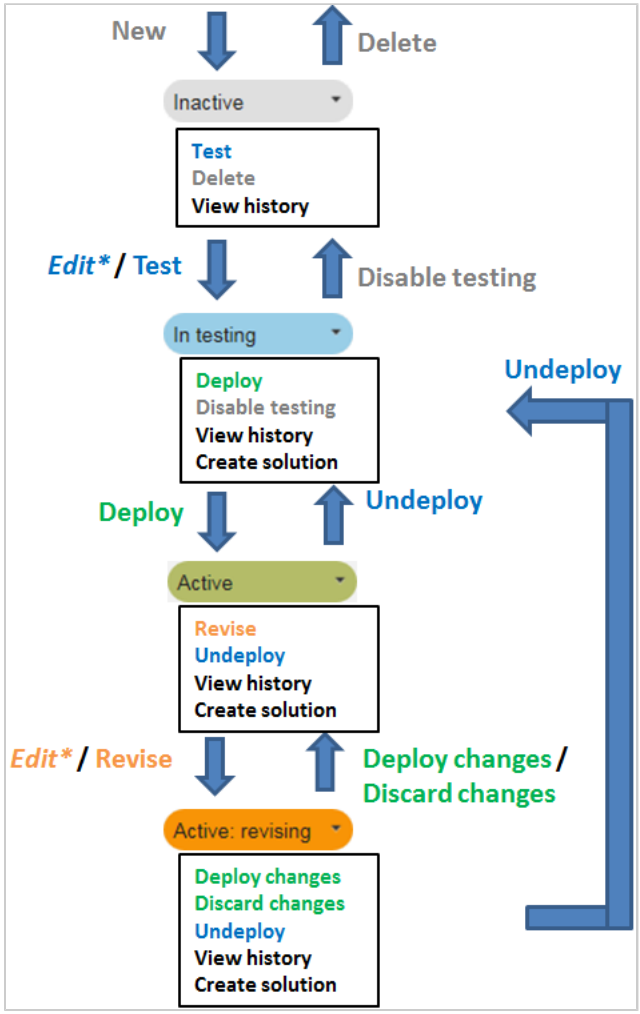
View history



The script deployment history is available by selecting **View history** from the Scripting Center **Status** menu. From this form you can browse through each revision of deployed code and download a selected document revision.

Important: A new history entry is only created when you **Deploy** a script. A new history entry is not created every time you **SAVE** your script changes.

Scripting Workflow

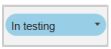
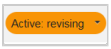


Note: * **Edit** is actioned by clicking the script link and saving from the Scripting Studio

A color coded status indicator shows the position of the script in the scripting workflow:

- **Inactive** scripts are not triggered at all.
- **In testing** scripts are only triggered by the user selected to test the code.
- **Active** scripts are triggered for all users.
- **Active: revising** scripts have separate deployed code and test code. The test code is triggered by the user selected to test the code and the deployed code is triggered by all other users.

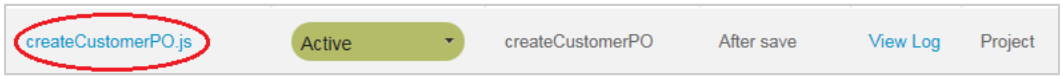
Depending on the scripts status in the workflow a list of options are available by clicking on the status.

Status	Actions
	■ **Test** — Prompts for test settings and on **SAVE** moves the script to **In testing**. ■ **Delete** — Prompts for confirmation and on **OK** deletes the script code and all associated history. ■ **View history** — see View history. ■ Click the script link to make changes in the Scripting Studio. On **SAVE** the script moves to **In testing**.
	■ **Deploy** — Prompts for confirmation and on **SAVE** moves the script to **Active**. ■ **Disable testing** — Prompts for confirmation and on **OK** moves the script to **In active**. ■ **Manage libraries** — see Manage libraries. ■ **Manage parameters** — see Manage parameters. ■ **View history** — see View history. ■ **Export solution** — see Creating Solutions. ■ Click the script link to make changes in the Scripting Studio. On **SAVE** the status in not changed.
	■ **Revise** — Prompts for a new JS file with the required content and then launches the Scripting Studio with this new content. On **SAVE** the script is moved to **Active: revising**. ■ **Undeploy** — Prompts for confirmation and on **OK** moves the script to **In testing**. ■ **View history** — see View history. ■ **Create solution** — see Creating Solutions. ■ Click the script link to make changes in the Scripting Studio. On **SAVE** the script moves to **Active: revising**.
	■ **Deploy changes** — Prompts for confirmation and on **SAVE** moves the script to **Active**. ■ **Discard changes** — Prompts for confirmation and on **OK** moves the script to **Active** ignoring any changes made. ■ **Manage libraries** — see Manage libraries. ■ **Manage parameters** — see Manage parameters. ■ **Undeploy** — Prompts for confirmation and on **OK** moves the script to **In testing**. ■ **View history** — see View history. ■ **Create solution** — see Creating Solutions. ■ Click the script link to make changes in the Scripting Studio. On **SAVE** the script moves to **Active: revising**.

Creating Form Scripts

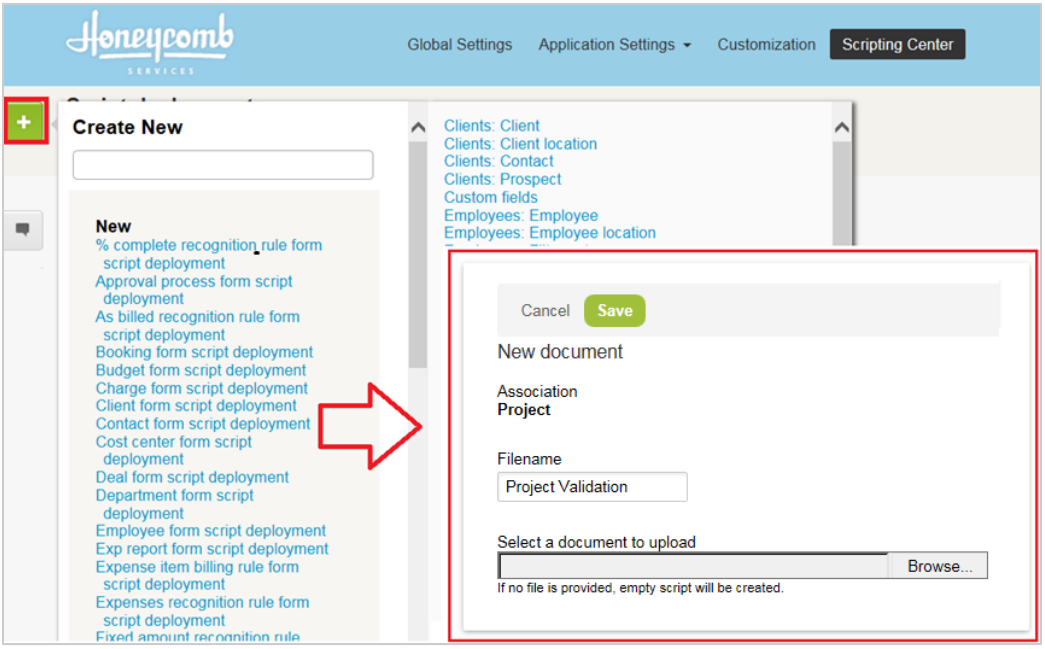
Form scripts are created from the **Create Button**. You need to specify a unique filename for the script in the dedicated scripting workspace. You can optionally select a document that already has the script you need otherwise a blank script file will be created. If you specify a document to upload then a new script file is created from the specified file and the original file left untouched.

Note: An individual script can only be associated with one form. The same script cannot be triggered by two different forms or even form events. An individual form may trigger as many scripts as necessary.



To create a form script:

1. Go to Administration > Scripting Center > Form. The list view for form scripts appears.
2. Click the **Create** button.



3. Click the type of form script you want to create under “New”. The “New document” dialog appears.
4. Type a filename for the script into the “Filename” dialog.
5. If you want to import an already written form script, click **Choose File** and select the script’s file.
6. Click **Save**. The list view for form scripts appears.
7. Click on the **Script** link in the Scripting Center to open the script in the Scripting Studio.
8. Type the script into the editor and then fill out fields in the Scripting Studio Tools and Settings:
    a. Select the user that the script will run for ‘In testing’ state, see Testing & debugging.
    b. Select any libraries referenced by this script.
    c. Select whether the script is executed **On Submit**, **Before save**, or **After save**.
    d. Select the **Entrance function**, the name of your function to run in the editor, see Entrance Function.
    e. Use the **Code revision comments** to comment the script changes made.
    f. Click **SAVE**.

> **Note:** The act of saving a script in the "Inactive" state will move the script to the "In testing" state, see Scripting Workflow.

Once a script has been created, you can edit the script by clicking on the script link, move the script through the Scripting Workflow, or view any log messages the script has generated via the "View Log" link, see Testing Form Scripts.

> **Tip: To reduce the errors in your scripts, see** Scripting Best Practices.

Scripts need to be carefully tested before being deployed to production. See Testing Form Scripts and Scripting Workflow for details.

For more details see:

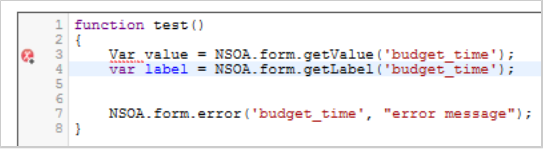
- Scripting Studio — Details on the OpenAir IDE.
- NSOA Functions — Details on the functions provided to access OpenAir.
- JavaScript — How to use the JavaScript language.
- Code Samples — OpenAir user script examples.
- Real World Use Cases — Larger examples provided to assist you in developing your own scripts.

Testing Form Scripts

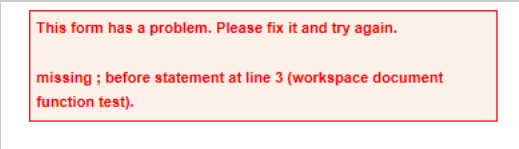
There are three types of errors you need to remove from your scripts.

- **Syntax errors** — These errors can be caught before your script is executed. Syntax errors are displayed in the Editor.

  For example:

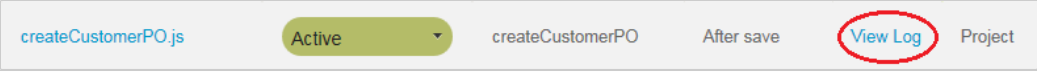


  OpenAir checks scripts for correct syntax before allowing them to be deployed. An error is displayed if you attempt to deploy a script with syntax errors.

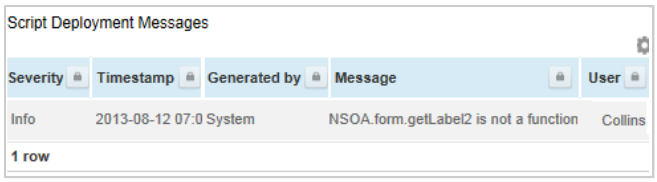


> **Note:** This error is caused because **Var** had been typed in place of **var**, JavaScript is case sensitive. See Variables for more details.

- **Runtime errors** — These errors occur during run time. OpenAir report runtime errors in the log.



  Click on the **View Log** link to see the log messages. See also Reporting.


Script Deployment Messages

Severity	Timestamp	Generated by	Message	User
Info	2013-08-12 07:0	System	NSOA.form.getLabel2 is not a function	Collins

1 row

This error was caused because the script attempted to call a method that doesn't exist, i.e., **NSOA.form.getLabel2** does not exist.

```
function test() {
    var value = NSOA.form.getValue('budget_time');
    var label = NSOA.form.getLabel2('budget_time');
}
```

In JavaScript missing methods can only be detected at runtime.

Tip: If you load the script into the Editor you can quickly find the line number reported in the log.

- **Logic errors** — These errors are the most difficult type to track down. They are not the result of a syntax or runtime error. Instead, they occur when you make a mistake in the logic that drives your script and you do not get the result you expected.

Tip: You can use the NSOA.meta.alert(message) function to log debugging information.

Important: It is strongly recommended to test scripts thoroughly in a Sandbox account before deploying to a Production account.

See also Testing & debugging.

Deploying Form Scripts

To deploy a form script:

1. Go to Administration > Scripting Center > Form. The list view for form scripts appears.
2. In the status column, click the drop-down list for the form script you want to deploy and select "Deploy". A deploy script dialog appears.
3. Add notes for the script deployment (optional).
4. Select an employee to execute the script.

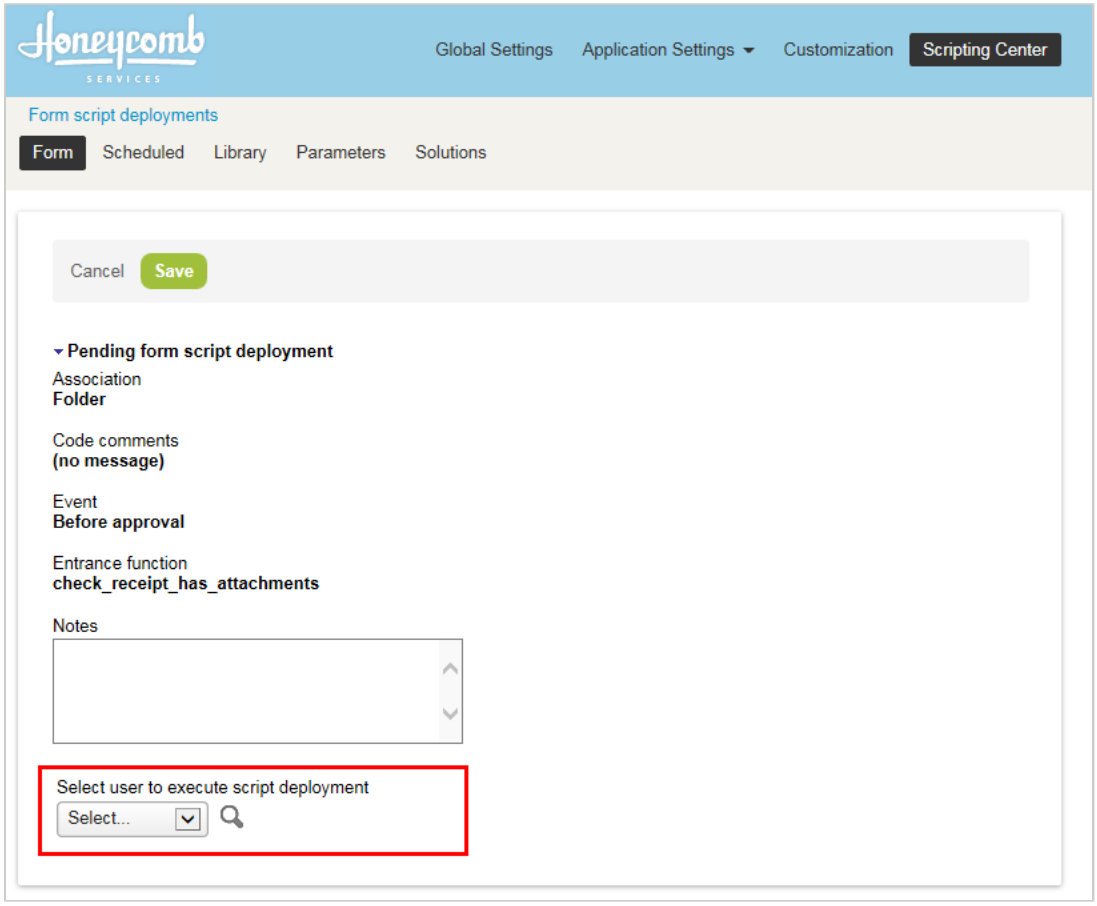
   Note: Form scripts cannot be executed as an Administrator.
5. Click **Save**. A message will confirm that the script was deployed, and the list view for the selected script type appears.

Execute as User when Deploying Form Scripts

When deploying a script, you must select a user to execute the deployment. This user acts as a proxy, and is needed when one user does not have the access permissions a script needs to run successfully.

The "Execute as User" feature is not intended as a replacement for using NSOA.wsapi.disableFilterSet([flag]).

Administrators will not appear in the “Execute as User” list. Form scripts are explicitly prevented from being deployed by Administrators.



Tip: Create a dedicated user with the minimum necessary permissions to execute the script for the “Select user to execute script deployment” feature.

Creating Scheduled Scripts

Scheduled Scripts are accessed from the **Scheduled** tab of the Scripting Center. See Scripting Switches to enable this feature.

Scheduled scripts are created in a similar same way to form scripts and follow the same Scripting Workflow. Notice that scheduled scripts have additional menu options available from the **Status** menu:

- **Run script deployment** — Prompts for confirmation and on **OK** will add a one-time schedule event to the platform script deployment job queue.
- **Cancel queued runs** — Prompts for confirmation and on **OK** will cancel any jobs queued to run for this script.

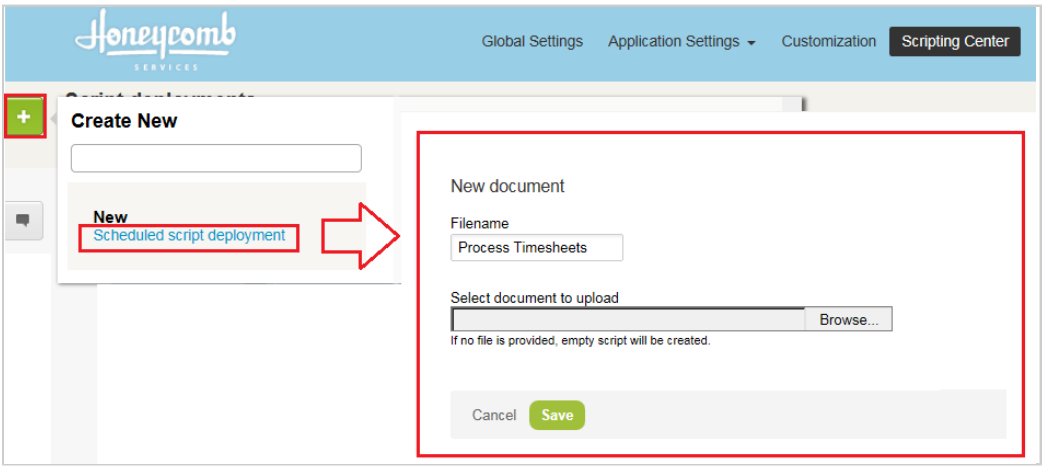
Scheduled scripts are not associated with a form and cannot access the NSOA.form functions.

To create a scheduled script:

1. Log in as an Administrator and navigate to the **Scheduled** tab on the Scripting Center.

Note: Make sure you have the necessary switches enabled, see Scripting Switches.

2. Create a new scheduled script from the **Create Button**.



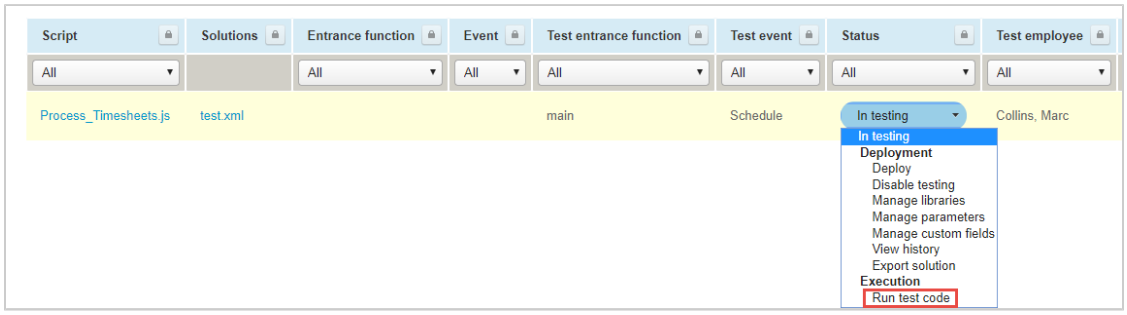
You need to specify a unique filename for the script in the Dedicated Scripting Workspace. You can optionally select a document that already has the script you need otherwise an empty script file will be created. If you specify a document to upload then a new script file is created from the specified file and the original file left untouched.

3. Click on the **Script** link in the Scripting Center to open the script in the Scripting Studio.
4. Type the script into the editor and then fill out fields in the Scripting Studio Tools and Settings:
   a. Select the user that the script will run for 'In testing' state, see Testing & debugging.
   b. Select any libraries referenced by this script.
   c. **Event** is fixed as 'Scheduled'.
   d. Select the **Entrance function**, the name of your function to run in the editor, see Entrance Function.
   e. Use the **Code revision comments** to comment the script changes made.
   f. Click **SAVE**.

Note: The act of saving a script in the "Inactive" state will move the script to the "In testing" state, see Scripting Workflow.

Testing Scheduled Scripts

Scheduled scripts can be run from the **Run test code** menu option form the **Status** menu.



Important: By default scheduled triggers are disabled on sandboxes. If you need to test scheduled triggers in your sandbox account, create a support case in SuiteAnswers and request the run_schedule_script trigger to be enabled for your sandbox account.

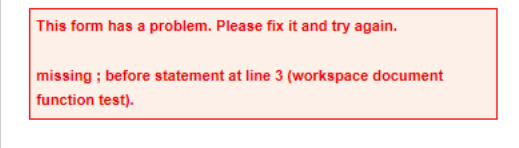
There are three types of errors you need to remove from your scripts.

- **Syntax errors** — These errors can be caught before your script is executed. Syntax errors are displayed in the Editor.

  For example:

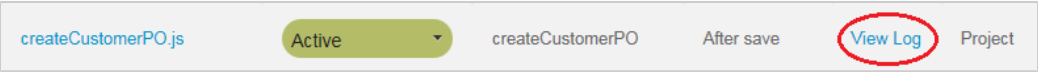
```
1  function main() {
2      // TODO Add Your Code Here
3
4      // TODO Handle Errors
5
6      // Notify The Owner
7      Var me = NSOA.wsapi.whoami();
8      var msg = {
9          to: [me.id],
10         subject: "Script completed",
11         format: "HTML",
12         body: "<b>Your script completed</b><br/>" +
13               "<hr/><i>Automatically sent by the system</i>"
14     };
15 }
```

  OpenAir checks scripts for correct syntax before allowing them to be deployed. An error is displayed if you attempt to deploy a script with syntax errors.

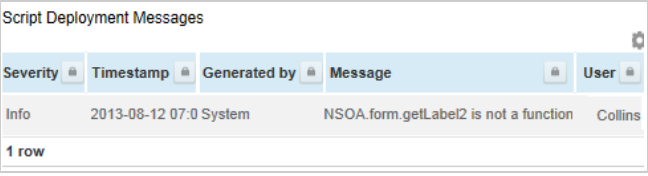


Note: This error is caused because **Var** had been typed in place of **var**, JavaScript is case sensitive. See Variables for more details.

- **Runtime errors** — These errors occur during run time. OpenAir report runtime errors in the log.



  Click on the **View Log** link to see the log messages. See also Reporting.



  This error was caused because the script attempted to call a method that doesn't exist, i.e., **NSOA.form.getLabel2** does not exist.

```
function test() {
    var value = NSOA.form.getValue('budget_time');
    var label = NSOA.form.getLabel2('budget_time');
```

```
}
```

In JavaScript missing methods can only be detected at runtime.

Tip: If you load the script into the Editor you can quickly find the line number reported in the log.

- **Logic errors** — These errors are the most difficult type to track down. They are not the result of a syntax or runtime error. Instead, they occur when you make a mistake in the logic that drives your script and you do not get the result you expected.

  Tip: You can use the NSOA.meta.alert(message) function to log debugging information.

Important: It is strongly recommended to test scripts thoroughly in a Sandbox account before deploying to a Production account.

See also Testing & debugging.

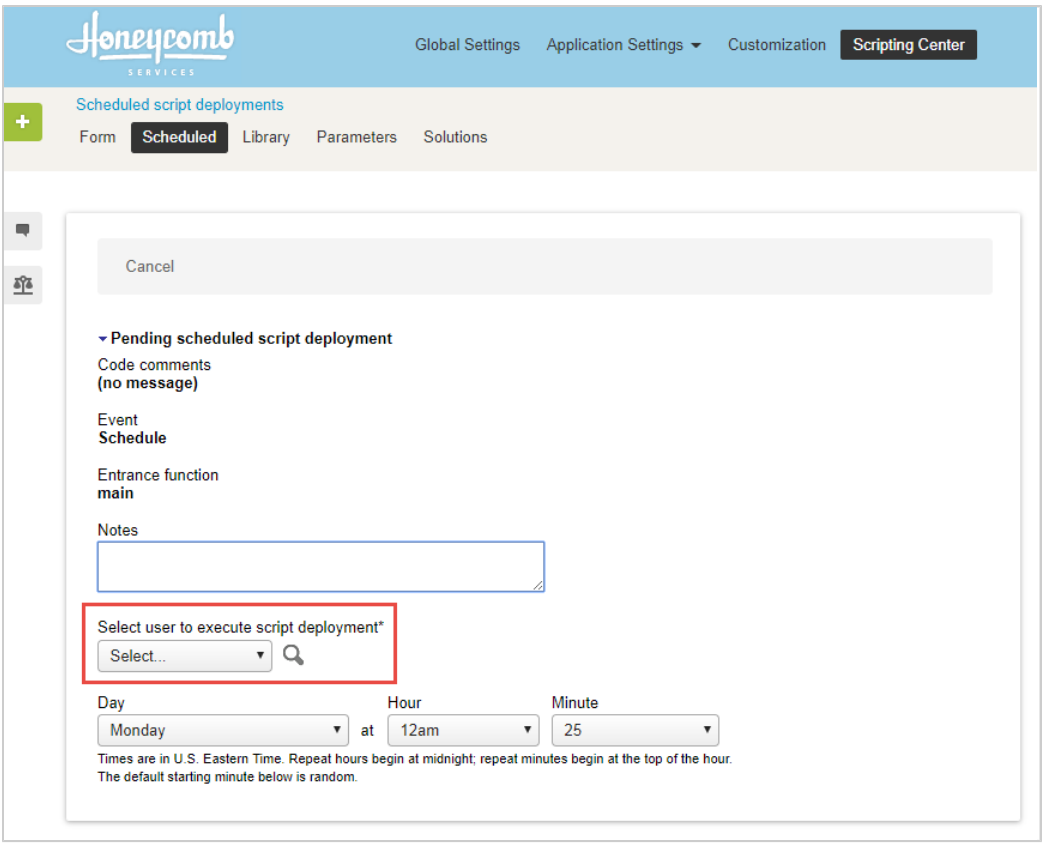
Deploying Scheduled Scripts

To deploy a scheduled script:

1. To deploy a scheduled script, select the **Deploy** option from the **Status** menu, see Scripting Workflow.

   Scheduled scripts are executed within the context of a user. You need to specify the user under which the script is to be executed when you deploy the script.

   As of the April 16, 2016 release, you can select a non-admin user who acts as a proxy to execute a script deployment. This is especially useful when a user does not have the access permissions a script needs to run successfully. With this feature, you need only assign the minimum-necessary permissions.



Scripts can be scheduled to run at any interval:

Example	The script will run...
1st of the month at 12am 00	On the first day of every month at 00:00
Monday at 11am 00	Every Monday at 11:00
Monday at 11am 15	Every Monday at 11:15
Monday at 11am Every 15th minute	Every Monday at 11:15, 11:30, and 11:45
Monday at Every hour 00	Every Monday at the top of each hour, e.g. 00:00, 01:00, ... , 22:00, 23:00
Every day 10am 30	Every day at 10:30

Scheduled Scripts and Scheduled Queue Status

The **Started** and **Duration [sec]** columns on the Scripting Center > Scheduled tab allows administrators to monitor the processing of scheduled scripts in the queue. Refresh your screen to see the progress. The Started and Duration [sec] columns are cleared when the script completes. For more information on scheduled scripts, see User Scripting and Creating Scheduled Scripts.

Creating Library Scripts

Library Scripts are accessed from the **Library** tab of the Scripting Center. Libraries can be called from both form and scheduled scripts. One library can call another library but circular relationships are not allowed. Libraries are automatically available when form and / or scheduled scripts are enabled, see Scripting Switches.

Library scripts are created in a similar way to form and scheduled scripts and follow the same Scripting Workflow.

Library scripts are not associated with a form or event and can only access NSOA.form functions if called from a form script.

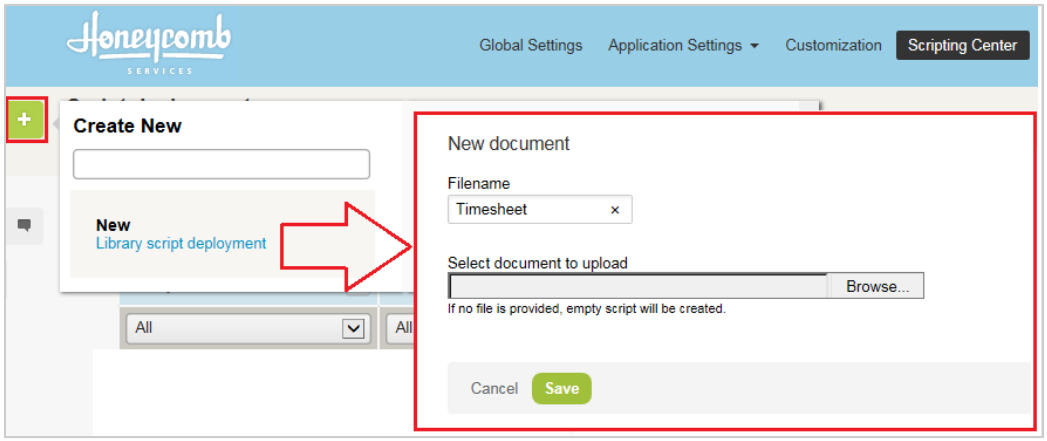
References to libraries can be set from the Scripting Center Manage libraries or from the Scripting Studio Scripting Studio Tools and Settings.

To create a library script:

1. Log in as an Administrator and navigate to the **Library** tab on the Scripting Center.

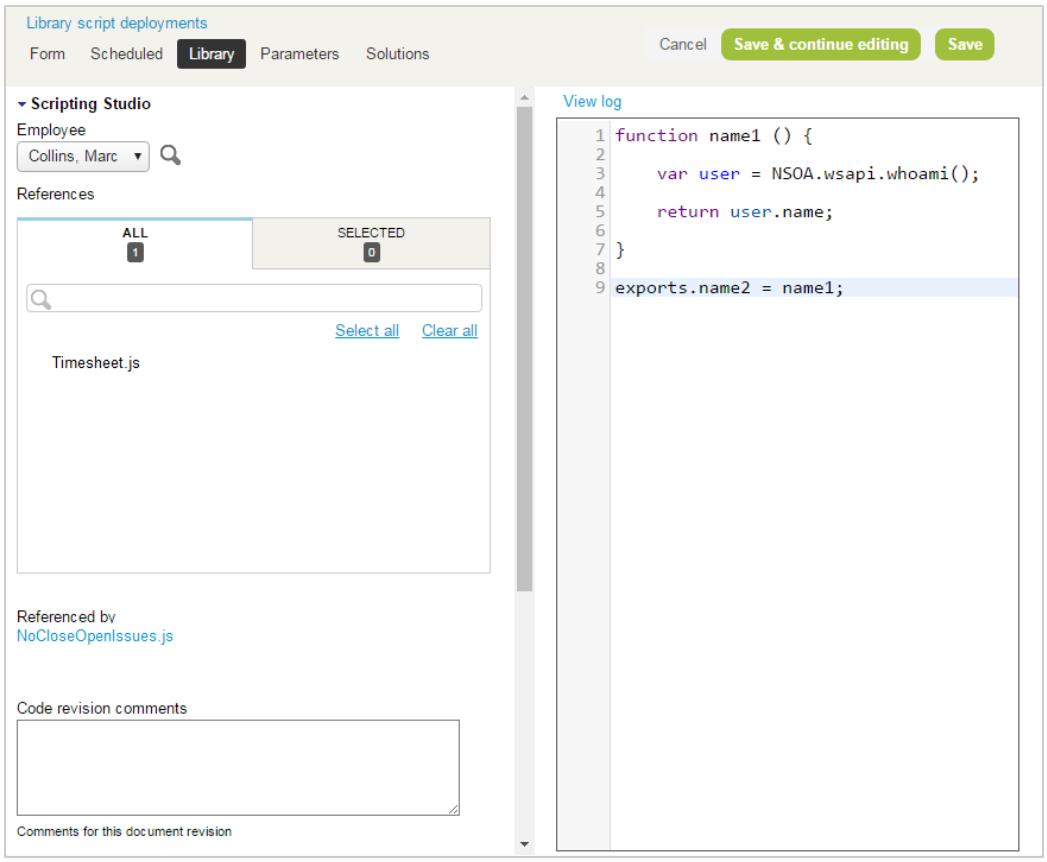
   Note: Make sure you have the necessary switches enabled, see Scripting Switches.

2. Create a new library script from the **Create Button**.



   You need to specify a unique filename for the script in the Dedicated Scripting Workspace. You can optionally select a document that already has the script you need otherwise an empty script file will be created. If you specify a document to upload then a new script file is created from the specified file and the original file left untouched.

3. Click on the **Script** link in the Scripting Center to open the script in the Scripting Studio.
4. Type the script into the editor.



Create functions in exactly the same way as for form and scheduled script and then use **exports** to make the function available. You have the option to change the name of the function that is exported.

Important: Functions created in a library are private to that library by default. You need to use the **exports** keyword to make the function available to scripts calling the library.

Tip: If you don't see a function you are expecting in the Functions explorer check that the function has been exported and that the library does not contain any syntax errors.

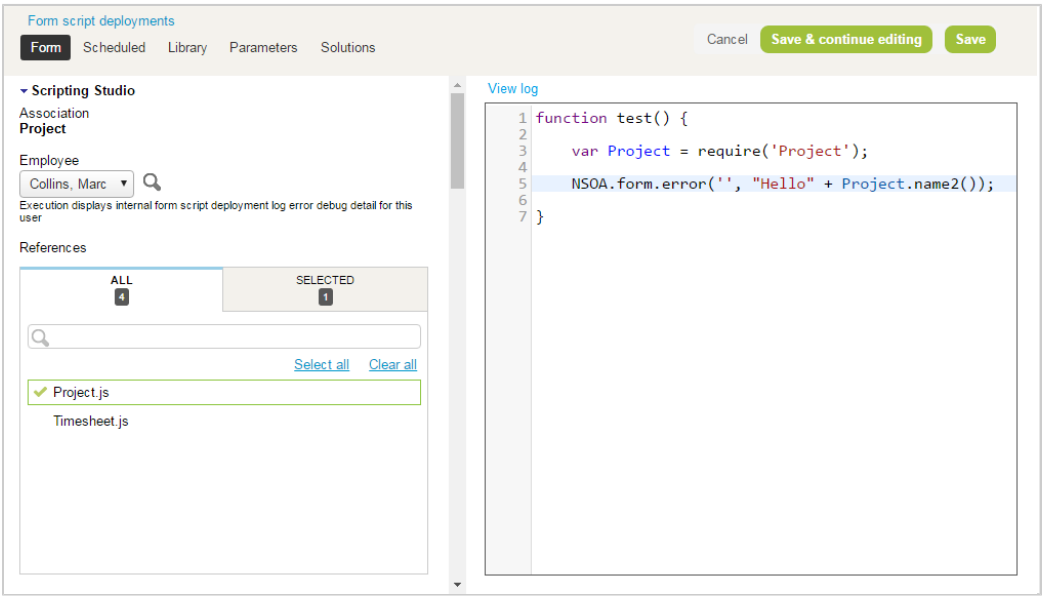
5. Fill out the fields in the Scripting Studio Tools and Settings:
   a. Select the user that the script will run for 'In testing' state, see Testing & debugging.
   b. Select any libraries referenced by this library.
   c. Use the **Code revision comments** to comment the script changes made.
   d. Click **SAVE**.

Note: The act of saving a script in the "Inactive" state will move the script to the "In testing" state, see Scripting Workflow.

Important: You cannot deploy a script that references a library that is not deployed.

To use a library script:

1. Create a form or scheduled script, see Creating Form Scripts.
2. Reference the library either from the Scripting Center Manage libraries or from the Scripting Studio Tools and Settings.
3. Use the library in the script.



a. Use the **require** keyword to create a variable referencing the library.

b. Use methods of the variable to access the functions exported by the library, see Objects.

Creating Parameters

Script Parameters are accessed from the **Parameters** tab of the Scripting Center. Parameters can be used by form, scheduled, and library scripts. Parameters are automatically available when form and / or scheduled scripts are enabled, see Scripting Switches.

Parameters have account wide scope, changing the value for a parameter will affect all scripts using that parameter.

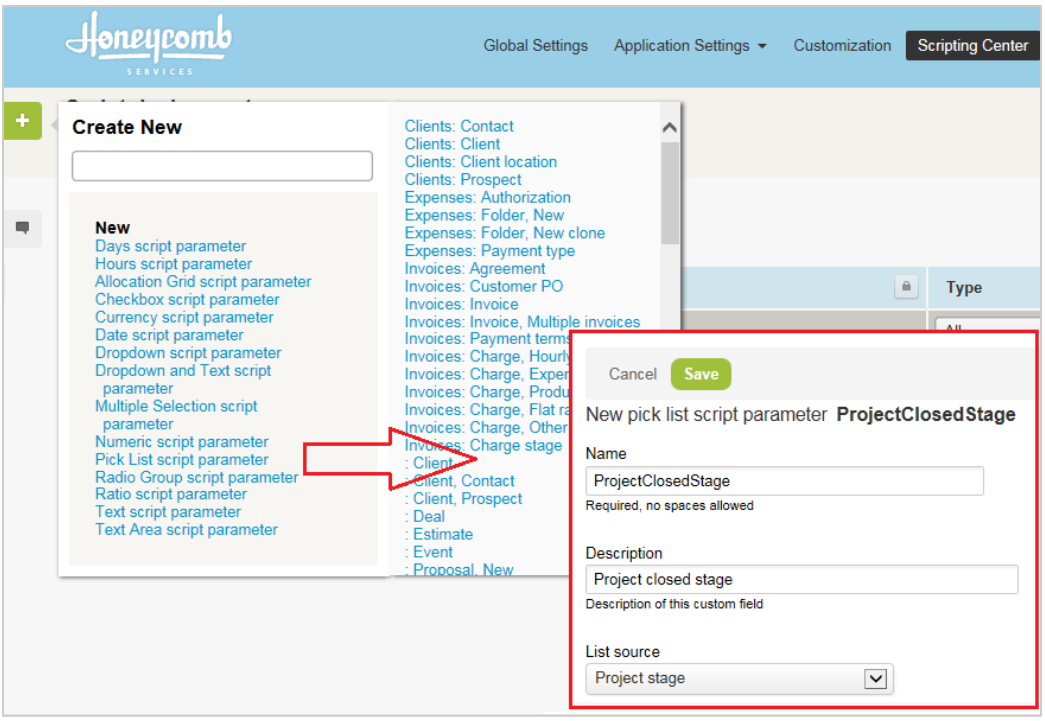
References to parameters can be set from the Scripting Center Manage parameters or from the Scripting Studio Script parameters section.

