



OpenAir

User Scripting

Copyright © 2013, 2024, Oracle and/or its affiliates.

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

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

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

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

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

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

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

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

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

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

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

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

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

Documentation Accessibility

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

Access to Oracle Support

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

Sample Code

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

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

No Excessive Use of the Service

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

Table of Contents

Introduction	1
User Scripting Overview	1
Getting Started	4
Logs	6
Reporting	11
Platform Role Permissions	12
Scripting and OpenAir Mobile	14
Scripting and OpenAir NetSuite Connector	15
User Scripting	16
Scripting Center	16
Scripting Workflow	20
Creating Form Scripts	21
Testing Form Scripts	23
Deploying Form Scripts	24
Creating Scheduled Scripts	25
Testing Scheduled Scripts	26
Deploying Scheduled Scripts	28
Scheduled Scripts and Scheduled Queue Status	29
Creating Library Scripts	29
Creating Parameters	31
Creating Solutions	33
Accessing Terminology	36
Scripting Studio	37
Scripting Studio Tools and Settings	38
SOAP Explorer	39
Functions Explorer	40
OData Explorer	40
Script Parameters	41
Terminology	42
Form Schema	42
Testing and Debugging	45
Editor	46
Scripting Studio Options	48
Entrance Function	49
Events	50
Scripting Governance	52
SOAP API	54
Making SOAP Calls	55
Using SOAP Results	59
Handling SOAP Errors	61
Outbound Calling	62
Request	63
Response	63
Limits	63
Password Script Parameters	64
Scripting Approvals	64
Working with the Approvals System	65
Using Approval Results	67
Handling Approval Errors	67
Custom Fields	68
Creating Custom Fields	68
Reading Custom Fields	70
Updating Custom Fields	71

NSOA Functions	72
NSOA.context.getAllParameters()	74
NSOA.context.getAllTerms()	75
NSOA.context.getLanguage()	75
NSOA.context.getParameter(name)	76
NSOA.context.getTerm(termid)	77
NSOA.context.isTestMode()	78
NSOA.context.parseTerminology(message)	78
NSOA.context.remainingTime()	79
NSOA.context.remainingUnits()	80
NSOA.form.confirmation(message)	81
NSOA.form.error(field, message)	82
NSOA.form.getAllValues()	83
NSOA.form.getLabel(field)	84
NSOA.form.getName(field)	85
NSOA.form.getNewRecord()	86
NSOA.form.getOldRecord()	87
NSOA.form.getValue(field)	88
NSOA.form.get_value(field)	89
NSOA.form.setValue(field, value)	90
NSOA.form.warning(message)	94
NSOA.https.delete(request)	95
NSOA.https.get(request)	96
NSOA.https.patch(request)	97
NSOA.https.post(request)	99
NSOA.https.put(request)	100
NSOA.listview.data(listviewId)	101
NSOA.listview.list()	103
NSOA.meta.alert(message)	104
NSOA.meta.log(severity, message)	105
NSOA.meta.sendMail(message)	106
NSOA.NSConnector.integrateAllNow()	107
NSOA.NSConnector.integrateRecord()	108
NSOA.NSConnector.integrateWorkflowGroup(name)	109
NSOA.record.<complex type>([id])	110
NSOA.report.data(reportId,optionalParameters)	112
NSOA.report.list()	114
NSOA.wsapi.add(objects)	115
NSOA.wsapi.approve(approveRequest)	116
NSOA.wsapi.delete(objects)	117
NSOA.wsapi.disableFilterSet([flag])	118
NSOA.wsapi.enableLog([flag])	119
NSOA.wsapi.modify(attributes, objects)	120
NSOA.wsapi.read(readRequest)	121
NSOA.wsapi.reject(rejectRequest)	122
NSOA.wsapi.remainingTime()	123
NSOA.wsapi.submit(submitRequest)	124
NSOA.wsapi.unapprove(unapproveRequest)	125
NSOA.wsapi.upsert(attributes,objects)	126
NSOA.wsapi.whoami()	127
Code Samples	128
Comparing Date Fields	128
Validating Numeric Fields	128
Requiring Minimum Values	129
Creating Error Log Entries	129

Sending email	129
SOAP API — Prevent closing a project with an open issue	130
SOAP API — Append notes to a project	130
SOAP API — Require task assignment	131
Submitting a Timesheet for Approval	132
Outbound Calling — SOAP Call Using HTTPS POST	132
Outbound Calling — Post a Slack Message	133
Outbound Calling — HTTPS GET with Authorization	134
JavaScript	135
JavaScript Overview	135
Variables	135
Variable Scope	136
Dynamic Data Types	137
Arrays	138
Associative Array	139
Objects	140
Functions	141
Loops	142
for	143
for in	143
forEach	143
do while	144
while	144
Conditional Statements	144
if ... else	145
switch	146
Error Handling	146
References	147
JavaScript Objects	147
JavaScript Operators	154
Reserved Words	156
Escape Sequences	157
Scripting Best Practices	158
Real World Use Cases	161
Validation	163
Ensure value of multiple commissions fields equals 100%	163
Require notes field to be populated on time entries when more than 8 hours in a day	165
When submitting an expense report, validate each ticket has an attachment (e.g. scanned receipt)	167
Ensure resource time entry matches booking planning and project worked hours	169
Automation	172
Optionally create a new Customer PO when editing a project	172
Create time entries from task assignments when the user creates a new timesheet	176
Control budgeted hours for a project using the project budget feature and a custom hours field	180
Workflow	182
Prevent a booking from being created if the selected resource has approved time off during the booking period	182
Prevent closing a project that has open issues	185
Automatically create a new issue when project stage is "at risk" and prevent project stage from changing until this issue is resolved	187
Send an alert email when a scheduled script completes	190
Send a Slack notification when issues are created or (re)assigned	191
User Scripting Release History	205

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, you should review [Scripting Best Practices](#).

 **Tip:** For a quick reference, see the  [OpenAir User Scripting Reference Card](#).

Scripting Switches

There are four 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 https methods** — enables you to access **NSOA.https** functions and call external APIs. See [Outbound Calling](#).
- **Enable user script support for Web Service API methods** — enables you to access the OpenAir SOAP API (Web Services) through the **NSOA.wsapi** functions. [SOAP API](#).

 **Note:** Contact OpenAir Customer 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])`

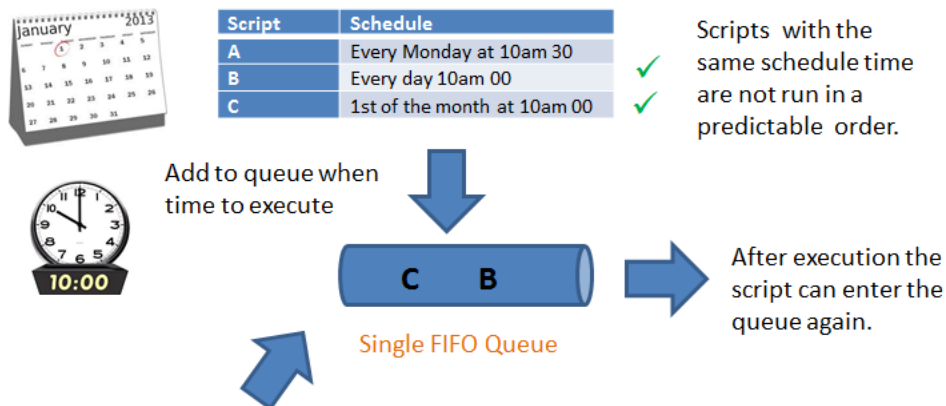
Important: Form scripts may be triggered by an event associated with user interaction — when a user clicks **Save**, for example.

Form scripts can also be triggered by an event associated with a process utilizing the form software logic — when importing project records from NetSuite using OpenAir NetSuite Connector, for example, depending on the integration configuration. For more information, see [Scripting and OpenAir NetSuite Connector](#).

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.



The Scripting Center > Scheduled > **Run script deployment / Run test code** places the script immediately in the 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, that is 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 Administrator Guide](#) Chapter 6 "Administration - Global Settings" for more information about 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 facilitate reading, transferring, archiving, and comparing them.

 **Tip:** All of the examples described in [Real World Use Cases](#) are provided as solutions, see [Creating Solutions](#).

Business Intelligence Connector


The OpenAir Business Intelligence Connector lets you publish OpenAir reports and list views to the OpenAir OData service. Reports can be published with different scope of use. All published reports are accessible with user scripting. You can publish reports for use with user scripting exclusively.


You can access the reports and list views published using OpenAir Business Intelligence Connector with the following functions:

- For reports: `NSOA.report.data(reportId,optionalParameters)` and `NSOA.report.list()`.
- For list views: `NSOA.listview.data(listviewId)` and `NSOA.listview.list()`

These functions give you access to the same information available when you use Business Intelligence tools to access your OpenAir OData feed.


You can use published list views like custom queries and read the latest list view data in your OpenAir form and scheduled scripts.

 **Note:** OpenAir Business Intelligence Connector must be enabled for your account to use `NSOA.listview` and `NSOA.report` functions. OpenAir Business Intelligence Connector is a licensed add-on. To enable this feature, contact your OpenAir account manager.

For more information about publishing list views and reports to the OpenAir OData service, see the  [OpenAir Business Intelligence Connector Guide](#).


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 go to the **Scripting Center** section.

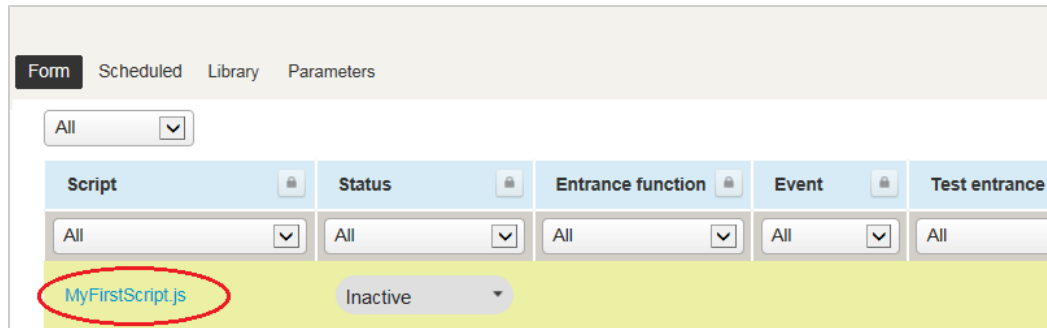
 **Note:** Make sure you have the necessary switches enabled, see [Scripting Switches](#).

2. Create a new script from the **Create Button**. See [Creating Form Scripts](#) and [Creating Scheduled Scripts](#).

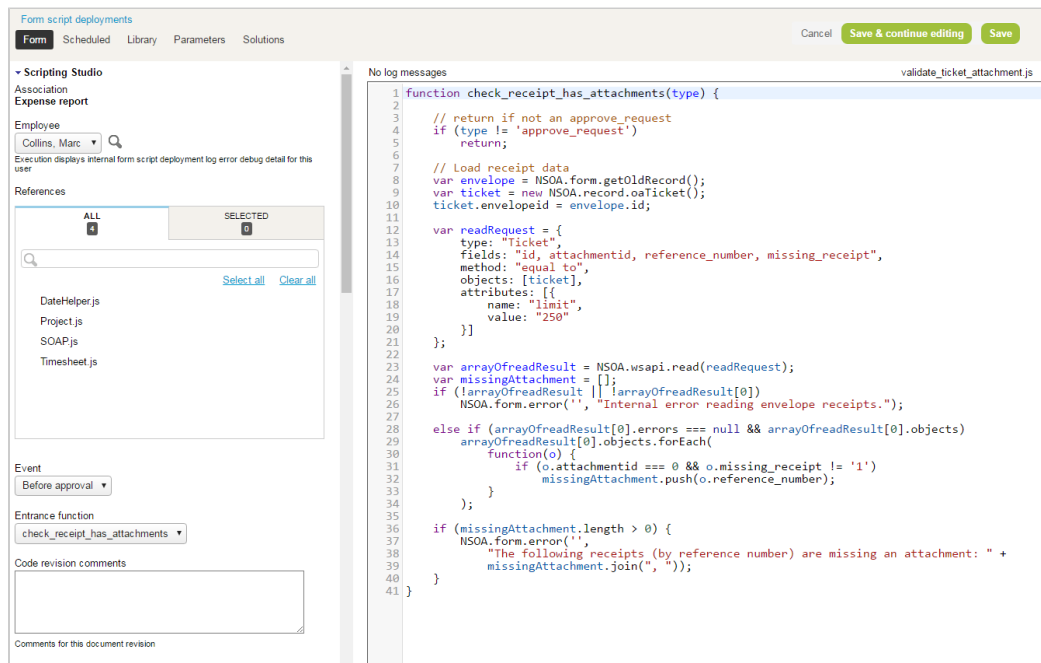
You need to specify a unique filename for the script in the [Dedicated Scripting Workspace](#). You can optionally select a document that already has the script you need otherwise an empty script file will be created. If you specify a document to upload then a new script file is created from the specified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

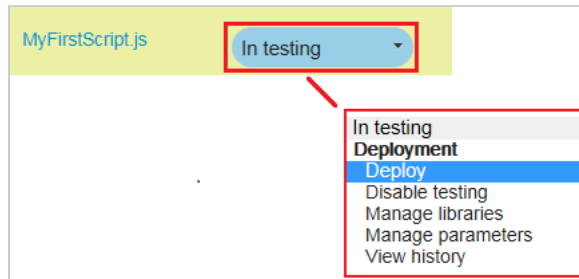


Note: The act of saving a script in the "Inactive" state will move the script to the "In testing" state, see [Scripting Workflow](#).

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 and 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 by OpenAir.
- **Message** — The full message text.

Note: OpenAir adds a log entry when one of the following script properties is changed, with an indication of what was changed: Source code, Event, Entrance function, Deployed status, Employee.

- **User** — The user who triggered the script.
- **Internal ID** — The internal ID of the log message. Sorting log entries by their Internal ID may be useful for debugging scripts when multiple log messages are recorded in the same second and you need to know the order the messages were recorded in.

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 by clicking the "View Log" link from the [Scripting Center](#) and [Scripting Studio](#), see also [Reporting](#).

The screenshot shows a log view interface with the following components:

- Filter:** A dropdown menu at the top left is set to "All" (labeled 1).
- Table Headers:** The table has columns for "Severity", "Internal ID" (sorted descending, labeled 2), "Timestamp", "Generated by", and "Message".
- Table Content:** The table displays 10 rows of log entries, each with an "Info" severity level, an "Internal ID", a "Timestamp", a "Generated by" field (mostly "System"), and a "Message" describing script changes.
- Summary:** Below the table, it shows "10 rows on page" and "23 total rows".
- Settings Menu:** A settings menu is open on the right side (labeled 3), showing options for "Customize list view", "Download list data" (labeled 4), and "Rows per page" (labeled 5). The "Rows per page" options are 10, 20, 50, 100, and All. The "Density" options are Compact, Comfortable (selected), and Disabled. The "Resize columns" options are Enabled and Disabled (selected).
- Navigation:** At the bottom, there are navigation buttons for "Previous", "1", "2", "3", "Next", and "Next Page".

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.

Script deployments				
Form	Scheduled	Library	Parameters	Solutions
All				
Script	Status	Form name	Log	
All	All	Project		
ProjectCF.js	In testing	Project	View Log (3)	
TestScript.js	Inactive	Project	No log messages	

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

View Log (3)	
1	<code>function main(type) {</code>
2	<code> NSOA.form.confirmation('confirmation message');</code>
3	<code>}</code>

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:

```
1 NSOA.meta.log('debug', "message");
2 NSOA.meta.log('Debug', "message");
3 NSOA.meta.log('DEBUG', "message");
```

The following are also treated as the same:

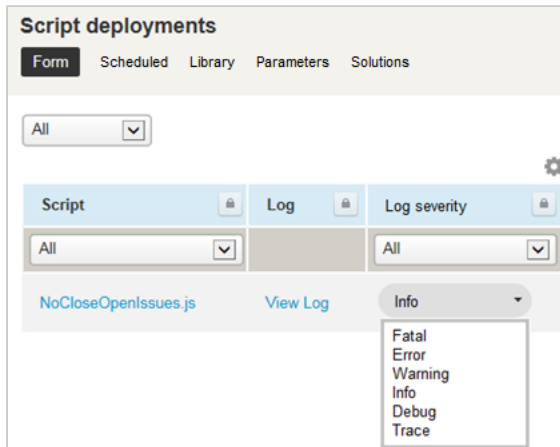
```
1 NSOA.meta.log('myseverity', "message");
2 NSOA.meta.log('Info', "message");
```

This is the same as calling:

```
1 | 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.

All

Form script deployment Messages

Severity	Timestamp	Generated by	Message
Fatal	2017-03-06 05:57:06	System	User script timed out (exceeded 10s, start: 2017-03-06 05:56:13, end: 2017-03-06 05:57:06) (terdf.js function setCustomCenterField).
Trace	2017-03-06 05:57:06	System	JS function 'NSOA.meta.log' started at 2017-03-06 05:56:15.34914, ended at 2017-03-06 05:56:15.35183 (dur. 0.00269s)
Trace	2017-03-06 05:57:06	System	JS function 'NSOA.wsapi.modify' started at 2017-03-06 05:56:13.91, ended at 2017-03-06 05:56:15.34211 (dur. 1.43212s)
Trace	2017-03-06 05:57:06	System	JS function 'NSOA.wsapi.disableFilterSel' started at 2017-03-06 05:56:13.90615, ended at 2017-03-06 05:56:13.90967 (dur. 0.00352s)
Trace	2017-03-06 05:57:06	System	JS function 'NSOA.meta.log' started at 2017-03-06 05:56:13.90323, ended at 2017-03-06 05:56:13.90585 (dur. 0.00263s)
Trace	2017-03-06 05:57:06	System	JS function 'NSOA.form.getNewRecord' started at 2017-03-06 05:56:13.58962, ended at 2017-03-06 05:56:13.61895 (dur. 0.02933s)
Trace	2017-03-06 05:57:06	System	Script 'terdf.js' started at 2017-03-06 05:56:13.572168

Clear Log Entries for a Specific Script

You can clear all log entries for a specific script from the Scripting Center.

To clear log entries for a script

1. Go to Administration > Scripting Center.
2. Click the status dropdown for the script in the **Status** column, then click **Clear log**.

A confirmation dialog appears.

3. Click **OK** to clear the logs.

If there were any log messages to be cleared, the log now contains a single entry indicating that the log was cleared manually.

Script	Status	Log	Form name
All	All		Project
LanguageTest.js	In testing	View Log (1)	Project
ListView.js	Deployment	View Log (7572)	Project
postSlackMessage.oa.js	Deploy	log messages	Project
SOAPCall.js	Disable testing	log messages	Project
SOAPTest.js	Manage libraries	log messages	Project
Test.js	Manage parameters	log messages	Project
	Manage custom fields	log messages	Project
	View history	log messages	Project
	Export solution	log messages	Project
	Logging		
	Clear log		

9 rows

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.

- No maintenance task
- Regenerate all utilization tables for booked utilization reports
- Regenerate pending utilization tables for booked utilization reports
- Set the PO purchaser to its original value
- Generate planned hours for each employee assigned to a task
- Update percent complete and recalculate all active projects
- Recalculate all projects and assigned utilization tables
- Generate cost center associations for receipts and time entries without cost centers
- Delete saved reports from inactive employees
- Calculated field entity determination and validation
- Rebuild registry
- Recalculate import/export state support tables
- Delete temporary pending bookings for all projects
- Delete all ▼ script deployment user logs older than 30 days with log level at or below Debug ▼

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

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

Important: You should 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

Form script deployment log detail report Clear sort 								
modify report		re-run report						
Generated by	Severity	Message	Entrance function	Form name	Event	User	Document	Workspace
System	Info	isDebugMode is not defined at user script line 3 (test.js function test).	test	project_edit_form	Before save	Collins, Marc	test.js	UserScripts
System	Info	oaAddress is not defined at user script line 5 (test.js function test).	test	project_edit_form	Before save	Collins, Marc	test.js	UserScripts
System	Info	oaEstimatephase is not defined at user script line 6 (test.js function test).	test	project_edit_form	Before save	Collins, Marc	test.js	UserScripts
System	Info	oaPurchaser is not defined at user script line 7 (test.js function test).	test	project_edit_form	Before save	Collins, Marc	test.js	UserScripts
System	Info	oaDate is not defined at user script line 8 (test.js function test).	test	project_edit_form	Before save	Collins, Marc	test.js	UserScripts

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

Scheduled script deployment log detail report		
Generated by	Severity	Message
System	Fatal	Neither document revision nor code exists for schedule script deployment78
System	Fatal	Cannot save this form due to error in schedule script deployment -1. Please contact account administrator with this error.

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. You should create the following roles:

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

Roles can be assigned several 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				

Permissions	Script Administrator	Script Developer	Script QA	Script Deploy
Enable script testing	✓	✓	✓	
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 and OpenAir Mobile

OpenAir Mobile 4.0 or later version supports:

- All form scripts associated with the expense report and receipt entity forms.
- "Before approval" and "After approval" scripts associated with the timesheet entity form.


Note: "On submit," "Before save," or "After save" scripts associated with the timesheet entity form are not supported.

For an example of script that is executed both in OpenAir and OpenAir Mobile, see

Scripting and OpenAir NetSuite Connector

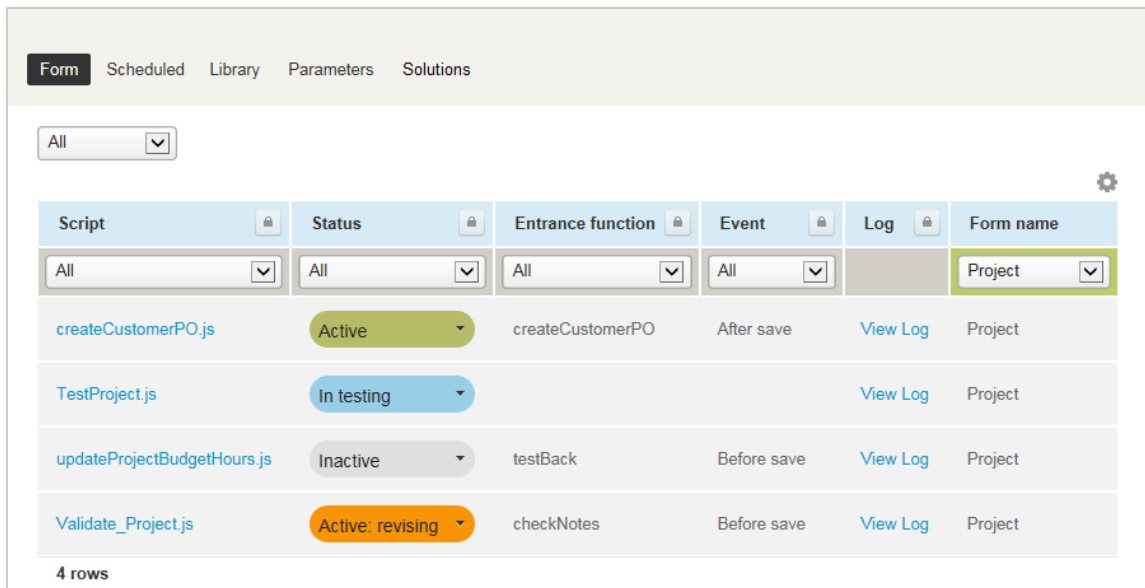
You can use the following OpenAir user scripting functions to trigger an integration run:

- `NSOA.NSConnector.integrateAllNow()` — Use this function to import and export records in bulk from your scheduled scripts. The run will include all integration workflows that are active at the time the run is triggered. See [NSOA.NSConnector.integrateAllNow\(\)](#).
- `NSOA.NSConnector.integrateWorkflowGroup(name)` — Use this function to import and export records in bulk from your scheduled scripts. The run will include only integration workflows in the workflow group specified by name. See [NSOA.NSConnector.integrateWorkflowGroup\(name\)](#).
- `NSOA.NSConnector.integrateRecord()` — Use this function to export a single OpenAir record from your form scripts. See [NSOA.NSConnector.integrateRecord\(\)](#).

If you are using the NetSuite <> OpenAir integration, and depending on the integration configuration, NetSuite Connector may use software logic associated with the Project form when importing project records from NetSuite to OpenAir. In this case, form scripts associated with the Project form and triggered by an “On submit”, “Before save”, or “After save” event will run for each imported project record. This will impact the performance of your integration runs and may result in errors related to scripting governance limits. For more information about configuration options that result in the integration triggering form scripts, see  [OpenAir NetSuite Connector Guide](#).

User Scripting

Scripting Center



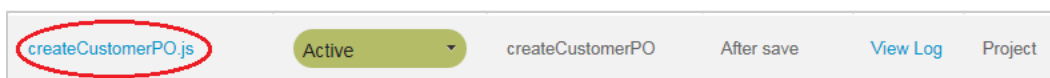
Script	Status	Entrance function	Event	Log	Form name
All	All	All	All		Project
createCustomerPO.js	Active	createCustomerPO	After save	View Log	Project
TestProject.js	In testing			View Log	Project
updateProjectBudgetHours.js	Inactive	testBack	Before save	View Log	Project
Validate_Project.js	Active: revising	checkNotes	Before save	View Log	Project

4 rows

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.

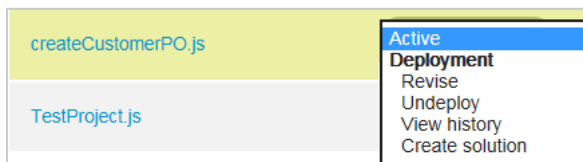
The Scripting Center has five tabs:

- **Form** — See [Creating Form Scripts](#).
- **Scheduled** — See [Creating Scheduled Scripts](#).
- **Library** — See [Creating Library Scripts](#).
- **Parameters** — See [Creating Parameters](#).
- **Solutions** — See [Creating Solutions](#).



createCustomerPO.js	Active	createCustomerPO	After save	View Log	Project
-------------------------------------	--------	------------------	------------	--------------------------	---------

From the Scripting Center you can launch the [Scripting Studio](#) by clicking on a script link.



createCustomerPO.js	<ul style="list-style-type: none"> Active Deployment Revise Undeploy View history Create solution
TestProject.js	

Scripts are moved through the [Scripting Workflow](#) from the **Status** menu.

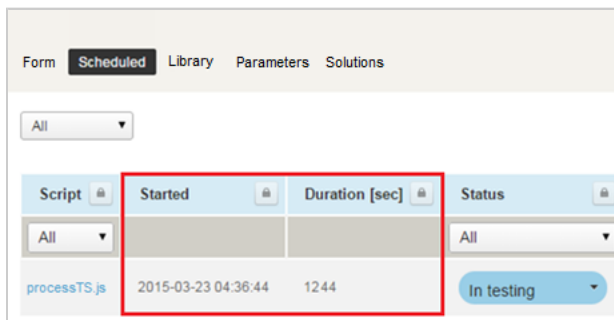
Note: Customers that choose not to use the Scripting Studio in favor of another editor are still fully supported from the Scripting Center.

You can view any log messages the script has generated using the “View Log” link, see [Logs](#).

You can clear all log entries for a specific script from the Scripting Center using the **Clear log** option in the **Status** dropdown list. See [Clear Log Entries for a Specific Script](#).

- **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



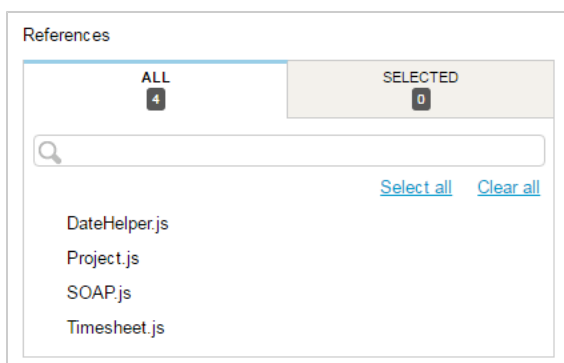
Script	Started	Duration [sec]	Status
processTS.js	2015-03-23 04:36:44	1244	In testing

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



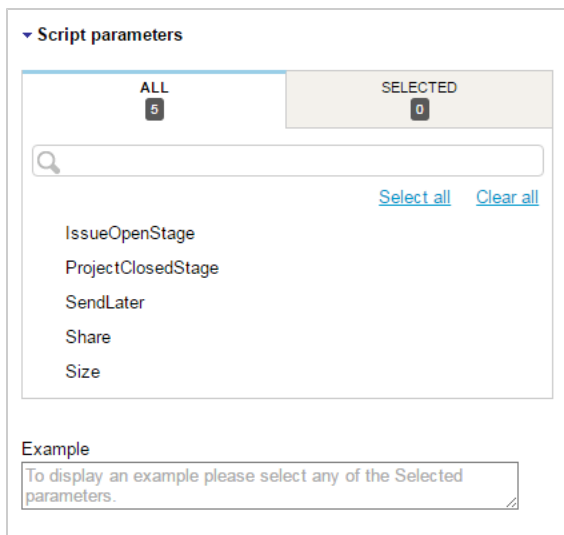
References
<div style="display: flex; justify-content: space-between;"> ALL 4 SELECTED 0 </div> <div style="margin-top: 10px;"> <input type="text"/> <div style="text-align: right; margin-top: 5px;"> Select all Clear all </div> <ul style="list-style-type: none"> DateHelper.js Project.js SOAP.js Timesheet.js </div>

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



▼ Script parameters

ALL 5 SELECTED 0

Search

[Select all](#) [Clear all](#)

IssueOpenStage
ProjectClosedStage
SendLater
Share
Size

Example

To display an example please select any of the Selected 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. For each version of a deployed script (document revision), the Script deployment history page shows:

- The deployed script source code
- Deployment comments
- When the script was deployed and by whom. The date and time is given as Eastern Time (UTC – 5).



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.

Script deployment history

▼ Document revision LanguageTest.js

Document : Revision
 :

Document download
[Download selected document](#)

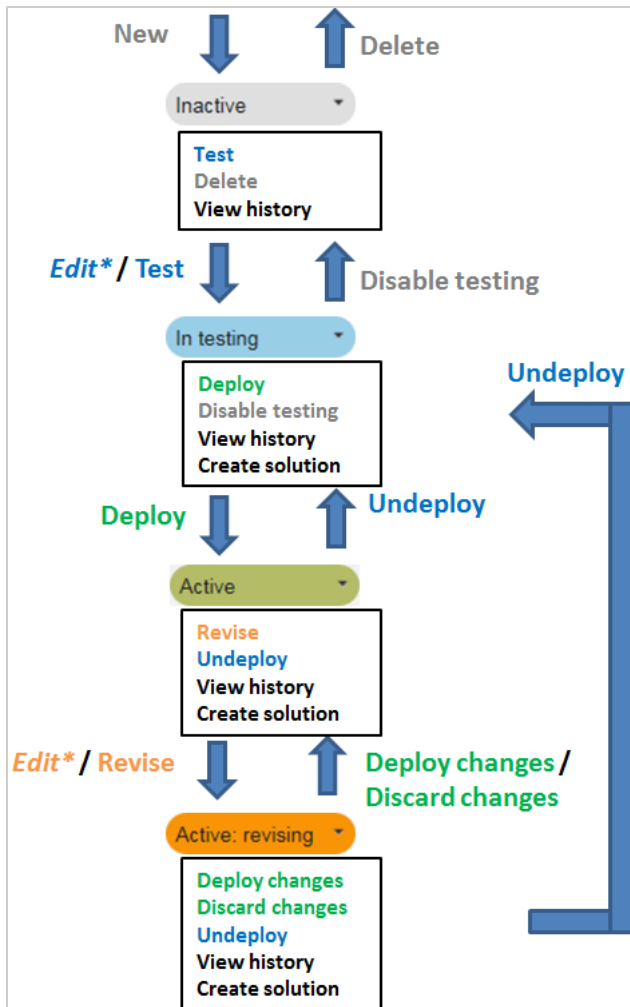
Document comments
 Removed unused library references from source code to prepare for production release

Created
 2023-01-18 03:08:58 by Collins, Marc