To create a parameter:

1. Log in as an Administrator and navigate to the **Parameters** tab on the Scripting Center.

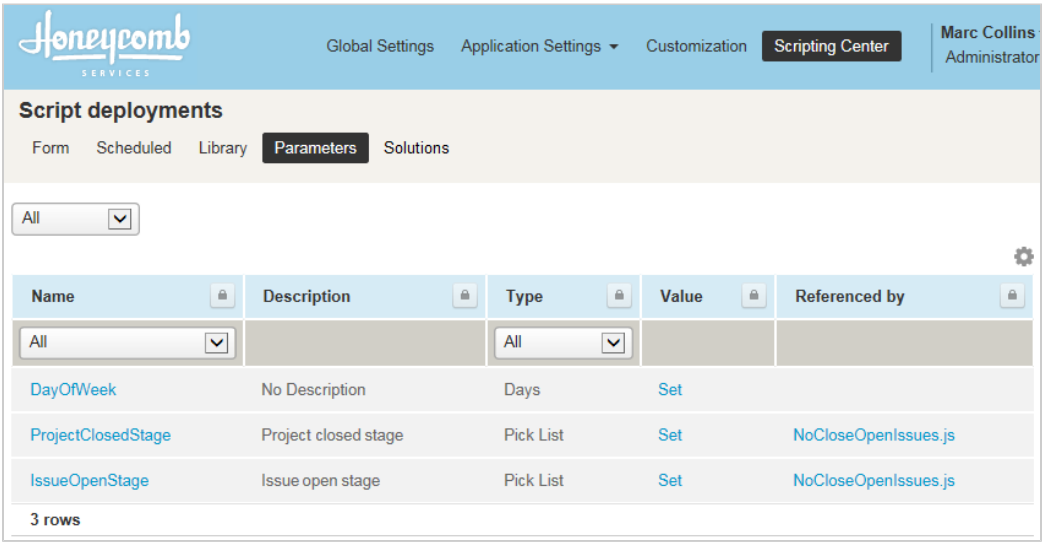
Note: Make sure you have the necessary switches enabled, see Scripting Switches.

2. Create a new parameter from the **Create Button**.



Create a parameter in the same way as you would create a custom field.

3. You can manage all the parameters from the **Parameters** tab in the Scripting Center.



a. Click on the **Name** of a parameter to edit its definition.

> **Note:** You cannot delete a parameter or change the name of a parameter that is **Referenced by** a script.

b. Click on **Set** to change the value selected for the parameter.

> **Important:** A parameter can be referenced by more than one script. Changing the value affects all scripts referencing the parameter. Form, scheduled, and library scripts can reference parameters.

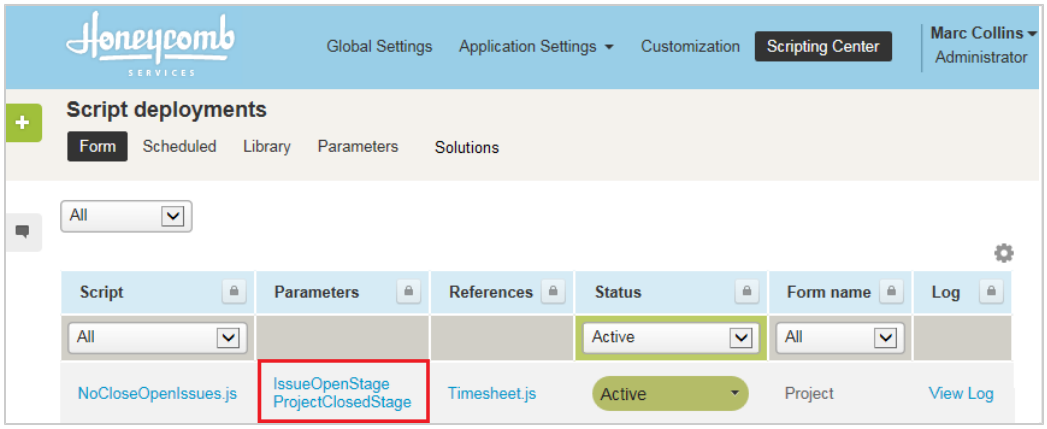
c. Click on the **Referenced by** script to open the script in the Scripting Studio.

To use a parameter:

1. First create any parameters you need, see To create a parameter:.
2. Reference the parameter either from the Scripting Center Manage parameters or from the Scripting Studio Script parameters.
3. You can use the NSOA.context.getParameter(name) or NSOA.context.getAllParameters() functions to read the parameter values in your script.

```
// project_stage_id and issue_stage_id depend on account settings
function test_prevent_project_close_with_open_issue() {

  // return if new stage is not closed
  if (NSOA.form.getValue('project_stage_id') != NSOA.context.getParameter('ProjectClosedStage'))
    return;

  // Load issue data
  var issue = new NSOA.record.oaIssue();
  issue.project_id = NSOA.form.getValue('id');
  issue.issue_stage_id = NSOA.context.getParameter('IssueOpenStage');

  var readRequest = {
    type : "Issue",
    fields : "id, date",
    method : "equal to",
    objects : [ issue ],
    attributes : [{
    name : "limit",
    value : "1"
    }]
  };
```

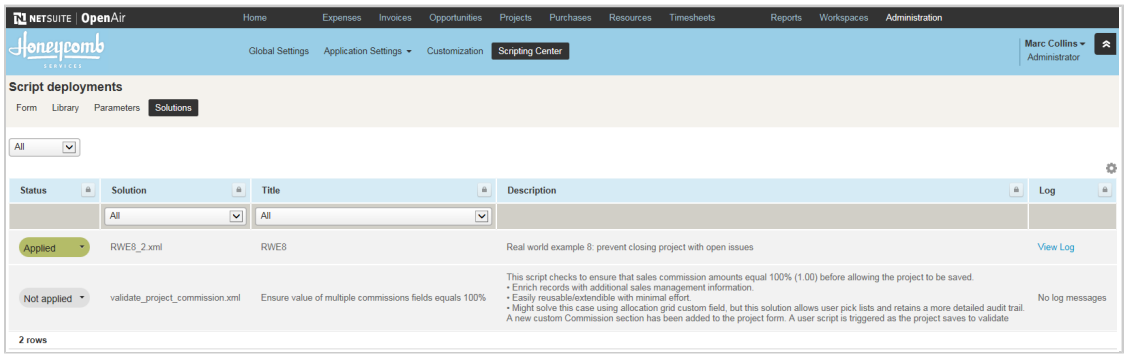
4. Administrators can change the script values from the calling script in the Scripting Center.



Click on the parameter name to change the value.

Important: A parameter can be referenced by more than one script. Changing the value affects all scripts referencing the parameter. Form, scheduled, and library scripts can reference parameters.

Creating Solutions



Platform solutions are accessed from the **Solutions** tab of the Scripting Center. Solutions can be created for form, scheduled, and library scripts. Solutions can also be used to create custom fields, script libraries, and script parameters. Solutions are automatically available when form and / or scheduled scripts are enabled, see Scripting Switches.

Solutions are stored in XML files to make them easy to read, transfer, archive, and compare. Solutions contain a version tag to allow the system to check that the solution file is compatible before applying.

A log is created when a solution is applied to show exactly what the solution created and to record any errors.

Tip: Add the “Solutions” column on the “Form” or “Scheduled” screens to see which scripts are contained in a solution.

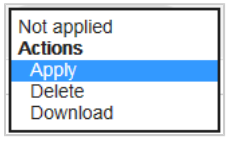
Solution Status

A solution can be in one of three states:

- **Not applied** — The solution has been uploaded.
- **Applied** — The solution has been successfully applied.
- **Failed** — The solution was applied but encountered errors.

Solutions create a log when they are applied to an account. For ‘Applied’ solutions you can view the log to see which objects (scripts or parameters) the solution created. For ‘Failed’ solutions you can also see the errors that occurred when the solution was applied.

Solution Actions



A solution can have the following actions:

- **Apply** — Creates all objects specified in the solution and creates a log file. If successful the solution status changes to 'Applied'. If unsuccessful an error message is displayed and the solution status changes to 'Failed'. See To apply a platform solution:.

  Note: This action is only available for solutions with the 'Not applied' status.

- **Delete** — Deletes the solution with all its history and logs.

  Important: This does not delete any objects created by the solution.

- **Download** — Downloads the solution XML file that was uploaded.

To create a solution:

1. Navigate to Administration > Scripting Center > Solutions.
2. Click the global **Create** button and select **Create solution**.
3. Name the solution and give it a title and description. Select the scripts to include in the solution and select any additional parameters or custom fields. Solutions are built from existing active scripts.
4. Click the **> Create** link under **Documentation URL** if you want to link to documentation describing the solution. Once the link is created, click the link in the Documentation URL column in the Solutions tab to open the document.
5. Select which scripts (including Library scripts) the solution will run from the **Scripts** selection list.
6. Select which custom fields the solution will create from the **Custom fields** selection list.
7. Select which script parameters the solution will create from the **Script parameters** selection list.
8. Click **Save**.

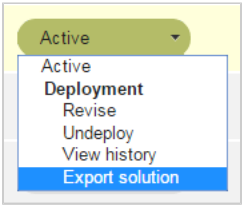
Note: You only need to select additional custom fields and parameters. When you select a script, the solution will automatically pull in the script's required libraries and parameters. The system ignores duplicate selections.

To create a platform solution from a single script:

1. Log in as an Administrator and navigate to the Scripting Center.

   Note: Make sure you have the necessary switches enabled, see Scripting Switches.

2. Navigate to the **Form**, **Scheduled**, or **Library** script you want to create the solution for.
3. Select the **Export solution** option from the script status menu.



   Note: You can create a solution for any script that is not 'Inactive'. See Scripting Workflow.

4. Enter the name, title, and description for the solution file and SAVE.

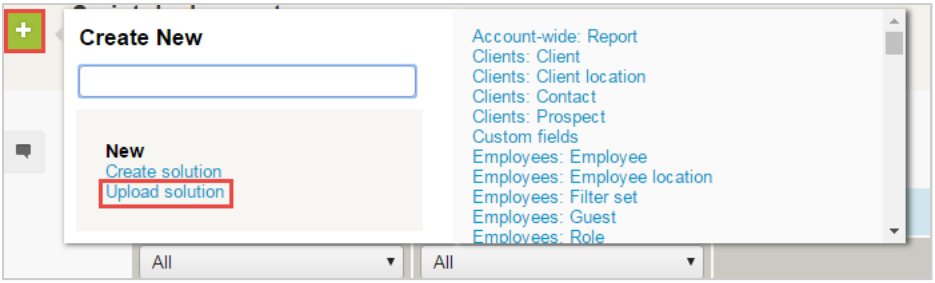
Tip: The solution will contain all library scripts and parameters referenced by the script. To create a solution without a certain reference, first remove the reference from the script and then create the solution.

To apply a platform solution:

1. Log in as an Administrator and navigate to the **Solutions** tab on the Scripting Center.

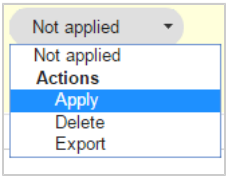
   Note: Make sure you have the necessary switches enabled, see Scripting Switches.

2. Select **Upload solution** from the **Create Button**.



   Note: You are not allowed to upload an invalid solution file.

3. Select the **Apply** option from the status menu.



Accessing Terminology

Remember, all terminology can be customized to meet the unique needs of your company, see Script Terminology. You can allow for changes in terminology by using terminology phrases in your script.

A terminology phrase takes the form "%project%" which is the internal ID for the term surrounded by '%' characters. Use the Terminology section in the Scripting Studio to lookup the internal identifiers to use.

Notice that there are two forms for a term. For example, project and A_project. The second form will return the correct indefinite article (a/an) required for the term.

Tip: Singular/plural and capitalization are respected in parsing the terminology.

You can access terminology with the following functions:

- NSOA.context.getAllTerms()
- NSOA.context.getTerm(termid)

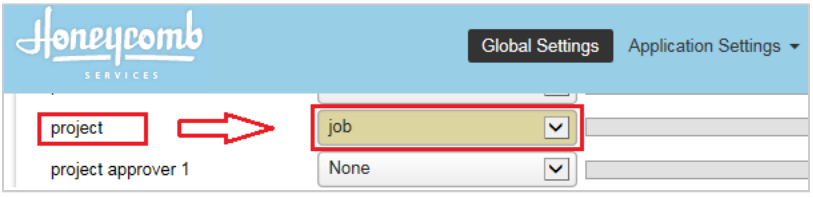
- NSOA.context.parseTerminology(message)

Terminology phrases are directly parsed in log and error messages:

- NSOA.form.error(field, message)
- NSOA.meta.alert(message)
- NSOA.meta.log(severity, message)

To use terminology in scripts:

1. Administrator set account terminology from Administration > Global Settings > Interface: Terminology.



> **Note:** You only need to enter the replacement term in its singular form. OpenAir automatically generates the plural term where applicable.

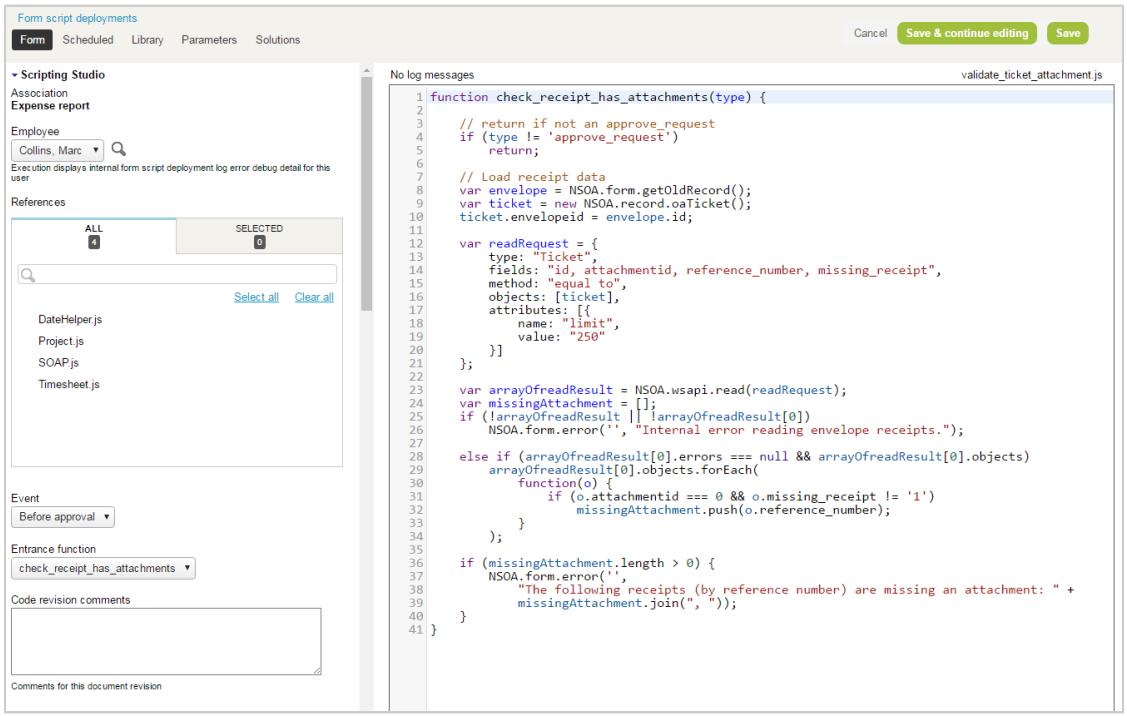
2. Scripts can look up a term using the NSOA.context.getTerm(termid) function and can use "%TERMID%" phrases in strings and parse them with the NSOA.context.parseTerminology(message), NSOA.form.error(field, message), NSOA.meta.alert(message), and NSOA.meta.log(severity, message) functions.

```
var proj_term = NSOA.context.getTerm("Projects");
// proj_term = "Jobs"

var msg1 = NSOA.context.parseTerminology("%Project% saved!");
// msg1 = "Job saved!"

var msg2 = NSOA.context.parseTerminology("Notes attached to %project%.");
// msg2 = "Notes attached to job."

// Automatic terminology parsing
NSOA.form.error("", "%Project% saved!");
NSOA.meta.alert("%Project% saved!");
NSOA.meta.log("Info", "%Project% saved!");
```

> **Note:** Singular/plural and capitalization are respected in parsing the terminology.

3. Users see the messages displayed with the correct account terminology.

Scripting Studio



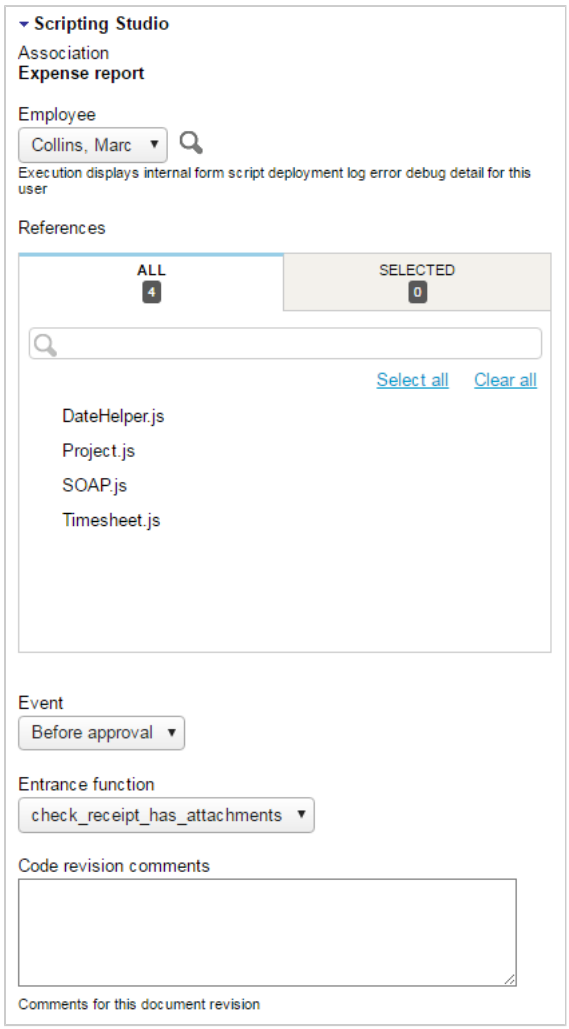
The Scripting Studio is accessed by clicking on a script link in the Scripting Center.

From the Scripting Studio a script developer can quickly create scripts with a full screen editor supported by intuitive tools with drag-and-drop:

- Scripting Studio Tools and Settings
- SOAP explorer
- Functions explorer
- Script parameters
- Terminology
- Form schema
- Testing & debugging
- Editor

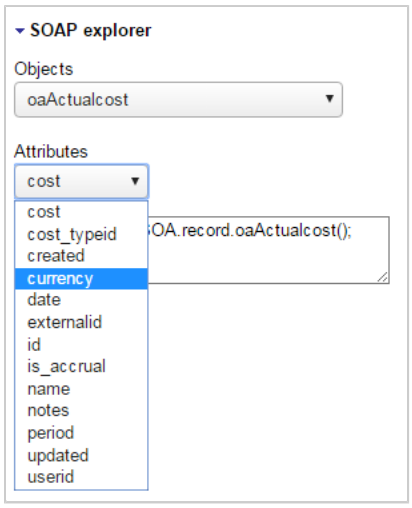
You can view any log messages the script has generated by clicking on the **View Log** link at the top—left of the editor, see also Testing Form Scripts

Scripting Studio Tools and Settings

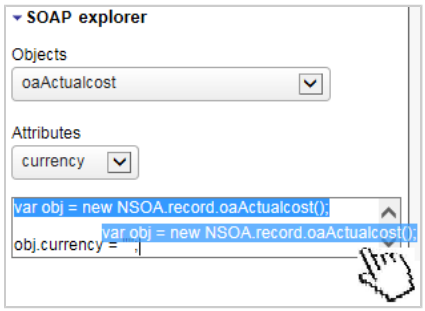


- **Association** — An individual script can only be associated with one form and this is set when the script is created. The same script cannot be triggered by two different forms or even form events. An individual form may trigger as many scripts as necessary.
- **Employee** — This is the user that will test the script, see Testing & debugging for more details.
- **References** — Select the libraries that are used by this script.
- **Event** — This is the event that will trigger the script to run, see Events.
- **Entrance function** — This is the name of the function to run, see Entrance Function.
- **Code revision comments** — These are optional notes that the developer can add.

SOAP explorer

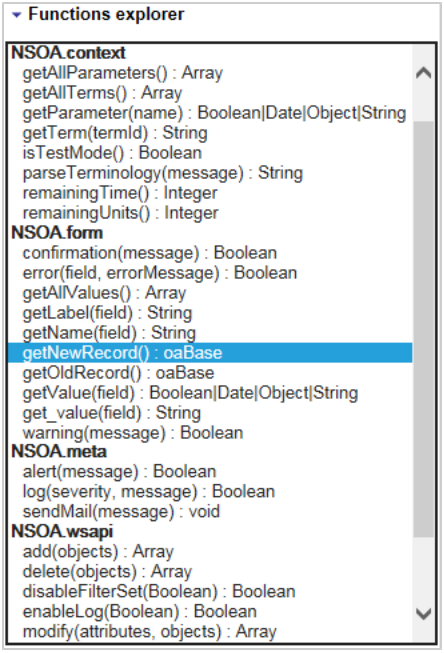


From the SOAP explorer you can quickly browse through the SOAP API objects and attributes, and view examples of usage. First select the SOAP object an then you can select an attribute for the object.

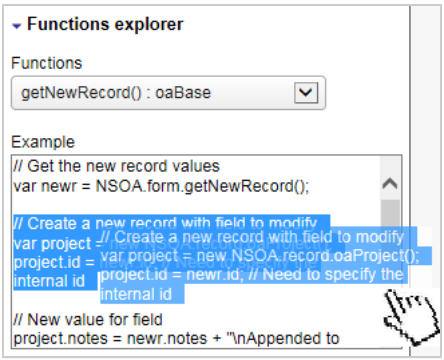


You can drag and drop code examples directly into the editor. See also the Auto List & Complete feature.

Functions explorer



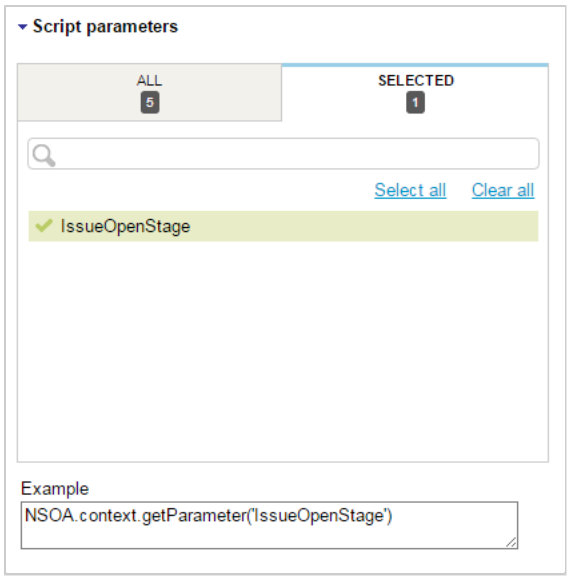
The functions explorer acts as an online cheat sheet showing the syntax for all the available NSOA functions and for any selected library. Select a function to view an example of usage.



You can drag and drop code examples directly into the editor. See also the Auto List & Complete feature.

Script parameters

From the script parameters section you can see all the parameters available in the account and select parameters used in the script.



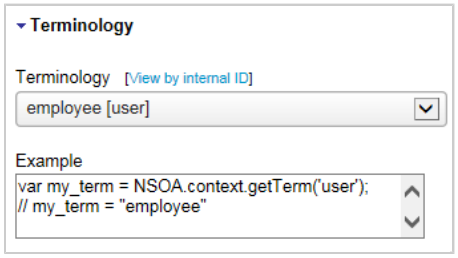
Click on a selected parameter to see an example. You can drag and drop code examples directly into the editor.

See also Creating Parameters.

Note: Referencing a parameter prevents the parameter from being deleted or changed in a way that will affect the script. See also the Auto List & Complete feature.

Terminology

From the terminology section you can browse through all the terminology available in OpenAir and see the terms set for the current account.

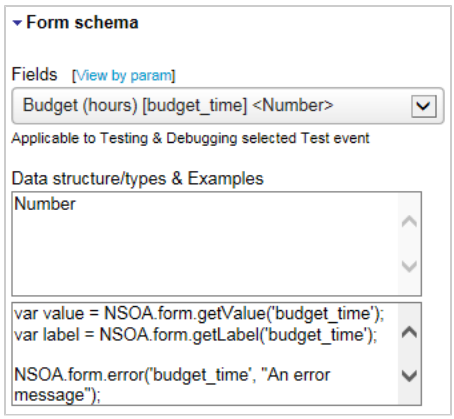


Select a term to see an example. You can drag and drop code examples directly into the editor.

See Accessing Terminology for details.

Form schema

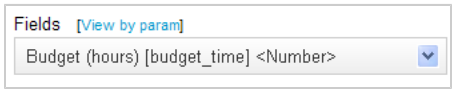
The **Form schema** allows you to explore the form you are creating the script for. This provides vital information as you need to know the names of the fields and the structure of the objects so you can reference them in your scripts.



The **Fields** drop-down list at the top gives a complete list of the available fields on the form. You can select between **View by param** or **View by label** by clicking on the link.

- If **View by label** is selected (default), each entry in the drop-down list has the following format:

  Label [Field] <Data Type>

  - **Budget (hours)** is the label the user sees on the form.
  - **Budget_time** is the field name you need to use when calling NSOA Functions.
  - **Number** is the internal data type JavaScript will use for a variable created for this field. See Dynamic Data Types.
- If **View by param** is selected, each entry in the drop-down list has the following format:

  Field [Label] <Data Type>

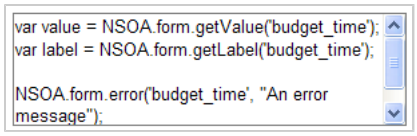
  - **Budget_time** is the field name you need to use when calling NSOA Functions.
  - **Budget (hours)** is the label the user sees on the form.
  - **Number** is the internal data type JavaScript will use for a variable created for this field. See Dynamic Data Types.

Tip: If you have added a new custom field and this is not listed in the **Form schema**, open the form with the new custom field to refresh the custom field list, and then open the Scripting Studio again. See Custom Fields for more details.

When you select a field from the drop down list the **Data structure/types & Examples** area is filled.

The **Data structure/types & Examples** area has two text fields.

The text field on the left shows the data type or data structure (if the data type is an object) for the field. See Object Fields.

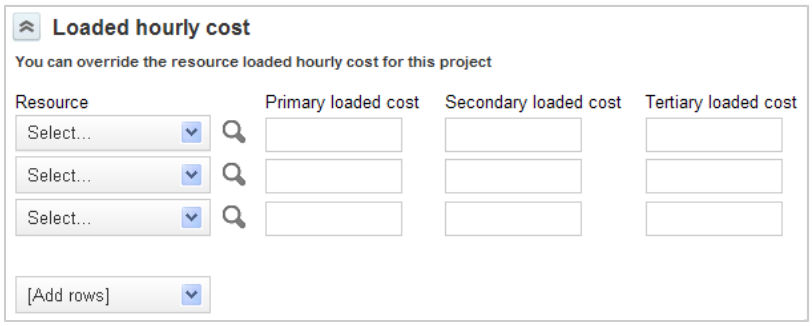


The text field on the right shows correctly formatted code samples using the NSOA functions for the selected field. You can directly copy and paste these samples into your script. See Object Fields.

Object Fields

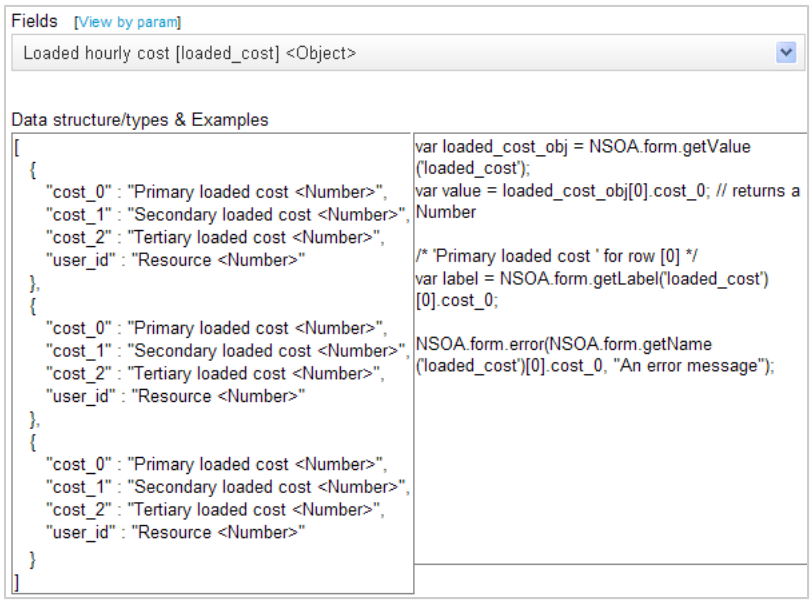
Fields with the **Object** type expose properties that allow you to access their internal data structure.

For example, consider the **Loaded hourly cost** form section.



All these fields are exposed through one **loaded_cost** field <Object>.

Notice the way the Loaded hourly cost fields map to the data structure you need for your script.



The data structure has three blocks that correspond to the three rows of fields in **Loaded hourly cost** form section.

Getting a value from an object field is a two step process:

1. First you need to get the object variable for the field:

```
var loaded_cost_obj = NSOA.form.getValue("loaded_cost");
```

2. Then you can use the object variable to get the value:

```
var value = loaded_cost_obj[0].cost_0;
```

You can combine these two steps into one line:

```
var value = NSOA.form.getValue("loaded_cost")[0].cost_0;
```

Take a closer look at the syntax: var value = **loaded_cost_obj[0].cost_0**.

Each row in the data structure is accessed in the same way as an array, i.e., **loaded_cost_obj[0]** is this first row. Each column of the row is accessed by the field name i.e. **loaded_cost_obj[0].cost_0** is the "Primary loaded cost" for the first row.

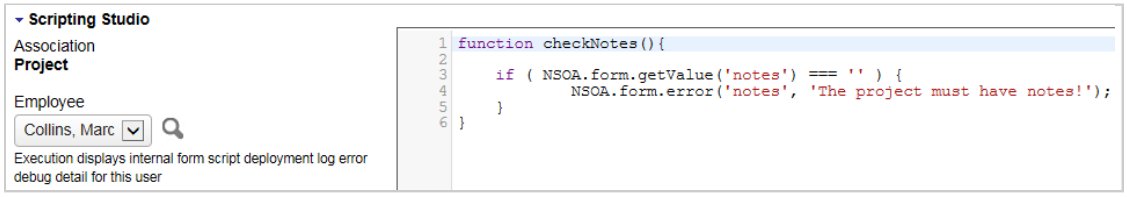
Note: Remember Arrays are zero-based.

Take a closer look at the syntax: var value = **NSOA.form.getValue("loaded_cost")[0].cost_0**

Notice this is actually **NSOA.form.getValue("loaded_cost")** with **[0].cost_0** appended.

This is referencing "loaded_cost" in the same way as a simple field and then using **[0].cost_0** to view the required property of the returned object.

Testing & debugging



From the Scripting Studio Tools and Settings you must specify a test user. You can determine if your script is running in Test mode within your script by calling NSOA.context.isTestMode().

Tip: Calls to NSOA.meta.log(severity, message) with a "debug" or "trace" severity are only executed in Test mode and do not consume Scripting Governance units but are limited to a maximum of 1000 per script.

If you are seeing a problem that is only happening with a particular user you can select that user to be the one that the test code runs for.

To set a test user

1. Select the user from the drop-down list.

2. Click on **SAVE**.

> **Note:** The named user will also be able to access error debug detail.

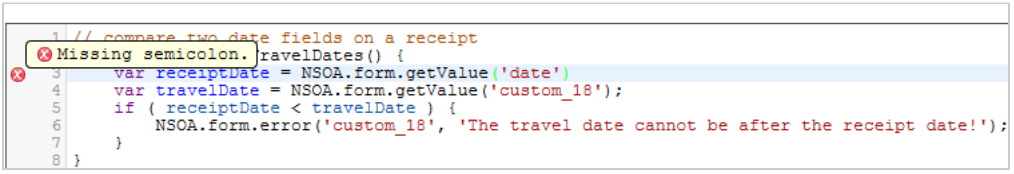
For more information on **Testing & Debugging**, see Testing Form Scripts.

Editor

```
1 // compare two date fields on a receipt
2 function validateTravelDates() {
3     var receiptDate = NSOA.form.getValue('date');
4     var travelDate = NSOA.form.getValue('custom_18');
5     if ( receiptDate < travelDate ) {
6         NSOA.form.error('custom_18', 'The travel date cannot be after the receipt date!');
7     }
8 }
```

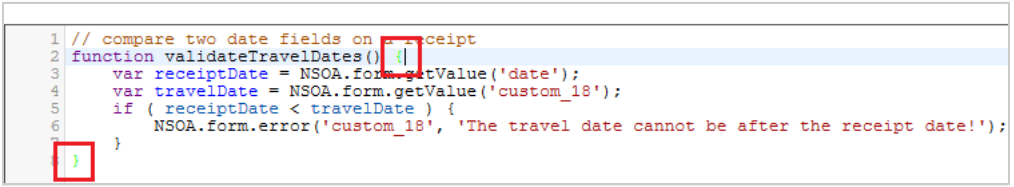
Editor Features:

- **Auto List & Complete** — Type '.' after "NSOA" and the Auto List window appears showing all the available options, see Auto List & Complete.
- **Color coding** — Keywords, variables, literals, comments, etc. are highlighted in different colors to aid correct coding.
- **Line numbers** — Line numbers are listed in the left margin to assist in development and debugging.
- **Line highlighting** — The line the cursor is on is highlighted to assist in editing the code.
- **Syntax checking** — Errors and warning are displayed as you type into the editor.

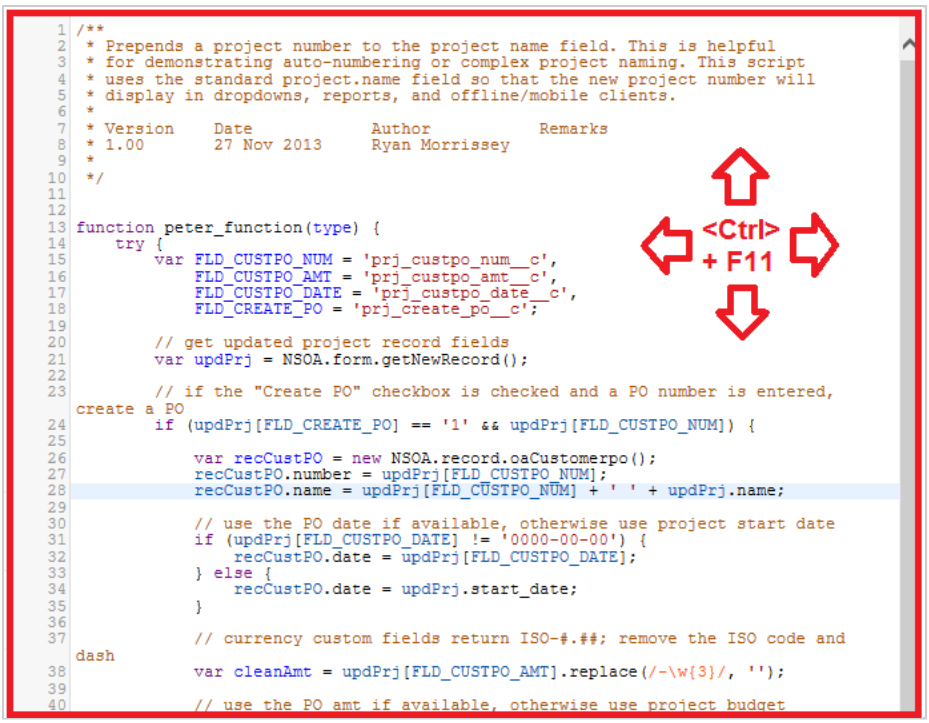


> **Note:** Hover your mouse over the error icon to display the error message.

- **Bracket completion** — If you type an opening bracket the matching closing bracket is automatically created.
- **Matching brackets** If you place your cursor next to a bracket then the matching brackets are highlighted



- **Full-screen** — Click into the editor and press **<Ctrl> + F11** to full-screen the script editor




```
/**
 * Prepends a project number to the project name field. This is helpful
 * for demonstrating auto-numbering or complex project naming. This script
 * uses the standard project.name field so that the new project number will
 * display in dropdowns, reports, and offline/mobile clients.
 *
 * Version    Date            Author          Remarks
 * 1.00       27 Nov 2013     Ryan Morrissey
 *
 */


function peter_function(type) {
    try {
        var FLD_CUSTPO_NUM = 'prj_custpo_num__c',
            FLD_CUSTPO_AMT = 'prj_custpo_amt__c',
            FLD_CUSTPO_DATE = 'prj_custpo_date__c',
            FLD_CREATE_PO = 'prj_create_po__c';

        // get updated project record fields
        var updPrj = NSOA.form.getNewRecord();

        // if the "Create PO" checkbox is checked and a PO number is entered, create a PO
        if (updPrj[FLD_CREATE_PO] == '1' && updPrj[FLD_CUSTPO_NUM]) {

            var recCustPO = new NSOA.record.oaCustomerpo();
            recCustPO.number = updPrj[FLD_CUSTPO_NUM];
            recCustPO.name = updPrj[FLD_CUSTPO_NUM] + ' ' + updPrj.name;

            // use the PO date if available, otherwise use project start date
            if (updPrj[FLD_CUSTPO_DATE] != '0000-00-00') {
                recCustPO.date = updPrj[FLD_CUSTPO_DATE];
            } else {
                recCustPO.date = updPrj.start_date;
            }

            // currency custom fields return ISO-#.##; remove the ISO code and dash
            var cleanAmt = updPrj[FLD_CUSTPO_AMT].replace(/-\w{3}/, '');

            // use the PO amt if available, otherwise use project budget
```

While working in full-screen mode you can still continue using the Auto List & Complete feature.

Press **Esc** to exit full-screen mode and save your changes in the usual way.

- **Search and Replace Functions** — Search through scripts using simple or regexp search expressions. Use the following key shortcuts for searches within the Script Editor:
  - **Start a Search** — Ctrl+F / Cmd+F
  - **Find Next** — Ctrl+G / Cmd+G
  - **Find Previous** — Shift+Ctrl+G / Shift+Cmd+G
  - **Replace** — Shift+Ctrl+F / Cmd+Option+F
  - **Replace All** — Shift+Ctrl+R / Shift+Cmd+Option+F

  These shortcuts can also be found in the Tips menu from within the Script editor.

  You can use regexp to search for more complex strings. For example, entering **/envelope|ticket/** in the search field searches for both “envelope” and “ticket”.