```

1 function main(type) {
2     NSOA.meta.alert("Script started for language test");
3
4
5     // return user language code : en | fr | de | es | zh | ja | cs
6     var before_units = NSOA.context.remainingUnits();
7     var lang = NSOA.context.getLanguage();
8     var after_units = NSOA.context.remainingUnits();
9
10    if (lang == 'cs') {
11        // display messages in the Czech language"
12        var msg = NSOA.context.getParameter('CSMessage');
13        NSOA.form.error("", "The message was: " + msg + " before units " + before_units + "
after units " + after_units);
14    }
15    else
16        NSOA.form.error("", "unexpected language: " + lang);
17
18 }
```

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.

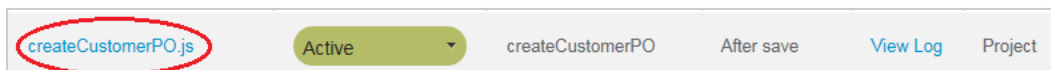
Status	Actions
Inactive	<ul style="list-style-type: none"> ■ Test — Prompts for test settings and on SAVE moves the script to In testing. ■ Delete — Prompts for confirmation and on OK deletes the script code and all associated history.

	<ul style="list-style-type: none"> View history — see View history. Click the script link to make changes in the Scripting Studio. On SAVE the script moves to In testing.
In testing	<ul style="list-style-type: none"> 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.
Active	<ul style="list-style-type: none"> 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.
Active: revising	<ul style="list-style-type: none"> 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

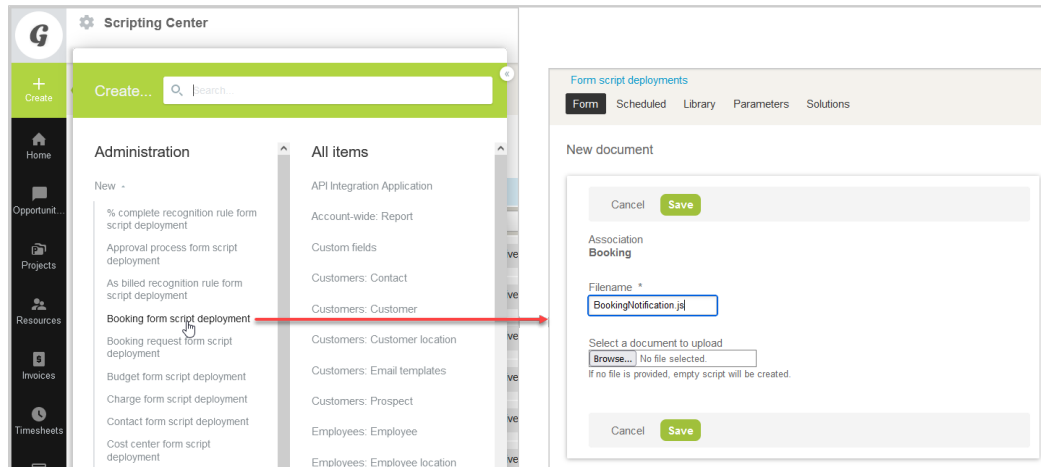
Form scripts are created from the **Create Button**. You need to specify a unique filename for the script in the dedicated scripting workspace. You can optionally select a document that already has the script you need otherwise a blank script file will be created. If you specify a document to upload 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 and 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](#).

After a script is 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 using 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:

```

1 function test()
2 {
3   Var_value = NSOA.form.getValue('budget_time');
4   var label = NSOA.form.getLabel('budget_time');
5
6   NSOA.form.error('budget_time', "error message");
7 }
8

```

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.

This form has a problem. Please fix it and try again.

missing ; before statement at line 3 (workspace document function test).

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.

createCustomerPO.js Active createCustomerPO After save [View Log](#) Project

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, that is, **NSOA.form.getLabel2** does not exist.

```


1 function test() {
2   var value = NSOA.form.getValue('budget_time');
3   var label = NSOA.form.getLabel2('budget_time');
4 }


```

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:** Test scripts thoroughly in a Sandbox account before deploying to a Production account.

See also [Testing and 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.

Form script deployments

Form Scheduled Library Parameters Solutions

Cancel Save

▼ Pending form script deployment

Association
Folder

Code comments
(no message)

Event
Before approval

Entrance function
check_receipt_has_attachments

Notes

Select user to execute script deployment

Select... 🔍

✓ **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.

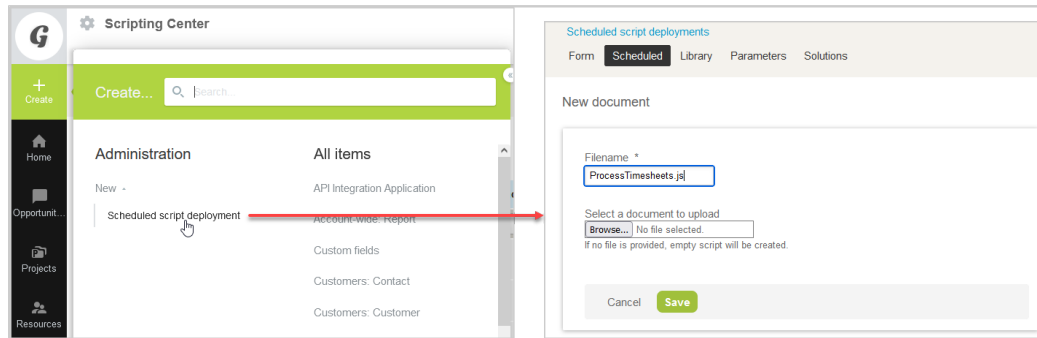
Scheduled scripts are not associated with a form and cannot access the NSOA.form functions.

To create a scheduled script:

1. Log in as an Administrator and go 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 and 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 from the **Status** menu.

Script	Solutions	Entrance function	Event	Test entrance function	Test event	Status	Test employee	
All		All	All	All	All	All	All	
Process_Timesheets.js	test.xml			main	Schedule	In testing	Collins, Marc	
							<ul style="list-style-type: none"> In testing Deployment <ul style="list-style-type: none"> Deploy Disable testing Manage libraries Manage parameters Manage custom fields View history Export solution Execution <ul style="list-style-type: none"> Run test code 	

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.

This form has a problem. Please fix it and try again.
 missing ; before statement at line 3 (workspace document function test).

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.

createCustomerPO.js Active createCustomerPO After save **View Log** Project

Click on the **View Log** link to see the log messages. See also [Reporting](#).

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, that is, **NSOA.form.getLabel2** does not exist.

```

1 function test() {
2     var value = NSOA.form.getValue('budget_time');
3     var label = NSOA.form.getLabel2('budget_time');
4 }

```

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:** Test scripts thoroughly in a Sandbox account before deploying to a Production account.

See also [Testing and Debugging](#).

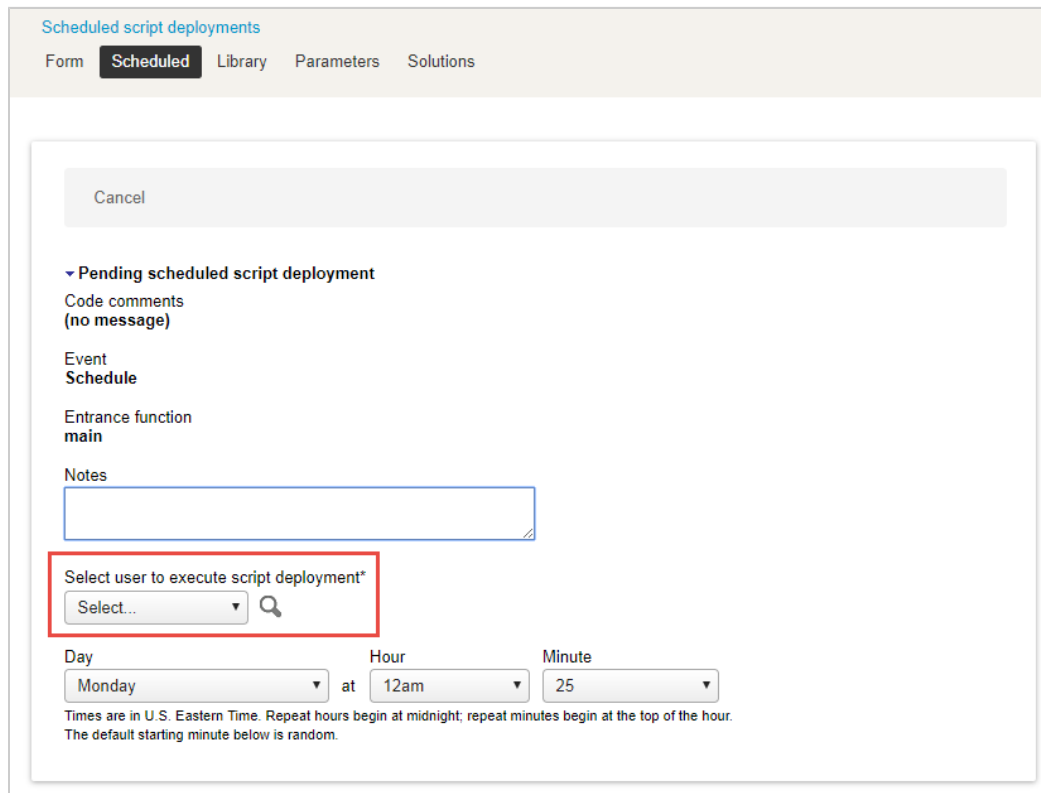
Deploying Scheduled Scripts

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-administrator 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.



Scheduled script deployments

Form **Scheduled** Library Parameters Solutions

Cancel

▼ Pending scheduled script deployment

Code comments
(no message)

Event
Schedule

Entrance function
main

Notes

Select user to execute script deployment*

Select... 🔍

Day: Monday at Hour: 12am Minute: 25

Times are in U.S. Eastern Time. Repeat hours begin at midnight; repeat minutes begin at the top of the hour.
The default starting minute below is random.

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, for example 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 about 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.

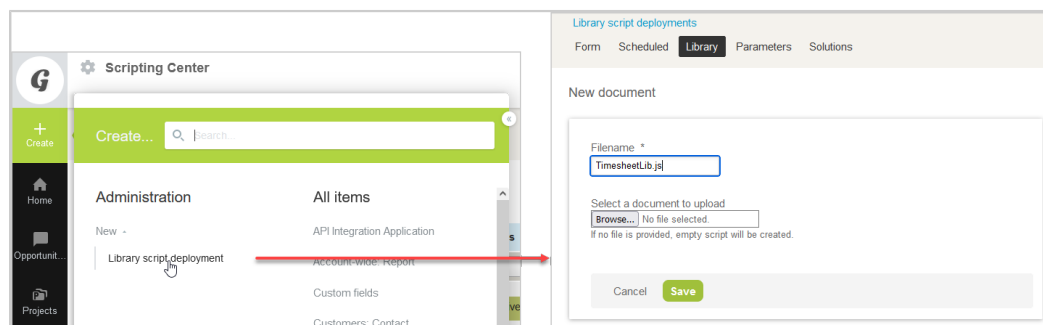
References to libraries can be set from the Scripting Center [Manage libraries](#) or from the Scripting Studio [Scripting Studio Tools and Settings](#).

To create a library script:

1. Log in as an Administrator and go 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.

The screenshot displays the 'Library script deployments' window. At the top, there are tabs for 'Form', 'Scheduled', 'Library' (selected), 'Parameters', and 'Solutions'. Action buttons for 'Cancel', 'Save & continue editing', and 'Save' are visible. The main area is divided into several sections: 'Scripting Studio' with an 'Employee' dropdown set to 'Collins, Marc'; 'References' with 'ALL' (1) and 'SELECTED' (0) counts, a search bar, and a list containing 'Timesheet.js'; 'Referenced by' showing 'NoCloseOpenIssues.js'; and 'Code revision comments' with an empty text area. On the right, a 'View log' pane shows the following code:

```

1 function name1 () {
2
3   var user = NSOA.wsapi.whoami();
4
5   return user.name;
6
7 }
8
9 exports.name2 = name1;

```

Create functions in 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.

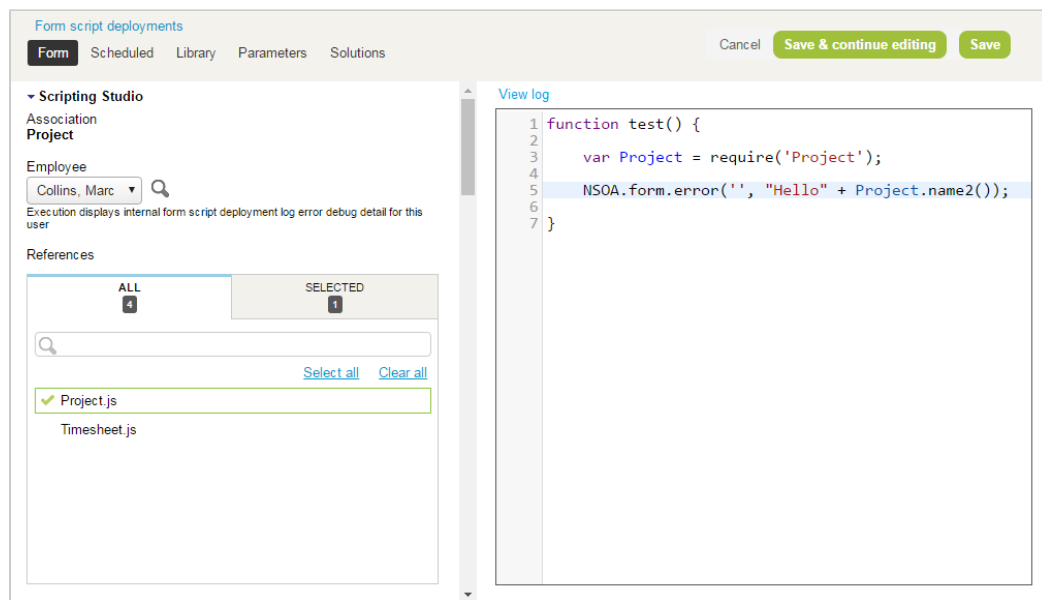
5. Fill out the fields in the Scripting Studio Tools and Settings:
 - a. Select the user that the script will run for 'In testing' state, see [Testing and Debugging](#).
 - b. Select any libraries referenced by this library.
 - c. Use the **Code revision comments** to comment the script changes made.
 - 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.

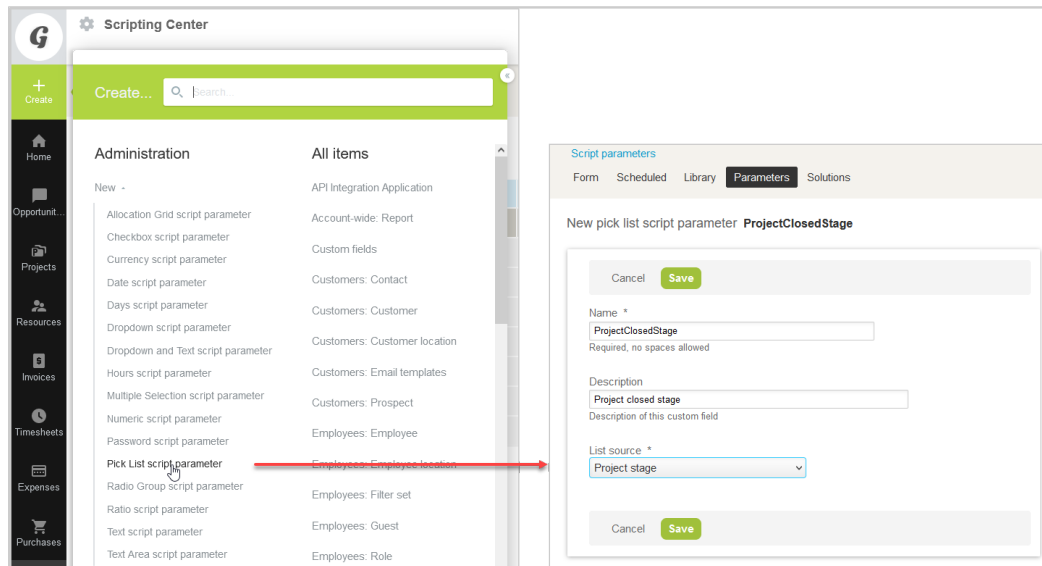
References to parameters can be set from the Scripting Center [Manage parameters](#) or from the Scripting Studio [Script Parameters](#) section.

To create a parameter:

1. Log in as an Administrator and go 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.

The screenshot shows the 'Parameters' tab in the Scripting Center. It displays a table with the following columns: Name, Description, Type, Value, and Referenced by. The table contains three rows of parameters.

Name	Description	Type	Value	Referenced by
All		All		
DayOfWeek	No Description	Days	Set	
ProjectClosedStage	Project closed stage	Pick List	Set	NoCloseOpenIssues.js
IssueOpenStage	Issue open stage	Pick List	Set	NoCloseOpenIssues.js

3 rows

- 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.

```

1 // project_stage_id and issue_stage_id depend on account settings
2 function test_prevent_project_close_with_open_issue() {
3
4 // return if new stage is not closed
5 if (NSOA.form.getValue('project_stage_id') != NSOA.context.getParameter('ProjectClosedStage'))
6   return;
7
8 // Load issue data
9 var issue = new NSOA.record.oaIssue();
10 issue.project_id = NSOA.form.getValue('id');
11 issue.issue_sStage_id = NSOA.context.getParameter('IssueOpenStage');
12
13 var readRequest = {
14   type : "Issue",
15   fields : "id, date",
16   method : "equal to",
17   objects : [ issue ],
18   attributes : [{
19     name : "limit",
20     value : "1"
21   }]
22 };

```

4. Administrators can change the script values from the calling script in the [Scripting Center](#).

Script	Parameters	References	Status	Form name	Log
All			Active	All	
NoCloseOpenIssues.js	IssueOpenStage ProjectClosedStage	Timesheet.js	Active	Project	View Log

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


Status	Solution	Title	Description	Log
All	All	All		
Applied	RWEB_2.xml	RWEB	Real world example 8: prevent closing project with open issues	View Log
Not applied	validate_project_commission.xml	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. • 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	No log messages

2 rows

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 so you can read, transfer, archive, and compare them. Solutions contain a version tag to allow OpenAir 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.

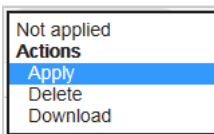
Solution Status

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. Go 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. After 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**.

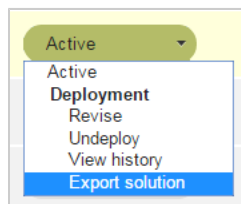
Note: You only need to select additional custom fields and parameters. When you select a script, the solution will automatically pull in the script's required libraries and parameters. OpenAir ignores duplicate selections.

To create a platform solution from a single script:

1. Log in as an Administrator and go to the [Scripting Center](#).

Note: Make sure you have the necessary switches enabled, see [Scripting Switches](#).

2. Go 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.

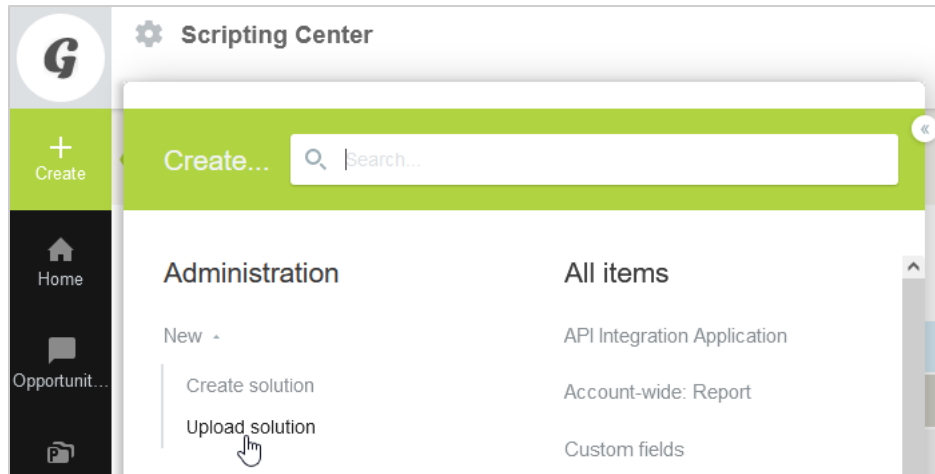
Tip: The solution will contain all library scripts and parameters referenced by the script. To create a solution without a certain reference, first remove the reference from the script and then create the solution.

To apply a platform solution:

1. Log in as an Administrator and go to the **Solutions** tab on the [Scripting Center](#).

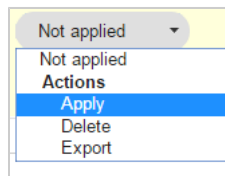
Note: Make sure you have the necessary switches enabled, see [Scripting Switches](#).

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 > Display > 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 "%TERMINID%" 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.

```

1 var proj_term = NSOA.context.getTerm("Projects");
2 // proj_term = "Jobs"
3
4 var msg1 = NSOA.context.parseTerminology("%Project% saved!");
5 // msg1 = "Job saved!"
6
7 var msg2 = NSOA.context.parseTerminology("Notes attached to %project%.");
8 // msg2 = "Notes attached to job."
9
10 // Automatic terminology parsing
11 NSOA.form.error("", "%Project% saved!");
12 NSOA.meta.alert("%Project% saved!");
13 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 screenshot shows the Scripting Studio interface. On the left, there is a sidebar with the following sections:

- Form script deployments:** Form, Scheduled, Library, Parameters, Solutions.
- Scripting Studio:** Association (Expense report), Employee (Collins, Marc), References (DateHelper.js, Project.js, SOAP.js, Timesheet.js), Event (Before approval), Entrance function (check_receipt_has_attachments), Code revision comments.

The main area displays a script editor with the following code:

```

1 function check_receipt_has_attachments(type) {
2
3 // return if not an approve_request
4 if (type != 'approve_request')
5     return;
6
7 // Load receipt data
8 var envelope = NSOA.form.getOldRecord();
9 var ticket = new NSOA.record.oaTicket();
10 ticket.envelopeid = envelope.id;
11
12 var readRequest = {
13     type: "Ticket",
14     fields: "id, attachmentid, reference_number, missing_receipt",
15     method: "equal to",
16     objects: [ticket],
17     attributes: [{
18         name: "limit",
19         value: "250"
20     }]
21 };
22
23 var arrayOfreadResult = NSOA.wsapi.read(readRequest);
24 var missingAttachment = [];
25 if (!arrayOfreadResult || !arrayOfreadResult[0])
26     NSOA.form.error('', "Internal error reading envelope receipts.");
27
28 else if (arrayOfreadResult[0].errors == null && arrayOfreadResult[0].objects)
29     arrayOfreadResult[0].objects.forEach(
30         function(o) {
31             if (o.attachmentid == 0 && o.missing_receipt != '1')
32                 missingAttachment.push(o.reference_number);
33         }
34     );
35
36 if (missingAttachment.length > 0) {
37     NSOA.form.error('',
38         "The following receipts (by reference number) are missing an attachment: " +
39         missingAttachment.join(", ");
40     );
41 }

```

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](#)
- [OData Explorer](#)
- [Script Parameters](#)
- [Terminology](#)
- [Form Schema](#)
- [Testing and 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

The screenshot displays the 'Scripting Studio' configuration panel for an 'Expense report' association. It includes a search field for the 'Employee' (set to 'Collins, Marc'), a 'References' section with a list of scripts (DateHelper.js, Project.js, SOAP.js, Timesheet.js) and 'Select all'/'Clear all' links, an 'Event' dropdown (set to 'Before approval'), an 'Entrance function' dropdown (set to 'check_receipt_has_attachments'), and a 'Code revision comments' text area.

▼ **Scripting Studio**
Association
Expense report

Employee
Collins, Marc

Execution displays internal form script deployment log error debug detail for this user

References

ALL 4 | SELECTED 0

Select all Clear all

DateHelper.js
Project.js
SOAP.js
Timesheet.js

Event
Before approval

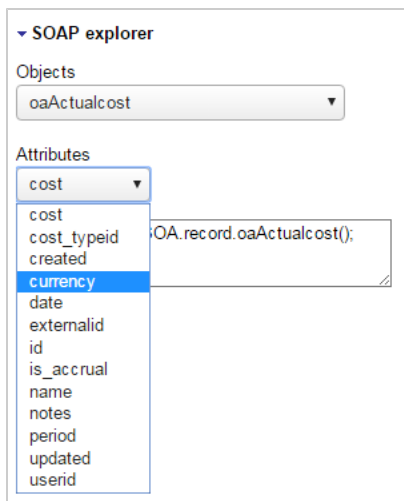
Entrance function
check_receipt_has_attachments

Code revision comments

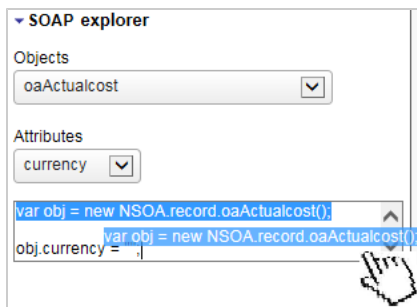
Comments for this document revision

- **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 and 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

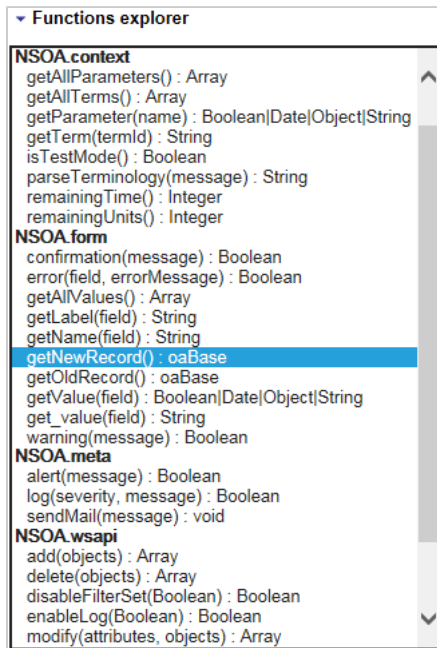


From the SOAP explorer you can browse through the SOAP API objects and attributes, and view examples of usage. Select a SOAP object, then select an attribute to view a code example using information for that object and attribute.

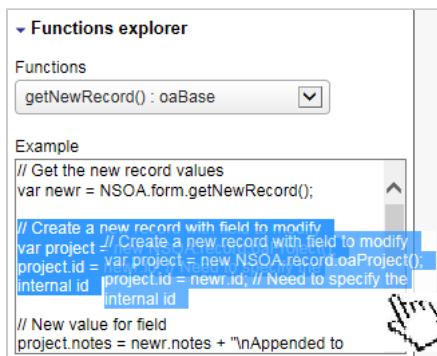


You can drag and drop code examples directly into the editor. See also the [Auto List & Complete](#) feature.

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.

OData Explorer

Use the OData explorer to browse your published OpenAir reports and list views, and the columns available in these resources. Select the resource type (published list view or published report), the resource, and the column, to view a code example using information in that resource and column. Select and drag the code example into the editor pane to use the snippet in your script.

The OData explorer shows both the OData resource ID and the saved list view or saved report title. This helps you identify and reference the correct OData resource by ID directly from the scripting studio.

See also [Business Intelligence Connector](#) and the following OpenAir User Scripting functions:

- For reports: [NSOA.report.data\(reportId,optionalParameters\)](#) and [NSOA.report.list\(\)](#).

- For list views: `NSOA.listview.data(listviewId)` and `NSOA.listview.list()`

Note: This functionality is available only if the OpenAir Business Intelligence Connector is enabled for your account. OpenAir Business Intelligence Connector is a licensed add-on. To enable this feature, contact your OpenAir account manager.

For more information about publishing list views and reports to the OpenAir OData service, see the [OpenAir Business Intelligence Connector Guide](#).

Form script deployments

Form Scheduled Library Parameters Solutions

▼ OData explorer

Types
Published list views ▼

Resources
[3] All Projects ▼

Fields
Project_owner ▼

```
var iterator = NSOA.listview.data(3);
// get the first record from iterator
var record = iterator.next();
var value = record["Project_owner"];
```

Script Parameters

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

▼ Script parameters

ALL 5 SELECTED 1

IssueOpenStage

Example

```
NSOA.context.getParameter("IssueOpenStage")
```

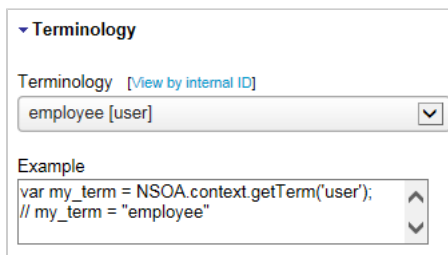
Click on a selected parameter to see an example. You can drag and drop code examples directly into the editor.

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.



▼ Terminology

Terminology [\[View by internal ID\]](#)

employee [user] ▼

Example

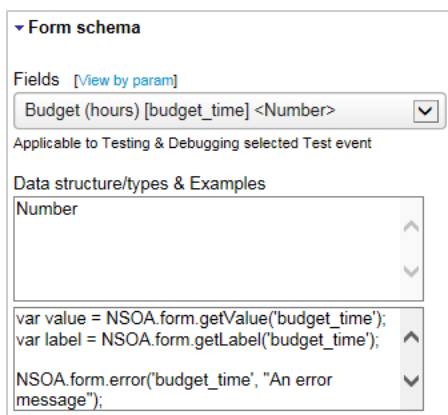
```
var my_term = NSOA.context.getTerm('user');
// my_term = "employee"
```

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.



▼ Form schema

Fields [\[View by param\]](#)

Budget (hours) [budget_time] <Number> ▼

Applicable to Testing & Debugging selected Test event

Data structure/types & Examples

Number

Example

```
var value = NSOA.form.getValue('budget_time');
var label = NSOA.form.getLabel('budget_time');
NSOA.form.error('budget_time', "An error message");
```

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.

Note: When editing a form script associated with the Receipt form, the hidden fields **Receipt subtype [subtype] <String>** and **Envelope [envelope_id] <Number>** are available in addition to the fields visible on the form:

- The **subtype** field indicates whether the receipt is a **Mileage receipt**, a regular **Receipt** or **Foreign currency receipt**. The receipt subtype is determined by the type of the associated expense item.
 - The function `NSOA.form.getValue('subtype')` will return a string with the value 'M' if it is a Mileage receipt, and 'R' otherwise (regular Receipt or Foreign currency receipt).
 - The functions `NSOA.form.getOldRecord()` and `NSOA.form.getNewRecord()` will return an object of type `oaTicket` with the associated expense item referenced by ID `oaTicket.item_id`. You can also use the item type field `oaItem.type` to determine the subtype of a receipt.
- The **envelope_id** field is the ID of the envelope the receipt is associated to.
 - The function `NSOA.form.getValue('envelope_id')` will return the envelope ID.
 - The functions `NSOA.form.getOldRecord()` and `NSOA.form.getNewRecord()` will return an object of type `oaTicket` with the associated envelope referenced by ID `oaTicket.envelopeid`.

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](#).

```
var value = NSOA.form.getValue("budget_time");
var label = NSOA.form.getLabel("budget_time");

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

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.

```
Fields [View by param]
Loaded hourly cost [loaded_cost] <Object>

Data structure/types & Examples
[
  {
    "cost_0": "Primary loaded cost <Number>",
    "cost_1": "Secondary loaded cost <Number>",
    "cost_2": "Tertiary loaded cost <Number>",
    "user_id": "Resource <Number>"
  },
  {
    "cost_0": "Primary loaded cost <Number>",
    "cost_1": "Secondary loaded cost <Number>",
    "cost_2": "Tertiary loaded cost <Number>",
    "user_id": "Resource <Number>"
  },
  {
    "cost_0": "Primary loaded cost <Number>",
    "cost_1": "Secondary loaded cost <Number>",
    "cost_2": "Tertiary loaded cost <Number>",
    "user_id": "Resource <Number>"
  }
]

var loaded_cost_obj = NSOA.form.getValue
("loaded_cost");
var value = loaded_cost_obj[0].cost_0; // returns a
Number

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

NSOA.form.error(NSOA.form.getName
("loaded_cost")[0].cost_0, "An error message");
```

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:

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

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


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

You can combine these two steps into one line:

```
1 | 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, that is, `loaded_cost_obj[0]` is this first row. Each column of the row is accessed by the field name that is `loaded_cost_obj[0].cost_0` is the "Primary loaded cost" for the first row.

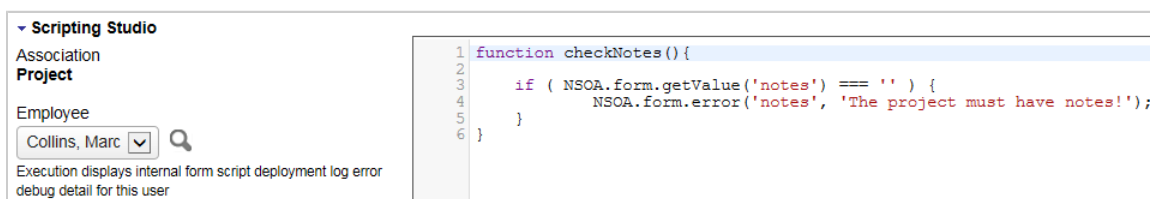
 **Note:** Remember [Arrays](#) are zero-based.