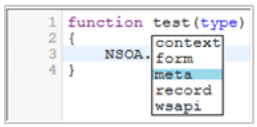
  Once a search dialog has been opened, press Escape to exit it without searching.
- Jump to Line Functions — You can move through your scripts quickly by entering Jump to Line functions. Press Alt+G to open the Jump to Line dialog. Then, enter the script line you want to move the cursor to. The following input formats are accepted:
  - Line — enter the line to move the cursor to. For example, entering 25 in the Jump to Line field moves the cursor to line 25
  - Line:column — enter both the line and column separated by a colon. For example, entering 25:9 moves the cursor to line 25, column 9.
  - +/-Line — enter how many lines forward or backward to move your cursor. For example, if the cursor is at line 5, and you enter +5 into the Jump to Line field, the cursor moves to line 10.
  - Scroll% — enter a percent of the document to move the cursor to. For example, entering 50% in the Jump to Line field moves the cursor 50%, to the middle of the script. Add + or _ to the

percentage to move forward or backward. For example, if the cursor is at the end of the script, — 50% moves the cursor backwards to the middle of the script.

Once the Jump to Line dialog has been opened, press Esc to exit it without moving the cursor.

Auto List & Complete

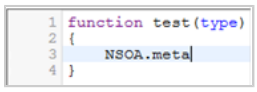
When a user types the text "NSOA" into the **Scripting Studio** editor and then hits the '.' character the Auto List window appears showing all the available options:



The user has the following options:

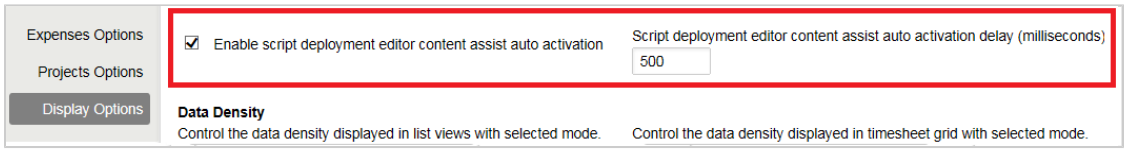
- Click on the required item with the mouse and double-click to select it.
- Use the up and down arrow keys to select the required item and then hit 'Enter' to select it.
- Type the first character of the required item (e.g. 'm') to highlight it and then hit 'Enter' to select it. If more than one item starts with the same letter then the first item will be highlighted and the list of options filtered.
- Hit 'Esc' to close the Auto List window and type as normal. Clicking outside of the editor window will also close the Auto List window.

> **Tip:** Press <Ctrl> + <Space> to show the Auto List window at any point in the editor.



On selecting an item from the Auto List window, the value will be copied into the editor and typing continues after the inserted value.

> **Tip:** Auto List & Complete is enabled by default. You can change your settings from **User center > Personal settings**.

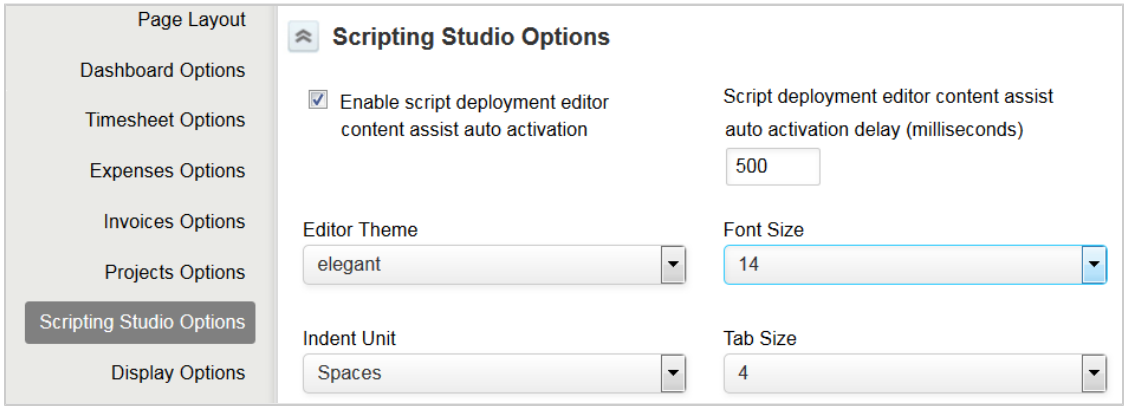


Scripting Studio Options

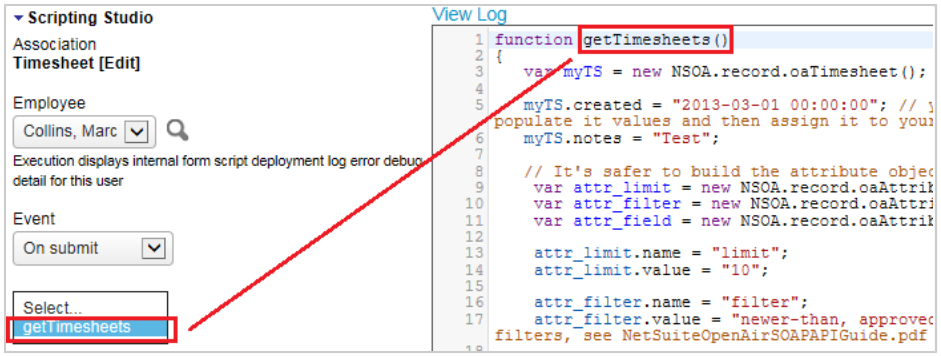
The scripting studio can be customized with various display options. To customize your scripting studio and editor, navigate to User Center > Personal settings > Display Options > Scripting Studio Options. From here, you can customize the following:

- Editor Theme — choose from a variety of color schemes for the script editor

- Indent Unit — select whether an indent unit is a space or a tab in the script editor
- Font Size — select the size of the text font in the script editor
- Tab Size — set how many spaces a tab uses in the script editor



Entrance Function



An **Entrance function** serves as the starting point in your script. For more on functions see Functions.

The **Entrance function** is associated with a form event in the Scripting Studio Tools and Settings options, see Events.

Note: The **Entrance function** field value is the name of the function without parenthesis (or parameter, if used).

Entrance Function Type Parameter

The entrance function can optionally take a **type** parameter which will be passed one of:

- **'new'** — this is passed when saving a new form.

  Note: This applies to **New** and **New, from another** actions.

- **'edit'** — this is passed when updating an existing form.

> **Note:** This also applies when creating a clone.

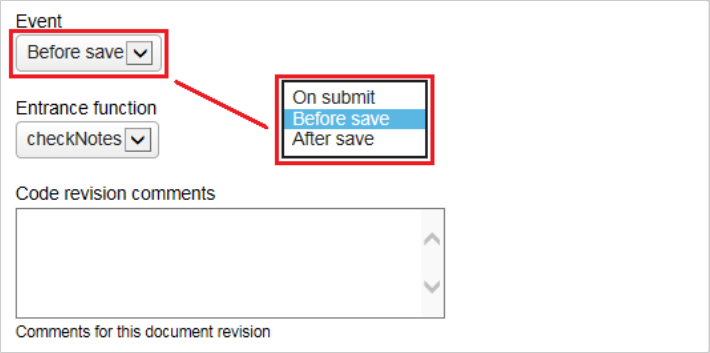
- **'approve_request'** — this is passed when a record is submitted for approval.
- **'approve'** — this is passed when a record is approved.
- **'reject'** — this is passed when a record is rejected.
- **'unapprove'** — this is passed when a record is unapproved, and is supported by bookings, booking requests, expense reports, invoices, purchase orders, purchase requests, schedule requests, and timesheets.

> **Note:** To enable the **Unapprove** type, please contact OpenAir Support and ask them to enable the "Enable user scripts to use an unapproval context in the after approval event" switch.

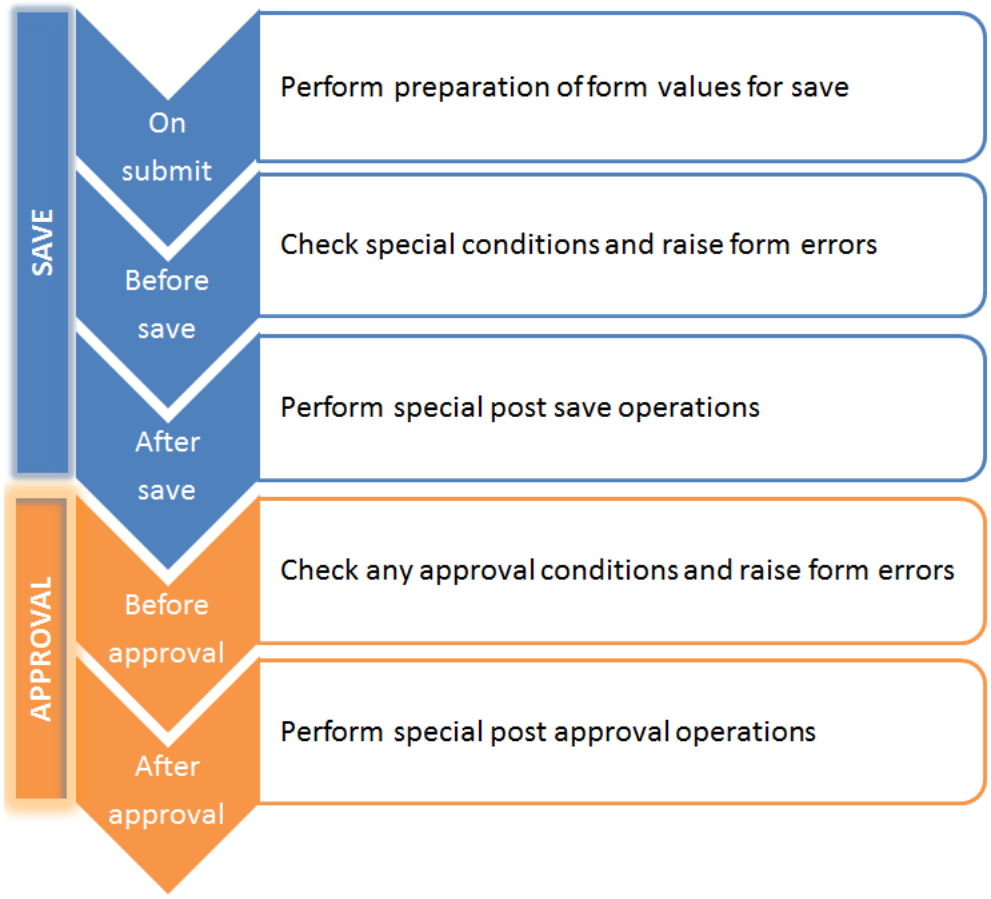
The type value will be the same regardless of the form **Event**. For example, if you call your entrance function on **"After save"** when creating a new form you will still be passed a type value of **'new'**.

Events

Scripts are triggered by events. The required **Event** is specified in the Scripting Studio Tools and Settings options.



You should select the **Event** according to the purpose of the Entrance Function, see the diagram below.



Note: Only forms that can take part in an approval process receive approval events.

- **On submit** — Always executed. This is the first event that occurs when the user click SAVE.
- **Before save** — Always executed. This is where you should check that any special conditions on the form are valid and raise form errors if required by calling NSOA.form.error(field, message).

  Note: The record does not exist in the database at this stage so you can't call **wsapi** functions to change any of the record values.

- **After save** — Executed if no form errors are raised. This is where you should call **wsapi** functions to modify the data held on this or related records.
- **Before approval** — This is where you can perform additional checks and prevent a record from being sent for approval by calling NSOA.form.error(field, message).
- **After approval** — This is where you can carry out additional actions following a record approval or reject.

Note: Only scripts which are triggered by “Before approval” and “After approval” events are supported on mobile devices. Scripts which are triggered by “On submit,” “Before save,” or “After save” are not supported for mobile devices. For an example of a script which is compatible with mobile devices, see Ensure resource time entry matches booking planning and project worked hours.

Scripting Governance

To prevent scripts from consuming excessive resources or running out of control, limitations are placed on scripts:

- **Time Limit** — If a form script runs for more than 5 seconds (not including wsapi call time) it is automatically terminated with a form error. If a request (all scripts triggered by the same form save) uses more than 60 seconds of wsapi time it will also be automatically terminated.

  Note: Scheduled scripts are allowed 1 hour of JS runtime and 1 hour of wsapi.

- **Units Limit** — NSOA functions are assigned a unit value. Each time an NSOA function is called its unit value is consumed. A script is allowed to consume a maximum of 1000 units with each run before it is automatically terminated with a form error. You can determine how many units you have remaining by calling NSOA.context.remainingUnits(). See the table below for the unit value of each NSOA function.

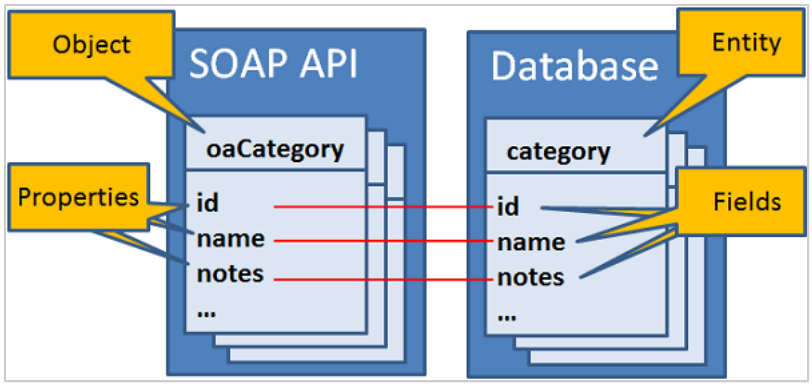
  Note: Scheduled scripts are allowed 1, 000, 000 units.

- **SendMail Limit** — Additional limits are placed on the NSOA.meta.sendMail(message) function. Form Scripts are allowed to send a maximum of 3 emails. Scheduled Scripts are allowed to send a maximum of 100 emails. The email body is not allowed to exceed 30,000 characters. The email subject is trimmed to the first line passed.

NSOA Function	Units
▪ NSOA.context.getTerm(termid) ▪ NSOA.context.isTestMode() ▪ NSOA.context.remainingTime() ▪ NSOA.context.remainingUnits() ▪ NSOA.record.<complex type>([id]) ▪ NSOA.wsapi.remainingTime()	0
▪ NSOA.context.getParameter(name) ▪ NSOA.form.confirmation(message) ▪ NSOA.form.error(field, message) ▪ NSOA.form.getLabel(field) ▪ NSOA.form.getName(field) ▪ NSOA.form.getNewRecord() ▪ NSOA.form.getOldRecord() ▪ NSOA.form.getValue(field)	1

NSOA Function	Units
▪ NSOA.form.get_value(field) ▪ NSOA.form.warning(message) ▪ NSOA.record.<complex type>([id]) ▪ NSOA.wsapi.disableFilterSet([flag]) ▪ NSOA.wsapi.enableLog([flag]) ▪ NSOA.wsapi.whoami()	
▪ NSOA.context.parseTerminology(message) ▪ NSOA.meta.alert(message) ▪ NSOA.meta.log(severity, message)	4 **Note:** Calls to NSOA.meta.log(severity, message) with the severity parameter set to "debug" or "trace" do not consume units but are limited to a maximum of 1000 per script.
▪ NSOA.context.getAllParameters( ) ▪ NSOA.context.getAllTerms() ▪ NSOA.form.getAllValues() ▪ NSOA.meta.sendMail(message) ▪ NSOA.NSConnector.integrateRecord()	10
▪ NSOA.wsapi.read(readRequest) ▪ NSOA.wsapi.add(objects) ▪ NSOA.wsapi.delete(objects) ▪ NSOA.wsapi.submit(submitRequest) ▪ NSOA.wsapi.approve(approveRequest) ▪ NSOA.wsapi.reject(rejectRequest) ▪ NSOA.wsapi.unapprove(unapproveRequest)	20 +10 for each additional object passed.
▪ NSOA.wsapi.modify(attributes, objects) ▪ NSOA.wsapi.upsert(attributes, objects)	40 +20 for each additional object passed.
▪ NSOA.NSConnector.integrateAllNow()	1000

SOAP API



OpenAir user scripting provides access to the OpenAir SOAP API (Web Services) through the NSOA.wsapi functions, see NSOA Functions. Before you begin using these functions, we recommend that you refer to the OpenAir SOAP API Reference Guide.

Important: Pay attention to **Appendix C Best Practices** in the **OpenAir SOAP API Reference** guide.

Tip: All OpenAir Complex types start "oa", e.g. "oaCategory". You can look up the OpenAir Complex Types and their properties from the following link: http://www.openair.com/wsdl.pl?wsdl. If you strip away the "oa" you are left with the table name, e.g. "Issue", see http://www.openair.com/database/single_user.html.

Note: You need the **Enable user script support for Web Service API methods** switch enabled to use the NSOA.wsapi functions, see Scripting Switches.

Tip: Scripts are executed within the context a user. This means that the user filter sets for the logged in user will be applied unless disabled, see NSOA.wsapi.disableFilterSet([flag]).

Using the SOAP API:

- Making SOAP Calls
- Using SOAP Results
- Handling SOAP Errors

Making SOAP Calls

The SOAP API is an object-oriented interface.

- You pass in an array of OpenAir Complex Type objects as a parameter. An error will be returned if the array contains more than 1000 objects.

  Note: You create a complex type object with the NSOA.record.<complex type>([id]) function. See **Chapter 6 OpenAir Complex Types** in the OpenAir SOAP API Reference Guide for more details.

- Some functions also require an array of Attribute objects as the first parameter.

- Functions return either an object or array of objects:
  - NSOA.record.<complex type>([id]) returns an OpenAir Complex Type object.
  - NSOA.wsapi.read(readRequest) returns an array of ReadResult objects.
  - NSOA.wsapi.add(objects), NSOA.wsapi.delete(objects), NSOA.wsapi.modify(attributes, objects), and NSOA.wsapi.upsert(attributes,objects) return an array of UpdateResult objects.

Important: The updated and created fields are maintained automatically by the system. You can read these values, but they cannot be modified.

Tip: It is more efficient to batch a series of objects together into a single SOAP API call rather than making a separate call for each object. The objects in the array are processed according to their order in the array.

Adding data

You add data to OpenAir by creating one or more OpenAir Complex Type objects, placing them into an array, and passing the array to the NSOA.wsapi.add(objects) function. You must specify all the mandatory fields for the objects passed. The id, updated and created fields are set automatically by the system.

To add data to OpenAir

1. Create an OpenAir Complex Type object with the NSOA.record.<complex type>([id]) function.

```
var category = new NSOA.record.oaCategory();
```

2. Fill out the properties for the object, see Objects.

```
category.name = "New Category";
category.cost_centerid = "123";
category.currency = "USD";
```

3. Place the object into an array of objects, see Arrays.

```
// To turn an object into an array of object, simply place it inside square brackets
var objects = [category]; // or just pass [category]
```

4. Pass the objects as a parameter to the NSOA.wsapi.add(objects).

```
var results = NSOA.wsapi.add( [category] );
```

5. Check for any errors, see Handling SOAP Errors.

Modifying data

You modify data to OpenAir by creating one or more OpenAir Complex Type objects, placing them into an array, and passing the array to the NSOA.wsapi.modify(attributes, objects) function. In each object passed, you need to specify the internal **id** and just the properties (fields) in the objects that you want to change. The **updated** field is set automatically by the system.

To modify data in OpenAir

1. Create an OpenAir Complex Type object with the NSOA.record.<complex type>([id]) function.

```
var category = new NSOA.record.oaCategory();
```

2. Fill out the internal id for the object and the properties you want to change, see Objects.

```
category.id = 79; // This is the id of the existing customer
category.cost_centerid = "453"; // The new value
```

3. Place the object into an array of objects, see Arrays.

```
// To turn an object into an array of object, simply place it inside square brackets
var objects = [category]; // or just pass [category]
```

4. Optionally create an array of attributes to pass.

```
var attributes = [];
```

5. Pass the objects and attributes as parameters to the NSOA.wsapi.modify(attributes, objects).

```
var results = NSOA.wsapi.modify( [attributes], [category] );
```

6. Check for any errors, see Handling SOAP Errors.

Deleting data

You delete data from OpenAir by creating one or more OpenAir Complex Type objects, placing them into an array, and passing the array to the NSOA.wsapi.delete(objects) function. In each object passed, you need to specify the internal **id**.

Important: You cannot delete an entity (database record) that has dependent records. You must first delete all the dependent records.

To delete data in OpenAir

1. Create an OpenAir Complex Type object with the NSOA.record.<complex type>([id]) function.

```
var category = new NSOA.record.oaCategory();
```

2. Fill out the properties for the object, see Objects.

```
category.id = 79;
```

3. Place the object into an array of objects, see Arrays.

```
// To turn an object into an array of object, simply place it inside square brackets
var objects = [category]; // or just pass [category]
```

4. Pass the objects as a parameter to the NSOA.wsapi.add(objects).

```
var results = NSOA.wsapi.delete( [category] );
```

5. Check for any errors, see Handling SOAP Errors.

Reading data

You read data from OpenAir by creating a ReadRequest object and passing it to the NSOA.wsapi.read(readRequest) function.

Important: You must specify a **limit** Attribute.

To read data from OpenAir

1. Create an OpenAir Complex Type object with the NSOA.record.<complex type>([id]) function and fill out the properties for the object to specify the search criteria.

```
var user = new NSOA.record.oaUser();
user.nickname = "jsmith";
```

2. Create a **limit** Attribute.

```
var attribute = {
    name  : "limit",
    value : "0,1000"
}
```

3. Create a ReadRequest object and fill out the properties.

```
var readRequest = {
    type : "User",
    method : "equal to", // return only records that match search criteria
    fields : "id, nickname, updated", // specify fields to be returned
    attributes : [ attribute ], // Limit attribute is required; type is Attribute
    objects : [ user ] // One object with search criteria
}
```

4. Pass the ReadRequest object to the NSOA.wsapi.read(readRequest) function.

```
var results = NSOA.wsapi.read(readRequest);
```

5. Check for any errors, see Handling SOAP Errors.
6. Process the results, see ReadResult.

See also the SOAP API — Prevent closing a project with an open issue code sample.

ReadRequest

The **ReadRequest** object is use to specify the required data to return in the NSOA.wsapi.read(readRequest) function.

```
// example read request - assumes attribute and user objects have been defined
var readRequest = {
    type : "User",
    method : "equal to", // return only records that match search criteria
    fields : "id, nickname, updated", // specify fields to be returned
    attributes : [ attribute ], // Limit attribute is required; type is Attribute
    objects : [ user ] // One object with search criteria
}
```

Property	Allowed Values
type	Any OpenAir Complex Type without the oa prefix e.g. “Issue”. See NSOA.record.<complex type>([id]) for the list of types.
method	▪ “all” — Returns all available records. **Note:** Use this cautiously as too many records may be requested for the server or client to handle. ▪ “equal to” — return only records that match search. ▪ “custom equal to” — return associated custom fields. ▪ “not equal to” — return only records that do not match.
fields	Comma separated list of fields to be returned e.g. "id, nickname, updated”. See the OpenAir SOAP API Reference Guide or http://www.openair.com/wsdl.pl?wsdl for the list of fields.
attributes	Array of attribute objects, see Attribute. **Important:** The “limit” attribute is required.
objects	Array of OpenAir Complex Type objects, see NSOA.record.<complex type>([id]).

Attribute

The attribute object is used to set additional criteria in the following NSOA methods:

- NSOA.wsapi.modify(attributes, objects)
- NSOA.wsapi.read(readRequest)
- NSOA.wsapi.upsert(attributes,objects)

The attribute object is simply a pair of **name** and **value** properties.

```
var attribute = {
    name  : "limit",
    value : "10"
}
```

See the table below for valid combinations of name and value.

name	value
“limit”	A single value (e.g. “500”) or range (e.g. “0, 1000”). Single value: "1", "500", "1000" - simply restricts the number of records returned. Range: "0, 1000" - the first integer specifies the offset of the first record to return and the second integer limits the number of records to return. To request data in consecutive batches, only the first part of the limit attribute should be incremented - "0,1000", "1000,1000", "2000,1000", etc. Sequence requests should be submitted until the result comes back empty or has less than 1000 items. See Reading data.
“filter”	▪ “newer-than”

name	value
	▪ "not-exported" ▪ "older-than" **Note:** Options can be placed into a comma separated list e.g. "newer-than,older-than,not-exported".
"update_custom"	Set to "1" to enable the updating of custom fields. See Updating Custom Fields.

Using SOAP Results

There are three types of results that can be returned from a successful **wsapi** SOAP call:

- OpenAir Complex Type object
- Array of ReadResult objects
- Array of UpdateResult objects

ReadResult

The **ReadResult** object is returned as a result of calling the NSOA.wsapi.read(readRequest) function.

Property	Value
objects	Array of OpenAir Complex Type objects, see NSOA.record.<complex type>([id]).
errors	Array of oaError objects.

UpdateResult

The **UpdateResult** object is returned as a result of calling the following functions.

- NSOA.wsapi.add(objects)
- NSOA.wsapi.delete(objects)
- NSOA.wsapi.modify(attributes, objects)
- NSOA.wsapi.upsert(attributes,objects)

Property	Value
id	Internal id of the record created or updated.
errors	Array of oaError objects.
status	▪ "U" — record was updated. ▪ "A" — record was added. ▪ "D" — record was deleted. ▪ "-1" — one or more errors occurred.

Also see Handling SOAP Errors.

Handling SOAP Errors

You should always check that any SOAP API call was successful before using the results.

- For calls to NSOA.record.<complex type>([id]), you just need to check that an object was returned.

```
var category = new NSOA.record.oaCategory();
if( !category )
    // An unexpected error has occurred!
```

- For all other calls you need to check that a result was returned and did not contain any errors.
  This is a two step process:
  - First check that you have an array of responses.

```
if (!result || !result[0])
```

  - If OK, then check if you have an errors property and you have at least one error.

```
else if (result[0].errors !== null && result[0].errors.length > 0)
```

```
// example assuming readRequest has already been defined
var result = NSOA.wsapi.read(readRequest);
// Check for errors
if (!result || !result[0]) {
    // An unexpected error has occurred!
} else if (result[0].errors !== null && result[0].errors.length > 0) {
    // There are errors to handle!
} else {
    // Process the response as expected
}
```

The **errors** property is an array of oaError objects.

See Code Samples for more examples.

oaError

An array of **oaError** objects is returned in the ReadResult and UpdateResult objects.

Note: In this version, only the **code** property is available from user script.

Property	Value
attributes	An array of additional attributes for this complex type.
comment	**More Information** for the error.
text	**Short Message** for the error.
code	**Error code** returned by the SOAP API.

Tip: See **Chapter 9 Appendix A Error Code Listing** in the OpenAir SOAP API Reference Guide for the full list of errors.

See also Error Handling.

Who Am I

You can get information about the currently logged in user by calling the NSOA.wsapi.whoami() function.

The NSOA.wsapi.whoami() function returns an oaUser object.

```
function test() {
    var user = NSOA.wsapi.whoami();
    NSOA.meta.alert( "User id " + user.id + " saved this record");
}
```

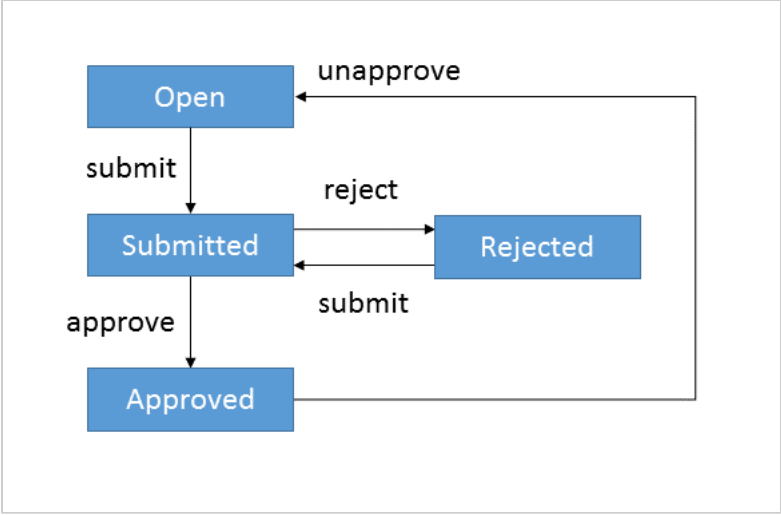
oaUser

The **oaUser** object has more than 100 attributes defining user specific information. See the OpenAir SOAP API Reference Guide for details.

An oaUser object is returned by the NSOA.wsapi.whoami() function.

Scripting Approvals

You can use scripts to submit, approve, reject, and unapprove timesheets, invoices, and envelopes in OpenAir. The approvals workflow is shown below:



Working with the Approvals System

Submitting Timesheets, Envelopes, and Invoices

Submit open or rejected timesheets, envelopes, and invoices which you have rights to in OpenAir by creating an approval object, preparing the record for submission, defining an array of submit requests, and passing the requests to the NSOA.wsapi.submit(submitRequest) function.

submitRequest

The NSOA.wsapi.submit(submitRequest) function takes an array of up to 1000 submitRequest objects. Each submitRequest object contains an oaTimesheet, oaEnvelope, or oaInvoice object to submit and additional approval process information passed to an oaApproval object.

```
// Define the submit requests
var requests = [{
     submit: objectToProcess,
     attributes [], // attributes only apply when submitting timesheets
     approval: approvalObj
}];
```

Property	Allowed Values
submit	**oaTimesheet**, **oaEnvelope**, or **oaInvoice** object
attributes	Only accepts “submit_warning” for oaTimesheet
approval	oaApproval

Approving Timesheets, Envelopes, and Invoices

Approve submitted timesheets, envelopes, and invoices to which you have rights to in OpenAir by creating an approval object, preparing the record for approval, defining an array of approve requests, and passing the requests to the NSOA.wsapi.approve(approveRequest) function.

approveRequest

The NSOA.wsapi.approve(approveRequest) function takes an array of up to 1000 approveRequest objects. Each approveRequest object contains an oaTimesheet, oaEnvelope, or oaInvoice object to approve and additional approval process information passed to an oaApproval object.

```
// Define the approve requests
var requests = [{
     approve: objectToProcess,
     attributes [], // pass an empty array for attributes when using approveRequest
     approval: approvalObj
}];
```

Property	Allowed Values
approve	**oaTimesheet**, **oaEnvelope**, or **oaInvoice** object
attributes	Pass an empty array
approval	oaApproval

Rejecting Timesheets, Envelopes, and Invoices

rejectRequest

The NSOA.wsapi.reject(rejectRequest) function takes an array of up to 1,000 rejectRequest objects. Each rejectRequest object contains an oaTimesheet, oaEnvelope, or oaInvoice object to submit and

additional approval process information passed to an oaApproval object. The **rejectRequest** object is used to specify the required data to return in the NSOA.wsapi.reject(rejectRequest) function.

```
// Define the reject requests
var requests = [{
     reject: objectToProcess,
     attributes [], // pass an empty array for attributes when using rejectRequest
     approval: approvalObj
}];
```

Property	Allowed Values
reject	**oaTimesheet**, **oaEnvelope**, or **oaInvoice** object
attributes	Pass an empty array
approval	oaApproval

Unapproving Timesheets, Envelopes, and Invoices

The NSOA.wsapi.unapprove(unapproveRequest) function takes an array of up to 1,000 unapproveRequest objects. Each unapproveRequest object contains an oaTimesheet, oaEnvelope, or oaInvoice object to unapprove and additional approval process information passed to an oaApproval object. The **unapproveRequest** object is used to specify the required data to return in the NSOA.wsapi.unapprove(unapproveRequest) function.

The NSOA.wsapi.unapprove(unapproveRequest) function takes an array of up to 1,000 unapproveRequest objects. Each unapproveRequest object contains an oaTimesheet, oaEnvelope, or oaInvoice object to unsubmit and additional approval process information passed to an oaApproval object.

unapproveRequest

The **unapproveRequest** object is used to specify the required data to return in the NSOA.wsapi.unapprove(unapproveRequest) function.

```
// Define the unapprove requests
var requests = [{
     unapprove: objectToProcess,
     attributes [], // pass an empty array for attributes when using unapproveRequest
     approval: approvalObj
}];
```

Property	Allowed Values
unapprove	**oaTimesheet**, **oaEnvelope**, or **oaInvoice** object
attributes	Pass an empty array
approval	oaApproval

Using Approval Results

There is one type of result which can be returned from a successful **wsapi** approval call:

- Array of ApprovalResult objects

ApprovalResult

The ApprovalResult object is returned as a result of calling the following functions.

- NSOA.wsapi.submit(submitRequest)
- NSOA.wsapi.approve(approveRequest)
- NSOA.wsapi.reject(rejectRequest)
- NSOA.wsapi.unapprove(unapproveRequest)

Property	Value
id	Internal id of the object for approval action.
approval_warnings	String representing any warnings.
approval_errors	String representing any errors
log	String representing the log of actions.
errors	Array of oaError objects.
approval_status	The approval status of the record - "O" — Open - "S" — Submitted - "A" — Approved - "R" — Rejected - "X" — Archived

Also see Handling Approval Errors.

Handling Approval Errors

You should always check that any approval API call was successful before using the results.

- For calls to NSOA.wsapi.submit(submitRequest), NSOA.wsapi.approve(approveRequest), NSOA.wsapi.reject(rejectRequest), and NSOA.wsapi.unapprove(unapproveRequest), you should check that a result was returned and did not have any errors.

This is a two-step process:

- First, check that you have an array of responses.

```
if (!result || !result[0])
```

- If OK, then check if you have an errors property and you have at least one error.

```
else if (result[0].errors !== null && result[0].errors.lenth > 0)
```

The following example checks for errors when using the NSOA.wsapi.approve(approveRequest) function:

```
// example assuming requests have already been defined
var results = NSOA.wsapi.approve(requests);
```

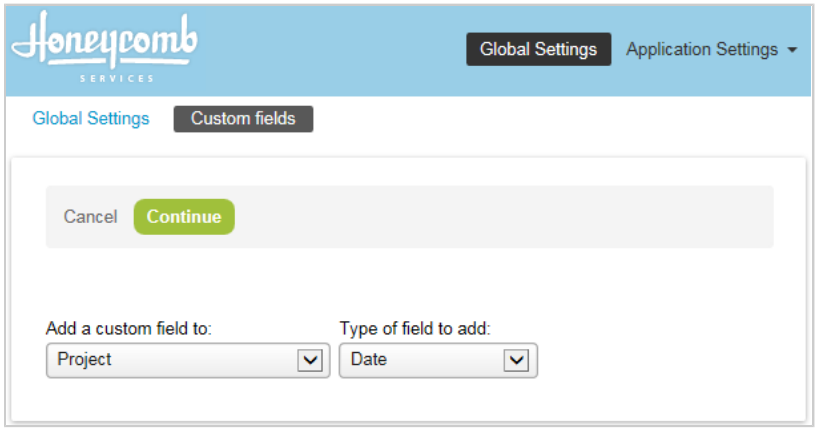
```
// Check for errors
if (!result || !result[0]) {
     // An unexpected error has occurred!
} else if (result[0].errors !== null && result[0].errors.length > 0) {
     // There are errors to handle!
} else {
     // Process the response as expected
}
```

The **errors** property is an array of oaError objects.

See Code Samples for more examples.

Custom Fields

Creating Custom Fields



To create a Custom Field:

1. Go to Administration > Global Settings > Custom Fields.
2. Select **New Custom field** from the **Create Button**.
3. Select the entity the custom field is associated with along with the type of field you are creating. Click Continue.
4. Type the Field name. This is required. The name cannot have any spaces, but you can use underscores.
5. Select the Active check box.
6. Type a Description. This is optional and is used for adding information about the new custom field.
7. Type the Display name. This is what displays on the form associated with the entity.
8. If desired, type a Hint to help your OpenAir employees understand the intent of the custom field.
9. If you select the Required check box, the field is required on the form and the form cannot to be saved without supplying a value.
10. If you select the Unique check box, a unique value must be entered in the field to be able to save the form.
11. If you select the check box to Hide on data entry forms, this custom field does not display on the form.

12. If you select the check box for Add Notes, a text box displays under the custom field for employees to add any additional notes.
13. If you select the check box for Divider, a divider line displays before the custom field. You can also type Divider text that displays in the Divider Line.

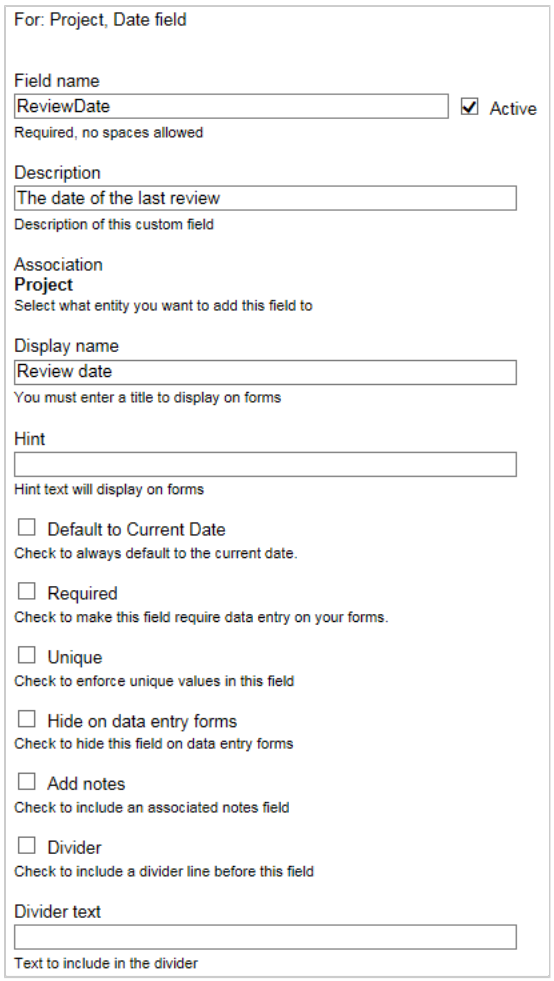
> **Note:** You may want to use Divider lines when you are defining a new section that needs to stand out on the form. For example, a series of custom fields defining a topic such as contract management may start with a Contract received check box. The divider line indicates the start of the contract management fields.

14. Click Save. Once you save the form, a Position field displays. Position determines the order of the custom field on the entity's form. To change the position, adjust the value using the drop-down list and click Save.

See the OpenAir Admin Guide for more details on creating customer fields.

> **Tip:** If you have added a new custom field and this is not listed in the Form schema of the Scripting Studio, open the form with the new custom field to refresh the custom field list, and then open the Scripting Studio again.

Example Date field for Project forms

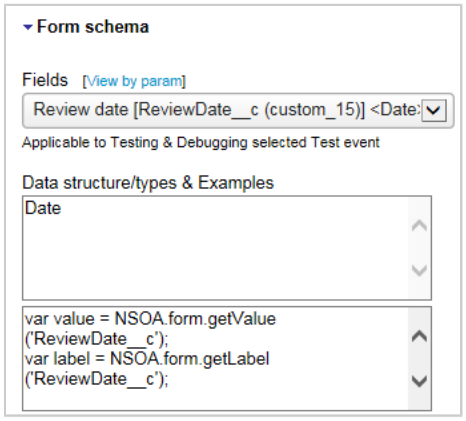


Note: This custom field is referred to in the code examples that follow.

The custom field will then be visible on the project form.

Reading Custom Fields

Use the Form schema to find the correct field name for the custom field.