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

Notice this is `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 and Debugging



The screenshot shows the Scripting Studio interface. On the left, there is a sidebar with a tree view containing 'Association', 'Project', and 'Employee'. Under 'Employee', a dropdown menu is open showing 'Collins, Marc' selected. Below the dropdown, there is a search icon and a note: 'Execution displays internal form script deployment log error debug detail for this user'. On the right, a code editor displays the following JavaScript code:

```
1 function checkNotes(){
2
3     if ( NSOA.form.getValue('notes') === '' ) {
4         NSOA.form.error('notes', 'The project must have notes!');
5     }
6 }
```

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 about **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.

```

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 }

```

Note: Point to 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

```

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 }

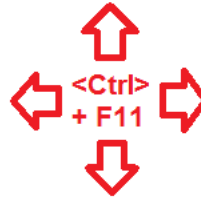
```

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

```

1  /**
2  * Prepends a project number to the project name field. This is helpful
3  * for demonstrating auto-numbering or complex project naming. This script
4  * uses the standard project.name field so that the new project number will
5  * display in dropdowns, reports, and offline/mobile clients.
6  *
7  * Version      Date          Author          Remarks
8  * 1.00         27 Nov 2013    Ryan Morrissey
9  *
10 */
11
12
13 function peter_function(type) {
14     try {
15         var FLD_CUSTPO_NUM = 'prj_custpo_num__c',
16             FLD_CUSTPO_AMT = 'prj_custpo_amt__c',
17             FLD_CUSTPO_DATE = 'prj_custpo_date__c',
18             FLD_CREATE_PO = 'prj_create_po__c';
19
20         // get updated project record fields
21         var updPrj = NSOA.form.getNewRecord();
22
23         // if the "Create PO" checkbox is checked and a PO number is entered,
24         // create a PO
25         if (updPrj[FLD_CREATE_PO] == '1' && updPrj[FLD_CUSTPO_NUM]) {
26             var recCustPO = new NSOA.record.oaCustomerpo();
27             recCustPO.number = updPrj[FLD_CUSTPO_NUM];
28             recCustPO.name = updPrj[FLD_CUSTPO_NUM] + ' ' + updPrj.name;
29
30             // use the PO date if available, otherwise use project start date
31             if (updPrj[FLD_CUSTPO_DATE] != '0000-00-00') {
32                 recCustPO.date = updPrj[FLD_CUSTPO_DATE];
33             } else {
34                 recCustPO.date = updPrj.start_date;
35             }
36
37             // currency custom fields return ISO-#.##; remove the ISO code and
38             dash      var cleanAmt = updPrj[FLD_CUSTPO_AMT].replace(/-\w{3}/, '');
39
40             // 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”.

After a search dialog is opened, press Escape to exit it without searching.

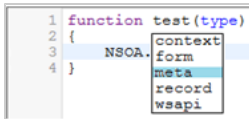
- **Jump to Line Functions** — You can move through your scripts quickly by entering Jump to Line functions. Press Alt+G to open the Jump to Line dialog. Then, enter the script line you want to move the cursor to. The following input formats are accepted:
 - **Line** — enter the line to move the cursor 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 backward to the middle of the script.

After the Jump to Line dialog is opened, press Esc to exit it without moving the cursor.

Auto List & Complete

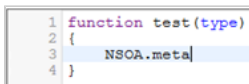
When a user types the text “NSOA” into the **Scripting Studio** editor and then hits the ‘.’ character the Auto List window appears showing all the available options:



The user has the following options:

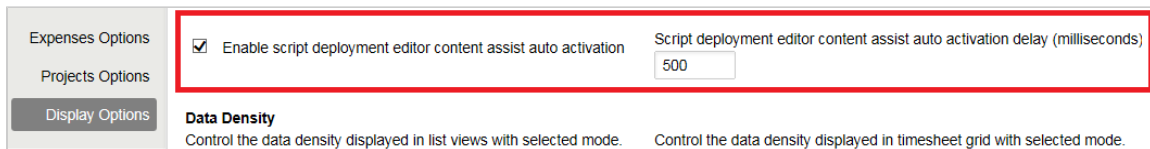
- Click on the required item with the mouse and double-click to select it.
- Use the up and down arrow keys to select the required item and then hit ‘Enter’ to select it.
- Type the first character of the required item (for example ‘m’) to highlight it and then hit ‘Enter’ to select it. If more than 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, go 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

<ul style="list-style-type: none"> Page Layout Dashboard Options Timesheet Options Expenses Options Invoices Options Projects Options <li style="background-color: #ccc;">Scripting Studio Options Display Options 	<h3>Scripting Studio Options</h3> <p><input checked="" type="checkbox"/> Enable script deployment editor content assist auto activation</p> <p>Script deployment editor content assist auto activation delay (milliseconds) <input type="text" value="500"/></p> <p>Editor Theme: <input type="text" value="elegant"/></p> <p>Font Size: <input type="text" value="14"/></p> <p>Indent Unit: <input type="text" value="Spaces"/></p> <p>Tab Size: <input type="text" value="4"/></p>
--	---

Entrance Function

The screenshot shows the Scripting Studio interface. On the left, the 'Scripting Studio' panel is open, showing the configuration for a 'Timesheet' form. The 'Employee' field is set to 'Collins, Marc' and the 'Event' is 'On submit'. The 'Entrance Function' field is set to 'getTimesheets'. On the right, the 'View Log' panel shows the JavaScript code for the 'getTimesheets()' function. A red arrow points from the 'getTimesheets' field in the form configuration to the function name in the code editor.

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, contact OpenAir Customer 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

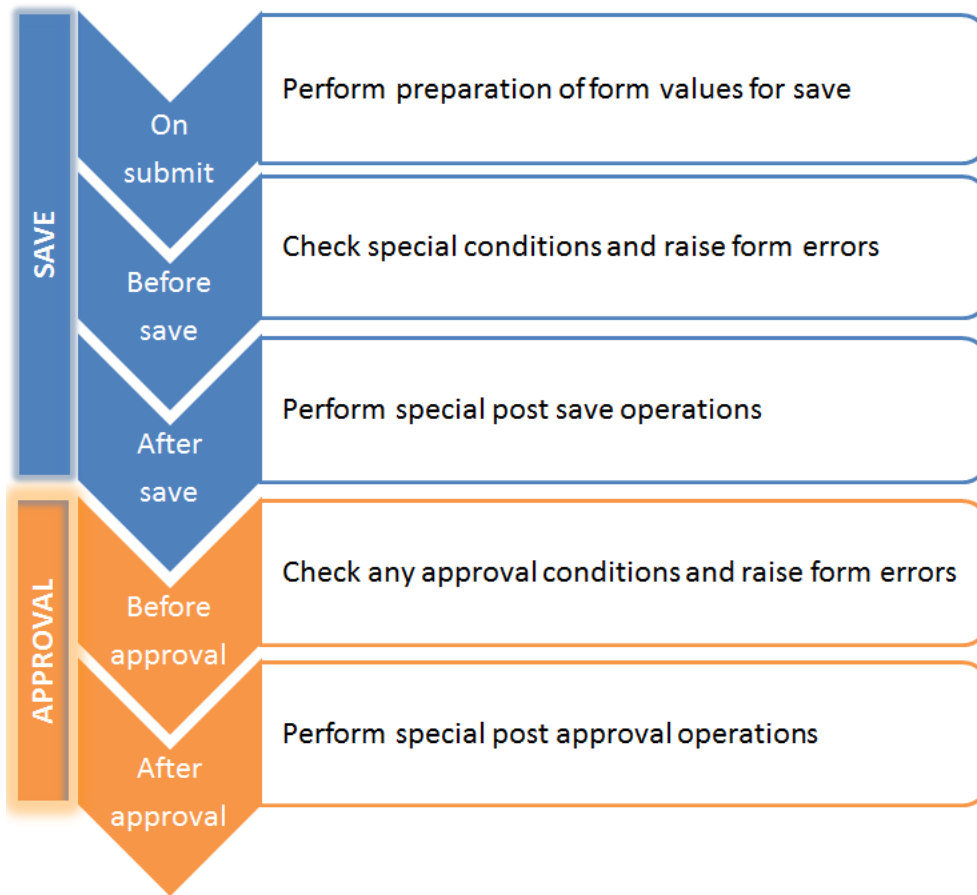
Scripts are triggered by events. The required **Event** is specified in the [Scripting Studio Tools and Settings](#) options.

The screenshot shows a configuration window for an entrance function. It contains the following elements:

- Event:** A dropdown menu with 'Before save' selected. A red box highlights this dropdown, and a red arrow points from it to the 'On submit' dropdown.
- Entrance function:** A dropdown menu with 'checkNotes' selected.
- Code revision comments:** A text area with a vertical scrollbar.
- Comments for this document revision:** A label at the bottom of the text area.

The 'On submit' dropdown menu is also highlighted with a red box and contains the following options: 'On submit', 'Before save', and 'After save'.

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 perform additional actions following a record approval or reject.

Note: The OpenAir mobile app does not support “On submit,” “Before save,” or “After save” scripts associated with the timesheet entity form.

OpenAir Mobile 4.0 or later version supports:

- All form scripts associated with the expense report and receipt entity forms.
- “Before approval” and “After approval” scripts associated with the timesheet entity form.

For an example of script that is executed both in OpenAir and OpenAir Mobile, see

Important: Form scripts may be triggered by an event associated with user interaction — when a user clicks **Save**, for example.

Form scripts can also be triggered by an event associated with a process utilizing the form software logic — when importing project records from NetSuite using OpenAir NetSuite Connector, for example, depending on the integration configuration. For more information, see [Scripting and OpenAir NetSuite Connector](#).

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 subject is trimmed to the first line passed.

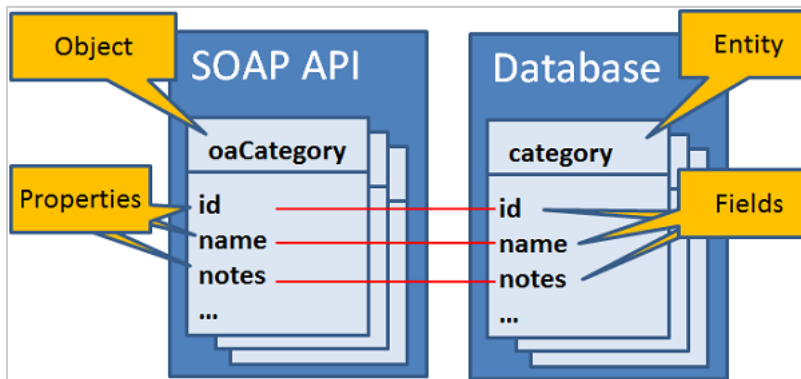


Important: There is a maximum body length set for your emails sent by form and scheduled scripts. Email messages with bodies above that maximum body length are not sent. The maximum body length is set to 30,000 characters by default and can be changed to suit your requirements. To review or to increase the maximum body length set for your account, contact OpenAir Customer Support.

NSOA Function	Units
<ul style="list-style-type: none"> ■ NSOA.record.<complex type>([id]) 	0
<ul style="list-style-type: none"> ■ 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) ■ NSOA.form.get_value(field) ■ NSOA.form.warning(message) ■ NSOA.listview.list() ■ NSOA.report.list() ■ NSOA.wsapi.disableFilterSet([flag]) ■ NSOA.wsapi.enableLog([flag]) ■ NSOA.wsapi.whoami() 	1
<ul style="list-style-type: none"> ■ NSOA.context.parseTerminology(message) ■ NSOA.meta.alert(message) ■ NSOA.meta.log(severity, message) 	4
<div style="border: 1px solid #0070C0; padding: 5px;">  <p>Note: Calls to <code>NSOA.meta.log(severity, message)</code> with the severity parameter set to "debug" or "trace" do not consume units but are limited to a maximum of 1000 per script.</p> </div>	
<ul style="list-style-type: none"> ■ NSOA.context.getAllParameters() ■ NSOA.context.getAllTerms() ■ NSOA.form.getAllValues() ■ NSOA.https.get(request) ■ NSOA.https.post(request) ■ NSOA.https.put(request) ■ NSOA.https.patch(request) ■ NSOA.https.delete(request) ■ NSOA.meta.sendMail(message) ■ NSOA.NSConnector.integrateRecord() 	10
<ul style="list-style-type: none"> ■ NSOA.listview.data(listviewId) ■ NSOA.report.data(reportId,optionalParameters) 	10 for each page loaded into iterator on demand

NSOA Function	Units
	(max 1000 items per page)
<ul style="list-style-type: none"> ■ 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.
<ul style="list-style-type: none"> ■ NSOA.wsapi.modify(attributes, objects) ■ NSOA.wsapi.upsert(attributes,objects) 	40 +20 for each additional object passed.
<ul style="list-style-type: none"> ■ NSOA.NSConnector.integrateAllNow() ■ NSOA.NSConnector.integrateWorkflowGroup(name) 	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, consult the OpenAir API documentation. See [OpenAir XML API & SOAP API Guide](#).



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



Tip: All OpenAir Complex types start “oa”, for example “oaCategory”. You can look up the OpenAir Complex Types and their properties from the following link: <https://app.openair.com/wSDL.pl>.

If you strip away the “oa” you are left with the table name, for example “Issue”. You can look up tables in the OpenAir data dictionary. To access the OpenAir data dictionary, use the link in the navigation bar of the OpenAir Help Center or use the following URL `https://<account-domain>/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 of 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. For the list of supported object types, see the help topic [XML and SOAP API Business Object Reference](#).

- 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 OpenAir. 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 required fields for the objects passed. The ID, updated and created fields are set automatically by OpenAir.

To add data to OpenAir

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

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

2. Fill out the properties for the object, see [Objects](#).

```
1 | category.name = "New Category";
2 | category.cost_centerid = "123";
3 | category.currency = "USD";
```

3. Place the object into an array of objects, see [Arrays](#).

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

4. Pass the objects as a parameter to the [NSOA.wsapi.add\(objects\)](#).

```
1 | 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 only the properties (fields) in the objects that you want to change. The **updated** field is set automatically by OpenAir.

To modify data in OpenAir

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

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

2. Fill out the internal ID for the object and the properties you want to change, see [Objects](#).

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

3. Place the object into an array of objects, see [Arrays](#).

```

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

```

4. Optionally create an array of attributes to pass.

```

1 | var attributes = [];

```

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

```

1 | 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.

```

1 | var category = new NSOA.record.ooCategory();

```

2. Fill out the properties for the object, see [Objects](#).

```

1 | category.id = 79;

```

3. Place the object into an array of objects, see [Arrays](#).

```

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

```

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

```

1 | 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.

```

1 | var user = new NSOA.record.oaUser();
2 | user.nickname = "jsmith";

```

2. Create a **limit** Attribute.

```

1 | var attribute = {
2 |     name : "limit",
3 |     value : "0,1000"
4 | }

```

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

```

1 | var readRequest = {
2 |     type : "User",
3 |     method : "equal to", // return only records that match search criteria
4 |     fields : "id, nickname, updated", // specify fields to be returned
5 |     attributes : [ attribute ], // Limit attribute is required; type is Attribute
6 |     objects : [ user ] // One object with search criteria
7 | }

```

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

```

1 | 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 used to specify the required data to return in the **NSOA.wsapi.read(readRequest)** function.

```

1 | // example read request - assumes attribute and user objects have been defined
2 | var readRequest = {
3 |     type : "User",
4 |     method : "equal to", // return only records that match search criteria
5 |     fields : "id, nickname, updated", // specify fields to be returned
6 |     attributes : [ attribute ], // Limit attribute is required; type is Attribute
7 |     objects : [ user ] // One object with search criteria
8 | }

```

Property	Allowed Values
type	Any OpenAir Complex Type without the oa prefix for example "Issue". See NSOA.record.<complex type>([id]) for the list of types.
method	<ul style="list-style-type: none"> ■ "all" — Returns all available records. <div style="border: 1px solid #00a0e3; padding: 5px; margin: 5px 0;"> <p> Note: Use this cautiously as too many records may be requested for the server or client to handle.</p> </div> <ul style="list-style-type: none"> ■ "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 for example "id, nickname, updated".

Property	Allowed Values
	For more information about supported fields, see the help topic XML and SOAP API Business Object Reference or https://app.openair.com/wsd/pl .
attributes	Array of attribute objects, see Attribute . <div style="border: 1px solid #ccc; background-color: #fff9c4; padding: 5px; margin-top: 5px;">  Important: The "limit" attribute is required. </div>
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.

```

1 |   var attribute = {
2 |     name : "limit",
3 |     value : "10"
4 |   }

```

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

name	value
"limit"	<p>A single value (for example "500") or range (for example "0, 1000").</p> <p>Single value: "1", "500", "1000" - simply restricts the number of records returned.</p> <p>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.</p> <p>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.</p> <p>See Reading data.</p>
"filter"	<ul style="list-style-type: none"> ■ "newer-than" ■ "not-exported" ■ "older-than" <div style="border: 1px solid #0070c0; background-color: #e6f2ff; padding: 5px; margin-top: 5px;">  Note: Options can be placed into a comma separated list, for example "newer-than,older-than,not-exported". </div>
"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 `NSOA.wsapi.read(readRequest)` function returns the **ReadResult** object.

For example, if the fields to be returned were "id, nickname, updated", the objects property for the returned ReadResult object would be:

```

1 // example ReadResult object - assumes the fields to be returned were "id, nickname, updated"
2 [
3   {
4     "errors": null,
5     "objects": [
6       {
7         "id": "26",
8         "nickname": "mcollins",
9         "updated": "2019-10-03 15:42:23",
10        "return_fields": {
11          "id": "1",
12          "nickname": "1",
13          "updated": "1"
14        }
15      },
16      {
17        "id": "33",
18        "nickname": "jadmin",
19        "updated": "2019-09-03 09:12:46",
20        "return_fields": {
21          "id": "1",
22          "nickname": "1",
23          "updated": "1"
24        }
25      }
26    ]
27  }
28 ]

```

Property	Value
objects	Array of Complex Type objects, each with the following properties: <ul style="list-style-type: none"> ■ One property for each field to be returned as listed in the ReadRequest object. Each property has: <ul style="list-style-type: none"> □ key: <i>field_name</i> — name of the field to be returned □ value: <i>field_value</i> — returned field value ■ return_fields — object with the following properties for each field to be returned: <ul style="list-style-type: none"> □ key: <i>field_name</i> — name of the field to be returned □ value: 1
errors	Array of oaError objects.

UpdateResult

The following functions return the **UpdateResult** object.

- [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 <code>oaError</code> objects.
status	<ul style="list-style-type: none"> ■ "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])`, only check that an object was returned.

```
1 | var category = new NSOA.record.oaCategory();
2 | if( !category )
3 |     // 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.

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

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

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

```
1 | // example assuming readRequest has already been defined
2 | var result = NSOA.wsapi.read(readRequest);
3 | // Check for errors
4 | if (!result || !result[0]) {
5 |     // An unexpected error has occurred!
6 | } else if (result[0].errors !== null && result[0].errors.length > 0) {
7 |     // There are errors to handle!
8 | } else {
9 |     // Process the response as expected
10 | }
```

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: For the full list of errors, see the help topic [Error Codes](#).

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.

```

1  function test() {
2      var user = NSOA.wsapi.whoami();
3      NSOA.meta.alert( "User ID " + user.id + " saved this record");
4  }

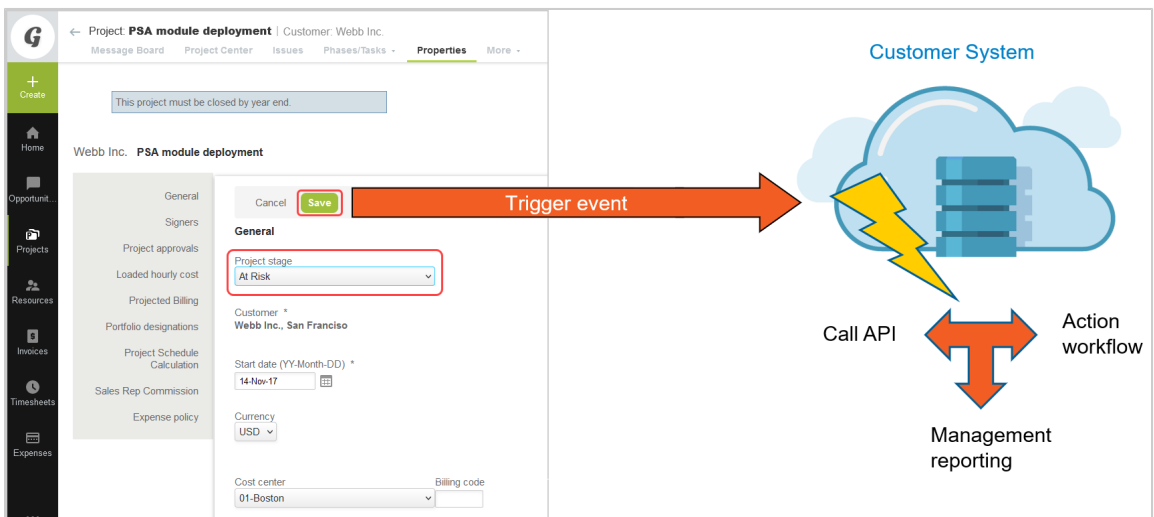
```

oaUser


The `oaUser` object has more than 100 attributes defining user specific information. See [User](#).

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

Outbound Calling



OpenAir user scripting enables calls to external APIs through the NSOA.https functions, see [NSOA Functions](#). Calls to REST, XML and SOAP APIs are supported.

 **Note:** You need the **Enable user script support for https methods** switch enabled to use the NSOA.https functions, see [Scripting Switches](#).

The following request methods are currently supported for form and scheduled scripts:

- GET — [NSOA.https.get\(request\)](#) function
- POST — [NSOA.https.post\(request\)](#) function
- PUT — [NSOA.https.put\(request\)](#) function
- PATCH — [NSOA.https.patch\(request\)](#) function
- DELETE — [NSOA.https.delete\(request\)](#) function

The functions take the [Request](#) object as a single parameter and return the [Response](#) object.

OpenAir user scripting also enables to create [Password Script Parameters](#).

Request

The **request** object is used to set the request parameters.

Property	Type	Required / Optional	Description
url	string	required	The HTTPS URL being requested. Parameters can be passed as part of the URL using a query string.
body	array object string	optional	The data passed to the server. If the data is passed as an array or object, it will be JSON serialized and URL encoded automatically. Nested arrays and objects are supported. If the data is passed as a string, it must be encoded correctly by the user. The GET method does not support this property.
headers	object	optional	The HTTPS headers. The MIME type is set automatically to application/json when body is an array or object.

Response

NSOA.https functions return the **response** object.

Property	Type	Description
body	object string	The response body.
code	string	The HTTP response status code.
headers	object	The response headers.

Limits

The following limits apply to all NSOA.https functions:

- The requested URL must use the HTTPS protocol and the server certificate must be validated.
- The functions will follow redirects up to a maximum of 7.
- If the client doesn't start receiving a response from the server within 45 seconds of the request being fully sent, a connection timeout occurs. If the request times out, a response object is returned with a standard HTTP Status Code (500) and a "Client-Warning" header set.
- The response must not exceed 1MB in size.
- The functions use 10 units per call. See [Scripting Governance](#).

Password Script Parameters

You can create password script parameters to store credentials such as passwords and access tokens as encrypted values and use them in your scripts for secure deployment and maintenance. See [Creating Parameters](#).

The value is hidden both on the form used to set the parameter value and in the parameters list view.

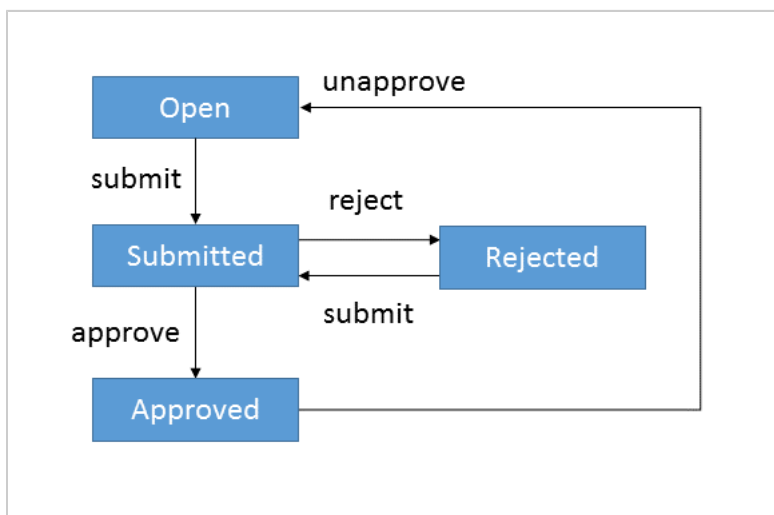
You can use the `NSOA.context.getParameter(name)` function to read the value for the specified password parameter in your outbound calling scripts.

Note: Review the following guidelines:

- Parameters need to be referenced before they can be used in a given script. This is done from the [Scripting Studio](#).
- Password script parameter definitions may be saved as part of a platform solution. See [Creating Solutions](#).

Scripting Approvals

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



Working with the Approvals System

Submitting Booking, Timesheets, Envelopes, and Invoices

Submit open or rejected bookings, 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 oaBooking, oaTimesheet, oaEnvelope, or oaInvoice object to submit and additional approval process information passed to an oaApproval object.

```

1 // Define the submit requests
2 var requests = [{
3   submit: objectToProcess,
4   attributes [], // attributes only apply when submitting timesheets
5   approval: approvalObj
6 }];

```

Property	Allowed Values
submit	oaBooking , oaTimesheet , oaEnvelope , or oaInvoice object
attributes	Only accepts "submit_warning" for oaTimesheet
approval	oaApproval

Approving Bookings, Timesheets, Envelopes, and Invoices

Approve submitted bookings, 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 oaBooking, oaTimesheet, oaEnvelope, or oaInvoice object to approve and additional approval process information passed to an oaApproval object.

```

1 // Define the approve requests
2 var requests = [{
3   approve: objectToProcess,
4   attributes [], // pass an empty array for attributes when using approveRequest
5   approval: approvalObj
6 }];

```

Property	Allowed Values
----------	----------------

approve	oaBooking , oaTimesheet , oaEnvelope , or oaInvoice object
attributes	Pass an empty array
approval	oaApproval

Rejecting Bookings, 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 oaBooking, 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.

```

1 // Define the reject requests
2 var requests = [{
3     reject: objectToProcess,
4     attributes [], // pass an empty array for attributes when using rejectRequest
5     approval: approvalObj
6 }];

```

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

Unapproving Bookings, 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 oaBooking, 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.

unapproveRequest

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

```

1 // Define the unapprove requests
2 var requests = [{
3     unapprove: objectToProcess,
4     attributes [], // pass an empty array for attributes when using unapproveRequest
5     approval: approvalObj
6 }];

```

Property	Allowed Values
----------	----------------

unapprove	oaBooking , 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 following functions return the ApprovalResult object.

- [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 <ul style="list-style-type: none"> ■ "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.

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

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

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

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

```
1 | // example assuming requests have already been defined
2 | var results = NSOA.wsapi.approve(requests);
3 | // Check for errors
4 | if (!result || !result[0]) {
5 |     // An unexpected error has occurred!
6 | } else if (result[0].errors !== null && result[0].errors.length > 0) {
7 |     // There are errors to handle!
8 | } else {
9 |     // Process the response as expected
10 | }
```

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. Check the Active 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 check the Required box, the field is required on the form and the form cannot to be saved without supplying a value.
10. If you check the Unique box, a unique value must be entered in the field to be able to save the form.
11. If you check the box to Hide on data entry forms, this custom field does not display on the form.

12. If you check the box for Add Notes, a text box displays under the custom field for employees to add any additional notes.
13. If you check the 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 box. The divider line indicates the start of the contract management fields.

14. Click Save. After 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 Administrator 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

For: Project, Date field

Field name
 Active
 Required, no spaces allowed

Description

 Description of this custom field

Association
Project
 Select what entity you want to add this field to

Display name

 You must enter a title to display on forms

Hint

 Hint text will display on forms

Default to Current Date
 Check to always default to the current date.

Required
 Check to make this field require data entry on your forms.

Unique
 Check to enforce unique values in this field

Hide on data entry forms
 Check to hide this field on data entry forms

Add notes
 Check to include an associated notes field

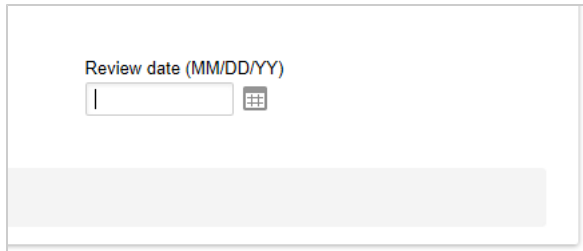
Divider
 Check to include a divider line before this field

Divider text

 Text to include in the divider

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

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

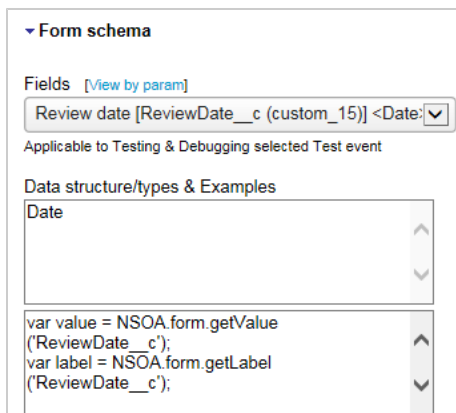


Review date (MM/DD/YY)

| 

Reading Custom Fields

Use the [Form Schema](#) to find the correct field name for the custom field.



▼ Form schema

Fields [\[View by param\]](#)

Review date [ReviewDate__c (custom_15)] <Date:▼

Applicable to Testing & Debugging selected Test event

Data structure/types & Examples

Date

```
var value = NSOA.form.getValue('ReviewDate__c');
var label = NSOA.form.getLabel('ReviewDate__c');
```

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

```
1 // Read the date value and log the value if the date is not empty
2 function logReviewDate(){
3     var reviewDate = NSOA.form.getValue('ReviewDate__c');
4     if( reviewDate !== null ) {
5         NSOA.meta.alert(reviewDate.toString());
6     }
7 }
```

Note: The old approach to reading custom fields using custom_ with the internally assigned custom field number appended is d. Use the custom fields name followed by __c (underscore underscore c) instead.

```
1 // Supported but NOT RECOMMENDED
2 var reviewDate = NSOA.form.getValue('custom_15');
3 }
```

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.

```
1 | 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.

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

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

```
1 | 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.