You can read the custom field value and label in the same way as for standard fields.

```
// Read the date value and log the value if the date is not empty
function logReviewDate(){
    var reviewDate = NSOA.form.getValue('ReviewDate__c');
    if( reviewDate !== null ) {
        NSOA.meta.alert(reviewDate.toString());
    }
}
```

Note: The old approach to read custom fields using **custom_** with the internally assigned custom field number appended is still supported but NOT recommended.

```
// Supported but NOT RECOMMENDED
var reviewDate = NSOA.form.getValue('custom_15');
}
```

To read a custom field value using record functions

1. Create an OpenAir record object with the NSOA.record.<complex type>([id]), NSOA.form.getNewRecord() or NSOA.form.getOldRecord() functions.

```
var proj = NSOA.form.getOldRecord(); // Call on  'After save' event
```

2. The custom field name is the **Field name** defined for the custom field with the special **'__c'** suffix appended to identify it as a custom field.

```
// custom field name = 'ReviewDate' + '__c';
```

3. Use the name to access the custom field value in the record object.

```
var reviewDate = proj.ReviewDate__c;
```

See also Updating Custom Fields.

Updating Custom Fields

For **NSOA.wsapi** functions, the name of the custom field is the **Field name** defined for the custom field with the special **'__c'** suffix appended to identify it as a custom field. It is also necessary to explicitly enable custom field updating.

Important: It is not possible to rename, change, or delete a custom field which is being used by an active script. This prevents unintended script problems.

To update a custom field

1. Create an OpenAir record object with the NSOA.record.<complex type>([id]), NSOA.form.getNewRecord() or NSOA.form.getOldRecord() functions.

```
var updProj = NSOA.form.getNewRecord(); // Get the record to modify
var recProj = new NSOA.record.oaProject(); // Record to specify just the values to update
recProj.id = updProj.id; // We need the id to update the correct record
```

2. Use the correct name format for the custom field, i.e. **Field name** defined for the custom field + **'__c'**.

```
recProj.ReviewDate__c = '2014-01-16'; // Notice the date format YYYY-MM-DD
```

3. The 'update_custom' Attribute must be specified.

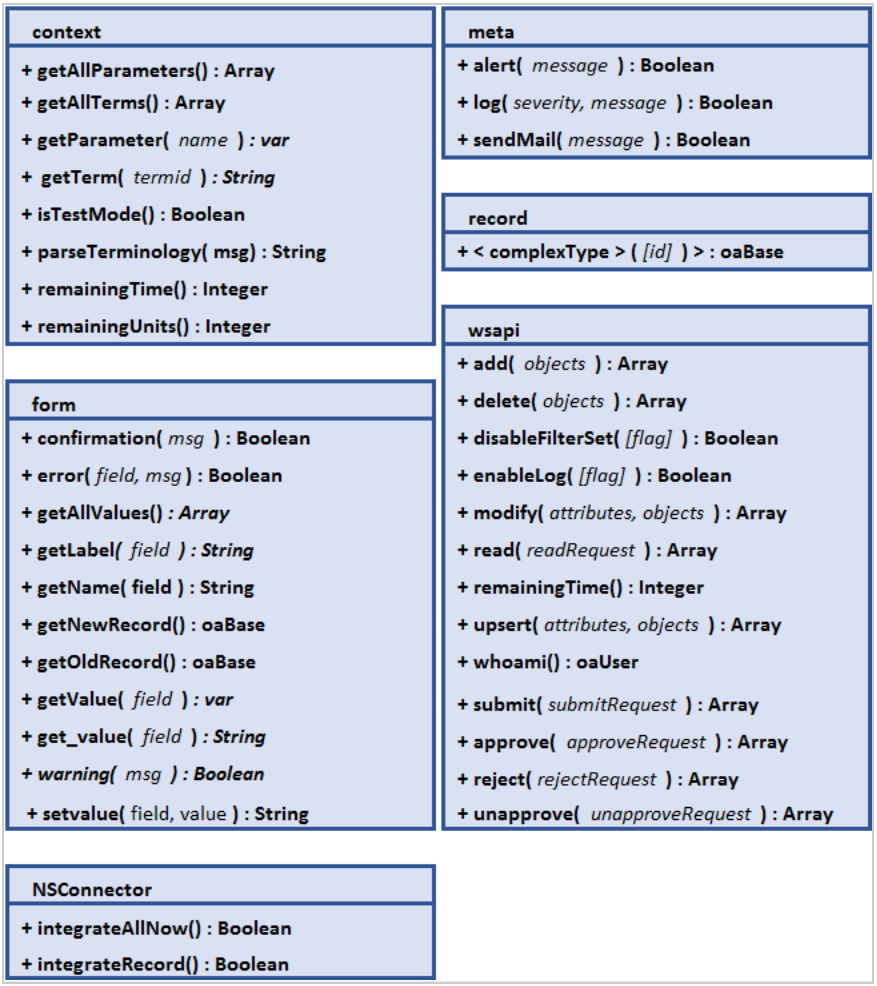
```
var attribute = {
    name  : 'update_custom',
    value : "1"
}
```

4. Call NSOA.wsapi.modify(attributes, objects).

```
var projResults = NSOA.wsapi.modify([attribute], [recProj]);
```

5. Check for any errors, see Handling SOAP Errors.
6. Process the results, see UpdateResult.

NSOA Functions



The following functions are provided to allow you to interact with OpenAir:

- NSOA.context
  - NSOA.context.getAllParameters()
  - NSOA.context.getAllTerms()
  - NSOA.context.getParameter(name)
  - NSOA.context.getTerm(termid)
  - NSOA.context.isTestMode()
  - NSOA.context.parseTerminology(message)
  - NSOA.context.remainingTime()
  - NSOA.context.remainingUnits()
- NSOA.form
  - NSOA.form.confirmation(message)
  - NSOA.form.error(field, message)
  - NSOA.form.getAllValues()

- NSOA.form.getLabel(field)
- NSOA.form.getName(field)
- NSOA.form.getNewRecord()
- NSOA.form.getOldRecord()
- NSOA.form.getValue(field)
- NSOA.form.get_value(field)
- NSOA.form.setValue(field, value)
- NSOA.form.warning(message)

- NSOA.meta
  - NSOA.meta.alert(message)
  - NSOA.meta.log(severity, message)
  - NSOA.meta.sendMail(message)
- NSOA.record
  - NSOA.record.<complex type>([id])
- NSOA.wsapi
  - NSOA.wsapi.add(objects)
  - NSOA.wsapi.delete(objects)
  - NSOA.wsapi.disableFilterSet([flag])
  - NSOA.wsapi.enableLog([flag])
  - NSOA.wsapi.modify(attributes, objects)
  - NSOA.wsapi.read(readRequest)
  - NSOA.wsapi.remainingTime()
  - NSOA.wsapi.upsert(attributes,objects)
  - NSOA.wsapi.whoami()
  - NSOA.wsapi.submit(submitRequest)
  - NSOA.wsapi.approve(approveRequest)
  - NSOA.wsapi.reject(rejectRequest)
  - NSOA.wsapi.unapprove(unapproveRequest)
- NSOA.NSConnector
  - NSOA.NSConnector.integrateRecord()
  - NSOA.NSConnector.integrateAllNow()

NSOA.context.getAllParameters()

Use this function to get an Associative Array of all the script parameters and values set for the script.

See Script parameters.

Parameters

- (none)

Returns

- An Associative Array of all the script parameters and values for the script.

Units Limit

- 10 units

For more information, see Scripting Governance.

Since

- April 18, 2015

Example

- This example creates a local variable called **allParams** with an Associative Array of all the script parameters and values for the script. It then uses a for in loop to log each parameter name and current value.

```
// Get all the parameters available for the script
var allParams = NSOA.context.getAllParameters();

// Loop through all the parameters
for (var key in allParams) {
    NSOA.meta.alert(key + ' has value ' + allParams[key]);
}
```

See NSOA.meta.alert(message).

See also NSOA.context.getParameter(name).

NSOA.context.getAllTerms()

Use this function to get an Associative Array of all the terminology identifiers and values set for the account.

See Script Terminology.

Parameters

- (none)

Returns

- An Associative Array of all the terminology identifiers and values for the account.

Units Limit

- 10 units

For more information, see Scripting Governance.

Since

- April 18, 2015

Example

- This example creates a local variable called **allTerms** with an Associative Array of all the terminology and values for the account. It then uses a for in loop to log each term and current value.

```
// Get all the terminology available for the script
var allTerms = NSOA.context.getAllTerms();

// Loop through all the terminology
for (var key in allTerms) {
    NSOA.meta.alert(key + ' has value ' + allTerms[key]);
}
```

See NSOA.context.parseTerminology(message) and NSOA.context.getTerm(termid).

See also Accessing Terminology.

NSOA.context.getParameter(name)

Use this function to get the value set for the specified parameter.

See Script Parameters.

Parameters

- *name* {string} [required] — The name of the parameter.

> **Note:** Use the Script parameters section in the Scripting Studio or the Scripting Center to lookup the parameter name to use.

Returns

- The value of the specified parameter.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

- April 18, 2015

Example

- This example shows a field value being checked against a parameter value.

```
// return if new stage is not closed
if (NSOA.form.getValue('project_stage_id') !=
    NSOA.context.getParameter('ProjectClosedStage'))
return;
```

See Prevent closing a project that has open issues.

NSOA.context.getTerm(termid)

Use this function to get the term used for the specified terminology identifier.

See Script Terminology.

Parameters

- *termid* {string} [required] — The internal identifier for the term.

> **Note:** Use the Terminology section in the Scripting Studio to lookup the parameter names to use.

Returns

- The term used for the specified terminology identifier.

Units Limit

- 0 units

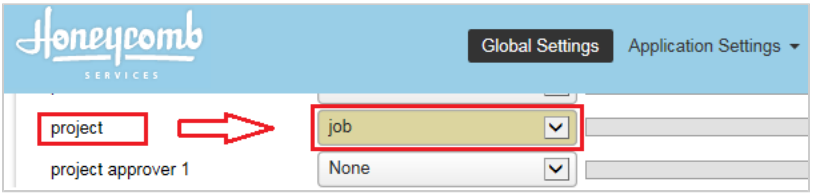
For more information, see Scripting Governance.

Since

- April 18, 2015

Example

- This example shows what would be returned if the account terminology had redefined project to job.



```
var proj_term = NSOA.context.getTerm('Projects');
// proj_term = "Jobs"
```

See NSOA.context.parseTerminology(message) and NSOA.context.getAllTerms().

See also Accessing Terminology.

NSOA.context.isTestMode()

Use this function to determine if the script is being run in test mode.

For more information see Testing & debugging.

Parameters

- (none)

Returns

- Boolean **true** if the script is running in test mode and **false** otherwise.

Units Limit

- 0 units

For more information, see Scripting Governance.

Since

- November 16, 2013

Example

- This example shows some code that only runs in "Test mode", e.g. an assertion.

```
if(NSOA.context.isTestMode() && someVar==null)
    throw new Error("someVar should never be null");
```

NSOA.context.parseTerminology(message)

Use this function to convert a string containing terminology phrases (terminology identifiers surrounded by '%' characters) into a string using the correct terminology set for the account.

See Script Terminology.

Parameters

- *message* {string} [required] — The message containing terminology phrases to replace with terms used in the account.

Returns

- The passed string with all the terminology phrases replaced by the terms used in the account.

Units Limit

- 4 units

> **Note:** Calls to NSOA.meta.log(severity, message) with the severity parameter set to "debug" or "trace" do not consume units but are limited to a maximum of 1000 per script.

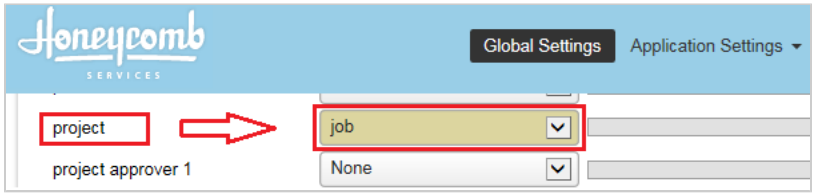
For more information, see Scripting Governance.

Since

- April 18, 2015

Example

- This example shows what would be returned if the account terminology had redefined project to job.




```
var msg = NSOA.context.parseTerminology("Notes attached to %project%.")
// msg = "Notes attached to job.";
```

See NSOA.context.getTerm(termid) and NSOA.context.getAllTerms().

See Accessing Terminology.

NSOA.context.remainingTime()

Use this function to determine how much time your script has remaining to execute (excluding wsapi call time) before it is terminated by Scripting Governance.

You can use this function to help you create more efficient scripts and also to take corrective action if a script is at risk of consuming excessive resources.

Parameters

- (none)

Returns

- Amount of time remaining allowed for the script to execute in milliseconds (excluding wsapi call time).

Tip: Always try to reduce the amount of time your scripts take to execute.

Units Limit

- 0 units

For more information, see Scripting Governance.

Since

- October 18, 2014

Example

- This example logs the amount of time remaining for the script to execute in milliseconds (excluding wsapi call time).

```
NSOA.meta.log('info', 'Remaining script time: '
        + NSOA.context.remainingTime() + ' milliseconds');
```

See also NSOA.wsapi.remainingTime().

For more information see Scripting Governance.

NSOA.context.remainingUnits()

Use this function to determine how many units your script has left before it will be halted by the system. Each script is allowed to consume a maximum of 1000 units.

For more information see Scripting Governance.

Parameters

- (none)

Returns

- The number of units remaining.

Tip: Always try to reduce the number of units your scripts consume. Notice that NSOA.record functions consume zero units, but NSOA.wsapi functions consume 10 units for each call.

Units Limit

- 0 units

For more information, see Scripting Governance.

Since

- August 17, 2013

Example

- This example displays the number of units consumed at the top of the form as an error message.

```
NSOA.form.error('','Units consumed: ' + NSOA.context.remainingUnits());
```

See also NSOA.form.error(field, message).

For more information see Scripting Governance.

NSOA.form.confirmation(message)

Use this function to print a confirmation message on the OpenAir form. The message that appears will look exactly like the OpenAir system-generated confirmation messages.

Note: This function will only have an affect on the **After save** and **After approval** events, see Events.

Parameters

- *message* {string} [required] — The confirmation message to display on the form.

Note: This message will be displayed instead of the system-generated confirmation message for the form.

Returns

- True if the function was successful and false otherwise.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

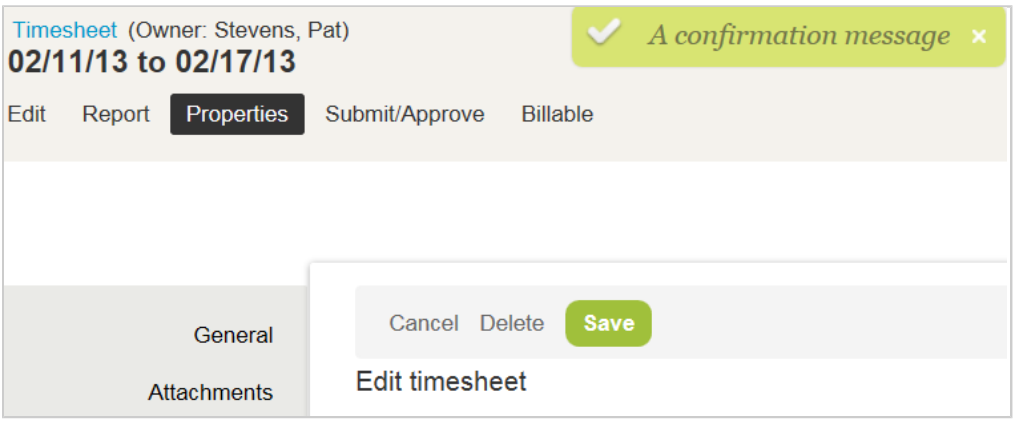
- October 17, 2015

Example

- This example displays the confirmation message 'A confirmation message' at the top of the form after the form is saved.

```
NSOA.form.confirmation("A confirmation message");
```

The message appears as a OpenAir system-generated confirmation.



See Code Samples for more examples.

NSOA.form.error(field, message)

Use this function to print an error message associated to the selected form field on the OpenAir form. The first argument is the field name on the form where you want the message to show up. The message that appears will look exactly like the OpenAir system-generated errors.

Note: The is function has no affect on the **After save** form event, see Events.

Parameters

- *field* {string} [required] — The name of the field on the form to display the error next to, or an empty string to display the message at the top of the form.

  Note: This is not the label the user sees displayed next to the field on the form. Use the Form schema to find the correct field name value.

- *message* {string} [required] — The error message to display on the form.

Returns

- True if the function was successful and false otherwise.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

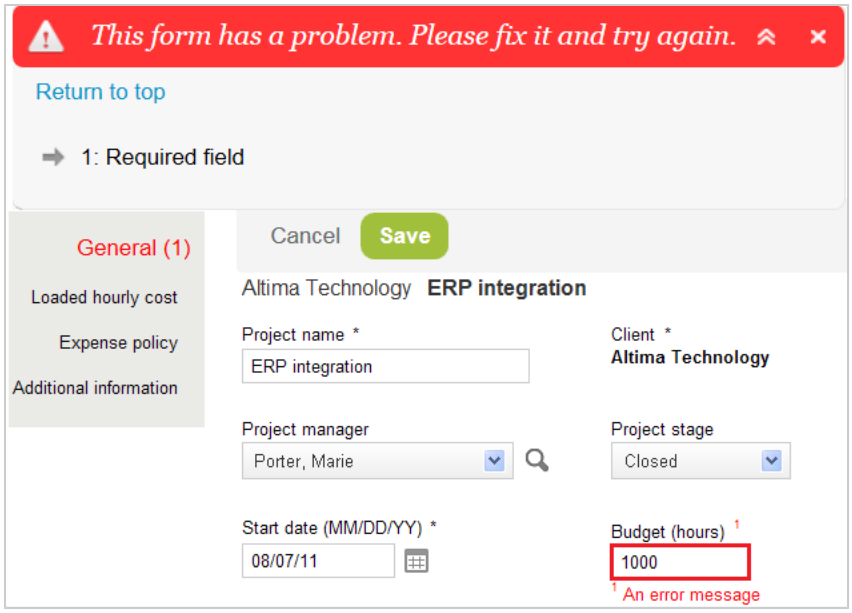
- August 17, 2013

Example

- This example displays the error message 'An error message' next to the budget_time field.

```
NSOA.form.error('budget_time', "An error message");
```

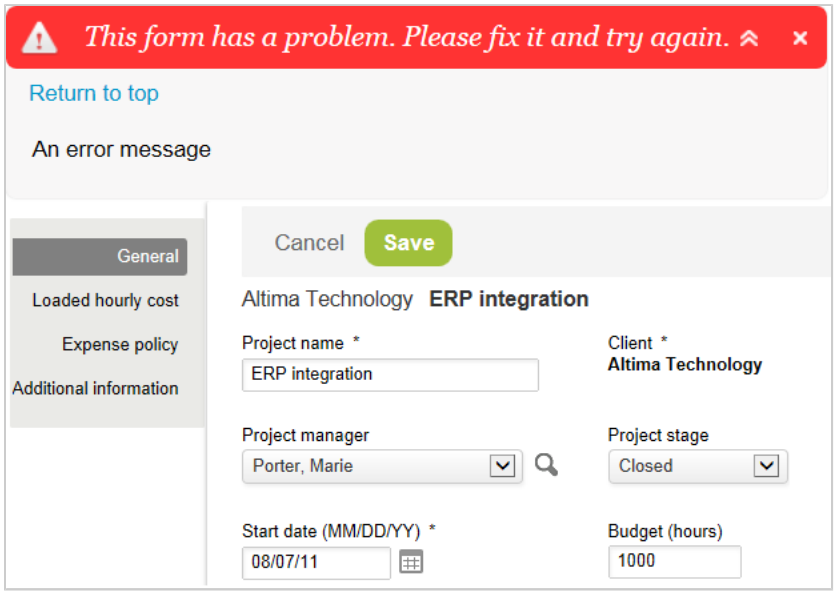
The message appears as a OpenAir system-generated error.



- This example displays the error message 'An error message' at the top of the form.

```
NSOA.form.error('', "An error message");
```

The message appears as a OpenAir system-generated error.



See Code Samples for more examples.

NSOA.form.getAllValues()

Use this function to get an Associative Array of all the fields and values on the OpenAir form. Keep in mind, any pick lists (e.g. Customer:Project, Employee, Expense item) will return an internal_id and not a text value. In this release, only fields directly related to the form are available (e.g. no related table lookups are available at this time). See also NSOA.form.getValue(field).

Parameters

- (none)

Returns

- An Associative Array of all the fields and values on the form. Use the Form schema to find the names and data types returned.

  Note: Some fields return an object. See Object Fields for more details.

Units Limit

- 10 units

For more information, see Scripting Governance.

Since

- August 17, 2013

Example

- This example creates a local variable called **allValues** with an Associative Array of all the fields and values on the form. It then reads the project_name and start_date from the **allValues** variable.

```
var allValues = NSOA.form.getAllValues();

var project_name = allValues.name;     // equivalent to getValue('name');
var start_date = allValues.start_date; // equivalent to getValue('start_date');
```

  See also NSOA.form.getValue(field).

  Note: Some fields return an object. See Object Fields for more details.

- You can loop through the keys of an associative array with the for in loop.

```
// Get all the values on the fields on the form
var allValues = NSOA.form.getAllValues();

//Loop through all the values
for( var key in allValues ) {
    NSOA.meta.alert(key + ' has value ' + allValues[key]);
```

```
}
```

See NSOA.form.getAllValues() and NSOA.meta.alert(message).

See Code Samples for more examples.

NSOA.form.getLabel(field)

Use this function to get the label the user sees for a field on the OpenAir form.

Parameters

- *field* {string} [required] — The name of the field on the form.

> **Note:** This is not the label the user sees displayed next to the field on the form. Use the Form schema to find the correct field name value.

Returns

- The text value the users sees for specified field.

> **Note:** Some fields return an object. See Object Fields for more details.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

- August 17, 2013

Example

- This example gets the label for the **date** field on the form the script is attached to.

```
    var receiptDateLabel = NSOA.form.getLabel('date');
```

- This example gets the label for a field that returns an object. See Object Fields for more details.

```
// 'Primary loaded cost ' for the first row
var label = NSOA.form.getLabel('loaded_cost')[0].cost_0;
```

See Code Samples for more examples.

NSOA.form.getName(field)

Use this function to get the parameter name of the field.

Note: This is generally the same as the field name i.e. the required parameter to call this function. The function is useful when working with Object Fields.

Parameters

- *field* {string} [required] — The name of the field on the form.

  Note: This is not the label the user sees displayed next to the field on the form. This is the name of the field displayed in the Form schema.

Returns

- The parameter name needed to refer to this field in user scripts.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

- August 17, 2013

Example

- In this example the name returned is the same as the field name passed in i.e. **budget_time**.

```
var name = NSOA.form.getName('budget_time');
```

- In this example the name is the field name for the row and column specified for the **loaded_cost** object. See Object Fields for more details.

```
// 'Primary loaded cost ' for the first row
var name = NSOA.form.getName('loaded_cost')[0].cost_0;
```

See Code Samples for more examples.

NSOA.form.getNewRecord()

Use this function to get the entity record for a form with the newly saved values, e.g. oaProject.

This function should be called on the **After save event**, see Events.

See also NSOA.form.getOldRecord().

Parameters

- (none)

Returns

- OpenAir Complex Type object, see NSOA.record.<complex type>([id]).

Note: This function will return **null** if called before the form has been saved.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

- November 16, 2013

Example

- This example modifies the project notes after the project has been saved.

Note: This script would be called on the "After save" event for the Project form

```
    // Get the new record values
    var newr = NSOA.form.getNewRecord();

    // Create a new record with field to modify
    var project = new NSOA.record.oaProject();
    project.id = newr.id; // Need to specify the internal id
    project.notes = newr.notes + "\nAppended to notes: " + (new Date().toString()); // New valu
e for field

    // Modify the notes
    NSOA.wsapi.disableFilterSet(true); // Drop user filters - make this a generic script
    var arrayOfupdateResult = NSOA.wsapi.modify([], [project]);
```

Note: This simple example does not show error checking, see Handling SOAP Errors.

See Code Samples for more examples.

NSOA.form.getOldRecord()

Use this function to get the entity record for a form with the current (not yet saved) values, e.g. oaProject.

See also NSOA.form.getNewRecord().

Tip: An Entrance Function can optionally receive a **type** string parameter. Check if the value of this parameter is 'update' to determine if the form is being modified.

Parameters

- (none)

Returns

- OpenAir Complex Type object, see NSOA.record.<complex type>([id]).

> **Note:** This function will return **null** if called for a form that is being created.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

- November 16, 2013

Example

- This example checks to see if the project name has been modified.

```
var oldr = NSOA.form.getOldRecord();
var newr = NSOA.form.getNewRecord();
if (oldr.name !== newr.name)
    NSOA.meta.alert("Project name changed to: " + newr.name);
```

> **Note:** This simple example does not show error checking, see Handling SOAP Errors.

See Code Samples for more examples.

NSOA.form.getValue(field)

Use this function to get the value of the field on the OpenAir form. Keep in mind, any pick lists (e.g. Customer:Project, Employee, Expense item) will return an internal_id and not a text value. In this release, only fields directly related to the form are available (e.g. no related table lookups are available at this time). See also NSOA.form.getAllValues() and NSOA.form.get_value(field).

Parameters

- *field* {string} [required] — The name of the field on the form.

> **Note:** This is not the label the user sees displayed next to the field on the form. Use the Form schema to find the correct field name value.

Returns

- The value in the specified field. Use the Form schema to find the data type of the returned value.

Note: Some fields return an object. See Object Fields for more details.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

- August 17, 2013

Example

- This example creates a local variable called **receiptDate** and sets its value to the content of the **date** field on the form the script is attached to.

```
var receiptDate = NSOA.form.getValue('date');
```

- This sample gets a value from a field that returns an object. See Object Fields for more details.

```
// First get the object variable for the field and then get the cost_0 value for the first row
var loaded_cost_obj = NSOA.form.getValue("loaded_cost");
var value = loaded_cost_obj[0].cost_0;

//  You can combine these two steps into one line
var value = NSOA.form.getValue("loaded_cost")[0].cost_0;
```

See Code Samples for more examples.

NSOA.form.get_value(field)

Use this function to get the value of the field on the OpenAir form. Keep in mind, any pick lists (e.g. Customer:Project, Employee, Expense item) will return an internal_id and not a text value. In this release, only fields directly related to the form are available (e.g. no related table lookups are available at this time).

Note: You are recommend to use NSOA.form.getValue(field) or NSOA.form.getAllValues() in preference to using **NSOA.form.get_value(field)**.

Parameters

- *field* {string} [required] — The name of the field on the form.

Returns

- The value of the field on the form as a string.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

- March 17, 2012

Example

- This example creates a local variable called **receiptDate** and sets its value to the content of the **date** field on the form the script is attached to.

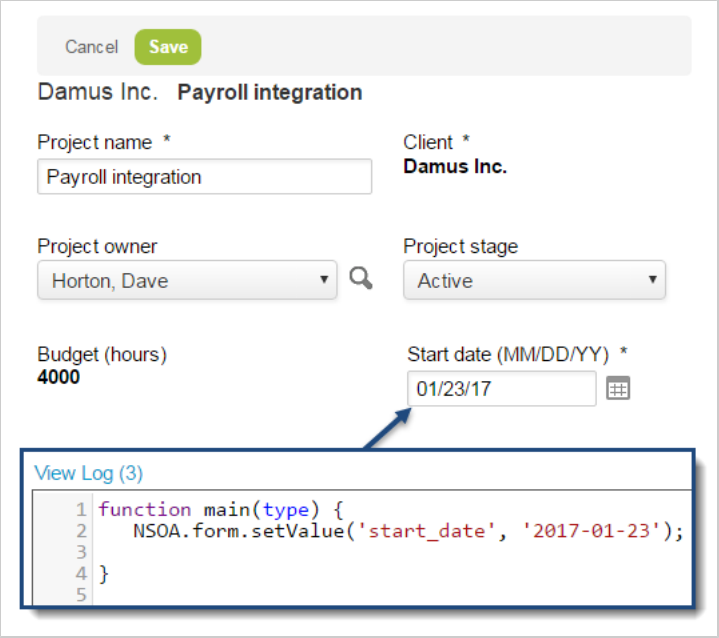
```
var receiptDate = NSOA.form.get_value('date');
```

See also NSOA.form.getValue(field) and NSOA.form.getAllValues().

See Code Samples for more examples.

NSOA.form.setValue(field, value)

Use this command to set form values on the submit scripting event and to update values as part of the main form save, without needing to write WSAPI (SOAP) calls. The effect is the same as a user making manual changes to a field.



Full validation from your other scripts or rules is applied after the changes are made, ensuring your changes are safe. Form default values are applied before the script is run, and any permission rules or "After save" scripts are applied after the "On submit" script runs.

Error messages can be raised on the submit event. If errors are raised, the script will still run to completion, and the errors will be logged.

The function takes two parameters:

- The field you want to change

  Note: SetValue supports changing the values in text fields, including text areas, dates, and numeric fields, as well as checkboxes. Dropdown lists, pick lists, password or sequence fields, radio groups, and multiple selections are not supported.

- The value to set in the field (either literal or variable)

See Examples for individual use cases.

Parameters

- field {string} [required] — The name of the field on the form to set the value for
- value {permitted value type for field} [required] — The value to set in the field. May be text, numbers, date values or ISO-8601-formatted strings, the NSOA.form.getValue command, true or false, or null values, depending on the field affected.

Returns

- True if the function was successful and false otherwise. If the function fails, it writes a descriptive message to the script log.

Units Limit

- 1 unit

Since

April 15, 2017

Examples

- This example enters a text string into a "Notes" field.

```
NSOA.form.setValue('notes', 'Note text here');
```

- To clear a text field, use an empty string in the second parameter.

```
NSOA.form.setValue('notes', '');
```

  Note: The example above only uses single quotes, not double quotes.

  Setting the value to **null** clears the text field.

```
NSOA.form.setValue('notes', null);
```

  This example sets a “Notes” field using the getValue command.

Note: NSOA.form.setValue supports both the new (prj_custpo_num__c) and old (custom_24) methods of referencing custom fields. When creating portable scripts, always use the new format for referencing custom fields.

```
NSOA.form.setValue('notes', NSOA.form.getValue('name'));
```

- SetValue also works with number fields. Number fields accept number values, but not strings. Number values must be written using the base U.S. number format, for example, "23.67". Number values are displayed according to the user's settings in Regional Settings > Number format.

```
NSOA.form.setValue("prj_sales_rep_ratio_1__c", 23.67);
```

Use a null value to clear a number field using setValue.

```
NSOA.form.setValue("prj_sales_rep_ratio_1__c", null);
```

- SetValue can set the value of <date> fields using date values or ISO-8601-formatted strings, for example, YYYY-MM-DD. Date values are displayed according to the user's settings in Regional Settings > Date format.

Note: An error is logged when you attempt to set a date field with an invalid date string.

```
NSOA.form.setValue('start_date', '2017-01-23');
```

Use a null value to clear a date field:

```
NSOA.form.setValue('start_date', null);
```

- Use **true** or **false** values to set checkboxes.

```
NSOA.form.setValue("active", true);
```

Use **false** to clear a checkbox:

Note: If you use setValue to set a null value for a checkbox, an error will be logged.

```
NSOA.form.setValue("active", false);
```

NSOA.form.warning(message)

Use this function to print a warning message on the OpenAir form. The message that appears will look exactly like the OpenAir system-generated warning messages.

Note: This function will only have an affect on the **After save** and **After approval** events, see Events.

Parameters

- *message* {string} [required] — The warning message to display on the form.

Note: This message will be displayed instead of the system-generated warning message for the form.

Returns

- True if the function was successful and false otherwise.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

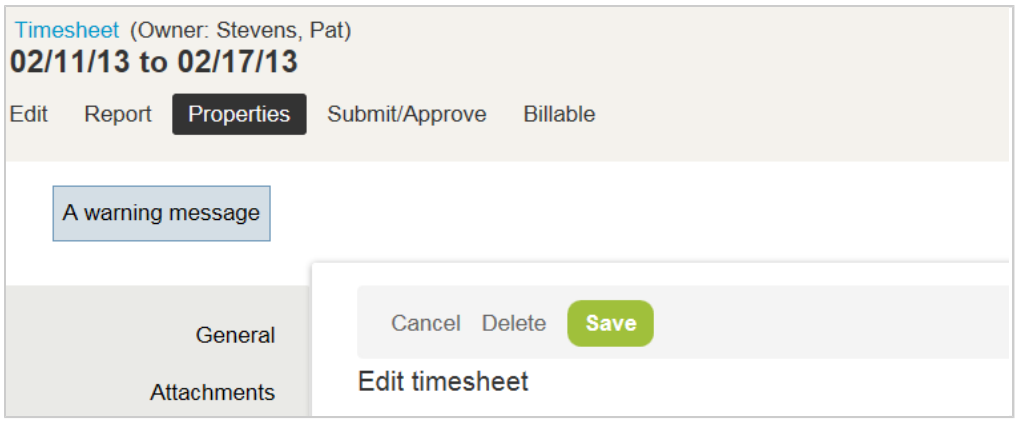
- October 17, 2015

Example

- This example displays the warning message 'A warning message' at the top of the form after the form is saved.

```
NSOA.form.warning("A warning message");
```

The message appears as a OpenAir system-generated warning.



See Code Samples for more examples.

NSOA.meta.alert(message)

Use this function to store an **Info** log entry. This is a short version of NSOA.meta.log(severity, message).

Parameters

- *message* {string} [required] — The message to be written to the log.

Returns

- True if the function was successful and false otherwise.

Units Limit

- 4 units

For more information, see Scripting Governance.

Since

- August 17, 2013

Example

- This sample writes the 'info' severity message 'Form error - travel date is after receipt date' to the log.

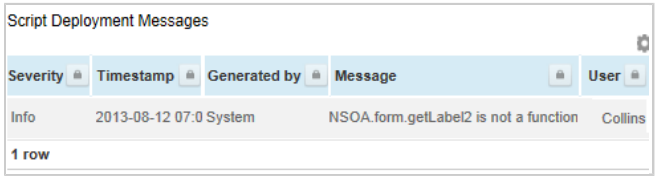
```
NSOA.meta.alert('Form error - travel date is after receipt date');
```

See also NSOA.meta.log(severity, message).

See Code Samples for more examples.

NSOA.meta.log(severity, message)

Use this function to store a log entry. The supported severities match those of the Log4j project.



The log indicates:

- **Severity** — The supplied severity: "fatal", "error", "warning", "info", "debug", or "trace".
- **Timestamp** — The time the message was logged.
- **Generated by** — For example, whether the message was generated by your script or the system
- **Message** — The full message text.
- **User** — For example, the user that was saving the form when the error happened.

> **Note:** If you have a syntax error or a runtime error you will see an error in the log generated by the system.

See also Form script deployment logs.

Parameters

- *severity* {string} [required] — The severity of the message: “fatal”, “error”, “warning”, “info”, “debug”, or “trace”.

  > **Note:** The “debug” and “trace” messages are only executed in test mode, see Testing & debugging. The “debug”, and “trace” messages do not consume Scripting Governance units but are limited to a maximum of 1000 per script.

- *message* {string} [required] — The message to be written to the log.

Returns

- True if the function was successful and false otherwise.

Units Limit

- 4 units

For more information, see Scripting Governance.

Since

- August 17, 2013

Example

- This sample writes the ‘error’ severity message 'Form error - travel date is after receipt date' to the log.

```
NSOA.meta.log('error', 'Form error - travel date is after receipt date');
```

  See also NSOA.meta.alert(message).

See Code Samples for more examples.

NSOA.meta.sendMail(message)

Use this function to send email messages from a form, library, or scheduled script. Form scripts are allowed to send a maximum of 3 emails and scheduled scripts a maximum of 100 email by Scripting Governance.

Parameters

- *msg* {object} [required] — An email message object with the following properties:
  - **to** — [optional] array of OpenAir User IDs / email addresses.
  - **cc** — [optional] array of OpenAir User IDs / email addresses.
  - **bcc** — [optional] array of OpenAir User IDs / email addresses.

> **Important:** At least one of to, cc, or bcc is required.

- **format** — [optional] if "HTML" the body will be treated as HTML. If this is set to any other value or omitted then the body will be treated as plain text.
- **subject** — [optional] string holding the email subject. The subject is trimmed to the first line if carriage return characters are used.

> **Important:** At least one of subject or body is required.

- **body** — [optional] the body has a maximum length of 30,000 characters. Governance will terminate the sendMail if this length is exceeded.

> **Tip:** If the **format** is set to "HTML" any tags you can place within the <body></body> section of an HTML file are valid.

- **author** — [optional] use to set one OpenAir user ID as the author of the emails.

See Code Samples for more examples.

Returns

- True if the email was placed in the queue for sending and false otherwise.

Units Limit

- 10 units

For more information, see Scripting Governance.

Since

- October 17, 2015

Example

- This sends a plain text email.

```
// Send a plain text message
var msg = {
    to: ["mcollins@openair.com"],
    cc: ["jadmin@openair.com"],
    subject: "Script alert",
    body: "Form saved"
};

NSOA.meta.sendMail(msg);
```

- This sends an HTML email.

```
// Send an HTML message
var msg = {
    to: ["mcollins@openair.com"],
```

```
        subject: "Project Assignment",
        format: "HTML",
        body: "<b>Client:</b> Altima Technologies</br>" +
              "<b>Project:</b> CRM Implementation</br>" +
              "<b>Project Manager:</b> Collins, Marc"
    };

    NSOA.meta.sendMail(msg);
```

See Code Samples for more examples.

NSOA.record.<complex type>([id])

This set of functions is used to create OpenAir Complex Type objects. If the Internal id is passed as a parameter then the object will be populated accordingly. The following objects are supported:

oaAddress	**oaEstimatephase**	**oaPurchaser**
oaAgreement	**oaEvent**	**oaPurchaserequest**
oaApproval	**oaHierarchy**	**oaRatecard**
oaBooking	**oaHierarchyNode**	**oaReimbursement**
oaBookingType	**oaHistory**	**oaRequest_item**
oaBudget	**oaImportExport**	**oaResourceprofile**
oaBudgetAllocation	**oaInvoice**	**oaResourceprofile_type**
oaCategory	**oaItem**	**oaRevenue_recognition_rule**
oaCcrate	**oaJobcode**	**oaRevenue_recognition_rule_amount**
oaCompany	**oaLeave_accrual_rule**	**oaRevenue_recognition_transaction**
oaContact	**oaLeave_accrual_rule_to_user**	**oaSchedulerequest**
oaCostcenter	**oaLeave_accrual_transaction**	**oaSchedulerequest_item**
oaCurrency	**oaLoadedCost**	**oaSlip**
oaCurrencyrate	**oaModule**	**oaSlipstage**
oaCustField	**oaPayment**	**oaSwitch**
oaCustomer	**oaPaymentterms**	**oaTask**
oaCustomerpo	**oaPaymenttype**	**oaTaskTimecard**
oaCustomerpo_to_project	**oaPayrolltype**	**oaTaxLocation**
oaCustomField	**oaPreference**	**oaTaxRate**
oaDate	**oaProduct**	**oaTerm**
oaDeal	**oaProject**	**oaTicket**
oaDealcontact	**oaProjectbillingrule**	**oaTimecard**
oaDealschedule	**oaProjectbillingtransaction**	**oaTimesheet**
oaDepartment	**oaProjectlocation**	**oaTimetype**

oaEntitytag	**oaProjectstage**	**oaTodo**
oaEnvelope	**oaProjecttask**	**oaUprate**
oaError	**oaProjecttask_type**	**oaUser**
oaEstimate	**oaProjecttaskassign**	**oaUserWorkschedule**
oaEstimateadjustment	**oaProposal**	**oaVendor**
oaEstimateexpense	**oaProposalblock**	**oaWorkspacelink**
oaEstimatelabor	**oaPurchase_item**	**oaWorkspaceuser**
oaEstimatemarkup	**oaPurchaseorder**	

Tip: You can lookup the OpenAir Complex Types and their properties from the following link http://www.openair.com/wsdl.pl?wsdl.

OpenAir Complex Type objects are required in the following wsapi functions:

- NSOA.wsapi.add(objects)
- NSOA.wsapi.delete(objects)
- NSOA.wsapi.modify(attributes, objects)
- NSOA.wsapi.read(readRequest)
- NSOA.wsapi.upsert(attributes,objects)

Note: For more information on the SOAP API (Web Services) see the OpenAir SOAP API Reference Guide.

Parameters

- *id* {var} [optional] — If specified, this (internal) id will be used to populate the new object.

Returns

- OpenAir Complex Type object.

Units Limit

- 0 units

For more information, see Scripting Governance.

Since

- November 16, 2013

Example

- This sample creates a customer object populates with the current values in the database.

```
// Create customer object populated with data for id = 66
var customer = NSOA.record.oaCustomer(66);
```

- This sample creates a new category in OpenAir.

```
// Create a new category object
var category = new NSOA.record.oaCategory(); // empty category
category.name = "New Category";
category.cost_centerid = "123";
category.currency = "USD";

// Invoke the add call
var results = NSOA.wsapi.add( [category] );
```

See also NSOA.wsapi.add(objects).

See Code Samples for more examples.

NSOA.wsapi.add(objects)

Use this function to add data to OpenAir. The function returns an error if more than 1000 objects are passed in.

You can use an external id field as a foreign key and add a record without querying first for an internal id.

Note: For more information on the SOAP API (Web Services) see the OpenAir SOAP API Reference Guide.

See also Making SOAP Calls.

Parameters

- *objects* {var} [required] — Array of OpenAir Complex Type objects, see NSOA.record.<complex type>([id]).

Returns

- Array of UpdateResult objects.

Units Limit

- 20 units

  +10 for each additional object passed

For more information, see Scripting Governance.

Since

- November 16, 2013

Example

- This sample creates a new category in OpenAir.

```
// Define a category object to create in OpenAir
var category = new NSOA.record.oaCategory();
category.name = "New Category";
category.cost_centerid = "123";
category.currency = "USD";

// Invoke the add call
var results = NSOA.wsapi.add( [category] );

// Get the new ID
var id = results[0].id;
```

Note: This simple example does not show error checking, see Handling SOAP Errors.

See Code Samples for more examples.

NSOA.wsapi.delete(objects)

Use this function to delete data in OpenAir based on an internal ID. The function returns an error if more than 1000 objects are passed in

Note: For more information on the SOAP API (Web Services) see the OpenAir SOAP API Reference Guide.

See also Making SOAP Calls.

Parameters

- *objects* {var} [required] — Array of OpenAir Complex Type objects, see NSOA.record.<complex type>([id]).

Returns

- Array of UpdateResult objects.

Units Limit

- 20 units

  +10 for each additional object passed

For more information, see Scripting Governance.

Since

- November 16, 2013

Example

- This sample deletes a customer from OpenAir.

```
// Delete customer with internal id 66
var customer = new NSOA.record.oaCustomer();
customer.id = 66;

// Invoke the delete call
var results = NSOA.wsapi.delete( [customer] );
```

Note: This simple example does not show error checking, see Handling SOAP Errors.

See Code Samples for more examples.

NSOA.wsapi.disableFilterSet([flag])

Use this function to check, enable, or disable user filter sets.

Note: Scripts are executed within the context of the user who is logged in. This means that the user filter sets for the logged in user will be applied unless disabled.

Tip: Disabling user filter sets allows you to write more generic scripts.

Parameters

- *flag* {Boolean} [optional] — If **true** is passed the user filter sets are disabled, if **false** is passed the user filter sets are enabled, and if **no parameter** is passed the function just returns the current filter setting.

Returns

- Boolean **true** if filter sets are disabled and **false** if user filter sets are enabled.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

- November 16, 2013

Example

- Disable user filter sets on .wsapi requests.

```
NSOA.wsapi.disableFilterSet(true);
```

- Enable user filterset on .wsapi requests.

```
NSOA.wsapi.disableFilterSet(false);
```

> **Note:** This the default OpenAir behavior, i.e. user filter sets enabled.

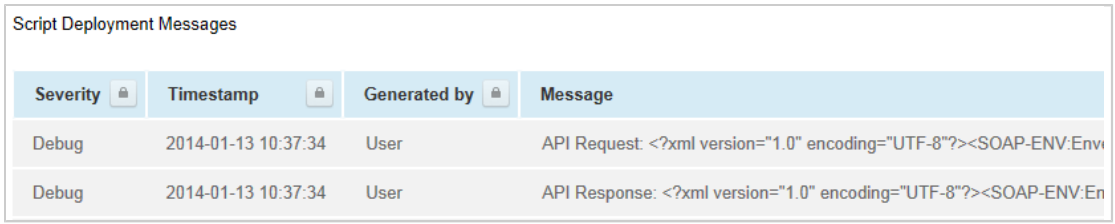
- Return the boolean state (without changing setting)

```
if( NSOA.wsapi.disableFilterSet()) {
    // The user filter sets are disabled
}
```

See Code Samples for more examples.

NSOA.wsapi.enableLog([flag])

Use this function to see the SOAP API request and response messages generated by NSOA.wsapi function calls.



Every call between enableLog(true) and enableLog(false) is logged and available for viewing in the same place as the NSOA.meta.log(severity, message) function.

> **Note:** This function only works in test mode and is ignored in production due to the size of the messages. See Testing & debugging.

Parameters

- *flag* {Boolean} [optional] — If **true** is passed then wsapi logging is enabled, if **false** is passed then wsapi logging is disabled, and if **no parameter** is passed the function just returns the current wsapi logging setting.

Returns

- Boolean **true** if wsapi logging is enabled and **false** if wsapi logging is disabled.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

- February 15, 2014

Example

- Enable wsapi logging.

```
NSOA.wsapi.enableLog(true);
```

- Disable wsapi logging.

```
NSOA.wsapi.enableLog(false);
```

Note: This is the default OpenAir behavior, i.e. wsapi logging disabled.

- Returns the boolean state (without changing setting)

```
if( NSOA.wsapi.enableLog()) {
    // wsapi logging is enabled
}
```

See Code Samples for more examples.

NSOA.wsapi.modify(attributes, objects)

Use this function to modify data in OpenAir. The function returns an error if more than 1000 objects are passed in.

You can use an external id field as a foreign key and add a record without querying first for an internal id. You can also modify data in OpenAir based on an internal ID.

Note: For more information on the SOAP API (Web Services) see the OpenAir SOAP API Reference Guide.

See also Making SOAP Calls.

Parameters

- *attributes* {var} [required] — Array of Attribute objects.
- *objects* {var} [required] — Array of OpenAir Complex Type objects, see NSOA.record.<complex type>([id]).

Returns

- Array of UpdateResult objects.

Units Limit

- 40 units

+20 for each additional object passed

For more information, see Scripting Governance.

Since

- November 16, 2013

Example

- This sample changes a customer's email address in OpenAir.

```
// Modify customer's email address
var customer = new NSOA.record.oaCustomer();
customer.id = 37;
customer.addr_email = "newest@email.com";

// Not attributes required
var attributes = [];

// Invoke the modify call
var results = NSOA.wsapi.modify( attributes, [customer] );
```

Note: This simple example does not show error checking, see Handling SOAP Errors.

See Code Samples for more examples.

NSOA.wsapi.read(readRequest)

Use this function to retrieve data from OpenAir.

Note: For more information on the SOAP API (Web Services) see the OpenAir SOAP API Reference Guide.

See also Making SOAP Calls.

Parameters

- *readRequest* {object} [required] — ReadRequest object.

Returns

- Array of ReadResult objects.

Units Limit

- 20 units

+10 for each additional object passed

For more information, see Scripting Governance.

Since

- November 16, 2013

Example

- This sample creates a new category in OpenAir.

```
// Create the issue object
var issue = new NSOA.record.oaIssue();
issue.project_id = NSOA.form.getValue('id');
issue.issue_stage_id = 1;

// Define the read request
var readRequest = {
    type   : "Issue",
    method : "equal to", // return only records that match search criteria
    fields : "id, date", // specify fields to be returned
    attributes : [       // Limit attribute is required; type is Attribute
       {
           name : "limit",
           value : "10"
       }
    ],
    objects : [          // One object with search criteria; type implied by rr 'type'
        issue
    ]
};

// Invoke the read call
var results = NSOA.wsapi.read(readRequest);
```

Note: This simple example does not show error checking, see Handling SOAP Errors.

See Code Samples for more examples.

NSOA.wsapi.remainingTime()

Use this function to determine how much time your script has remaining to execute inside wsapi functions before it is terminated by Scripting Governance.

You can use this function to help you create more efficient scripts and also to take corrective action if a script is at risk of consuming excessive resources.

Parameters

- (none)

Returns

- Amount of time remaining allowed for the script to execute inside wsapi calls in milliseconds.

> **Tip:** Always try to reduce the amount of time your scripts take to execute.

Units Limit

- 0 units

For more information, see Scripting Governance.

Since

- October 18, 2014

Example

- This example logs the amount of wsapi time remaining for the script to execute in milliseconds.

```
NSOA.meta.log('info', 'Remaining wsapi time: '
        + NSOA.wsapi.remainingTime() + ' milliseconds');
```

See also NSOA.context.remainingTime().

For more information see Scripting Governance.

NSOA.wsapi.upsert(attributes,objects)

Use this function to add or modify data in OpenAir based on lookup attributes. The function returns an error if more than 1000 objects are passed in.

You can use an externalid field as a foreign key and add a record without querying first for an internal id.

> **Note:** For more information on the SOAP API (Web Services) see the OpenAir SOAP API Reference Guide.

See also Making SOAP Calls.

Parameters

- *attributes* {var} [required] — Array of Attribute objects.
- *objects* {var} [required] — Array of OpenAir Complex Type objects, see NSOA.record.<complex type>([id]).

Returns

- Array of UpdateResult objects.

Units Limit

- 40 units

  +20 for each additional object passed

For more information, see Scripting Governance.

Since

- November 16, 2013

Example

- This sample creates a new category in OpenAir.

```
//Define a category object to create/update in OpenAir
var category = new NSOA.record.oaCategory();
category.name = "Updated Category";
category.externalid = "555";

// Specify that the lookup is done by external_id and not by (default) internal id
var attribute = {
    name  : "lookup",
    value : "externalid"
};

// Invoke the upsert call
var results = NSOA.wsapi.upsert( [attribute], [category] );
```

Note: This simple example does not show error checking, see Handling SOAP Errors.

See Code Samples for more examples.

NSOA.wsapi.whoami()

Use this function to add or modify data in OpenAir based on lookup attributes. The function returns an oaUser object, see Who Am I.

Note: For more information on the SOAP API (Web Services) see the OpenAir SOAP API Reference Guide.

See also Making SOAP Calls.

Parameters

- (none)

Returns

- An oaUser object.

Units Limit

- 1 unit

For more information, see Scripting Governance.

Since

- November 16, 2013

Example

- This sample logs the name of the user running the script.

```
function logUser() {
    var user = NSOA.wsapi.whoami();
    NSOA.meta.alert( "User id " + user.id + " saved this record");
}
```

Note: This simple example does not show error checking, see Handling SOAP Errors.

See Code Samples for more examples.

NSOA.NSConnector.integrateAllNow()

This function is the equivalent to clicking the **Run** button on the integration form. It can only be called for a "Scheduled" script, and allows 1 call per script.

This function supports the following:

- Expense reports
- Invoices
- Project information
- Task information
- Timesheets

SuiteCloud+ is required to use this function. This function also requires the "Enable multi-threading support. To be used with SuiteCloud+" administrative setting.

Note: For more information on the SOAP API (Web Services) see the OpenAir SOAP API Reference Guide.

See also Making SOAP Calls.

Parameters

- (none)

Returns

- Boolean **true** if integration was triggered and **false** if integration was not triggered.

Units Limit

- 1000 units

For more information, see Scripting Governance.

Since

- April 16, 2016

Example

- This example triggers the NetSuite integration for all fields using a scheduled script.

```
function main() {
    var records = NSOA.wsapi.read(...);

    // check if result is OK
    if (!records || !records[0])
        return;

    // trigger NetSuite integration if there is no error and more than 50 records
    else if (records[0].errors === null && records[0].objects &&records[0].objects.length > 50)
 {
        NSOA.NSConnector.integrateAllNow();
    }
}
```

Note: This simple example does not show error checking, see Handling SOAP Errors.

See Code Samples for more examples.

NSOA.NSConnector.integrateRecord()

This function is equivalent to clicking the **Export/Send** links in the Tips menu for a selected record. It can only be called for a "Form" script, and allows 1 call per script.

This function supports the following:

- Expense reports
- Invoices
- Project information
- Task information
- Timesheets

SuiteCloud+ is required to use this function. This function also requires the "Enable multi-threading support. To be used with SuiteCloud+" administrative setting.

Parameters

- (none)

Returns

- Boolean **true** if integration was triggered and **false** if integration was not triggered.

Units Limit

- 10 units

For more information, see Scripting Governance.

Since

- April 16, 2016

Example

- This example presents a common use case for after-approval events with envelopes.

```
function main() {
    // integrate current form object to NetSuite
    NSOA.NSConnector.integrateRecord();
```

See Code Samples for more examples.

NSOA.wsapi.submit(submitRequest)

Use this function to submit timesheets, invoices, and envelopes. It can take an array of up to 1,000 submit request objects.

Parameters

- *submitRequest*{object} [required] — submitRequest object

Returns

Array of ApprovalResult objects.

Units Limit

- 20 units

  +10 for each additional object passed

For more information see Scripting Governance.

Since

October 15, 2016

Example

In this example, the script creates the approval object, then prepares the timesheet with timesheet ID 45 for submitting, defines the submit requests, and invokes the action call.

```
// Create the approval object
var approvalObj = new NSOA.record.oaApproval();
approvalObj.notes = "submit from scripting";

// Prepare the record for submit
var timesheetToProcess = new NSOA.record.oaTimesheet();
timesheetToProcess.id = 45;

// Define the submit requests
var requests = [{
      submit: timesheetToProcess,
      attributes: [], // submit attributes are optional
      approval: approvalObj
}];

// Invoke the action call
var results = NSOA.wsapi.submit(requests);
```

Note: This simple example does not show error checking, see Handling Approval Errors

See Code Samples for more examples.

NSOA.wsapi.approve(approveRequest)

Use this function to approve timesheets, invoices, and envelopes. It can take an array of up to 1,000 approve request objects.

Parameters

- *approveRequest*{object} [required] — approveRequest object

Returns

Array of ApprovalResult objects.

Units Limit

- 20 units

  +10 for each additional object passed

For more information see Scripting Governance.

Since

October 15, 2016

Example

In this example, the script creates the approval object, then prepares the timesheet with timesheet ID 45 for approval, defines the approve requests, and invokes the action call.

```
// Create the approval object
var approvalObj = new NSOA.record.oaApproval();
approvalObj.notes = "approve from scripting";

// Prepare the record for approve
var timesheetToProcess = new NSOA.record.oaTimesheet();
timesheetToProcess.id = 45;

// Define the approve requests
var requests = [{
      approve: timesheetToProcess,
      attributes: [], // approve attrributes are optional
      approval: approvalObj
}];

// Invoke the action call
var results = NSOA.wsapi.approve(requests);
```

Note: This simple example does not show error checking, see Handling Approval Errors

See Code Samples for more examples.

NSOA.wsapi.reject(rejectRequest)

Use this function to reject timesheets, invoices, and envelopes. It can take an array of up to 1,000 reject request objects.

Parameters

- *rejectRequest*{object} [required] — rejectRequest object

Returns

Array of ApprovalResult objects.

Units Limit

- 20 units

  +10 for each additional object passed

For more information see Scripting Governance.

Since

October 15, 2016

Example

In this example, the script creates the approval object, then prepares the timesheet with timesheet ID 45 for rejection, defines the reject requests, and invokes the action call.

```
// Create the approval object
var approvalObj = new NSOA.record.oaApproval();
approvalObj.notes = "reject from scripting";

// Prepare the record for reject
var timesheetToProcess = new NSOA.record.oaTimesheet();
timesheetToProcess.id = 45;

// Define the reject requests
var requests = [{
      reject: timesheetToProcess,
      attributes: [], // reject attrributes are optional
      approval: approvalObj
}];

// Invoke the action call
var results = NSOA.wsapi.reject(requests);
```

Note: This simple example does not show error checking, see Handling Approval Errors

See Code Samples for more examples.

NSOA.wsapi.unapprove(unapproveRequest)

Use this function to unapprove timesheets, invoices, and envelopes. It can take an array of up to 1,000 unapprove request objects.

Parameters

- *unapproveRequest*{object} [required] — unapproveRequest object

Returns

Array of ApprovalResult objects.

Units Limit

- 20 units

  +10 for each additional object passed

For more information see Scripting Governance.

Since

October 15, 2016

Example

In this example, the script creates the approval object, then prepares the timesheet with timesheet ID 45 for unapproval, defines the unapprove requests, and invokes the action call.

```
// Create the approval object
var approvalObj = new NSOA.record.oaApproval();
approvalObj.notes = "unapprove from scripting";

// Prepare the record for unapprove
var timesheetToProcess = new NSOA.record.oaTimesheet();
timesheetToProcess.id = 45;

// Define the unapprove requests
var requests = [{
      unapprove: timesheetToProcess,
      attributes: [], // unapprove attrributes are optional
      approval: approvalObj
}];

// Invoke the action call
var results = NSOA.wsapi.unapprove(requests);
```

Note: This simple example does not show error checking, see Handling Approval Errors

See Code Samples for more examples.

Code Samples

The following code samples are provided:

- Comparing Date Fields
- Validating Numeric Fields
- Requiring Minimum Values
- Creating Error Log Entries
- Sending email
- SOAP API — Prevent closing a project with an open issue
- SOAP API — Append notes to a project
- SOAP API — Require task assignment
- Submitting a Timesheet for Approval

See also Real World Use Cases.

Comparing Date Fields

```
// compare two date fields on a receipt
function validateTravelDates() {
   var receiptDate = NSOA.form.getValue('date');
   var travelDate = NSOA.form.getValue('TravelDate__c');
```

```
   if ( receiptDate < travelDate ) {
      NSOA.form.error('TravelDate__c', 'The travel date cannot be after the receipt date!');
   }
}
```

See also:

- NSOA.form.getValue(field)
- NSOA.form.error(field, message)

Validating Numeric Fields

```
// validate a number entered into a custom numeric field
function projectRating() {
   var rating = NSOA.form.getValue('ProjectRating__c');

   if ( rating < 1 || rating > 5 ) {
      NSOA.form.error('ProjectRating__c', 'Ratings must be whole numbers between 1 and 5.');
   }
}
```

See also:

- NSOA.form.getValue(field)
- NSOA.form.error(field, message)

Requiring Minimum Values

```
// require notes on all airfare exceeding $1,000 dollars
function airfareCost() {
   var cost = NSOA.form.getValue('cost');
   var notes  = NSOA.form.getValue('notes');
   var item = NSOA.form.getValue('item_id');

   if ( cost > 1000 && notes.length < 1 && item == '4' ) {

      NSOA.form.error('notes', 'Notes are required for Airfare exceeding $1,000.');
   }
}
```

See also:

- NSOA.form.getValue(field)
- NSOA.form.error(field, message)

Creating Error Log Entries

```
// add an error log entry to the validateTravelDates() function above
function validateTravelDates() {
   var receiptDate = NSOA.form.getValue('date');
```

```
   var travelDate = NSOA.form.getValue('TravelDate__c');

   if ( receiptDate < travelDate ) {
      NSOA.form.error('TravelDate__c', 'The travel date cannot be after the receipt date!');
      NSOA.meta.log('error', 'Form error - travel date ' + travelDate + ' is after receipt date
 ' + receiptDate);
   }
}
```

See also:

- NSOA.form.getValue(field)
- NSOA.form.error(field, message)
- NSOA.meta.log(severity, message)

Sending email

```
function sendAlert() {
    // TODO Add Your Code Here

    // TODO Handle Errors

    // Notify The Owner
    var me = NSOA.wsapi.whoami();
    var msg = {
        to: [me.id],
        subject: "Script completed",
        format: "HTML",
        body: "<b>Your script completed</b><br/>" +
              "<hr/><i>Automatically sent by the system</i>"
    };

    NSOA.meta.sendMail(msg);
}
```

See also:

- NSOA.meta.sendMail(message)

SOAP API — Prevent closing a project with an open issue

```
function test_prevent_project_close_with_open_issue() {
    var project_stage_id = NSOA.form.getValue('project_stage_id');
    if (project_stage_id != 4) // if new stage is not closed, skip check
        return;

    //Read request
    var issue = new NSOA.record.oaIssue();
    issue.project_id = NSOA.form.getValue('id');
    issue.issue_stage_id = 1;
    var readRequest = {
    type : "Issue",
```

```
    method : "equal to", // return only records that match search criteria
    fields : "id, date", // specify fields to be returned
    attributes : [       // Limit attribute is required; type is Attribute
        {
            name : "limit",
            value : "10"
        }
    ],
    objects : [      // One object with search criteria; type implied by rr 'type'
        issue
    ]
    };
    var arrayOfreadResult = NSOA.wsapi.read(readRequest);
    if (!arrayOfreadResult || !arrayOfreadResult[0])
        NSOA.form.error('', "Internal error analyzing project. Please contact account admin.");

    else if (arrayOfreadResult[0].errors === null && arrayOfreadResult[0].objects)
        arrayOfreadResult[0].objects.forEach(
          function(o) {
              NSOA.form.error('', "Can't close project with open issues.");
          }
    );
}
```

See also:

- NSOA.form.getValue(field)
- NSOA.record.<complex type>([id])
- NSOA.wsapi.read(readRequest)
- NSOA.form.error(field, message)

SOAP API — Append notes to a project

```
    // This is called on the "After save" event for the Project form
    function append_to_project_notes() {
        var newr = NSOA.form.getNewRecord();

        var project = new NSOA.record.oaProject();
        project.id = newr.id;
        project.notes = newr.notes + "\nAppended to notes: " + (new Date().toString());

        NSOA.wsapi.disableFilterSet(true);
        var arrayOfupdateResult = NSOA.wsapi.modify([], [project]);
        NSOA.meta.alert("Got modify status: " + arrayOfupdateResult[0].status);
}
```

See also:

- NSOA.form.getNewRecord()
- NSOA.record.<complex type>([id])
- NSOA.wsapi.modify(attributes, objects)
- NSOA.meta.alert(message)

SOAP API — Require task assignment

```
// Add form error if user is not assigned to project task to which they're about to be booked.
function require_task_assignment() {

   // Prepare read query
    var pta = new NSOA.record.oaProjecttaskassign();
    pta.projecttaskid = NSOA.form.getValue('project_task_id');
    pta.userid = NSOA.form.getValue('user_id');

    var readRequest = {
        type : "Projecttaskassign",
        method : "equal to",               // return only records that match search criteria
        fields : "id",                     // specify fields to be returned
        attributes : [                     // Limit attribute is required; type is Attribute
            {
                name : "limit",
                value : "1"
            }
        ],
        objects : [                        // One object with search criteria
            pta
        ]
   };

    // Run query
    NSOA.wsapi.disableFilterSet(true);   // disable the current user's filter for read query
    var result = NSOA.wsapi.read(readRequest);

    // Check query results
    if (!result || !result[0])
        NSOA.form.error('', "Internal error analyzing booking. Please contact account admin.");

    else if (result[0].errors !== null || result[0].objects === null || result[0].objects.lengt
h === 0)
        NSOA.form.error('_user_id', "Can't book this user without being assigned to selected pr
oject task.");
}
```

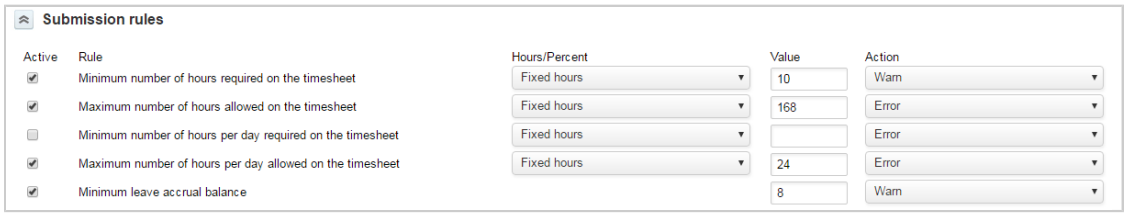
See also:

- NSOA.record.<complex type>([id])
- NSOA.form.getValue(field)
- NSOA.wsapi.read(readRequest)
- NSOA.wsapi.disableFilterSet([flag])
- NSOA.form.error(field, message)

Submitting a Timesheet for Approval

In the case below, the following timesheet submission rules have been configured. When submitting a timesheet (in this example, with timesheet ID 45), if any warnings occur, (for example, if the minimum number of hours on the timesheet is less than 10), the submit call from the script would fail. If you

want the submit call to occur despite the warnings, you would need to pass the “ignore_warnings” attribute as shown in the code example below.




```
function main(type) {

     // Create the approval object
     var approvalObj = new NSOA.record.oaApproval();
     approvalObj.notes = "submit from scripting";

     // Prepare the record for submit
     var timesheetToProcess = new NSOA.record.oaTimesheet();
     timesheetToProcess.id = 45;

     // Ignore any warnings
     var ignore_warnings = {
          name : "submit_warning",
          value : "1"
     };

     // Define the submit requests
     var requests = [{
          submit: timesheetToProcess,
          attributes: [ignore warnings],
          approval: approvalObj
     }];

     // Invoke the action call
     var results = NSOA.wsapi.submit(requests);

}
```

JavaScript

JavaScript Overview

OpenAir user scripts are external JavaScript files. OpenAir is compliant with ECMAScript 5.

JavaScript is a cross-platform, object-oriented scripting language.

Important: For OpenAir to use an external JavaScript file, it must be stored in a **Workspace** as an ASCII text file with the file extension **.js**.

JavaScript is easy to learn.

Key Points

- Semicolons to end statements are optional in JavaScript, but for clarity you are advised to always use them.
- JavaScript ignores extra white space. Use white space to make your scripts more readable.

  Note: The following statements are the same.

```
var receiptDate=NSOA.form.getValue('date');
var receiptDate = NSOA.form.getValue('date');
```

- JavaScript is case sensitive.

  Important: The following variables are NOT the same!

```
var receiptDate;
var ReceiptDate;
```

- JavaScript supports single and multliline comments. Use comments to make your scripts maintainable!

```
// This is a single line comment

/*
   This is a multiline comment
*/
```

  Tip: You can comment out lines of script to prevent them from being executed. This is a useful debugging technique.

Variables

Variables are usually declared in JavaScript with the **var** keyword.

```
var price;
```

Note: JavaScript is an untyped language, you cannot declare a variable to be a string or number. Variables can hold any type and data types are converted automatically behind the scenes. See Dynamic Data Types

Tip: If you don't use **var** the variable will be declared as **global**. You should avoid using global variables as they can result in unwanted side effects and are a frequent source of bugs! See Variable Scope.

After a variable is declared its value is undefined.

A value is assigned to a variable with the equals sign.

```
price = 500;
```

A variable can be assigned a value when it is declared.

```
var price = 500;
```

A variable can be emptied by setting its value to null.

```
price = null;
```

If you re-declare a variable, the variable will not lose its value.

```
var travelType = "Car";
var travelType; // travelType is still "Car"
```

Important: If you assign a value to variable that has not been declared with **var**, the variable will automatically be declared as a **global** variable. See Variable Scope.

Variables names must start with a letter or underscore and cannot use any Reserved Words.

Tip: Use short names for variables which you use only in nearest code.

Multi-word names add precision from right to left, adjectives are always at the left side.

Use camel-case.

Variable Scope

Variables in JavaScript can have **local** or **global** scope. The **scope** of a variable refers to the variable's visibility within a script. Variables accessible to a restricted part of a script are said be **local**. Variables that are accessible from anywhere, are said to be **global**.

Global variables can be created anywhere in JavaScript code, simply by assigning initial values to them. Once created, global variables can be accessed from any part of the script and retain their values until the script ends.

In JavaScript, newly created variables are assumed to be global, regardless of where they are created, unless explicitly defined with the **var** keyword.

Important: Ambiguity can arise when a global variable and local variable have the same names. JavaScript resolves this ambiguity by giving priority to local variables.

Dynamic Data Types

JavaScript has dynamic data types. The same JavaScript variable can be treated as having different data types depending on the context it is used in.

```
var travelType;          // travelType is undefined
var travelType = 5;      // travelType is a Number
var travelType = "Car";  // travelType is a String
```

Internal JavaScript data types:

- String
- Number
- Boolean
- null
- undefined

Note: See also Objects.

String

A string in JavaScript is series of characters enclosed in quotation marks. A string must be delimited by quotation marks of the same type, either single quotation marks ' or double quotation marks ".

```
var name = "John Smith";
var type = 'customer';
```

You can use quotes inside a string, as long as they don't match the quotes surrounding the string.

```
var responseText = "It was paid to 'John Smith'";
var responseText = 'It was paid to "John Smith"';
```

You can put a quote inside a string using the \ character.

```
var responseText = 'It\'s okay.';
```

You can access a character in a string by its zero-based position index.

```
var name = "John Smith";
var character = name[3]; // character == 'n'
```

In JavaScript a string is an object. See String for properties and methods.

Number

JavaScript has only one type of number. Large numbers can be written in scientific (exponential) notation.

```
var pi = 3.14;
var amount = 314e5;     // 31400000
var factor = 314e-5;    // 0.00314
```

JavaScript interprets numeric constants as octal if they are preceded by a zero, and as hexadecimal if they are preceded by a zero and x.

```
var x=0377;    // This is 255 in decimal
var y=0xFF;    // This is 255 in decimal
```

Important: When you assign a number to a variable, do not put quotes around the value. If you put quotes around a numeric value, the variable content will be treated as a string.

Never write a number with a leading zero, unless you want an octal conversion.

In JavaScript a number is an object. See Number for properties and methods.

Boolean

Booleans can only have two values: **true** or **false**.

```
var sent = true;
var paid = false;
```

In JavaScript a boolean is an object. See Boolean for properties and methods.

null

null is a special keyword denoting an empty value.

A variable can be emptied by setting it to **null**.

```
travelType = null;
```

undefined

This is a special keyword denoting an undefined value.

Before a variable is assigned a value it is undefined.

```
var travelType;    // variable is undefined
```

Arrays

In JavaScript an array is created as follows:

```
// Creating an array
```

```
var priority = new Array();
priority[0] = "Low";
priority[1] = "Normal";
priority[2] = "High";

// Literal array
var priority = ["Low","Normal","High"];
```

Note: JavaScript Arrays are zero base.

An element is accessed in the array by **index** number:

```
// To access the first element
var level = priority[0];

// To modify the first element
priority[0] = "Not required";
```

You can have different types in an array:

```
var entry = new Array();
entry[0] = Date.now;
entry[1] = "Book";
entry[2] = 5.99;
```

In JavaScript an array is an object, so an array can be an element in an array.

See Array for properties and methods.

See also Associative Array.

Associative Array

An associative array is a set of key value pairs. The value is stored in association with its key and if you provide the key the array will return the value.

```
// To create an associative array
contacts = {
    firstname : 'John',
    lastname  :'Smith'
};
```

Note: Notice the similarity to Objects.

An associative array is accessed by a key name.

```
// Use the key to access an entry
var value = contacts['firstname']; // value is 'John'

// You can also use dot notation
var value = contacts.lastname; // value is 'Smith'
```

Note: The key is always a string, but the value can be any type. See Dynamic Data Types and Objects.

You can loop through the keys of an associative array with the for in loop.

```
// Get all the values on the fields on the form
var allValues = NSOA.form.getAllValues();

//Loop through all the values
for( var key in allValues ) {
    NSOA.meta.alert(key + ' has value ' + allValues[key]);
}
```

See also:

- NSOA.form.getAllValues()
- NSOA.meta.alert(message)

You can change the value using assignment to a property.

```
// Using the array notation
contacts['firstname'] = 'Joe';

// Using dot notation
contacts.firstname = 'Joe';
```

You can add a new key/value pair by assigning to a property that doesn't exist.

```
// Using the array notation
contacts['company'] = 'NetSuite';

// Using dot notation
contacts.company ='NetSuite';
```

Important: Some fields return an object. See Object Fields.

Objects

An object is just a special kind of data, with Properties and Methods.

JavaScript allows you to define your own objects.

```
// To declare an object
var person={ firstname : "John", lastname  : "Smith", age:  25};

// Use spaces and line breaks to make your definition clearer
var person={
    firstname : "John",
    lastname  : "Smith",
    age       :  25
};
```

```
// You can access object properties in two way
var age = person.age;
var age = person["age"];
```

The object (person) in the example above has 3 properties: firstname, lastname, and age.

See also for in and forEach.

Properties

Properties are the values associated with an object.

The syntax for accessing the property of an object is:

```
objectName.propertyName
```

This example uses the length property of the String object to find the length of a string:

```
var message = "Hello World!";
var x = message.length;
```

The value of x, after execution of the code above will be 12.

Methods

Methods are the actions that can be performed on objects.

You can call a method with the following syntax:

```
objectName.methodName()
```

This example uses the toUpperCase() method of the String object, to convert a text to uppercase:

```
var message="Hello world!";
var x = message.toUpperCase();
```

The value of x, after execution of the code above will be “HELLO WORLD!”.

Functions

Functions are declared with the **function** keyword, they can be passed Arguments and can Return Values.

Function names must start with a letter or underscore and cannot use any Reserved Words.

```
// Declaring a function
function calcSum (x,y) {
    return x + y;
}
```

```
// Calling a function
var result = calcSum(15,25);
```

Note: Variables declared inside a function as **var** are local to the function. Variables defined inside a function without the **var** are global variables.

Arguments

Functions do not need arguments.

```
function validateTravelDates() {
   var receiptDate = NSOA.form.getValue('date');
   var travelDate = NSOA.form.getValue('TravelDate__c');

   if ( receiptDate < travelDate ) {
      NSOA.form.error('TravelDate__c', 'The travel date cannot be after the receipt date!');
   }
}
```

You can pass as many arguments as you like separated by commas.

```
function calcSum (x,y,z) {
    var result = x + y + z;
    return result;
}
```

See also:

- NSOA.form.getValue(field)
- NSOA.form.error(field, message)

Important: If you declare a function with arguments then the function must be called with all the arguments in the expected order.

Return Values

Functions do not need to return a value.

```
function logFormError(message) {
    NSOA.meta.log('error', 'Form error - ' + message);
}
```

See also:

- NSOA.meta.log(severity, message)

Use the **return** statement to return a variable from a function.

```
function calcProduct (x,y) {
```

```
    var result = x * y;

return result;
}
```

You can use the **return** statement to immediately exit a function. The return value is optional.

```
function reduceValue (x,y) {
    if ( x < y ) {

return; // exit function
    }

    var result = x - y;
    return result;
}
```

Loops

JavaScript supports the following types of loop:

- for
- for in
- forEach
- do while
- while

Key Points

- Use the **break** statement to terminate the current while or for loop and continue executing the statements after the loop..
- Use the **continue** statement to stop executing the current iteration and continue with the next iteration.

Important: Be careful not to create endless loops. Make sure your loops always have an exit condition!

for

The **for** loop executes a block of code a specified number of times.

Syntax

```
for (initialization; condition; increment) {
   // statements
}
```

Example

```
for (var i = 0; i < 5; i++) {
    x = x + "The number is " + i;
}
```

for in

The **for in** loop is for iterating through the enumerable properties of an object. See Objects and Associative Array.

Syntax

```
for (variable in object) {
    // code to be executed
}
```

Example

```
var person={firstName:"John",lastName:"Smith",age:21};

for (i in person) {
    s = s + person[i];
}
```

forEach

The **forEach** loop has the benefit that you don't have to declare indexing and entry variables in the containing scope, as they're supplied as arguments to the iteration function, and so nicely scoped to just that iteration.

```
var a = ["a", "b", "c"];
a.forEach(function(o) {
    NSOA.meta.alert(o);
});
```

See also:

- NSOA.meta.alert(message)

do while

The **do while** loop is a variant of the while loop. This block is first executed and then repeated as long as the condition is true.

Syntax

```
do {
  // statements
}
while (condition)
```

Example

```
var i=0;
do {
    x = x + "The number is " + i;
    i++;
}
while ( i < 5 )
```

while

The **while** loop iterates through a block of code as long as a specified condition is true.

Syntax

```
while (condition) {
  // statements
}
```

Example

```
var i = 0;
while ( i < 5 ) {
    x = x + "The number is " + i;
    i++;
}
```

Conditional Statements

JavaScript supports if ... else and switch conditional statements.

if ... else syntax

```
if (condition) {
   statements_1
} else {
   statements_2
}
```

switch syntax

```
switch (expression) {
   case label_1:
      statements_1
      [break;]
   case label_2:
      statements_2
      [break;]
   default:
     statements_n
     [break;]
```

```
}
```

if ... else

if is the fundamental control statement that allows JavaScript to make decisions and execute statements conditionally.

If the expression is true (i.e. today < endDate) then the block following **if** is executed.

```
var endDate = NSOA.form.getValue('end_date');
var today = new Date();
if (today < endDate) {
    // statements to execute if we haven't started this yet
}
```

If the expression is false (i.e. today < endDate in not true) the optional block following else is executed.

```
var endDate = NSOA.form.getValue('end_date');
var today = new Date();
if (today < endDate) {
    // statements to execute if we haven't started this yet
}
else {
    // statements to execute if we have started
}
```

Note: 'else' is optional, but 'if' must be present

You can chain together as many **if ... else** statements as required.

```
var type = NSOA.form.getValue('type');
if (type == 0) {
    // statements to handle type 0
} else if (type == 1) {
    // statements to handle type 1
} else if (;type == 2) {
    // statements to handle type 2
} else{
    // statements to handle all other types
}
```

Tip: Rather than creating a long **if .. else** chain, use the switch statement.

Note: This is just a series of **if** statements, where each **if** is part of the **else** clause of the previous statement. Each condition is evaluated in sequence. The first block with its condition to evaluate true is executed and then the whole chain is exited. If no condition is true then the final else block is executed.

See also:

- NSOA.form.getValue(field)

switch

The **switch** statements compares an expression against a list of case values. Execution jumps to the first case that matches. If nothing matches, execution jumps to the **default** condition.

```
var type = NSOA.form.getValue('type');
switch (type) {
    case 0:
        // statements to handle type 0
        break;
    case 1:
        // statements to handle type 1
        break;
    case 2:
        // statements to handle type 2
        break;
    default:
        // statements to handle all other types
}
```

Note: The **break** statements ends the case and execution jumps to the next statement after the switch block. If the break is omitted execution continues with the next case.