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

2. Use the correct name format for the custom field, that is **Field name** defined for the custom field + `'__c'`.

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

3. The `'update_custom'` **Attribute** must be specified.

```
1 | var attribute = {
2 |   name : 'update_custom',
3 |   value : "1"
4 | }
```

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

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

5. Check for any errors, see [Handling SOAP Errors](#).
6. Process the results, see [UpdateResult](#).

NSOA Functions

context + getAllParameters() : Array + getAllTerms() : Array + getParameter(name) : var + getTerm(terminid) : String + isTestMode() : Boolean + parseTerminology(message) : String + remainingTime() : Integer + remainingUnits() : Integer	meta + alert(message) : Boolean + log(severity, message) : Boolean + sendMail(message) : Boolean
form + confirmation(message) : Boolean + error(field, message) : Boolean + getAllValues() : Array + getLabel(field) : String + getName(field) : String + getNewRecord() : oaBase + getOldRecord() : oaBase + getValue(field) : var + get_value(field) : String + warning(message) : Boolean + setValue(field, value) : String	NSConnector + integrateAllNow() : Boolean + integrateRecord() : Boolean + integrateWorkflowGroup(name) : Boolean
https + delete(request) : Object + get(request) : Object + patch(request) : Object + post(request) : Object + put(request) : Object	record + < complexType > ([id]) > : oaBase
listview + data(listviewId) : Iterator + list() : Array	report + data(reportId) : Iterator + list() : Array
	wsapi + add(objects) : Array + approve(approveRequests) : Array + delete(objects) : Array + disableFilterSet([flag]) : Boolean + enableLog([flag]) : Boolean + modify(attributes, objects) : Array + read(readRequest) : Array + reject(rejectRequests) : Array + remainingTime() : Integer + submit(submitRequests) : Array + upsert(attributes, objects) : Array + unapprove(unapproveRequests) : Array + whoami() : oaUser

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

- NSOA.context
 - NSOA.context.getAllParameters()
 - NSOA.context.getAllTerms()
 - NSOA.context.getLanguage()
 - NSOA.context.getParameter(name)
 - NSOA.context.getTerm(terminid)
 - 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.https
 - NSOA.https.delete(request)
 - NSOA.https.get(request)
 - NSOA.https.patch(request)
 - NSOA.https.post(request)
 - NSOA.https.put(request)
- NSOA.listview
 - NSOA.listview.data(listviewId)
 - NSOA.listview.list()
- NSOA.meta
 - NSOA.meta.alert(message)
 - NSOA.meta.log(severity, message)
 - NSOA.meta.sendMail(message)
- NSOA.NSConnector
 - NSOA.NSConnector.integrateAllNow()
 - NSOA.NSConnector.integrateRecord()
 - NSOA.NSConnector.integrateWorkflowGroup(name)
- NSOA.record
 - NSOA.record.<complex type>([id])
- NSOA.report
 - NSOA.report.data(reportId,optionalParameters)
 - NSOA.report.list()
- NSOA.wsapi
 - NSOA.wsapi.add(objects)
 - NSOA.wsapi.approve(approveRequest)
 - NSOA.wsapi.delete(objects)
 - NSOA.wsapi.disableFilterSet([flag])
 - NSOA.wsapi.enableLog([flag])
 - NSOA.wsapi.modify(attributes, objects)

- `NSOA.wsapi.read(readRequest)`
- `NSOA.wsapi.reject(rejectRequest)`
- `NSOA.wsapi.remainingTime()`
- `NSOA.wsapi.submit(submitRequest)`
- `NSOA.wsapi.unapprove(unapproveRequest)`
- `NSOA.wsapi.upsert(attributes,objects)`
- `NSOA.wsapi.whoami()`

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.

```

1 | // Get all the parameters available for the script
2 | var allParams = NSOA.context.getAllParameters();
3 |
4 | // Loop through all the parameters
5 | for (var key in allParams) {
6 |     NSOA.meta.alert(key + ' has value ' + allParams[key]);
7 | }

```

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.

```

1 // Get all the terminology available for the script
2 var allTerms = NSOA.context.getAllTerms();
3
4 // Loop through all the terminology
5 for (var key in allTerms) {
6     NSOA.meta.alert(key + ' has value ' + allTerms[key]);
7 }

```

See [NSOA.context.parseTerminology\(message\)](#) and [NSOA.context.getTerm\(termid\)](#).

See also [Accessing Terminology](#).

NSOA.context.getLanguage()


Use this function to get the user's display language preference. You can then adapt your form scripts according to the user's language preference and show translated versions of the same message, for example.

Parameters

- (none)

Returns

- A two-character string — the ISO 639–1 two-letter code for the language selected in the authenticated user's personal settings, or xx, if the **Show language keys** view option is in use.

 **Note:** If the Multilanguage feature is not enabled, the function returns en.

Language	Two-letter code
Chinese (Simplified)	zh
Czech	cs
English	en
French	fr
German	de
Japanese	ja
Spanish	es

Units Limit

- 0 units

For more information, see [Scripting Governance](#).

Since

- April 9, 2022

Example

- This example creates a local variable called **lang** with the user's language preference. It then uses conditional branching to display a message in the user's preferred language.

```

1 // return user language code : en | fr | de | es | zh | ja | cs
2 var lang = NSOA.context.getLanguage();
3
4 if (lang == 'cs') {
5     // display messages in the Czech language"
6 }
7 if (lang == 'en') {
8     // display messages in the English language"
9 }

```

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.

```
1 // return if new stage is not closed
2 if (NSOA.form.getValue('project_stage_id') !=
3     NSOA.context.getParameter('ProjectClosedStage'))
4     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.

```
1 | var proj_term = NSOA.context.getTerm('Projects');
2 | // 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 and 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”, for example an assertion.

```
1 | if(NSOA.context.isTestMode() && someVar==null)
2 |   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.

```
1 | var msg = NSOA.context.parseTerminology("Notes attached to %project%.")
2 | // 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).

```
1 | NSOA.meta.log('info', 'Remaining script time: '
2 |     + 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 OpenAir. 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.


```
1 | 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

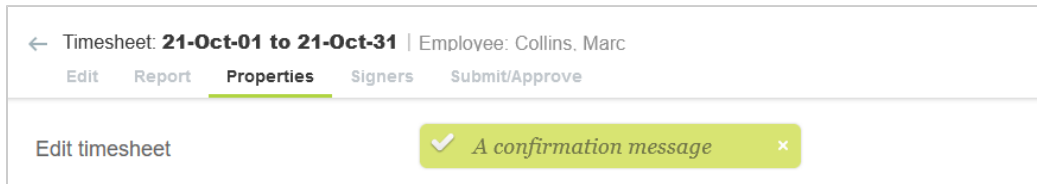
- October 17, 2015

Example

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

```
1 | 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: This function has no effect 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.

```
1 | 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.

```
1 | 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 (for example 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 (for example no related table lookups are available now). 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.

```
1  var allValues = NSOA.form.getAllValues();
2
3  var project_name = allValues.name;    // equivalent to getValue('name');
4  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.

```
1  // Get all the values on the fields on the form
2  var allValues = NSOA.form.getAllValues();
3
4  //Loop through all the values
5  for( var key in allValues ) {
6      NSOA.meta.alert(key + ' has value ' + allValues[key]);
7  }
```

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.

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


- This example gets the label for a field that returns an object. See [Object Fields](#) for more details.

```
1 | // 'Primary loaded cost ' for the first row
2 | 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, that is, 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, that is, **budget_time**.

```
1 | 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.

```
1 | // 'Primary loaded cost ' for the first row
2 | 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, for example 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

```

1 // Get the new record values
2 var newr = NSOA.form.getNewRecord();
3
4 // Create a new record with field to modify
5 var project = new NSOA.record.oaProject();
6 project.id = newr.id; // Need to specify the internal ID
7 project.notes = newr.notes + "\nAppended to notes: " + (new Date().toString()); // New value for field
8
9 // Modify the notes
10 NSOA.wsapi.disableFilterSet(true); // Drop user filters - make this a generic script
11 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, for example 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.

```

1 | var oldr = NSOA.form.getOldRecord();
2 | var newr = NSOA.form.getNewRecord();
3 | if (oldr.name != newr.name)
4 |     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 (for example 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 (for example no related table lookups are available now). 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.

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

- This sample gets a value from a field that returns an object. See [Object Fields](#) for more details.

```
1 | // First get the object variable for the field and then get the cost_0 value for the first row
2 | var loaded_cost_obj = NSOA.form.getValue("loaded_cost");
3 | var value = loaded_cost_obj[0].cost_0;
4 |
5 | // You can combine these two steps into one line
6 | 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 (for example 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 (for example no related table lookups are available now).

 **Note:** You should 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.

```
1 | 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.

The screenshot shows a form with the following fields:

- Project name ***: Payroll integration
- Client ***: Damus Inc.
- Project owner**: Horton, Dave
- Project stage**: Active
- Budget (hours)**: 4000
- Start date (MM/DD/YY) ***: 01/23/17

A blue arrow points from the 'Start date' field to a code editor window titled 'View Log (3)' containing the following code:

```
1 function main(type) {
2   NSOA.form.setValue('start_date', '2017-01-23');
3
4 }
5
```

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

The `NSOA.form.setValue` function supports the following field types: text, text area, date, numeric, currency, days, hours, ratio, checkbox, dropdown, dropdown and text, pick list and radio group. The following field types are not supported: password, sequence and multiple selection.

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.

Note: You can reference custom fields using either to the user-defined custom field name suffixed with `_c` (for example `my_custom_radio_group_c` or the internal custom field ID prefixed by `custom_` and the (for example, `custom_42`). You should reference custom fields by their user-defined names to ensure your scripts are portable.

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

Text field

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

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

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

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

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

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

```
1 | 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.

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

Numeric field

- Numeric 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.

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

- Use a null value to clear a numeric field using setValue.

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

■ Date field

- 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.

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

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

- Use a null value to clear a date field:

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

■ Checkbox field

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

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

- Use **false** to clear a checkbox:

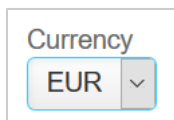
```
1 | NSOA.form.setValue("active", false);
```

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

■ Dropdown / Dropdown and text field

- To set a dropdown (or dropdown and text field) field value, use one of the values from the dropdown list, as it is displayed.

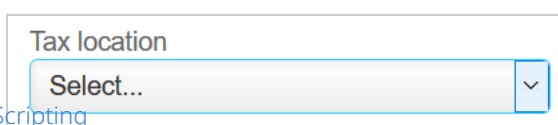
```
1 | NSOA.form.setValue("currency", "EUR");
```



A screenshot of a dropdown menu. The label "Currency" is at the top. Below it, a box contains the text "EUR" and a downward-pointing arrow icon.

- Set the value to "Select ..." to clear any selected value.

```
1 | NSOA.form.setValue("tax_location_id", "Select...");
```




A screenshot of a dropdown menu. The label "Tax location" is at the top. Below it, a box contains the text "Select..." and a downward-pointing arrow icon.

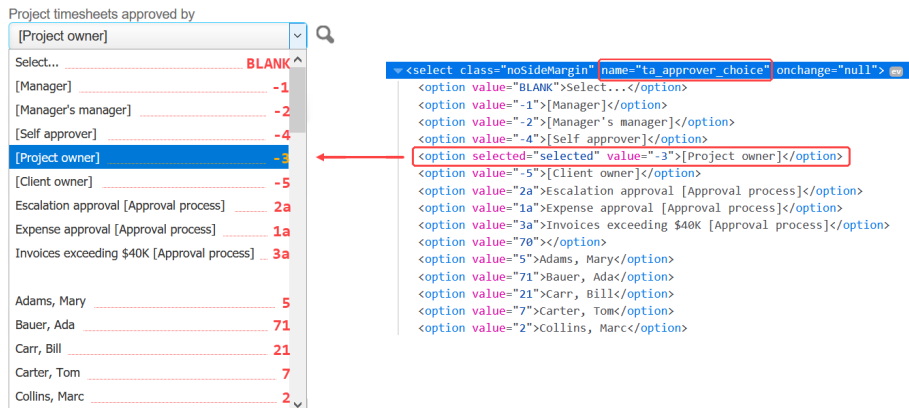
- When using `NSOA.form.setValue` to set a value for a dropdown field, an error is logged if the value parameter is not one of the available dropdown options.

■ Pick list field

- Set the pick list field value by its internal identifier. Meta values are also supported — in the example below, the internal identifier for [Project owner] is `-3`.

 **Tip:** To find the internal identifier for meta values, you can use developer tools on your browser. Point to the pick list, right-click and select **Inspect element** to display the HTML for the pick list element. You can retrieve the name of the field from the name attribute of the `<select>` element and the internal identifier from the value attribute of the `<option>` element.

```
1 | NSOA.form.setValue("ta_approver_choice", "-3");
```



Project timesheets approved by

[Project owner]	BLANK
[Manager]	-1
[Manager's manager]	-2
[Self approver]	-4
[Project owner]	-3
[Client owner]	-5
Escalation approval [Approval process]	2a
Expense approval [Approval process]	1a
Invoices exceeding \$40K [Approval process]	3a
Adams, Mary	5
Bauer, Ada	71
Carr, Bill	21
Carter, Tom	7
Collins, Marc	2

```
<select class="noSideMargin" name="ta_approver_choice" onchange="null">
  <option value="BLANK">Select...</option>
  <option value="-1">[Manager]</option>
  <option value="-2">[Manager's manager]</option>
  <option value="-4">[Self approver]</option>
  <option selected="selected" value="-3">[Project owner]</option>
  <option value="-5">[Client owner]</option>
  <option value="2a">Escalation approval [Approval process]</option>
  <option value="1a">Expense approval [Approval process]</option>
  <option value="3a">Invoices exceeding $40K [Approval process]</option>
  <option value="70"></option>
  <option value="5">Adams, Mary</option>
  <option value="71">Bauer, Ada</option>
  <option value="21">Carr, Bill</option>
  <option value="7">Carter, Tom</option>
  <option value="2">Collins, Marc</option>
</select>
```


- When using `NSOA.form.setValue` to set a value for a dropdown field, an error is logged if the value parameter is not one of the dropdown options.
- Set the value to `"BLANK"` to clear any selected value.

```
1 | NSOA.form.setValue("ta_approver_choice", "BLANK");
```

- An error is logged if the value parameter is not the internal identifier for one of the available pick list options.

■ Radio group field

- Set a radio group field value by the radio button name.

 **Note:** You can reference custom fields using the user-defined name — for example `prj_radio_group_c` — or the internal reference — for example `custom_42`. Always reference custom fields by their the user-defined names to create portable scripts.

```
1 | NSOA.form.setValue("prj_radio_group_c", "three");
```

or

```
1 | NSOA.form.setValue("custom_42", "three");
```



- Set the value to an empty string "" or to null to clear any selected value.

```
1 | NSOA.form.setValue("prj_radio_group_c", "");
```

or

```
1 | NSOA.form.setValue("prj_radio_group_c", null);
```

- An error is logged if the value parameter is invalid. Radio button values, as displayed on the screen, null and an empty string "" are the only valid values.

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

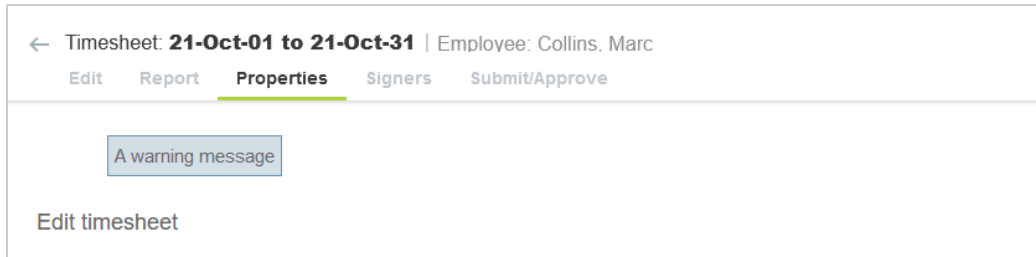
- October 17, 2015

Example

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

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

The message appears as an OpenAir system-generated warning.



See [Code Samples](#) for more examples.

NSOA.https.delete(request)

Use this function to send an HTTPS DELETE request to delete resources on a server. In general, DELETE requests support both query string parameters and a request body. The exact use of DELETE requests and what data is returned depends on the implementation of the server. The function will return an error if the URL requested does not use the HTTPS protocol. The function will follow redirects up to a maximum of 7. The response must not exceed 1MB in size.

Note: If the client doesn't start receiving a response from the server within 45 seconds of the request being fully sent, a connection timeout occurs. If the request times out, a response object is returned with a standard HTTP Status Code (500) and a "Client-Warning" header set.

Parameters

- *request* {object} [required] — The request object is used to set the DELETE request parameters

Property	Type	Required / Optional	Description
url	string	required	The HTTPS URL being requested.
body	array object string	optional	The DELETE data. If the data is passed as an array or object, it will be automatically JSON serialized and URL encoded.
headers	object	optional	The HTTPS headers.

Returns

- *response* {object} [read-only] — The NSOA.https.delete function returns the **response** object.

Property	Type	Description
body	string object	The response body
code	string	The HTTP response status code.
headers	object	The response headers.

Units Limit

- 10 units

For more information, see [Scripting Governance](#).

Since

- October 12, 2019

Example

- This example sends an HTTPS DELETE request, converts the response to a JSON string, displays it in a confirmation message and stores it as a log entry.

```

1 function main(type) {
2
3     var response = NSOA.https.delete({
4         url: 'https://postman-echo.com/delete',
5         body: 'This is expected to be sent back as part of response body.'
6     });
7
8     NSOA.meta.alert(JSON.stringify(response));
9     NSOA.form.confirmation(JSON.stringify(response));
10
11 }
```

See [Code Samples](#) for more examples.

NSOA.https.get(request)

Use this function to send an HTTPS GET request to retrieve data from a server. The data is identified by a unique URL and parameters can be passed to the server using query string parameters. What data is returned depends on the implementation of the server. The function will return an error if the URL requested does not use the HTTPS protocol. The function will follow redirects up to a maximum of 7. The response must not exceed 1MB in size.

Note: If the client doesn't start receiving a response from the server within 45 seconds of the request being fully sent, a connection timeout occurs. If the request times out, a response object is returned with a standard HTTP Status Code (500) and a "Client-Warning" header set.

Parameters

- request* {object} [required] — The request object is used to set the GET request parameters

Property	Type	Required / Optional	Description
url	string	required	The HTTPS URL being requested.
headers	object	optional	The HTTPS headers.