Error Handling

JavaScript supports **try** and **catch** blocks to handle errors. When something goes wrong, JavaScript will throw an error.

Syntax

```
try {
    // The code to run
}
catch(err) {
    // Code to handle any errors
}
```

Example

```
try {
    var receiptDate = NSOA.form.getValue('date');
    var travelDate = NSOA.form.getValue('TravelDate__c');

    if ( receiptDate < travelDate ) {
          NSOA.form.error('TravelDate__c', 'The travel date cannot be after the receipt date!')
;
    }
}
catch(err) {
    NSOA.meta.log('error', err.message);
}
```

See also:

- NSOA.form.getValue(field)
- NSOA.form.error(field, message)
- NSOA.meta.log(severity, message)

Key points:

- Use a **try** block to surround code that could throw an error.
- Use a **catch** block to contain code that handles any errors.
- You can use the **throw** statement to create custom errors.

throw

When an exception occurs JavaScript will throw an error that you can catch.

You can use the throw statement to raise your own custom exceptions. These exceptions can be captured and appropriate action taken.

```
// Function that throws a custom "Divide by zero" error
function divide(x,y) {
    if( y == 0 ) {
        throw( "Divide by zero" );
    } else {
        return x / y;
    }
}

//Function that catches the custom error as a string
function test() {
    try {
        return divide(10,0);
    }
    catch(err) {
        // err == "Divide by zero"
    }
}
```

Note: You can throw different types, e.g. String, Number, and Object.

References

JavaScript Objects

Array

An Array object is used to store multiple values in a single variable.

```
// Creating an array
```

```
var priority = new Array();
priority[0] = "Low";
priority[1] = "Normal";
priority[2] = "High";

// To access the first element
var level = priority[0];

// To modify the first element
priority[0] = "Not required";


// To find the length of an array
var x = priority.length

// To find the index position of an element in the array
var i = priority.indexOf("Normal")
```

See also Associative Array.

Array Properties

Property	Description
constructor	Returns the function that created the Array object's prototype.
length	Sets or returns the number of elements in an array.
prototype	Allows you to add properties and methods to an Array object.

Array Methods

Method	Description
concat()	Joins two or more arrays, and returns a copy of the joined arrays.
indexOf()	Search the array for an element and returns its position.
join()	Joins all elements of an array into a string.
lastIndexOf()	Search the array for an element, starting at the end, and returns its position.
pop()	Removes the last element of an array, and returns that element.
push()	Adds new elements to the end of an array, and returns the new length.
reverse()	Reverses the order of the elements in an array
shift()	Removes the first element of an array, and returns that element
slice()	Selects a part of an array, and returns the new array.
sort()	Sorts the elements of an array.
splice()	Adds/Removes elements from an array.
toString()	Converts an array to a string, and returns the result.
unshift()	Adds new elements to the beginning of an array, and returns the new length.
valueOf()	Returns the primitive value of an array

Boolean

A Boolean object is used to convert a non-boolean value to a boolean value (**true** or **false**).

```
var bool = new Boolean();
```

Boolean Properties

Property	Description
constructor	Returns the function that created the Boolean object's prototype.
prototype	Allows you to add properties and methods to an Boolean object.

Boolean Methods

Method	Description
toString()	Converts a Boolean value to a string, and returns the result (either “true” or ”false”). `bool.toString()`
valueOf()	Returns the primitive value of a Boolean object (either **true** or **false**). `bool.valueOf()`

Date

A Date object is used to work with dates and times.

Date objects are created with new Date().

There are four ways of creating a Date object:

```
var dt = new Date();
var dt = new Date(milliseconds);
var dt = new Date(dateString);
var dt = new Date(year, month, day, hours, minutes, seconds, milliseconds);
```

Example of setting a date

```
var startDate = new Date();
startDate.setFullYear(2013,0,14); // startDate == "Jan 14, 2013"
```

Note: month is zero-based i.e. 0 == ‘January’

Example of comparing two dates

```
var startDate = new Date();
```

```
startDate.setFullYear(2013,0,14);
var today = new Date();
if (startDate > today) {
    // startDate later than today's date
} else {
    // startDate is on or before today's date
}
```

Date Properties

Property	Description
constructor	Returns the function that created the Date object's prototype.
prototype	Allows you to add properties and methods to a Date object.

Date Methods

Method	Description
getDate()	Returns the day of the month (from 1-31).
getDay()	Returns the day of the week (from 0-6).
getFullYear()	Returns the year (four digits)
getHours()	Returns the hour (from 0-23).
getMilliseconds()	Returns the milliseconds (from 0-999).
getMinutes()	Returns the minutes (from 0-59).
getMonth()	Returns the month (from 0-11).
getSeconds()	Returns the seconds (from 0-59).
getTime()	Returns the number of milliseconds since midnight Jan 1, 1970.
getTimezoneOffset()	Returns the time difference between UTC time and local time, in minutes.
getUTCDate()	Returns the day of the month, according to universal time (from 1-31).
getUTCDay()	Returns the day of the week, according to universal time (from 0-6).
getUTCFullYear()	Returns the year, according to universal time (four digits).
getUTCHours()	Returns the hour, according to universal time (from 0-23).
getUTCMilliseconds()	Returns the milliseconds, according to universal time (from 0-999).
getUTCMinutes()	Returns the minutes, according to universal time (from 0-59).
getUTCMonth()	Returns the month, according to universal time (from 0-11).
getUTCSeconds()	Returns the seconds, according to universal time (from 0-59).
getYear()	Deprecated. Use the getFullYear() method instead.
parse()	Parses a date string and returns the number of milliseconds since midnight of January 1, 1970.
setDate()	Sets the day of the month of a date object.
setFullYear()	Sets the year (four digits) of a date object.

Method	Description
setHours()	Sets the hour of a date object .
setMilliseconds()	Sets the milliseconds of a date object.
setMinutes()	Set the minutes of a date object.
setMonth()	Sets the month of a date object.
setSeconds()	Sets the seconds of a date object.
setTime()	Sets a date and time by adding or subtracting a specified number of milliseconds to/from midnight January 1, 1970.
setUTCDate()	Sets the day of the month of a date object, according to universal time.
setUTCFullYear()	Sets the year of a date object, according to universal time (four digits).
setUTCHours()	Sets the hour of a date object, according to universal time.
setUTCMilliseconds()	Sets the milliseconds of a date object, according to universal time.
setUTCMinutes()	Set the minutes of a date object, according to universal time.
setUTCMonth()	Sets the month of a date object, according to universal time.
setUTCSeconds()	Set the seconds of a date object, according to universal time.
setYear()	**Deprecated.** Use the setFullYear() method instead
toDateString()	Converts the date portion of a Date object into a readable string
toGMTString()	**Deprecated.** Use the toUTCString() method instead.
toISOString()	Returns the date as a string, using the ISO standard.
toJSON()	Returns the date as a string, formated as a JSON date.
toLocaleDateString()	Returns the date portion of a Date object as a string, using locale conventions.
toLocaleTimeString()	Returns the time portion of a Date object as a string, using locale conventions.
toLocaleString()	Converts a Date object to a string, using locale conventions
toString()	Converts a Date object to a string.
toTimeString()	Converts the time portion of a Date object to a string
toUTCString()	Converts a Date object to a string, according to universal time
UTC()	Returns the number of milliseconds in a date string since midnight of January 1, 1970, according to universal time.
valueOf()	Returns the primitive value of a Date object.

Math

The Math object allows you to perform mathematical tasks.

The Math object does not need to be created to use it.

```
var pi = Math.PI;      // Returns PI value
var x = Math.sqrt(25); // Returns the square root of 25
```

Math Properties

Property	Description
E	Returns Euler's number (approx. 2.718).
LN2	Returns the natural logarithm of 2 (approx. 0.693).
LN10	Returns the natural logarithm of 10 (approx. 2.302).
LOG2E	Returns the base-2 logarithm of E (approx. 1.442).
LOG10E	Returns the base-10 logarithm of E (approx. 0.434).
PI	Returns the square root of 1/2 (approx. 0.707).
SQRT1_2	Allows you to add properties and methods to a Number object.
SQRT2	Returns the square root of 2 (approx. 1.414).

Math Methods

Method	Description
abs(x)	Returns the absolute value of x.
acos(x)	Returns the arccosine of x, in radians.
asin(x)	Returns the arcsine of x, in radians.
atan(x)	Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians.
atan2(y,x)	Returns the arctangent of the quotient of its arguments.
ceil(x)	Returns x, rounded upwards to the nearest integer.
cos(x)	Returns the cosine of x (x is in radians)
exp(x)	Returns the value of E^x.
floor(x)	Returns x, rounded downwards to the nearest integer.
log(x)	Returns the natural logarithm (base E) of x.
max(x,y,z,...,n)	Returns the number with the highest value.
min(x,y,z,...,n)	Returns the number with the lowest value.
pow(x,y)	Returns the value of x to the power of y.
random()	Returns a random number between 0 and 1.
round(x)	Rounds x to the nearest integer.
sin(x)	Returns the sine of x (x is in radians).
sqrt(x)	Returns the square root of x.
tan(x)	Returns the tangent of an angle.

Number

A Number object is an object wrapper for primitive numeric values.

```
var x = new Number(value);
```

Note: If the value parameter cannot be converted into a number, it returns NaN (Not-a-Number).

Number Properties

Property	Description
constructor	Returns the function that created the Number object's prototype.
MAX_VALUE	Returns the largest number possible in JavaScript.
MIN_VALUE	Returns the smallest number possible in JavaScript
NEGATIVE_INFINITY	Represents negative infinity (returned on overflow).
NaN	Represents a "Not-a-Number" value.
POSITIVE_INFINITY	Represents positive infinity (returned on overflow).
prototype	Allows you to add properties and methods to a Number object.

Number Methods

Method	Description
toExponential(x)	Converts a number into an exponential notation.
toFixed(x)	Formats a number with x numbers of digits after the decimal point.
toPrecision(x)	Formats a number to x length
toString()	Converts a Number object to a string.
valueOf()	Returns the primitive value of a Number object.

String

A String object is used to manipulate a series of characters.

A String object is created with new String() or by assigning a string to a variable.

```
var s = new String("Hello world!");
// or just:
var s = "Hello world!";

// Finding the length of a string
var message = "Hello World!";
var x = message.length; // x is 12

// Converting a string to uppercase
var message="Hello world!";
var x=message.toUpperCase(); // x is "HELLO WORLD!"

/* The indexOf()method returns the position (as a number) of the
first found occurrence of a specified text inside a string */
var str="Hello world, welcome to OpenAir.";
var n=str.indexOf("welcome");
```

String Properties

Property	Description
constructor	Returns the function that created the String object's prototype.
length	Returns the length of a string.
prototype	Allows you to add properties and methods to a String object.

String Methods

Method	Description
charAt()	Returns the character at the specified index.
charCodeAt()	Returns the Unicode of the character at the specified index.
concat()	Joins two or more strings, and returns a copy of the joined strings.
fromCharCode()	Converts Unicode values to characters.
indexOf()	Returns the position of the first found occurrence of a specified value in a string.
lastIndexOf()	Returns the position of the last found occurrence of a specified value in a string.
match()	Searches for a match between a regular expression and a string, and returns the matches.
replace()	Searches for a match between a substring (or regular expression) and a string, and replaces the matched substring with a new substring.
search()	Searches for a match between a regular expression and a string, and returns the position of the match.
slice()	Extracts a part of a string and returns a new string.
split()	Splits a string into an array of substrings.
substr()	Extracts the characters from a string, beginning at a specified start position, and through the specified number of character.
substring()	Extracts the characters from a string, between two specified indices.
toLowerCase()	Converts a string to lowercase letters.
toUpperCase()	Converts a string to uppercase letters.
trim()	Removes white space from both ends of a string.
valueOf()	Returns the primitive value of a String object.

JavaScript Operators

= is used to assign values.

+ is used to add values together.

JavaScript Operators:

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- Logical Operators

Note: JavaScript also contains a conditional operator that assigns a value to a variable based on a condition.

```
/* If the quantity ordered is more than 1000
 * then the packing cost is for a large packet,
 * otherwise the packing cost is for a small packet.
*/
packingCost=(quantity>1000)?largePacket:smallPacket;
```

Arithmetic Operators

Arithmetic operators are used to perform arithmetic between variables and/or values.

Operator	Description	Example
+	Addition	x=y+2;
-	Subtraction	x=y-2;
*	Multiplication	x=y*2;
/	Division	x=y/2;
%	Modulus (division remainder)	x=y%2;
++	Pre-Increment Post-Increment	x=++y; x=y++;
--	Pre-Decrement Post-Decrement	x=--y; x=y--;

Assignment Operators

Assignment operators are used to assign values to JavaScript variables.

Operator	Example	Equivalent to
=	x=y;	
+=	x+=y;	x=x+y;
-=	x-=y;	x=x-y;
*=	x*=y;	x=x*y;
/=	x/=y;	x=x/y;
%=	x%=y;	x=x%y;

Comparison Operators

Comparison operators are used in logical statements to determine equality or difference between variables or values.

Operator	Description
==	equal to

Operator	Description
===	exactly equal to (value and type)
!=	not equal
!==	not exactly equal (different value or type)
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Logical Operators

Logical operators are used to determine the logic between variables or values.

Operator	Description	Example
&&	and	(true && false) == false
||	or	(true || false) == true
!	not	!(false) == true

Reserved Words

The following words cannot be used as JavaScript variables, functions, methods, or object names:

- JavaScript Keywords
- JavaScript Reserved Keywords

JavaScript Keywords

break	for	throw
case	function	try
catch	if	typeof
continue	in	var
default	instanceof	void
delete	new	while
do	return	with
else	switch	finally
this		

JavaScript Reserved Keywords

abstract	export	long

synchronized	boolean	extends
native	throws	byte
final	package	transient
char	float	private
volatile	class	goto
protected	const	implements
public	debugger	import
short	double	int
static	enum	interface
super		

Escape Sequences

An escape sequence in created using a backslash to identify the special character.

JavaScript supports the following escape sequences:

Escape Sequence	Description
\'	Single quote or apostrophe
\"	Double quote
\\	Backslash
\0	Null character (that is backslash plus zero)
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\xXX	Latin-1 character specified by two hexadecimal digits. For example, the copyright symbol its \xa9
\uXXXX	Unicode character specified by four hexadecimal digits. For example, the π symbol is \u03c0.

Note: If you include the backslash in front of any other character than those shown in the table, JavaScript will ignore the backslash.

Scripting Best Practices

The following sections offer a series of best practices which you can apply to your scripting. These best practices are meant to help you succeed with scripting, and create scripts which:

- Can be verified and tracked as working correctly
- Can recover from errors
- Don't need continuous maintenance

Following these practices can maximize your investment in scripting.

Development

- Confirm that your development and production accounts have the necessary switches enabled. You must have the "Enable user script support for Web Service API methods" feature to use the NSOA.wsapi functions. See Scripting Switches.

  Note: By default, scheduled triggers are disabled on sandboxes.

- Test your scripts in a sandbox account before deploying them to a production account. See Testing Form Scripts.
- Use platform role permissions to control access to critical features of the Scripting Center and Scripting Studio. See Platform Role Permissions.
- Always check for errors and handle errors appropriately. See Error Handling.
- Consider mobile users when designing scripted solutions. Scripts triggered by "On submit", "Before save", or "After save" are not supported for mobile devices. See Scripting for Mobile Devices.
- Remember that some NSOA functions have no effect for certain script events. For example, NSOA.form.error has no effect on the "After save" form event. See NSOA Functions.
- Use NSOA.form.setValue instead of a wsapi call when possible. See NSOA.form.setValue(field, value).
- Use NSOA.form.confirmation / warning / error messages to give user feedback. See NSOA.form.confirmation(message), NSOA.form.warning(message), and NSOA.form.error(field, message).
- Write scripts which fail safely and are re-entrant to avoid data corruption.

Writing Scripts in JavaScript

- Use comments to explain the script. See JavaScript Overview.
- Use indentation and white space to make your code easy to read. See JavaScript Overview.
- Use meaningful names for variables and functions. See Variables.
- Be consistent in the way you name variables and functions. Use camel case. See Variables.
- Be careful with quotation marks. Quotation marks are used in pairs around strings and that both quotation marks must be of the same style (either single or double). See String.
- Be careful with equal signs. You should not use a single equal sign for comparisons. See Comparison Operators.
- Declare variables explicitly using the var keyword. See Variables.

- Be careful not to create endless loops. Your loops should always have an exit condition. See Loops.
- Rather than creating a long “if .. else” chain, use the “switch” statement. See Conditional Statements.
- Use “try and catch” blocks to handle errors. See Error Handling.

SOAP / WSAPI

- Always check that any SOAP API call was successful before using the results. See Handling SOAP Errors.
- Where possible, batch a series of objects together into a single SOAP API call rather than making a separate call for each object. See Making SOAP Calls.
- The updated and created fields are maintained automatically by the system. You can read these values, but they cannot be modified. See Making SOAP Calls.
- You cannot delete an entity (database record) which has dependent records. You must first delete all the dependent records. See Deleting data.
- You must specify a limit attribute to read data. Make this limit as small as possible if you will only access the first record (for example, set the limit attribute to 1). See Attribute and Reading data.
- Don’t forget to specify the ‘update_custom’ attribute to update a custom field. See Updating Custom Fields.

Logs

- Use log messages to verify your script is executing as expected and to help you to troubleshoot scripts which behave unexpectedly. Logs.
- Set the log severity to “Warning” or “Error” to save space and improve system performance. See Log Severity.
- Set the log severity of a deployed script to "Debug" or "Trace" to track down errors which only occur for a deployed script. See Log Severity.
- Use the “delete log entries” maintenance task to delete log entries which are no longer needed. Use this maintenance task when your system is not busy and be careful not to delete log entries which you may need. See Delete Log Entries.
- Always keep at least the last 30 days of logs. See Delete Log Entries.

Data Access

- Make sure your script can run correctly for any user that may trigger the script. Form scripts are executed within the context of the user who is logged in. See NSOA.wsapi.disableFilterSet([flag]).
- When setting “Select user to execute a script deployment”, create a dedicated user with the minimum necessary permissions dedicated to this purpose. See Platform Role Permissions.

Governance

- Make sure that none of the execution paths through your script will exceed the allowed units limit. See Scripting Governance.

- Don't try to do too much in a script (especially in a form script). Make sure your script can finish well within the allowed time limits. Your script needs to be able to run successfully even when the server is under load. See Scripting Governance.
- Always try to reduce the number of units your scripts consume. Notice that NSOA.record functions consume zero units, but NSOA.wsapi functions consume 10 units for each call. See NSOA Functions.

Maintainable Scripts

- Access custom fields using the **__c** notation (note the two underscore characters). The old approach to read custom fields using custom_ with the internally assigned custom field number appended is still supported but NOT recommended. In addition, scripts using the custom_ notation may not be portable between environments, for example, from sandbox to production. See Reading Custom Fields.
- Reference custom fields used by the script. This will prevent changes to custom fields from unintentionally breaking a script. See Updating Custom Fields.
- Reference parameters used by the script. Referencing a parameter prevents the parameter from being deleted or changed in a way which will affect the script. See Creating Parameters.
- Use library scripts to package the complexity of a scripted solution into calling scripts and supporting functions. This will result in scripts which are easier to build and maintain. You can build libraries of proven functions to reduce the cost of future development and maintenance. See Creating Library Scripts.
- Use script parameters to create scripts which can be configured without needing to change the script. See Creating Parameters.
- Use script terminology to allow scripts to immediately reflect any terminology changes made by the administrator. See Accessing Terminology.
- Use platform solutions to package all the elements of a script into a single file. See Creating Solutions.

Real World Use Cases

The following examples are provided to assist you in developing your own scripts. Please be aware of the disclaimer for these examples.

Important: NetSuite Inc. may provide sample code in SuiteAnswers, User Guides, or elsewhere through help links. All such sample code is provided “as is” and “as available”, for use only with an authorized OpenAir Service account. Sample code is made available subject to the SuiteCloud Terms of Service at www.netsuite.com/tos where the term “Service” shall mean the OpenAir Service and the sample code shall be included in the “SuiteCloud Technologies”. NetSuite may modify or remove sample code at any time without notice.

- Validation
  - Ensure value of multiple commissions fields equals 100%
  - Require notes field to be populated on time entries when more than 8 hours in a day
  - When submitting an expense report, validate each ticket has an attachment (e.g. scanned receipt)
  - Ensure resource time entry matches booking planning and project worked hours
- Automation
  - Optionally create a new Customer PO when editing a project
  - Create time entries from task assignments when the user creates a new timesheet
  - Control budgeted hours for a project using the project budget feature and a custom hours field
- Workflow
  - Prevent a booking from being created if the selected resource has approved time off during the booking period
  - Prevent closing a project that has open issues
  - Automatically create a new issue when project stage is "at risk" and prevent project stage from changing until this issue is resolved
  - Send an alert email when a scheduled script completes

Using the examples

Before you start, make sure you have the necessary switches enabled in your test account, see Getting Started.

Note: You need to be logged in as an administrator to work with the development environment.

To try out the examples:

1. Log in as an Administrator and navigate to the **Administration** > **Scripting Center**.
2. Follow the steps described in the **Setup** section for the example. See Quick Start for more details.

   Tip: Save time by using the solution file link provided at the top of each **setup** section, see Creating Solutions.
3. See Scripting Workflow and Testing Form Scripts.

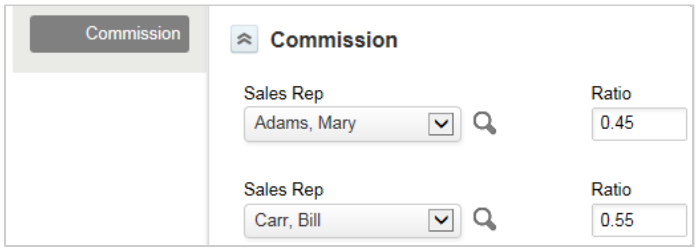
Validation

Ensure value of multiple commissions fields equals 100%

This script checks to ensure that sales commission amounts equal 100% (1.00) before allowing the project to be saved. It can be modified to support any number of sales rep commissions fields.

- Enrich records with additional sales management information.
- Easily reusable/extendible with minimal effort.
- Might solve this case using allocation grid custom field, but this solution allows user pick lists and retains a more detailed audit trail.

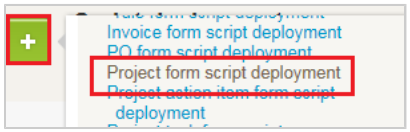
A new custom **Commission** section has been added to the project form. A user script is triggered as the project saves to validate the commission values entered.



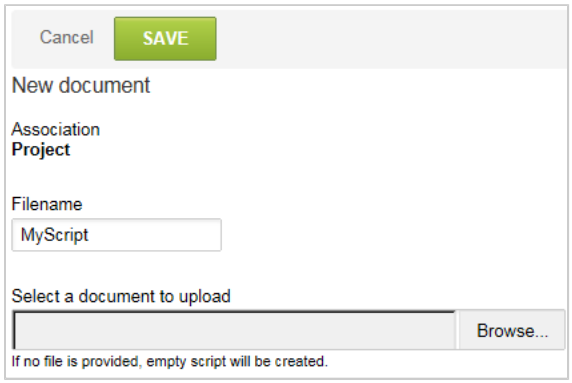
Setup

Follow the steps below or download the solutions file, see Creating Solutions for details.

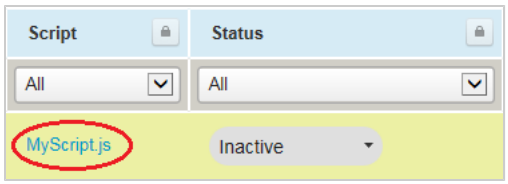
1. Create a new **Project form script deployment**.



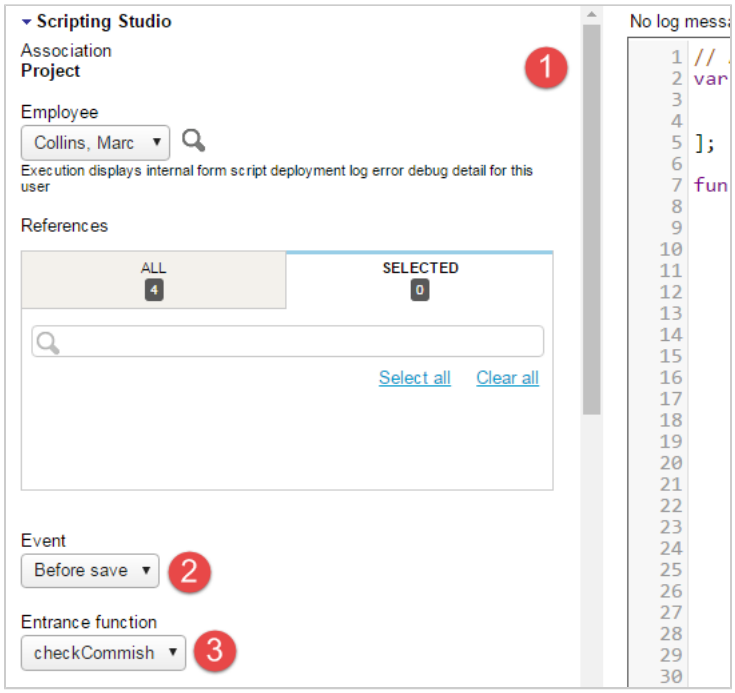
2. Enter a **Filename** and click **SAVE**. The extension '.js' is automatically appended if not supplied.



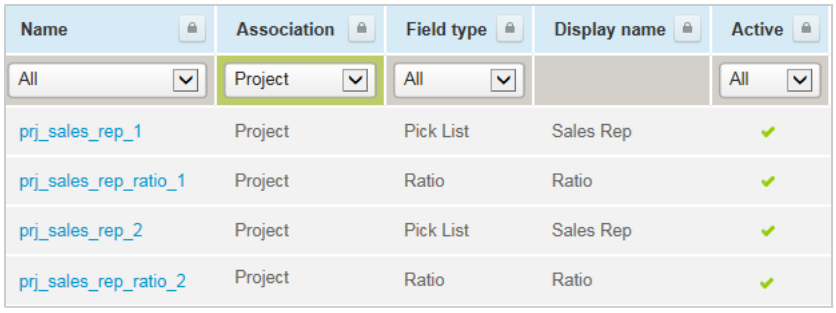
3. Click on the script link to launch the **Scripting Studio**.



4. (1) Copy the **Program Listing** below into the editor, (2) set the **Before save** event, and set **checkCommish** as the Entrance Function.

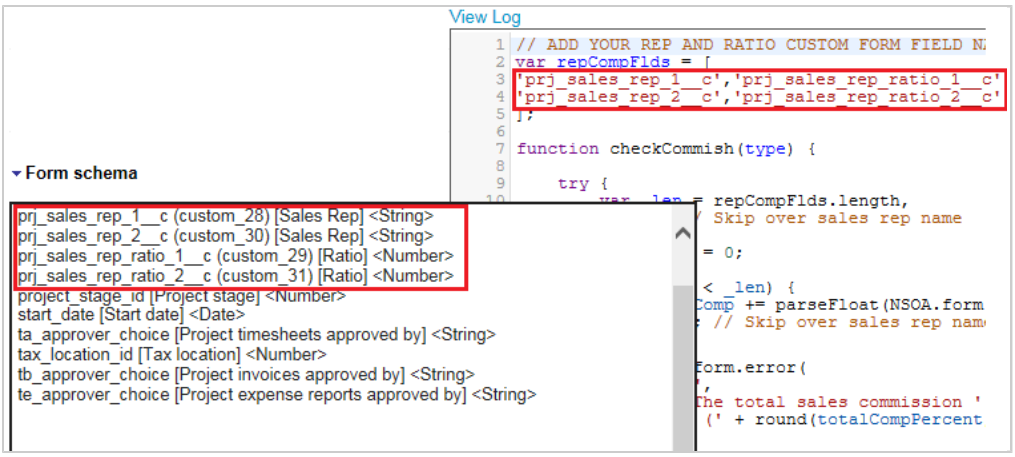


5. Set up the required number of custom field pairs for **Project**. The first in each pair is a **Pick List** with a **List source** of User. The second in each pair is a **Ratio**. You can set the **Divider text** for the very first custom field to **Commission** to place the custom fields in their own section.



Name	Association	Field type	Display name	Active
All	Project	All		All
prj_sales_rep_1	Project	Pick List	Sales Rep	✔
prj_sales_rep_ratio_1	Project	Ratio	Ratio	✔
prj_sales_rep_2	Project	Pick List	Sales Rep	✔
prj_sales_rep_ratio_2	Project	Ratio	Ratio	✔

6. Use the **Form schema** to identify the correct param names for the custom fields and change the array at the top of the script accordingly.



Tip: If the new custom fields are not listed in the **Form schema**, navigate to Projects, open a project form (this with will refresh the custom field list), and then open the Scripting Studio again.

Program Listing

```
// ADD YOUR REP AND RATIO CUSTOM FORM FIELD NAMES TO THE ARRAY BELOW
var repCompFlds = [
    'prj_sales_rep_1__c', 'prj_sales_rep_ratio_1__c', // Use Form schema to find param names
    'prj_sales_rep_2__c', 'prj_sales_rep_ratio_2__c'
];

function checkCommish(type) {

    try {
        var _len = repCompFlds.length,
            _i = 1, // Skip over sales rep name
            _j = 0,
            totalComp = 0;

        while (_i < _len) {
            totalComp += parseFloat(NSOA.form.getValue(repCompFlds[_i]));
            _i += 2; // Skip over sales rep name
        }

        var totalCompRound = round(totalComp, 2),
            totalCompPercent = totalCompRound * 100;
        if (totalCompRound !== 0 && totalCompRound != 1) {

            NSOA.form.error(
                '',
                'The total sales commission ' + totalCompRound +
                ' (' + round(totalCompPercent, 2) + '%) must equal 100%!'
            );

            for (_j; _j < _len; _j++) {
                NSOA.form.error(repCompFlds[_j], 'Please check and re-save.');
```

```
            }
        }

    } catch (e) {
        NSOA.meta.log('fatal', "Error running the script: " + e);
    }

}

function round(number, decimals) {
    return (Math.round(number * Math.pow(10, 2)) / Math.pow(10, 2)).toFixed(decimals);
}
```

Require notes field to be populated on time entries when more than 8 hours in a day

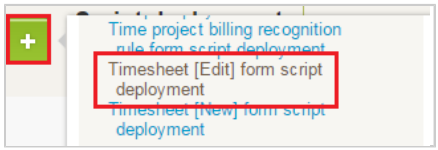
This script validates that the notes field has been populated on time entries when more than 8 hours in a day.

- Validation occurs before a timesheet may be submitted for approval
- Ensures daily overtime hours are annotated
- Easily customizable to support policy on different time periods, or groupings (e.g. by project)

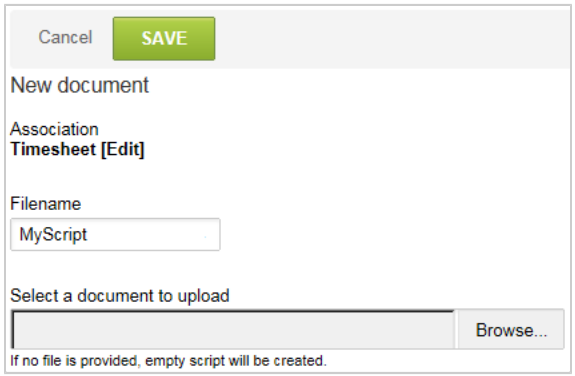
Setup

Follow the steps below or download the solutions file, see Creating Solutions for details.

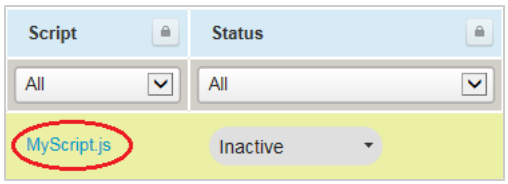
1. Create a new **Timesheet [Edit] form script deployment**.



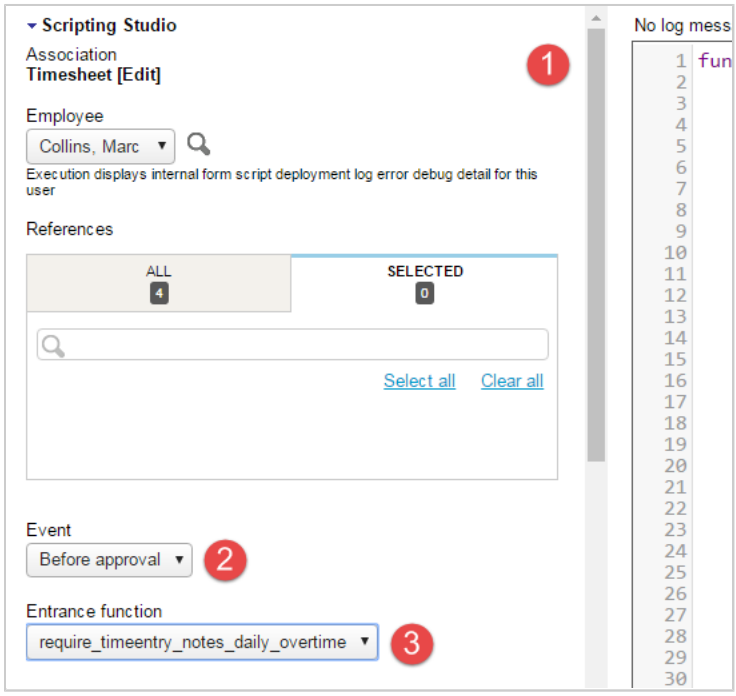
2. Enter a **Filename** and click **SAVE**. The extension '.js' is automatically appended if not supplied.



3. Click on the script link to launch the **Scripting Studio**.



4. (1) Copy the **Program Listing** below into the editor, (2) set the **Before approval** event, and set **require_timeentry_notes_daily_overtime** as the Entrance Function.



Program Listing

```
function require_timeentry_notes_daily_overtime() {

    // Load task data
    var task = new NSOA.record.oaTask();
    task.timesheetid = NSOA.form.getOldRecord().id;
    NSOA.meta.log('debug', "got ts id " + task.timesheetid);

    var readRequest = {
        type: "Task",
        fields: "id, date, decimal_hours, notes",
        method: "equal to",
        objects: [task],
        attributes: [{
            name: "limit",
            value: "1000"
        }]
    };

    var arrayOfreadResult = NSOA.wsapi.read(readRequest);
```

```
    // Analyze the timesheet
    var ts_total_by_day = {};
    var task_no_notes_by_day = {};
    if (!arrayOfreadResult || !arrayOfreadResult[0])
        NSOA.form.error('', "Internal error loading timesheet details.");

    else if (arrayOfreadResult[0].errors === null && arrayOfreadResult[0].objects)
        arrayOfreadResult[0].objects.forEach(
            function(o) {
                // Get yyyy-mm-dd part
                var date = o.date.substr(0, 10);

                // Track total hours for this day
                if (ts_total_by_day[date] === undefined)
                    ts_total_by_day[date] = Number(o.decimal_hours);
                else if (ts_total_by_day[date] <= 8)
                    ts_total_by_day[date] += Number(o.decimal_hours);
                else
                    return; // Already reported form error if we got here
                NSOA.meta.log('debug', date + " -> " + ts_total_by_day[date]);

                // Track time entries with no notes
                if (task_no_notes_by_day[date] === undefined)
                    task_no_notes_by_day[date] = [];
                if (o.notes === undefined || o.notes.length === 0)
                    task_no_notes_by_day[date].push(o.id);

                // Check the policy
                if (ts_total_by_day[date] > 8 && task_no_notes_by_day[date] !== undefined &&
                    task_no_notes_by_day[date].length > 0) {
                    NSOA.meta.log('trace', ts_total_by_day[date] + " -> " + task_no_notes_by_da
y[date].length);
                    NSOA.form.error('', "Notes are required on all " + date +
                        " time entries: total reported time that day is more than 8 hours.");
                }
            }
        );
}
```

When submitting an expense report, validate each ticket has an attachment (e.g. scanned receipt)

This script validates each receipt has an attachment when submitting an expense report.

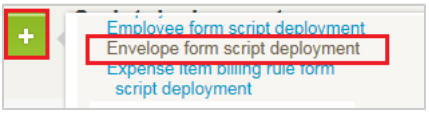
- Verifies whether document attachments exist on a ticket record
- Does not require an attachment if "Missing receipt" is checked

> **Note:** The **Enable the missing paper receipt feature** switch needs to be enabled for this option.

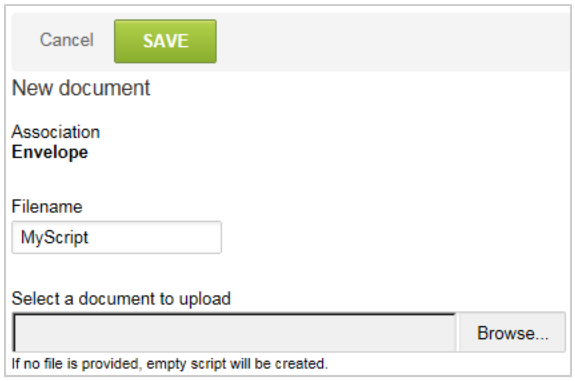
Setup

Follow the steps below or download the solutions file, see Creating Solutions for details.

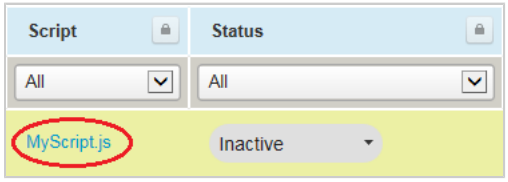
1. Create a new **Envelope form script deployment**.



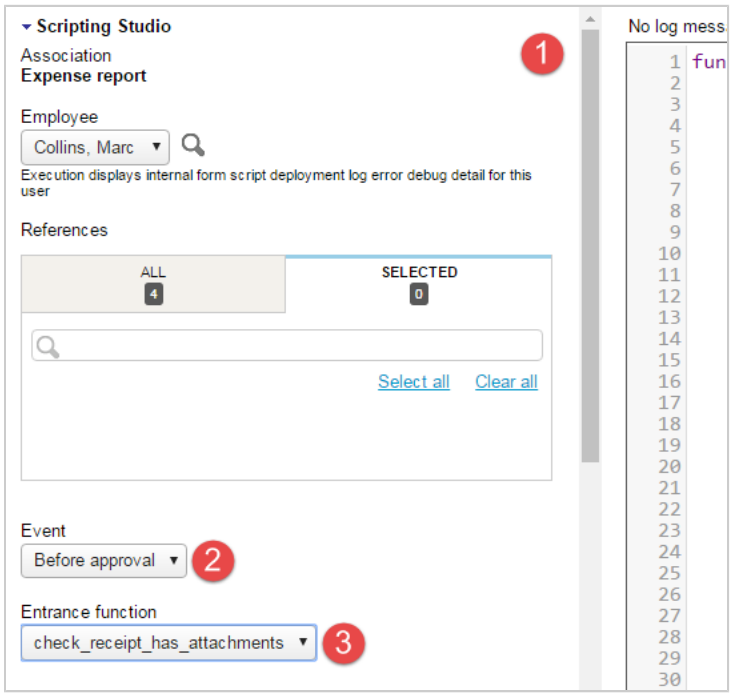
2. Enter a **Filename** and click **SAVE**. The extension '.js' is automatically appended if not supplied.



3. Click on the script link to launch the **Scripting Studio**.



4. (1) Copy the **Program Listing** below into the editor, (2) set the **Before approval** event, and set **check_receipt_has_attachments** as the Entrance Function.



Program Listing

```
function check_receipt_has_attachments(type) {

    // return if not an approve_request
    if (type != 'approve_request')
        return;

    // Load receipt data
    var envelope = NSOA.form.getOldRecord();
    var ticket = new NSOA.record.oaTicket();
    ticket.envelopeid = envelope.id;

    var readRequest = {
        type: "Ticket",
        fields: "id, attachmentid, reference_number, missing_receipt",
        method: "equal to",
        objects: [ticket],
        attributes: [{
            name: "limit",
            value: "250"
        }]
    };

    var arrayOfreadResult = NSOA.wsapi.read(readRequest);
    var missingAttachment = [];
    if (!arrayOfreadResult || !arrayOfreadResult[0])
        NSOA.form.error('', "Internal error reading envelope receipts.");

    else if (arrayOfreadResult[0].errors === null && arrayOfreadResult[0].objects)
        arrayOfreadResult[0].objects.forEach(
            function(o) {
                if (o.attachmentid === '0' && o.missing_receipt != '1')
                    missingAttachment.push(o.reference_number);
            }
        );

    if (missingAttachment.length > 0) {
        NSOA.form.error('',
            "The following receipts (by reference number) are missing an attachment: " +
            missingAttachment.join(", "));
    }
}
```

Ensure resource time entry matches booking planning and project worked hours

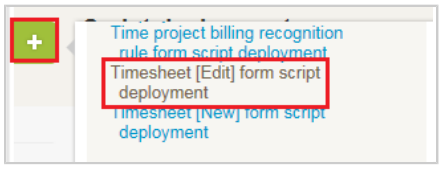
This script ensures that resources are not logging time after the task assignment related booking end date, or exceeding assigned planned hours.

- Keep worked time in-sync with planned allocation
- Stay within planned budget
- Works on mobile devices (iPhone/Android)

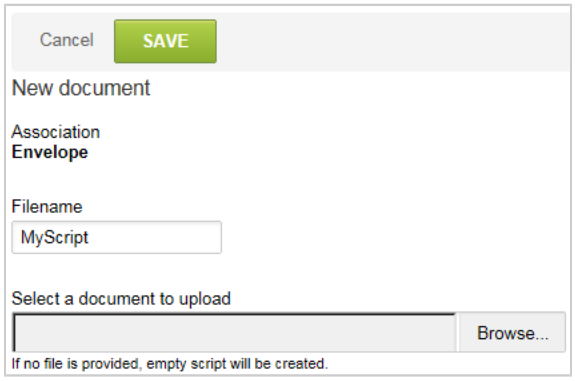
Setup

Follow the steps below or download the solutions file, see Creating Solutions for details.

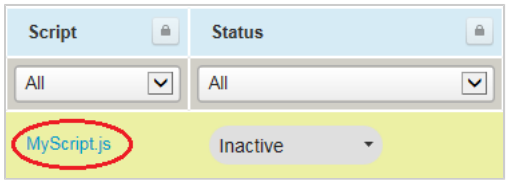
1. Create a new **Timesheet [Edit] form script deployment**.



2. Enter a **Filename** and click **SAVE**. The extension '.js' is automatically appended if not supplied.



3. Click on the script link to launch the **Scripting Studio**.



4. (1) Copy the **Program Listing** below into the editor, (2) set the **Before approval** event, and set **verify_timeentry_policy** as the Entrance Function.

Program Listing

```
function verify_timeentry_policy(type) {
    var timesheet = NSOA.form.getOldRecord();

    // Only check on approval request and if current user is the timesheet owner
    if (type != 'approve_request' || timesheet.userid != NSOA.wsapi.whoami().id)
        return;

    // Load task data
    var taskFilter = new NSOA.record.oaTask();
    taskFilter.timesheetid = timesheet.id;

    // disable current user's filter for this script
    NSOA.wsapi.disableFilterSet(true);

    // Analyze tasks to load related records
    var task_readRequest = {
```

```
        type: "Task",
        fields: "id, date, projecttaskid, decimal_hours",
        method: "equal to",
        objects: [taskFilter],
        attributes: [{
            name: "limit",
            value: "1000"
        }, {
            name: "filter",
            value: "current-user"
        }]
    };
    var task_arrayOfreadResult = NSOA.wsapi.read(task_readRequest);

    var tasks_by_uniqueKey = {};
    var ts_pta_worked_hours = {};
    var ptaFilters = {};
    var bookingFilters = {};
    if (!task_arrayOfreadResult || !task_arrayOfreadResult[0])
        NSOA.form.error('', "Internal error loading %task% details.");
    else if (task_arrayOfreadResult[0].errors === null && task_arrayOfreadResult[0].objects)
        task_arrayOfreadResult[0].objects.forEach(function(task) {
            NSOA.meta.log('debug', "Got task: " + JSON.stringify(task));

            // Only consider project task assignments
            if (!task.projecttaskid)
                return;

            // Correlate booking <=> project_task_assignment via task tuple(project_task_id,use
r_id)
            var uniqueKey = task.projecttaskid;

            // Store information about this time entry
            if (tasks_by_uniqueKey[uniqueKey])
                tasks_by_uniqueKey[uniqueKey].push(task);
            else
                tasks_by_uniqueKey[uniqueKey] = [task];
            ts_pta_worked_hours[uniqueKey] += parseFloat(task.decimal_hours);

            // Prepare related booking filters
            var bookingFilter = new NSOA.record.oaBooking();
            bookingFilter.project_taskid = task.projecttaskid;
            bookingFilters[uniqueKey] = bookingFilter; // elimiate duplicates

            // Prepare related project_task_assign filters
            var ptaFilter = new NSOA.record.oaProjecttaskassign();
            ptaFilter.projecttaskid = task.projecttaskid;
            ptaFilters[uniqueKey] = ptaFilter; // elimiate duplicates

        });
    else
        return; // assume no data found

    // Now load and analyze project task assignments (one read request)
    if (Object.keys(ptaFilters).length > 0) {
```

```
        var equalTo = [];
        for (var i = 0; i < Object.keys(ptaFilters).length; i++)
            equalTo.push("equal to");
        var ptaFilter = [];
        Object.keys(ptaFilters).forEach(function(k) {
            ptaFilter.push(ptaFilters[k]);
        });
        var pta_readRequest = {
            type: "Projecttaskassign",
            fields: "id, planned_hours, userid, projecttaskid",
            method: equalTo.join(', or '),
            objects: ptaFilter,
            attributes: [{
                name: "limit",
                value: "1000"
            }, {
                name: "filter",
                value: "current-user"
            }]
        };
        NSOA.meta.log('debug', "pta_readRequest=" + JSON.stringify(pta_readRequest));
        var pta_arrayOfreadResult = NSOA.wsapi.read(pta_readRequest);
        NSOA.meta.log('debug', "pta_arrayOfreadResult=" + JSON.stringify(pta_arrayOfreadResult)
);

        var pta_planned_hours = {};
        var pta_worked_hours = {};
        if (!pta_arrayOfreadResult || !pta_arrayOfreadResult[0])
            NSOA.form.error('', "Internal error loading %project_task% assignment details.");
        else if (pta_arrayOfreadResult[0].errors === null && pta_arrayOfreadResult[0].objects)
            pta_arrayOfreadResult[0].objects.forEach(function(pta) {
                var uniqueKey = pta.projecttaskid;
                var planned_hours = parseFloat(pta.planned_hours);

                // Skip assignment if no planned hours
                if (!planned_hours)
                    return;

                // Compute worked hours for current user's assignment
                var taskFilter = new NSOA.record.oaTask();
                taskFilter.projecttaskid = pta.projecttaskid;
                taskFilter.userid = pta.userid;
                var task_readRequest = {
                    type: "Task",
                    fields: "id, decimal_hours",
                    method: "equal to",
                    objects: [taskFilter],
                    attributes: [{
                        name: "limit",
                        value: "1000"
                    }, {
                        name: "filter",
                        value: "current-user"
                    }]
                };
```

```
                var task_arrayOfreadResult = NSOA.wsapi.read(task_readRequest);

                var worked_hours = 0;
                if (!task_arrayOfreadResult || !task_arrayOfreadResult[0])
                    NSOA.form.error('', "Internal error loading %timeentry% assignment details.
");
                else if (task_arrayOfreadResult[0].errors === null && task_arrayOfreadResult[0]
.objects)
                    task_arrayOfreadResult[0].objects.forEach(function(task) {
                        worked_hours += parseFloat(task.decimal_hours);
                    });

                // Verify user's worked hours haven't exceeded as a result of this timesheet
                NSOA.meta.log('debug', "worked=" + worked_hours + ",planned=" + planned_hours);

                if (worked_hours && worked_hours > planned_hours) {
                    var pt = NSOA.record.oaProjecttask(pta.projecttaskid);
                    var error = "Worked %hours% (" + worked_hours + ") including %timeentrys% o
n this %timesheet% exceeds your planned %hours% (" + planned_hours + ") for %project% '" + NSOA
.record.oaProject(pt.projectid).name + "' %project_task% '" + pt.name + "'.";
                    if (ts_pta_worked_hours[uniqueKey]) {
                        error += "This %timesheet% adds " + ts_pta_worked_hours[uniqueKey] + "
%hours%.";
                        var worked_excluding_ts = worked_hours - ts_pta_worked_hours[uniqueKey]
;
                        if (worked_excluding_ts <= planned_hours)
                            error += "Please reduce your worked %hours% by " + (worked_hours -
planned_hours) + ".";
                    }
                    NSOA.form.error('', error);
                }
            });
    }

    // Now load and analyze bookings
    var df = require('lib_date_format');
    if (Object.keys(bookingFilters).length > 0) {
        var equalTo = [];
        for (var i = 0; i < Object.keys(bookingFilters).length; i++)
            equalTo.push("equal to");
        var bookingFilter = [];
        Object.keys(bookingFilters).forEach(function(k) {
            bookingFilter.push(bookingFilters[k]);
        });
        var booking_readRequest = {
            type: "Booking",
            fields: "id, enddate, userid, project_taskid, projectid",
            method: equalTo.join(', or '),
            objects: bookingFilter,
            attributes: [{
                name: "limit",
                value: "1000"
            }, {
                name: "filter",
                value: "current-user"
```

```
            }]
        };
        NSOA.meta.log('debug', "booking_readRequest=" + JSON.stringify(booking_readRequest));
        var booking_arrayOfreadResult = NSOA.wsapi.read(booking_readRequest);
        NSOA.meta.log('debug', "booking_arrayOfreadResult=" + JSON.stringify(booking_arrayOfrea
dResult));

        if (!booking_arrayOfreadResult || !booking_arrayOfreadResult[0])
            NSOA.form.error('', "Internal error loading %project_task% assignment details.");
        else if (booking_arrayOfreadResult[0].errors === null && booking_arrayOfreadResult[0].o
bjects)
            booking_arrayOfreadResult[0].objects.forEach(function(booking) {
                var uniqueKey = booking.project_taskid;
                NSOA.meta.log('debug', uniqueKey + "," + JSON.stringify(tasks_by_uniqueKey));
                var tasks = tasks_by_uniqueKey[uniqueKey];
                if (!tasks)
                    return;
                tasks.forEach(function(task) {
                    var taskDate = new Date(task.date.substr(0, 10));
                    taskDate.setDate(taskDate.getDate() + 1);
                    NSOA.meta.log('debug', JSON.stringify(task));
                    var bookingDate = new Date(booking.enddate.substr(0, 10));
                    NSOA.meta.log('debug', "Check: " + taskDate + '>' + bookingDate);
                    if (taskDate && bookingDate) {
                        if (taskDate > bookingDate) {
                            var pt = NSOA.record.oaProjecttask(booking.project_taskid);
                            NSOA.form.error('', "Task on date " + df.userDateFormat(taskDate) +
 " exceeds booking end date " + df.userDateFormat(bookingDate) + " for for %project% '" + NSOA.
record.oaProject(pt.projectid).name + "' %project_task% '" + pt.name + "'.");
                            return;
                        }
                    }
                });
            });
    }
}
```

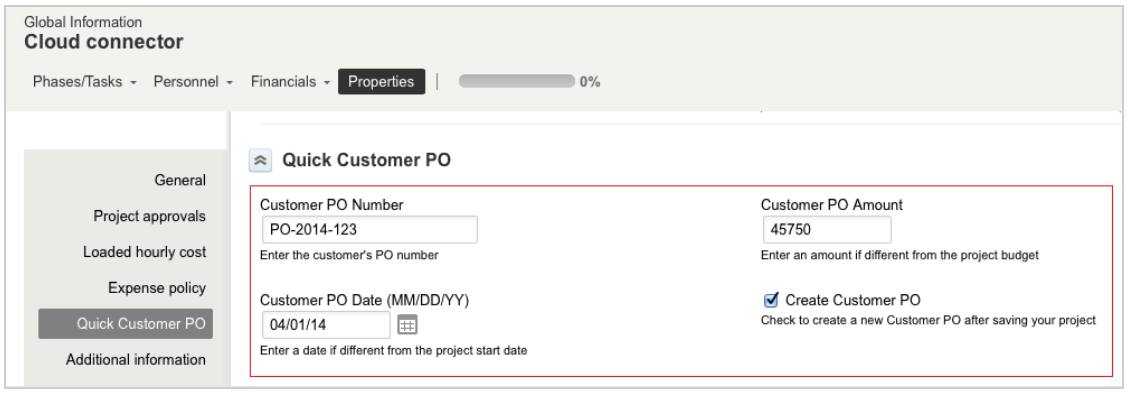
Automation

Optionally create a new Customer PO when editing a project

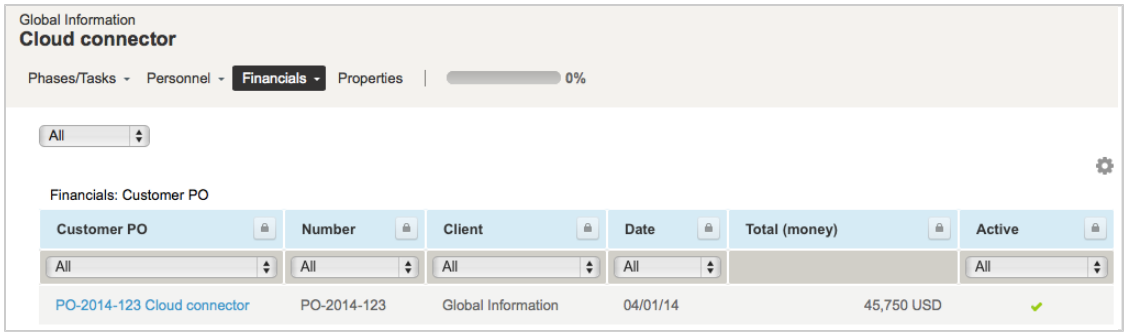
This example allows a customer to streamline their business processes by quickly creating customer POs as a part of saving a project.

- Saves ~7 mouse clicks
- Can be used on a per-project basis (not required)
- Can be used multiple times if many POs are required on one project

A new custom **Quick Customer PO** section has been added to the project form. A user script is triggered as the project saves to create the specified customer PO.



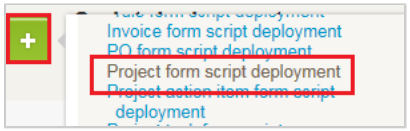
The **Create Customer PO** check box signals that a new customer PO record is to be created and the customer PO fields cleared allowing the user to quickly create additional customer POs. When the project is saved the specified **Customer PO** is then created.



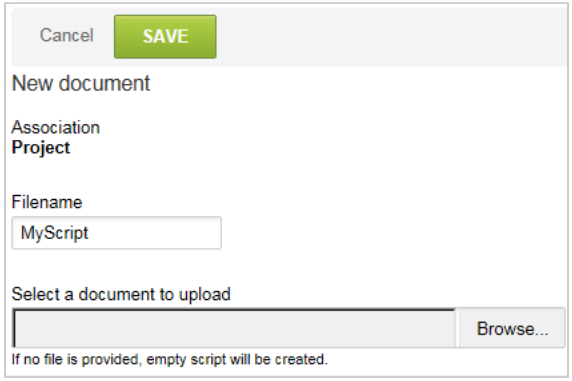
Setup

Follow the steps below or download the solutions file, see Creating Solutions for details.

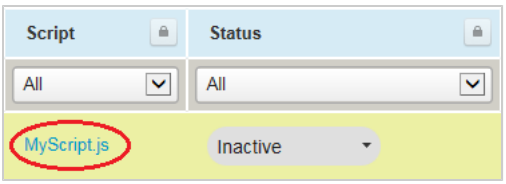
1. Create a new **Project form script deployment**.



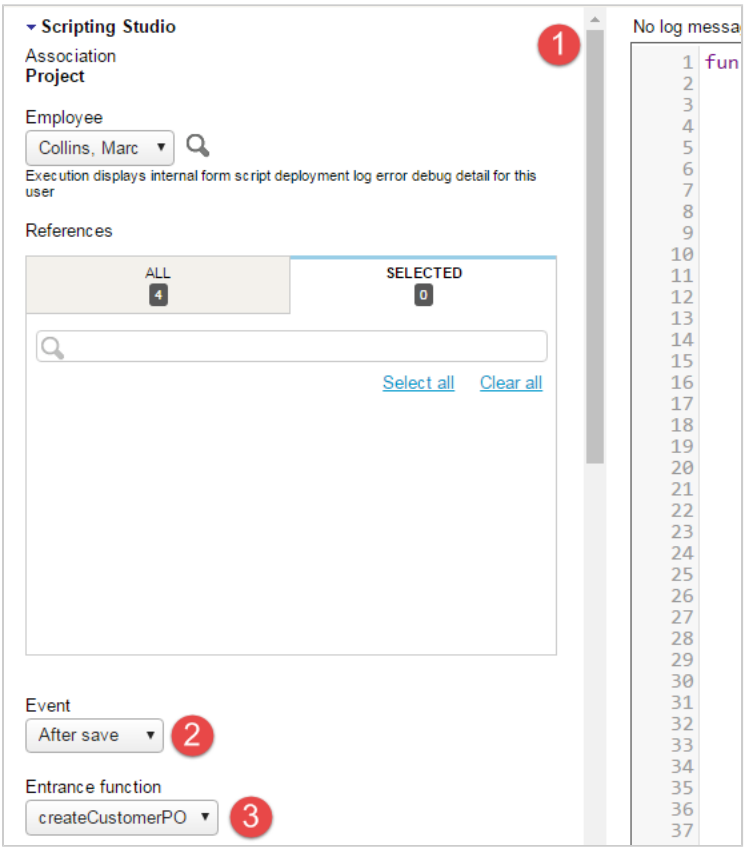
2. Enter a **Filename** and click **SAVE**. The extension '.js' is automatically appended if not supplied.



3. Click on the script link to launch the **Scripting Studio**.



4. (1) Copy the **Program Listing** below into the editor, (2) set the **After save** event, and set **createCustomerPO** as the Entrance Function.



5. Set up the following custom fields for **Project**. You can set the **Divider text** for the very first custom field to **Quick Customer PO** to place the custom fields in their own section.

Name	Association	Field type	Display name	Active
All	Project	All		All
prj_custpo_num	Project	Text	Customer PO Number	✔
prj_custpo_amt	Project	Numeric	Customer PO Amount	✔
prj_custpo_date	Project	Date	Customer PO Date (MM/DD/YY)	✔
prj_create_po	Project	Checkbox	Create Customer PO	✔

Add the following hints:

- prj_custpo_num — **Hint:** Enter the customer's PO number.
- prj_custpo_amt — **Hint:** Enter an amount if different from the project budget.

- prj_custpo_date — **Hint:** Enter a date if different from the project start date.
- prj_create_po — **Hint:** Check to create a new Customer PO after saving your project.

Program Listing

```
function createCustomerPO(type) {
    try {
        var FLD_CUSTPO_NUM = 'prj_custpo_num__c',
            FLD_CUSTPO_AMT = 'prj_custpo_amt__c',
            FLD_CUSTPO_DATE = 'prj_custpo_date__c',
            FLD_CREATE_PO = 'prj_create_po__c';

        // get updated project record fields
        var updPrj = NSOA.form.getNewRecord();

        // if the "Create PO" checkbox is checked and a PO number is entered, create a PO
        if (updPrj[FLD_CREATE_PO] == '1' && updPrj[FLD_CUSTPO_NUM]) {

            var recCustPO = new NSOA.record.oaCustomerpo();
            recCustPO.number = updPrj[FLD_CUSTPO_NUM];
            recCustPO.name = updPrj[FLD_CUSTPO_NUM] + ' ' + updPrj.name;

            // use the PO date if available, otherwise use project start date
            if (updPrj[FLD_CUSTPO_DATE] != '0000-00-00') {
                recCustPO.date = updPrj[FLD_CUSTPO_DATE];
            } else {
                recCustPO.date = updPrj.start_date;
            }

            // currency custom fields return ISO-#.##; remove the ISO code and dash
            var cleanAmt = updPrj[FLD_CUSTPO_AMT].replace(/-\w{3}/, '');

            // use the PO amt if available, otherwise use project budget
            if (cleanAmt && cleanAmt != '0.00') {
                recCustPO.total = cleanAmt;
            } else if (updPrj.budget && updPrj.budget > 0.00) {
                recCustPO.total = updPrj.budget;
            }

            recCustPO.currency = updPrj.currency;
            recCustPO.customerid = updPrj.customerid;
            recCustPO.active = 1;

            // disable the current user's filter set while script runs
            NSOA.wsapi.disableFilterSet(true);

            // add the new customer po to the project
            var custPOResults = NSOA.wsapi.add(
                [recCustPO]
            );

            if (!custPOResults || !custPOResults[0]) {
                NSOA.meta.log('error', 'Unexpected error! Customer PO was not created.');
            } else if (custPOResults[0].errors) {
```

```
        custPOResults[0].errors.forEach(function(err) {
            NSOA.meta.log('error', 'Error: ' + err.code + ' - ' + err.comment);
        });
    } else {
        // new customer po to project link object
        var recCustPOtoProj = new NSOA.record.oaCustomerpo_to_project();
        recCustPOtoProj.customerpoid = custPOResults[0].id;
        recCustPOtoProj.customerid = updPrj.customerid;
        recCustPOtoProj.projectid = updPrj.id;
        recCustPOtoProj.active = '1';

        // disable the current user's filter set while script runs
        NSOA.wsapi.disableFilterSet(true);

        // add the new customer po to the project
        var custPOtoProjResults = NSOA.wsapi.add(
            [recCustPOtoProj]
        );

        if (!custPOtoProjResults || !custPOtoProjResults[0]) {
            NSOA.meta.log('error',
                'Unexpected error! Customer PO was not linked to the project.');
        } else if (custPOtoProjResults[0].errors) {
            custPOtoProjResults[0].errors.forEach(function(err) {
                NSOA.meta.log('error', 'Error: ' + err.code + ' - ' + err.comment);
            });
        }

    }

    // create project object to hold update information and clear quick po
    var recProj = new NSOA.record.oaProject(updPrj.id);
    recProj[FLD_CUSTPO_NUM] = '';
    recProj[FLD_CREATE_PO] = '';
    recProj[FLD_CUSTPO_AMT] = '';
    recProj[FLD_CUSTPO_DATE] = '';

    // disable the current user's filter set while script runs
    NSOA.wsapi.disableFilterSet(true);

    // enable custom field editing
    var update_custom = {
        name: 'update_custom',
        value: '1'
    };

    // update the project to clear quick customer po create information
    var projResults = NSOA.wsapi.modify(
        [update_custom], [recProj]
    );

    if (!projResults || !projResults[0]) {
        NSOA.meta.log('error', 'Unexpected error! Project was not updated.');
    } else if (projResults[0].errors) {
        projResults[0].errors.forEach(function(err) {
```

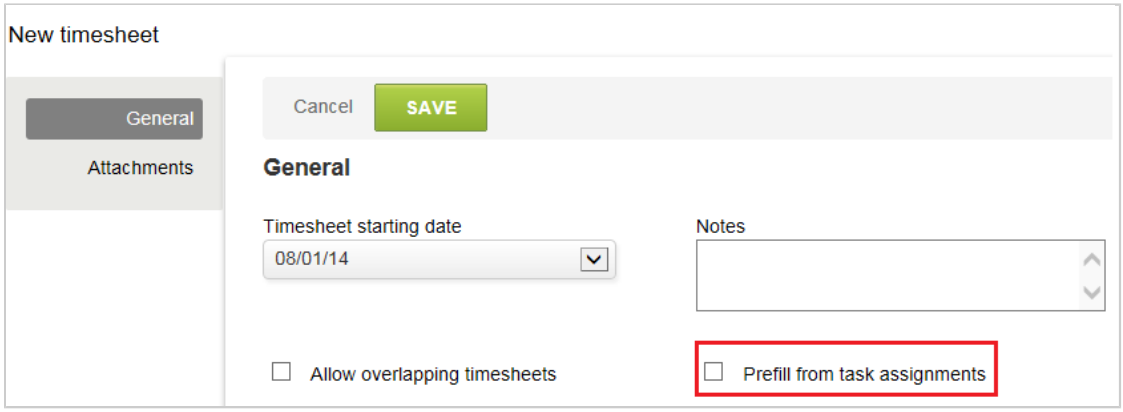
```
                    NSOA.meta.log('error', 'Error: ' + err.code + ' - ' + err.comment);
                });
            }

        } else {
            NSOA.meta.log('debug', 'Customer po creation was skipped.');
        }

    } catch (e) {
        NSOA.meta.log('error', 'Uncaught error: ' + e);
    }
}
```

Create time entries from task assignments when the user creates a new timesheet

This script creates time entries from task assignments when the user creates a new timesheet.

When the user creates a new timesheet, toggle checkbox to have it prefilled with data fetched from the current task assignments.

- Automate timesheet creation with relevant tasks an employee is working on.
- Saves a lot of time digging through pick lists, finding correct tasks.
- FUTURE: Could deploy project task afterSave script to auto-update existing open timesheets as assignments change (practical for monthly timesheets).

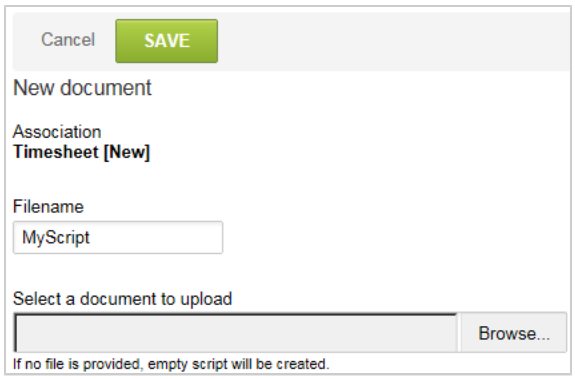


Setup

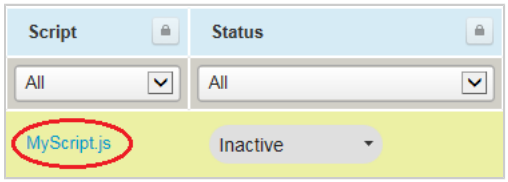
Follow the steps below or download the solutions file, see Creating Solutions for details.

1. Create a new **Timesheet [New] form script deployment**.

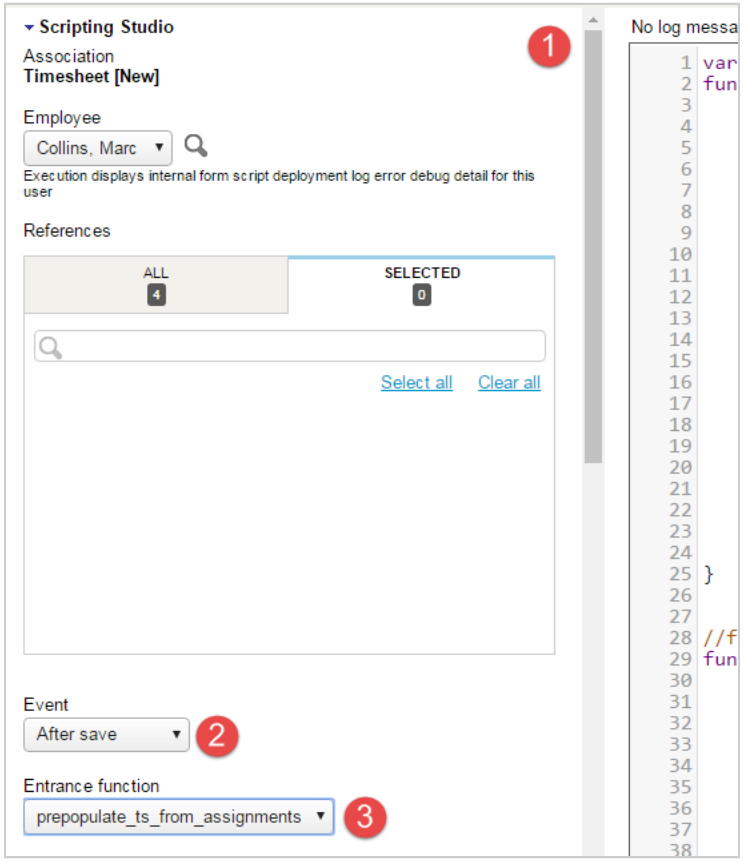
2. Enter a **Filename** and click **SAVE**. The extension '.js' is automatically appended if not supplied.



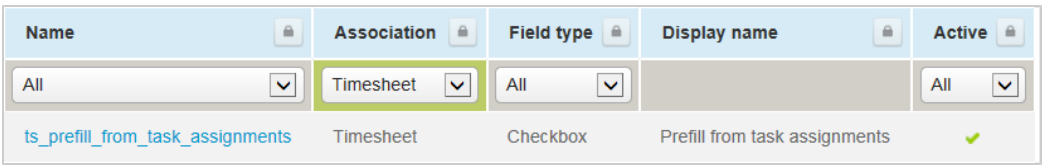
3. Click on the script link to launch the **Scripting Studio**.



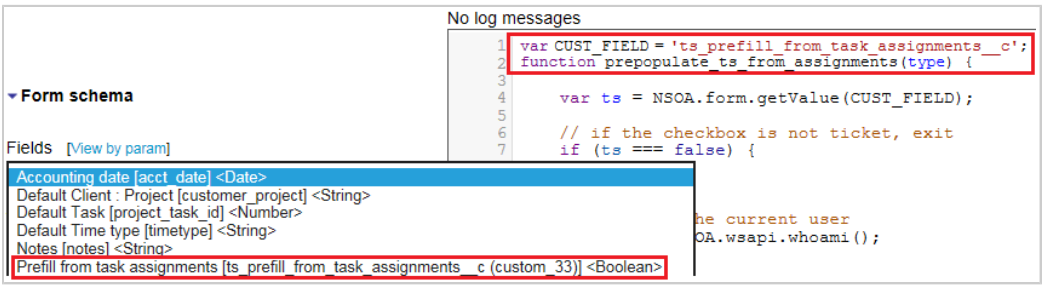
4. (1) Copy the **Program Listing** below into the editor, (2) set the **After save** event, and set **prepopulate_ts_from_assignments** as the Entrance Function.



5. Set up a **Checkbox** custom field for **Timesheet**. The first in each pair is a **Pick List** with a **List source** of User. The second in each pair is a **Ratio**. You can set the **Divider text** for the very first custom field to **Commission** to place the custom fields in their own section.



6. Use the **Form schema** to identify the correct param names for the custom fields and change the array at the top of the script accordingly.



Tip: If the new custom field is not listed in the **Form schema**, navigate to Timesheets, create a new Timesheet form without saving (this with will refresh the custom field list), and then open the Scripting Studio again.

Program Listing

```
var CUST_FIELD = 'ts_prefill_from_task_assignments__c'; // Use Form schema to find param name
function prepopulate_ts_from_assignments(type) {

    var ts = NSOA.form.getValue(CUST_FIELD);

    // if the checkbox is not ticked, exit
    if (ts === false) {
        return;
    }

    // retrieve the current user
    var user = NSOA.wsapi.whoami();

    //look for current assignments for this user
    var defaultPerRow = find_assignments(user.id);

    // retrieve the new object
    var newr = NSOA.form.getNewRecord();
    var timesheet = new NSOA.record.oaTimesheet();

    timesheet.id = newr.id;
    timesheet.default_per_row = defaultPerRow;

    var result = NSOA.wsapi.modify([], [timesheet]);
}


//find the assignments for this user and return a string for timesheet.default_per_row
```

```
function find_assignments(userid) {

    // fetch a list of task assignments for the current user
    var taskAssign = new NSOA.record.oaProjecttaskassign();
    taskAssign.userid = userid;

    var readTasksAssign = {
        type: "Projecttaskassign",
        method: "equal to",
        fields: "projecttaskid",
        attributes: [{
            name: "limit",
            value: "0,1000"
        }],
        objects: [taskAssign]
    };

    var CSV = {
        pt_id: [],
        cp_id: []
    };
    var resultTaskAssign = NSOA.wsapi.read(readTasksAssign);

    // iterate through all the task assignments and filter only current ones
    // retrieve all tasks that belong to user, started in the past and percent_complete < 100
    if (resultTaskAssign[0].errors === null && resultTaskAssign[0].objects) {

        for (var i = 0; i < resultTaskAssign[0].objects.length; i++) {

            var projectTask = new NSOA.record.oaProjecttask();
            projectTask.id = resultTaskAssign[0].objects[i].projecttaskid;

            var readTask = {
                type: "Projecttask",
                method: "equal to",
                fields: "calculated_starts,starts,percent_complete,customerid,id,projectid",
                attributes: [{
                    name: "limit",
                    value: "0,1000"
                }],
                objects: [projectTask]
            };

            var resultTask = NSOA.wsapi.read(readTask);

            // do we have results?
            if (resultTask[0].errors === null && resultTask[0].objects) {
                for (var k = 0; k < resultTask[0].objects.length; k++) {

                    var ptStartDate = resultTask[0].objects[k].starts.substring(0, 10);

                    // optimization: skip blank dates
                    if (ptStartDate === '0000-00-00') {
                        ptStartDate = resultTask[0].objects[k].calculated_starts.substring(0, 1
0);
```

```
                        if (ptStartDate === '0000-00-00') {
                            continue;
                        }
                    }

                    var currentDate = new Date();
                    var starts = new Date(ptStartDate);

                    // do we have a date?
                    // NSOA.meta.alert('currentdate=' + currentDate + ' starts='+starts);

                    if (starts <= currentDate &&
                        parseInt(resultTask[0].objects[k].percent_complete, 10) < 100) {
                        CSV.pt_id.push(resultTask[0].objects[k].id);
                        CSV.cp_id.push(resultTask[0].objects[k].customerid + ":" +
                            resultTask[0].objects[k].projectid);
                    }
                }
            }
        }
    }

    var retval = "pt_id," + CSV.pt_id.join(",") + "\n";
    retval = retval + "cp_id," + CSV.cp_id.join(",");
    return retval;

}
```

Control budgeted hours for a project using the project budget feature and a custom hours field

This script controls the budgeted hours for a project using the project budget feature and a custom hours field.

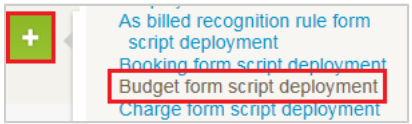
Note: Requires "Project budgets" feature enabled.

- Only requires you to manage your budgeted hours in one place
- Allows budgeted hours to be date stamped for better change order management
- Form permissions can be used to make the project budget hours field read only, which matches the budget money field behavior

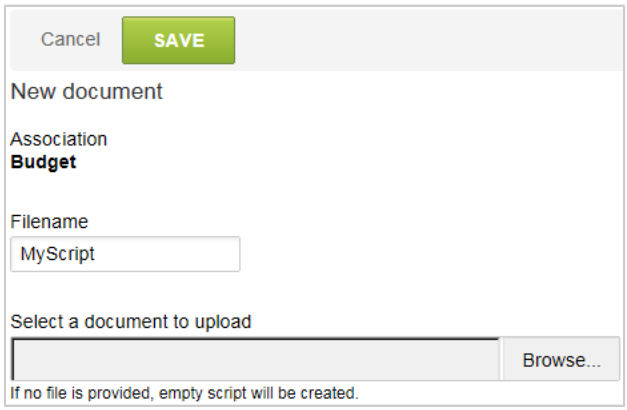
Setup

Follow the steps below or download the solutions file, see Creating Solutions for details.

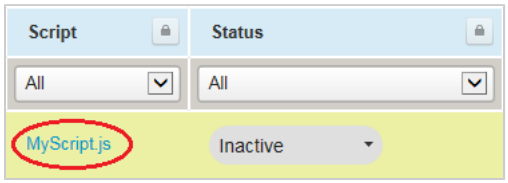
1. Create a new **Budget form script deployment**.



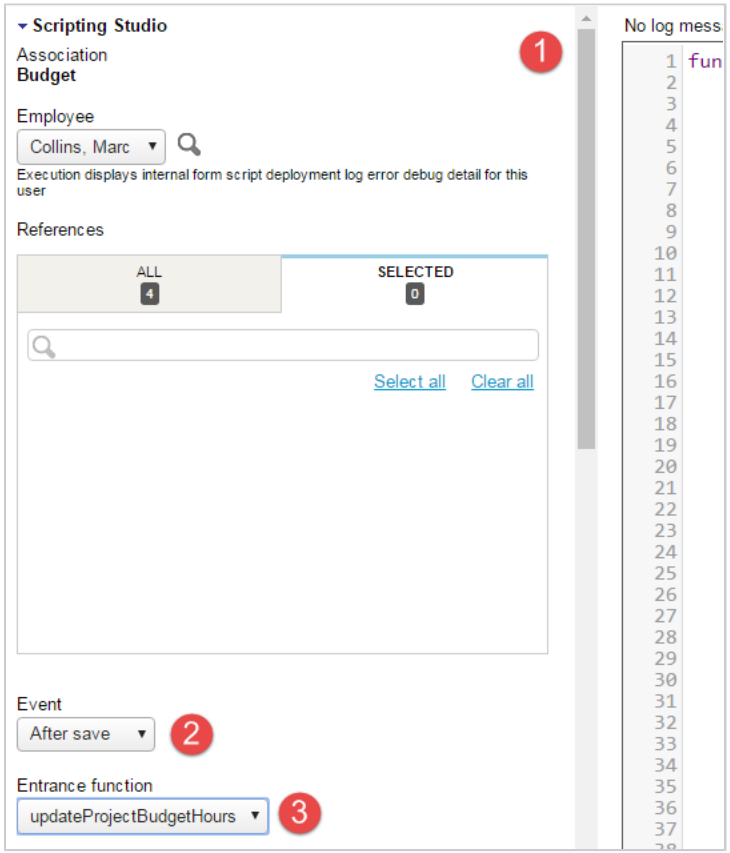
2. Enter a **Filename** and click **SAVE**. The extension '.js' is automatically appended if not supplied.



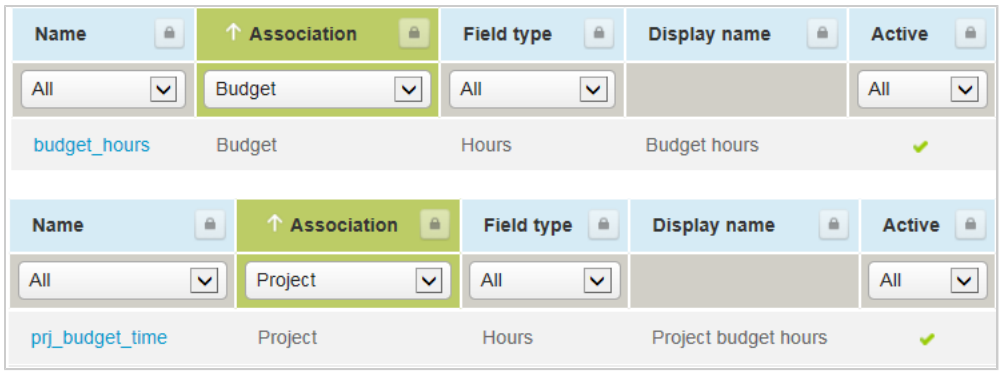
3. Click on the script link to launch the **Scripting Studio**.



4. (1) Copy the **Program Listing** below into the editor, (2) set the **After save** event, and set **updateProjectBudgetHours** as the Entrance Function.



5. Set up an **Hours** custom field for **Project** and an **Hours** custom field for **Budget**.



Name	Association	Field type	Display name	Active
All	Budget	All		All
budget_hours	Budget	Hours	Budget hours	✔

Name	Association	Field type	Display name	Active
All	Project	All		All
prj_budget_time	Project	Hours	Project budget hours	✔

Program Listing

```
function updateProjectBudgetHours(type) {
    try {
        // DEBUG: Uncomment next line to enable XML logging
        // var wsLog = NSOA.wsapi.enableLog(true);
        // list all fields used in script
        var FLD_PRJ_ID = 'id',
            FLD_PRJ_BUD_HRS = 'prj_budget_time__c',
            FLD_BUD_PID = 'projectid';

        // store newly saved budget record
        var updBudget = NSOA.form.getNewRecord();

        // create new budget object to store information
        var budRec = new NSOA.record.oaBudget();
        budRec.projectid = updBudget.projectid;
        var budRequest = {
            type: "Budget",
            method: "equal to",
            fields: "id,budget_hours__c", // budget_hours__c is a custom hours field
            attributes: [{
                name: "limit",
                value: "100"
            }],
            objects: [budRec]
        };

        // disable the current user's filter set while script runs
        NSOA.wsapi.disableFilterSet(true);

        // search for all budget transactions with current projectid
        var budResults = NSOA.wsapi.read(budRequest);
        if (!budResults || !budResults[0]) {
            NSOA.meta.log('error', 'Unexpected error! Could not return project budgets.');
            return;
        } else if (budResults[0].errors !== null && budResults[0].errors.length > 0) {
            prjResults[0].errors.forEach(function(err) {
                var fullError = err.code + ' - ' + err.comment + ' ' + err.text;
                NSOA.meta.log('error', 'Error: ' + fullError);
            });
```

```
            return;
        }
        var b,
            totalBudHrs = 0,
            budObj = budResults[0].objects,
            budObjLen = budObj.length;
        for (b = 0; b < budObjLen; b++) {
            var budHrs = parseInt(budObj[b].budget_hours__c, 10);
            totalBudHrs += budHrs; // add all hours together
        }
        // create new project object to store information
        var prjRec = new NSOA.record.oaProject();
        prjRec[FLD_PRJ_ID] = updBudget[FLD_BUD_PID];
        prjRec[FLD_PRJ_BUD_HRS] = totalBudHrs;

        // disable the current user's filter set while script runs
        NSOA.wsapi.disableFilterSet(true);

        // update the project budget
        var prjResults = NSOA.wsapi.modify(
            [{
                name: "update_custom",
                value: "1"
            }], [prjRec]
        );
        if (!prjResults || !prjResults[0]) {
            NSOA.meta.log('error', 'Unexpected error! Project was not updated.');
        } else if (prjResults[0].errors !== null && prjResults[0].errors.length > 0) {
            prjResults[0].errors.forEach(function(err) {
                var fullError = err.code + ' - ' + err.comment + ' ' + err.text;
                NSOA.meta.log('error', 'Error: ' + fullError);
            });
            return;
        }

    } catch (e) {
        NSOA.meta.log('error', 'Try/catch error: ' + e);
    }
}
```

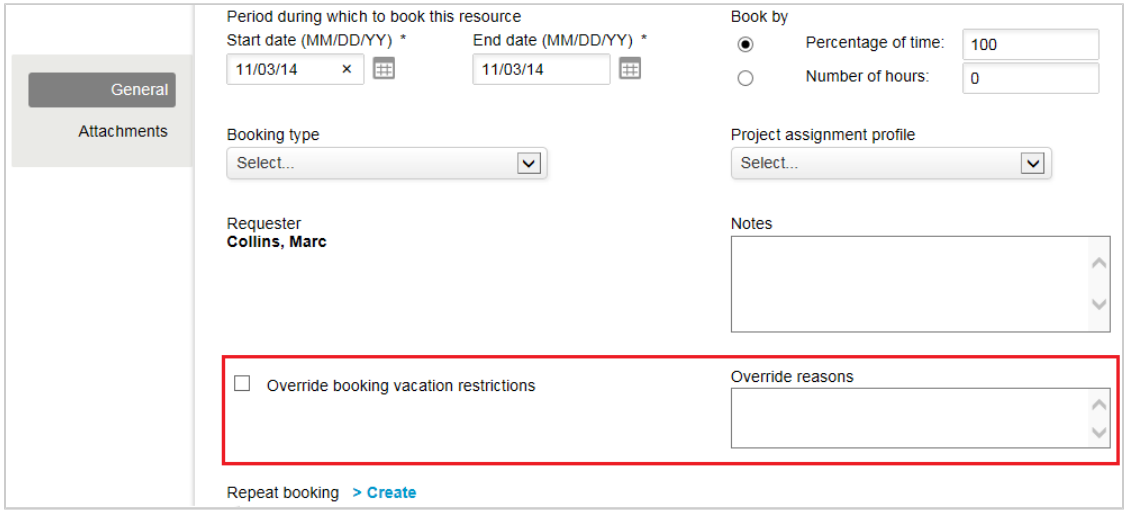
Workflow

Prevent a booking from being created if the selected resource has approved time off during the booking period

This script prevents a booking from being created if the selected resource has approved time off during the booking period.

- Improves accuracy of bookings

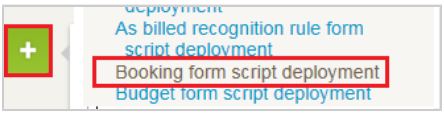
- Supports override flag to force booking



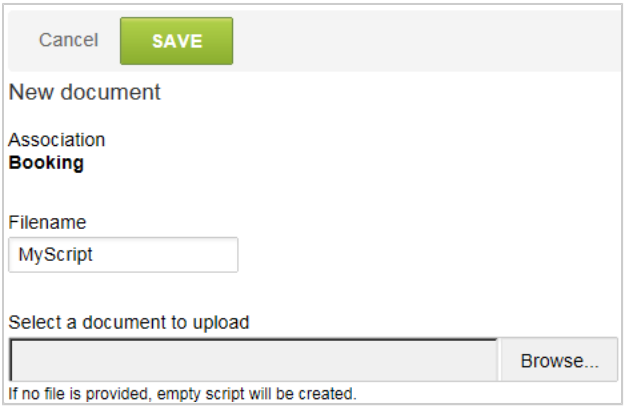
Setup

Follow the steps below or download the solutions file, see Creating Solutions for details.

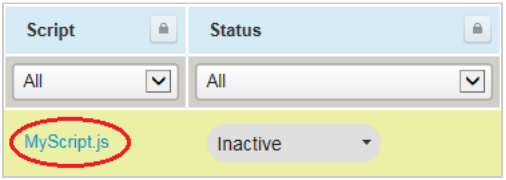
1. Create a new **Booking form script deployment**.



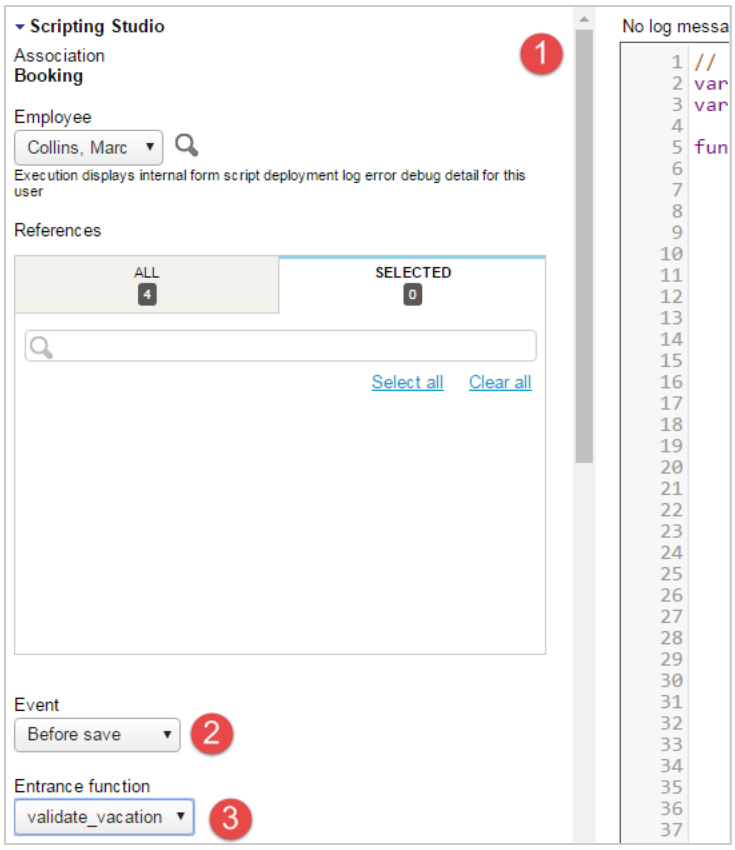
2. Enter a **Filename** and click **SAVE**. The extension '.js' is automatically appended if not supplied.



3. Click on the script link to launch the **Scripting Studio**.



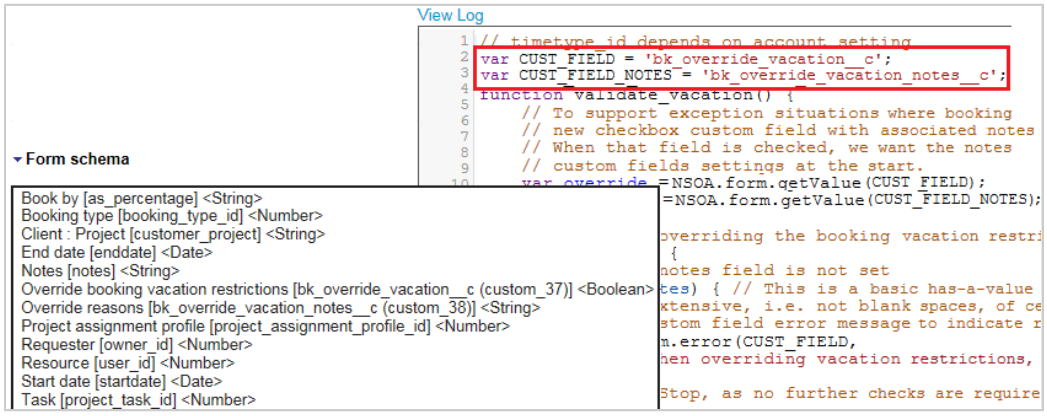
4. (1) Copy the **Program Listing** below into the editor, (2) set the **Before save** event, and set **validate_vacation** as the Entrance Function.



5. Set up a **Checkbox** and a **Text Area** custom field for **Booking**.

Name	Association	Field type	Display name	Active
All	Booking	All		All
bk_override_vacation	Booking	Checkbox	Override booking vacation restrictions	✔
bk_override_vacation_notes	Booking	Text Area	Override reasons	✔

6. Use the **Form schema** to identify the correct param names for the custom fields and change the array at the top of the script accordingly.



Tip: If the new custom fields are not listed in the **Form schema**, navigate to Resources, open a booking form (this with will refresh the custom field list), and then open the Scripting Studio again.

Program Listing

```
// timetype_id depends on account settings
var CUST_FIELD = 'bk_override_vacation__c';
var CUST_FIELD_NOTES = 'bk_override_vacation_notes__c';

function validate_vacation() {
    // To support exception situations where booking should be allowed over scheduled timeoff,
    // new checkbox custom field with associated notes field has been added to Booking form.
    // When that field is checked, we want the notes field to be required, so we validate the
    // custom fields settings at the start.
    var override = NSOA.form.getValue(CUST_FIELD);
    var req_notes = NSOA.form.getValue(CUST_FIELD_NOTES);

    // If we are overriding the booking vacation restrictions
    if (override) {
        // And the notes field is not set
        if (!req_notes) { // This is a basic has-a-value check, typically check should be
            // more extensive, i.e. not blank spaces, of certain length, etc.
            // Set custom field error message to indicate required, and prevent form saving
            NSOA.form.error(CUST_FIELD,
                'When overriding vacation restrictions, notes are required');
        }
        return; // Stop, as no further checks are required
    }

    // getValue returns JS Date objects for Date type fields
    var start = NSOA.form.getValue('startdate'); // While adding/changing a script,
    var end = NSOA.form.getValue('enddate'); // the Form Schema section provides a list
    // of available form fields and the expected
    // return values of those fields

    // Create the oaSchedulerequest object for the WSAPI read search
    // Information on available records can be found in the user scripting guide
    // Note the form field is user_id but the SOAP API field is userid
    var approvedSchedReq = new NSOA.record.oaSchedulerequest();
    approvedSchedReq.userid = NSOA.form.getValue('user_id');
    approvedSchedReq.approval_status = 'A'; // (A)pproved, (O)pen, (S)ubmitted, (R)ejected
    approvedSchedReq.timetypeid = '5'; // Personal time is timetypeid 5

    // Pull the start and end dates for Schedulerequests that match our criteria
    var aPTO = NSOA.wsapi.read({
        type: 'Schedulerequest', // The SOAP API complex type
        method: 'equal to',
        fields: 'startdate,enddate', // start & end fields for Schedulerequest complex type
        attributes: [{
            name: 'limit', // ReadRequest objects must have a limit specified
            value: '100' // '100' returns up to 100, '50,100' returns 50 - 100
        }],
```

```
        objects: [approvedSchedReq] // The previously created search object
    });

    // NSOA.wsapi.read() returns an array of objects with error and objects properties
    for (x = 0; x < aPTO.length; x++) {
        // If there were errors, notify the user and stop
        if (aPTO[x].errors) {
            var errorMsg = '';
            for (i = 0; i < aPTO[x].errors.length; i++) {
                errorMsg += 'SOAP error [' + aPTO[x].errors[i].code + ']:';
                errorMsg += aPTO[x].errors[i].text + ' - ';
                errorMsg += aPTO[x].errors[i].comment + "\n";
            }
            NSOA.form.error('', errorMsg); // Set the main form error message with the details
            return;
        }

        // If there were approved personal Schedulerequest objects found
        if (aPTO[x].objects) {
            NSOA.meta.alert(aPTO[x].objects.length);
            // Loop through and validate the time off doesn't overlap booking request period
            for (i = 0; i < aPTO[x].objects.length; i++) {
                var thisStart = convertToDate(aPTO[x].objects[i].startdate);
                var thisEnd = convertToDate(aPTO[x].objects[i].enddate);

                // If the PTO overlaps the start of the period
                if ((thisStart <= start && thisEnd <= end && thisEnd >= start) ||
                    (thisStart <= start && thisEnd >= end) || // Or overlaps whole period
                    (thisStart >= start && thisEnd <= end) || // Or is wrapped by the period
                    (thisStart >= start && thisStart <= end && thisEnd >= end)) { // Or end
                    var malDate;
                    if (thisStart.getTime() == thisEnd.getTime()) { // If the is a single day
                        malDate = thisStart.toDateString(); // Only display one date
                    } else { // Else start/end range
                        malDate = thisStart.toDateString() + ' - ' + thisEnd.toDateString();
                    }

                    // Set the form startdate error message accordingly, then stop.
                    NSOA.form.error('startdate', 'The requested resource has approved personal
time off' + ' during the selected booking period: ' + malDate);
                    return;
                }
            }
        }
    }
}

// Helper function for converting date strings to JS Date objects
function convertToDate(vDate) {
    // Expected date format is a string: YYYY-MM-DD 0:0:0
    var aYMD = vDate.split(' ');
    aYMD = aYMD[0].split('-');
    return new Date(aYMD[0], aYMD[1] - 1, aYMD[2]);
}
```

Prevent closing a project that has open issues

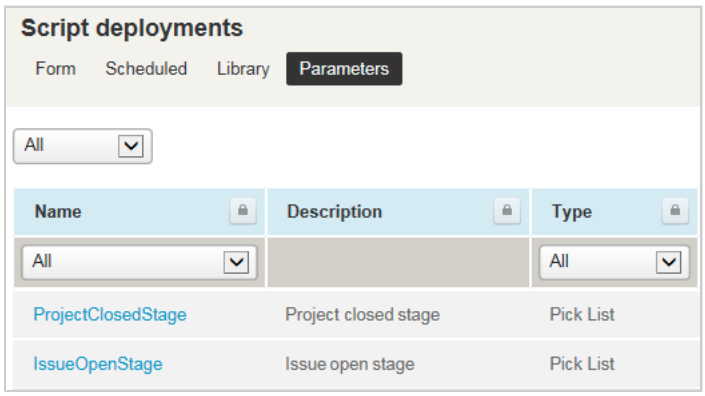
This script prevents the closing of a project that has open issues.

- Enforces workflow requirement that all open issues be addressed before a project can be closed

Setup

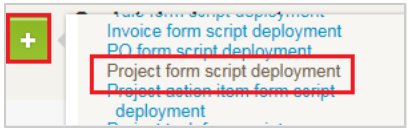
Follow the steps below or download the solutions file, see Creating Solutions for details.

1. Create two parameters, see Creating Parameters.

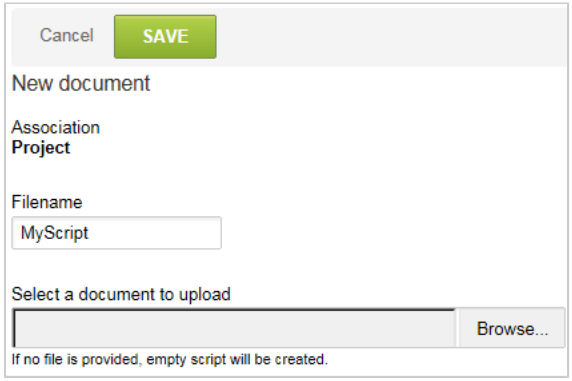


The **List source** for the ProjectClosedStage Pick List parameter is “Project stage”.
The **List source** for the IssueOpenStage Pick List parameter is “Stage”.

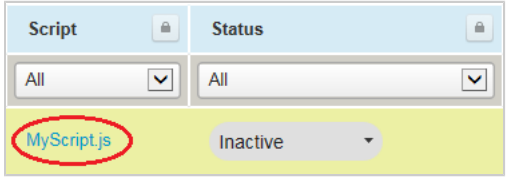
2. Create a new **Project form script deployment**.



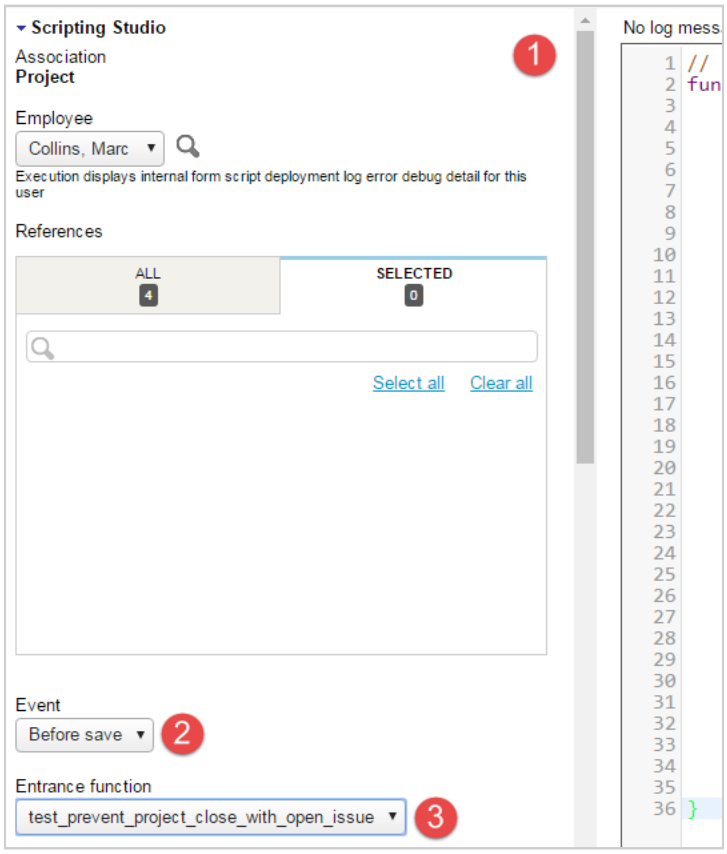
3. Enter a **Filename** and click **SAVE**. The extension ‘.js’ is automatically appended if not supplied.



4. Click on the script link to launch the **Scripting Studio**.



5. (1) Copy the **Program Listing** below into the editor, (2) set the **Before save** event, and set **test_prevent_project_close_with_open_issue** as the Entrance Function.



Program Listing

```
// project_stage_id and issue_stage_id depend on account settings
function test_prevent_project_close_with_open_issue() {

    // return if new stage is not closed
    if (NSOA.form.getValue('project_stage_id') !=
        NSOA.context.getParameter('ProjectClosedStage'))
        return;

    // Load issue data
    var issue = new NSOA.record.oaIssue();
    issue.project_id = NSOA.form.getValue('id');
    issue.issue_stage_id = NSOA.context.getParameter('IssueOpenStage');

    var readRequest = {
        type: "Issue",
        fields: "id, date",
        method: "equal to",
        objects: [issue],
        attributes: [{
            name: "limit",
            value: "1"
```

```
        }]
    };

    var arrayOfreadResult = NSOA.wsapi.read(readRequest);

    if (!arrayOfreadResult || !arrayOfreadResult[0])
        NSOA.form.error('', "Internal error analyzing project issues.");

    else if (arrayOfreadResult[0].errors === null && arrayOfreadResult[0].objects)
        arrayOfreadResult[0].objects.forEach(
            function(o) {
                NSOA.form.error('', "Can't close project with open issues.");
            }
        );
}
```

Automatically create a new issue when project stage is "at risk" and prevent project stage from changing until this issue is resolved

This script automatically create a new issue when the project stage is saved as "at risk" and prevents the project stage from changing until the issue is resolved.

- Enforces documentation trail for critical project concerns
- More complex variation of simple "project stage" validation example

Follow the steps below or download the solutions file, see Creating Solutions for details.

Note: You will still need to create the custom fields described in **Setup 1 — Custom Field**

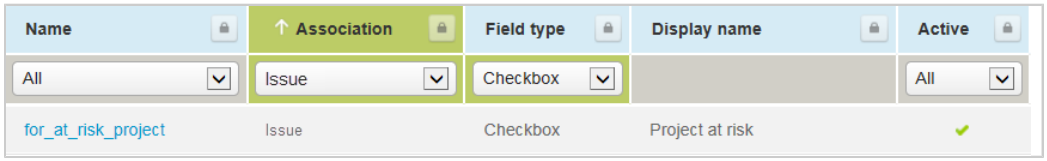
This example consists of a custom field and two scripts:

- Setup 1 — Custom Field is used by both the scripts.
- Setup 2 — Project After Save creates an issue with a custom field enabled.
- Setup 3 — Project Before Save prevents the project stage from changing until the issue is resolved.

Important: This example requires you to create a Project Stage. See the OpenAir Admin Guide for more details on Project Stages.

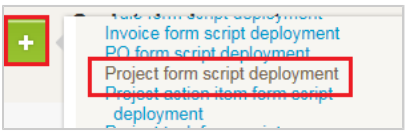
Setup 1 — Custom Field

1. Set up a **Checkbox** and a **Text Area** custom field for **Issue**.

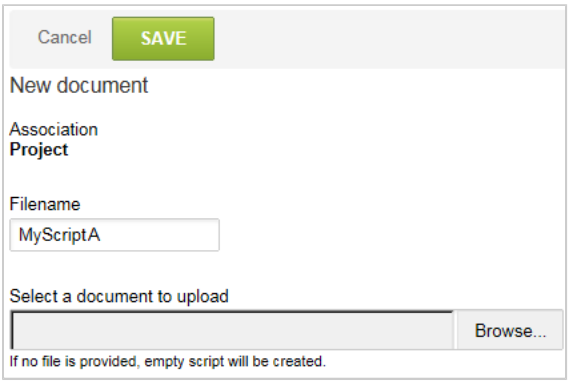


Setup 2 — Project After Save

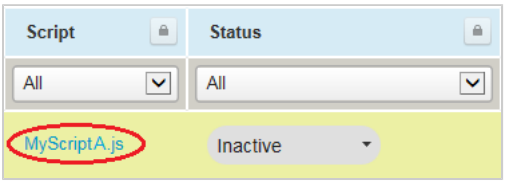
1. Create a new **Project form script deployment**.



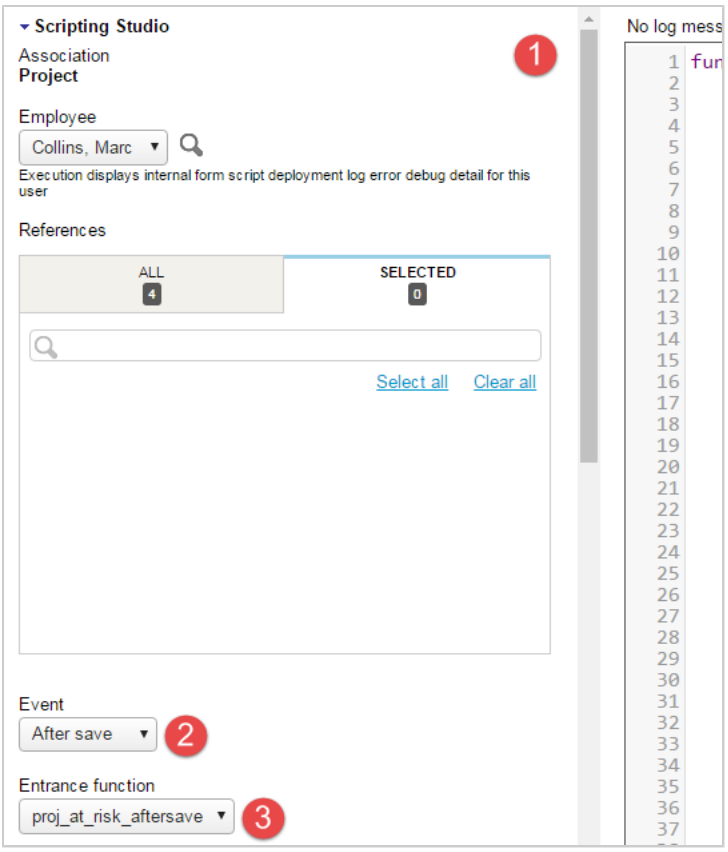
2. Enter a **Filename** and click **SAVE**. The extension '.js' is automatically appended if not supplied.



3. Click on the script link to launch the **Scripting Studio**.



4. (1) Copy the **Program Listing** below into the editor, (2) set the **After save** event, and set **proj_at_risk_aftersave** as the Entrance Function.



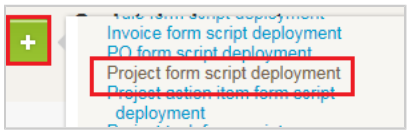
Program Listing for Setup 2

```
function proj_at_risk_aftersave() {
   var PROJECT_STAGE_AT_RISK = NSOA.context.getParameter('ProjectAtRiskStage');
   var ISSUE_STAGE_OPEN = NSOA.context.getParameter('IssueOpenStage');

    // return if new stage is changed and "at risk"
    var proj = NSOA.form.getNewRecord();
    var old_stage = NSOA.form.getOldRecord().project_stageid;
    var current_stage = proj.project_stageid;
    NSOA.meta.log("debug", "old=" + old_stage + ", new=" + current_stage);
    if (old_stage == current_stage || current_stage != PROJECT_STAGE_AT_RISK)
        return;

    // Check for an existing at-risk event
    var issue = new NSOA.record.oaIssue();
    issue.project_id = proj.id;
    issue.for_at_risk_project__c = '1';

    var readRequest = {
        type: "Issue",
        fields: "id, name, date",
        method: "equal to",
        objects: [issue],
        attributes: [{
```

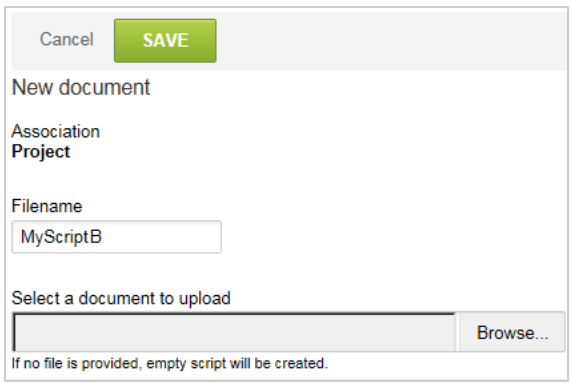
```
            name: "limit",
            value: "1"
        }]
    };

    var arrayOfreadResult = NSOA.wsapi.read(readRequest);
    if (!arrayOfreadResult || !arrayOfreadResult[0])
        NSOA.form.error('', "Internal error analyzing project issues.");

    else if (arrayOfreadResult[0].errors === null &&
        (!arrayOfreadResult[0].objects || arrayOfreadResult[0].objects.length === 0)) {
        issue.owner_id = NSOA.wsapi.whoami().id;
        issue.description = "Projected reported at risk";
        issue.issue_status_id = 1; // Unassigned
        issue.issue_stage_id = ISSUE_STAGE_OPEN;
        issue.date = (new Date()).toISOString().slice(0, 10);
        NSOA.meta.log('debug', JSON.stringify(issue));
        NSOA.wsapi.add(issue);
    }
}
```

Setup 3 — Project Before Save

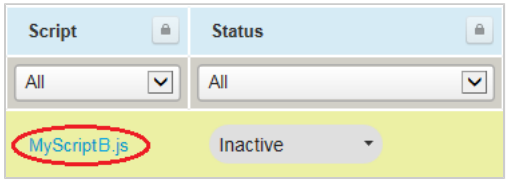
1. Create a new **Project form script deployment**.



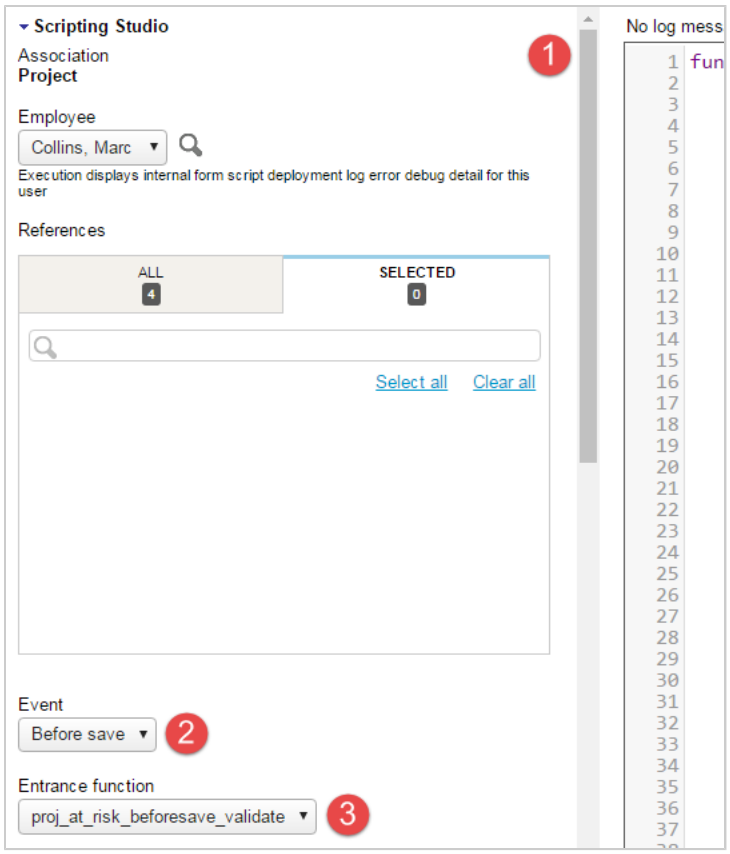
2. Enter a **Filename** and click **SAVE**. The extension '.js' is automatically appended if not supplied.



3. Click on the script link to launch the **Scripting Studio**.



4. (1) Copy the **Program Listing** below into the editor, (2) set the **Before save** event, and set **proj_at_risk_beforesave_validate** as the Entrance Function.



Program Listing for Setup 3

```
function proj_at_risk_beforesave_validate() {
   var PROJECT_STAGE_AT_RISK = NSOA.context.getParameter('ProjectAtRiskStage');
   var ISSUE_STAGE_OPEN = NSOA.context.getParameter('IssueOpenStage');

    // return if new stage is not changing from "at risk"
    var current_stage = NSOA.form.getOldRecord().project_stageid;
    var new_stage = NSOA.form.getValue('project_stage_id');
    if (!(current_stage == PROJECT_STAGE_AT_RISK && new_stage != PROJECT_STAGE_AT_RISK))
        return;

    // Load issue data
    var issue = new NSOA.record.oaIssue();
    issue.project_id = NSOA.form.getValue('id');
    issue.issue_stage_id = ISSUE_STAGE_OPEN;
    issue.for_at_risk_project__c = '1';

    var readRequest = {
        type: "Issue",
        fields: "id, name, date",
        method: "equal to",
        objects: [issue],
        attributes: [{
            name: "limit",
```

```
            value: "1"
        }]
    };

    var arrayOfreadResult = NSOA.wsapi.read(readRequest);

    if (!arrayOfreadResult || !arrayOfreadResult[0])
        NSOA.form.error('', "Internal error analyzing project issues.");

    else if (arrayOfreadResult[0].errors === null && arrayOfreadResult[0].objects)
        arrayOfreadResult[0].objects.forEach(
            function(o) {
                NSOA.form.error('', "Can't change project stage until " +
                    "the following issue is resolved: " + o.name);
            }
        );
}
```

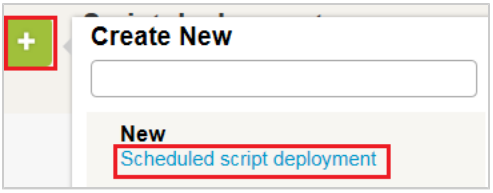
Send an alert email when a scheduled script completes

This script informs a user when a scheduled script completes successfully.

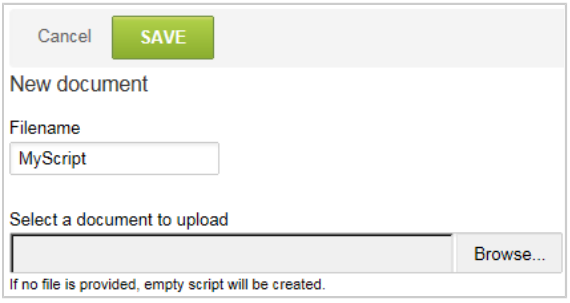
Setup

Follow the steps below or download the solutions file, see Creating Solutions for details.

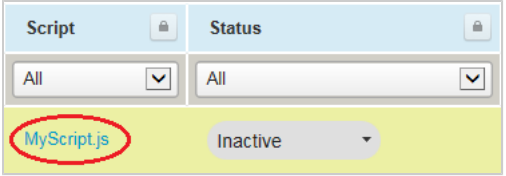
1. Create a new **Scheduled script deployment**.



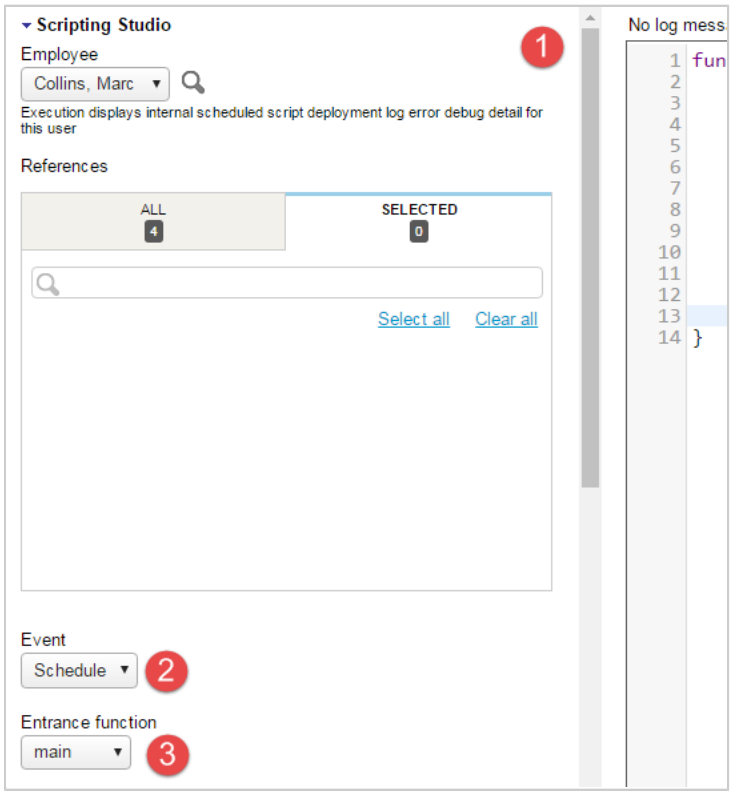
2. Enter a **Filename** and click **SAVE**. The extension '.js' is automatically appended if not supplied.



3. Click on the script link to launch the **Scripting Studio**.



4. (1) Copy the **Program Listing** below into the editor, (2) set the **Schedule** event, and set **main** as the Entrance Function.



Program Listing

```
function main() {
    // TODO Add Your Code Here

    // TODO Handle Errors

    // Notify The Owner
    var me = NSOA.wsapi.whoami();
    var msg = {
        to: [me.id],
        subject: "Script completed",
        format: "HTML",
        body: "<b>Your script completed</b><br/>" +
              "<hr/><i>Automatically sent by the system</i>"
    };

    NSOA.meta.sendMail(msg);
}
```

OpenAir User Scripting Release History

Here you can find all changes made to the OpenAir User Scripting guide and user scripting by release.

October 14, 2017

- Added **author** parameter for NSOA.meta.sendMail(message).
- In addition to creating scripts, solutions can now create custom fields, script libraries, and script parameters. Selection lists for these options have been added to the Solution form. See Creating Solutions.
- Scripts can now be deployed against Issue forms. The Project Issue Form and Issue Form are distinct script deployments. See Creating Form Scripts.