Returns

- response* {object} [read-only] — The NSOA.https.get function returns the **response** object.

Property	Type	Description
body	string object	The GET data.
code	string	The HTTP response status code.
headers	object	The response headers.

Units Limit

- 10 units

For more information, see [Scripting Governance](#).

Since

- April 13, 2019

Example

- This example sends an HTTPS GET request, converts the response to a JSON string, displays it in a confirmation message and stores it as a log entry.

```

1 function main(type) {
2
3     var response = NSOA.https.get({
4         url: 'https://postman-echo.com/get?foo1=bar1&foo2=bar2'
5     });
6
7     NSOA.meta.alert(JSON.stringify(response));
8     NSOA.form.confirmation(JSON.stringify(response));
9
10 }
```

- This example sends an HTTPS GET request to an endpoint simulating a basic-auth protected endpoint. It converts the response to a JSON string, stores it as a log entry and displays a confirmation message if the authentication is succesful.

```

1 function main(type) {
2
3     var response = NSOA.https.get({
4         url: 'https://postman-echo.com/basic-auth',
5         headers: {'Authorization': 'Basic cG9zdG1hbjpwYXNzd29yZA=='}
6     });
7     NSOA.meta.alert(JSON.stringify(response));
8     if (response.body.authenticated) {
9         NSOA.form.confirmation('Authentication successful');
10    }
11
12 }
```

See [Code Samples](#) for more examples.

NSOA.https.patch(request)

Use this function to send an HTTPS PATCH request to update or to make partial modifications to resources on a server. PATCH requests may support both query string parameters and a request body.

The exact use of PATCH requests and what data is returned depends on the implementation of the server. The function will return an error if the URL requested does not use the HTTPS protocol. The function will follow redirects up to a maximum of 7. The response must not exceed 1MB in size.

Note: If the client doesn't start receiving a response from the server within 45 seconds of the request being fully sent, a connection timeout occurs. If the request times out, a response object is returned with a standard HTTP Status Code (500) and a "Client-Warning" header set.

Parameters

- *request* {object} [required] — The **request** object is used to set the PATCH request parameters

Property	Type	Required / Optional	Description
url	string	required	The HTTPS URL being requested.
body	array object string	optional	The PATCH data. If the data is passed as an array or object, it will be automatically JSON serialized and URL encoded.
headers	object	optional	The HTTPS headers. The MIME type is set automatically to application/json when body is an array or object.

Returns

- *response* {object} [read-only] — The NSOA.https.patch function returns the **response** object.

Property	Type	Description
body	object string	The response body.
code	string	The HTTP response status code.
headers	object	The response headers.

Units Limit

- 10 units

For more information, see [Scripting Governance](#).

Since

- October 12, 2019

Example

- This example sends form data to an endpoint using the HTTPS PATCH method, converts the response to a JSON string, displays it in a confirmation message and stores it as a log entry.


```

1 function main(type) {
2
3     var response = NSOA.https.patch({
4         url: 'https://postman-echo.com/patch',
5         body: 'This is expected to be sent back as part of response body.'
6     });
7
8
9     NSOA.meta.alert(JSON.stringify(response));
10    NSOA.form.confirmation(JSON.stringify(response));
11
12 }

```

See [Code Samples](#) for more examples.

NSOA.https.post(request)

Use this function to send an HTTPS POST request to transfer data to a server (and elicit a response). Parameters can be passed to the server using query string parameters, as well as the request body. What data is returned depends on the implementation of the server. The function will return an error if the URL requested does not use the HTTPS protocol. The function will follow redirects up to a maximum of 7. The response must not exceed 1MB in size.

Note: If the client doesn't start receiving a response from the server within 45 seconds of the request being fully sent, a connection timeout occurs. If the request times out, a response object is returned with a standard HTTP Status Code (500) and a "Client-Warning" header set.

Parameters

- *request* {object} [required] — The **request** object is used to set the POST request parameters

Property	Type	Required / Optional	Description
url	string	required	The HTTPS URL being requested.
body	array object string	optional	The POST data. If the data is passed as an array or object, it will be automatically JSON serialized and URL encoded.
headers	object	optional	The HTTPS headers. The MIME type is set automatically to application/json when body is an array or object.

Returns

- *response* {object} [read-only] — The NSOA.https.post function returns the **response** object.

Property	Type	Description
body	object string	The response body.
code	string	The HTTP response status code.
headers	object	The response headers.

Units Limit

- 10 units

For more information, see [Scripting Governance](#).

Since

- April 13, 2019

Example

- This example sends form data to an endpoint using the HTTPS POST method, converts the response to a JSON string, displays it in a confirmation message and stores it as a log entry.

```

1 function main(type) {
2
3     var response = NSOA.https.post({
4         url: 'https://postman-echo.com/post',
5         body: 'foo1=bar1&foo2=bar2' ,
6         headers: {'Content-Type':'application/x-www-form-urlencoded'}
7     });
8
9     NSOA.meta.alert(JSON.stringify(response));
10    NSOA.form.confirmation(JSON.stringify(response));
11
12 }

```

See [Code Samples](#) for more examples.

NSOA.https.put(request)

Use this function to send an HTTPS PUT request to update or replace data on a server (and elicit a response). Parameters can be passed to the server using query string parameters, as well as the request body. What data is returned depends on the implementation of the server. The function will return an error if the URL requested does not use the HTTPS protocol. The function will follow redirects up to a maximum of 7. The response must not exceed 1MB in size.

Note: If the client doesn't start receiving a response from the server within 45 seconds of the request being fully sent, a connection timeout occurs. If the request times out, a response object is returned with a standard HTTP Status Code (500) and a "Client-Warning" header set.

Parameters

- request* {object} [required] — The **request** object is used to set the PUT request parameters

Property	Type	Required / Optional	Description
url	string	required	The HTTPS URL being requested.
body	array object string	optional	The PUT data. If the data is passed as an array or object, it will be automatically JSON serialized and URL encoded.
headers	object	optional	The HTTPS headers. The MIME type is set automatically to application/json when body is an array or object.

Returns

- `response {object}` [read-only] — The `NSOA.https.put` function returns the **response** object.

Property	Type	Description
body	object string	The response body.
code	string	The HTTP response status code.
headers	object	The response headers.

Units Limit

- 10 units

For more information, see [Scripting Governance](#).

Since

- October 12, 2019

Example

- This example sends form data to an endpoint using the HTTPS PUT method, converts the response to a JSON string, displays it in a confirmation message and stores it as a log entry.

```


1 function main(type) {
2
3     var response = NSOA.https.put({
4         url: 'https://postman-echo.com/put',
5         body: 'foo1=bar1&foo2=bar2' ,
6         headers: {'Content-Type': 'application/x-www-form-urlencoded'}
7     });
8
9     NSOA.meta.alert(JSON.stringify(response));
10    NSOA.form.confirmation(JSON.stringify(response));
11
12 }
```


See [Code Samples](#) for more examples.


NSOA.listview.data(listviewId)


Use this function to read the published list view data available to the user running the script. The function returns a specialized list view data iterator (length, index, next, each).

- The data read by your scripts is the same as the data you can see in your list view at any given time.
- Accessing published list views using the `NSOA.listview.data(listviewId)` and `NSOA.listview.list()` user scripting functions does not use any of your allocated OData requests.
- For more information, see [Business Intelligence Connector](#). See also [OData Explorer](#) to browse available OData resources and get started with a sample code, and [NSOA.report.data\(reportId,optionalParameters\)](#) to read published report data.

 **Tip:** Use published list views like custom queries and read only the necessary list view data in your OpenAir scripts.

 **Important:** Both form and scheduled scripts support the `NSOA.listview.data(listviewId)` function. However, the number of items you can process in form scripts is restricted by the scripting governance time limits. The function is best suited for reading published list view data in scheduled scripts, which allow up to 1 hour of JS runtime and 1 hour of web services API call time. For more information about scripting limits, see [Scripting Governance](#).

 **Note:** The OpenAir Business Intelligence Connector must be enabled for your account to use `NSOA.listview` and `NSOA.report` functions. OpenAir Business Intelligence Connector is a licensed add-on. To enable this feature, contact your OpenAir account manager.

For more information about publishing list views and reports to the OpenAir OData service, see the  [OpenAir Business Intelligence Connector Guide](#).

Parameters

- `listviewId` — the published list view OData resource ID (integer).

Returns

- A specialized list view data iterator (length, index, next, each).
 - `length` — number of items
 - `index` — index of last returned item
 - `next` — returns either the next item from the iterator or “undefined” when the iterator is done
 - `each` — calls the specified function for each item in the iterator

Units Limit

- 10 units for each 1000-item page loaded into iterator on-demand. Consumes 10 units for the first fetch even when the page is empty.

Since

- April 18, 2020

Example

```

1 // get the iterator for listview data, it has following members
2 // it consumes 10 units for each 1000-item page loaded into iterator on-demand
3 // * 'length' - number of items
4 // * 'index' - index of last returned item
5 // * 'next' - returns next item from iterator or undefined when iterator is done
6 // * 'each' - calls specified function for each item in the iterator
7 var iterator = NSOA.listview.data(7);
8
9 // get number of listview records
10 var row_count = iterator.length;
11
12 // grab first two records
13 var first = iterator.next();
14 var second = iterator.next();
15

```

```


16 // process rest of the listview
17 iterator.each(function(record, index) {
18
19     // search for particular name
20     if (record.Name === "Nathan Brown") {
21
22         // set the field value
23         NSOA.fozm.setValue("remaining_budget_c", record["Remaining Budget"]);
24
25         // stop iterating
26         return false;
27     }
28 });


```

NSOA.listview.list()

Use this function to read the list of published list views available from the OpenAir OData service. Returns the list of published list views.

- Accessing published list views using the [NSOA.listview.data\(listviewId\)](#) and `NSOA.listview.list()` user scripting functions does not use any of your allocated OData requests.
- For more information, see [Business Intelligence Connector](#). See also [NSOA.report.list\(\)](#) to read the list of published reports.

 **Note:** The OpenAir Business Intelligence Connector must be enabled for your account to use `NSOA.listview` and `NSOA.report` functions. OpenAir Business Intelligence Connector is a licensed add-on. To enable this feature, contact your OpenAir account manager.


For more information about publishing list views and reports to the OpenAir OData service, see the  [OpenAir Business Intelligence Connector Guide](#).

Parameters

N/A

Returns

- The list of published list views. Each item in the list has the following properties:
 - ID — The published list view OData resource ID
 - Name — The name of the list view
 - Last published — The date and time when the list view was last published

 **Important:** The Last published property will be `Last_published` with an underscore instead of the space if the optional feature **Replace Non-Alphanumeric Characters with Underscores in Column Titles and Metadata** is enabled for your account. Accommodate both possibilities in your scripts to ensure your scripts continue work whether the feature is enabled or not.

Units Limit

- 1 unit

Since

- April 18, 2020

Example

```

1 // get the list of published listviews
2 var listviews = NSOA.listview.list();
3
4 // each item in the list has following properties
5 // * 'ID'
6 // * 'Name'
7 // * 'Last_published'
8
9 // loop through all published listviews and find given listview ID
10 var i;
11 var listviewId;
12 for (i = 0; i < listviews.length; i++) {
13     if (listviews[i].Name === 'My Approved Bookings') {
14         listviewId = listviews[i].ID;
15         break;
16     }
17 }
18
19 // if listview ID was found get its data
20 if (listviewId > 0) {
21     var rows = NSOA.listview.data(listviewId);
22
23     // process all listview rows
24 }

```

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.

```

1 | 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.

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				

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 by OpenAir
- **Message** — The full message text.
- **User** — For example, the user that was saving the form when the error occurred.

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

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 and 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.

```
1 | 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] string holding the email body.

 **Important:** There is a maximum body length set for your emails sent by form and scheduled scripts. Email messages with bodies above that maximum body length are not sent.

The maximum body length is set to 30,000 characters by default and can be changed to suit your requirements. To review or to increase the maximum body length set for your account, contact OpenAir Customer Support.

 **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.

```

1 // Send a plain text message
2 var msg = {
3     to: ["mcollins@openair.com"],
4     cc: ["jadmin@openair.com"],
5     subject: "Script alert",
6     body: "Form saved"
7 };
8
9 NSOA.meta.sendMail(msg);

```

- This sends an HTML email.


```

1 // Send an HTML message
2 var msg = {
3     to: ["mcollins@openair.com"],
4     subject: "Project Assignment",
5     format: "HTML",
6     body: "<b>Client:</b> Altima Technologies</br>" +
7         "<b>Project:</b> CRM Implementation</br>" +
8         "<b>Project Manager:</b> Collins, Marc"
9 };
10
11 NSOA.meta.sendMail(msg);

```

See [Code Samples](#) for more examples.

NSOA.NSConnector.integrateAllNow()

This function lets you trigger the integration to run for all active integration workflows from your scheduled scripts. It is equivalent to clicking the **Run** button on the OpenAir NetSuite Connector screen and selecting all workflows. For more information about running the integration, see  [OpenAir NetSuite Connector Guide](#).



Important: This function:

- Must be called from a scheduled script.
- Cannot be called only once in the same scheduled script.
- Cannot be called if the `NSOA.NSConnector.integrateWorkflowGroup(name)` is called in the same scheduled script. See [NSOA.NSConnector.integrateWorkflowGroup\(name\)](#)

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.

```


1 function main() {
2     var records = NSOA.wsapi.read(...);
3
4     // check if result is OK
5     if (!records || !records[0])
6         return;
7
8     // trigger NetSuite integration if there is no error and more than 50 records
9     else if (records[0].errors === null && records[0].objects && records[0].objects.length > 50) {
10         NSOA.NSConnector.integrateAllNow();
11     }
12 }

```

 **Note:** This simple example does not show error checking, see [Handling SOAP Errors](#).

See [Code Samples](#) for more examples.

NSOA.NSConnector.integrateRecord()

This function let you export a single OpenAir record to NetSuite from your form scripts. It is equivalent to clicking the **Export/Send** links in the Tips menu for a selected record. For more information about exporting a single record from OpenAir to NetSuite, see  [OpenAir NetSuite Connector Guide](#).

This function applies only to the OpenAir records available for export to NetSuite. It will perform an action only if called for one of the following forms:

- Envelope
- Invoice
- Revenue Recognition Transaction
- Customer
- Timesheet
- Purchase Request

- Project
- Project Task



Important: This function:

- Must be called from a form script.
- Allows a maximum of one call per script.
- Performs no action and returns **false** if called for any form other than those listed above.

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.

```

1 | function main() {
2 |     // integrate current form object to NetSuite
3 |     NSOA.NSConnector.integrateRecord();

```

See [Code Samples](#) for more examples.

NSOA.NSConnector.integrateWorkflowGroup(name)

OpenAir NetSuite Connector lets you create workflow groups to include only the integration workflows you need in each scheduled integration run. For more information about workflow groups, see

 [OpenAir NetSuite Connector Guide](#).

This function lets you trigger the integration to run for a specific workflow group from your user scripts.



Important: This function:

- Must be called from a scheduled script.
- Cannot be called twice for the same workflow group in the same scheduled script.
- Cannot be called if the `NSOA.NSConnector.integrateAllNow()` is called in the same scheduled script. See [NSOA.NSConnector.integrateAllNow\(\)](#)

Parameters

- `name` (string) — The name of the workflow group. It must match exactly the name of an existing workflow group.

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

- October 9, 2021

Example

- This example triggers the NetSuite integration for the workflow group 'Timesheets Custom Export' using a scheduled script.

```
1 function main() {
2     NSOA.NSConnector.integrateWorkflowGroup('Timesheets Custom Export');
3
4 }
```



Note: This simple example does not show error checking, see [Handling SOAP Errors](#).

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 <https://app.openair.com/wSDL.pl>.

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 about the SOAP API (Web Services), see [OpenAir XML API & SOAP API 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.

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

- This sample creates a new category in OpenAir.

```
1 // Create a new category object
2 var category = new NSOA.record.oaCategory(); // empty category
3 category.name = "New Category";
4 category.cost_centerid = "123";
5 category.currency = "USD";
6
7 // Invoke the add call
8 var results = NSOA.wsapi.add( [category] );
```

See also [NSOA.wsapi.add\(objects\)](#).

See [Code Samples](#) for more examples.

NSOA.report.data(reportId,optionalParameters)

Use this function to read published report data available to the user executing the script. The function returns a specialized report data iterator (length, index, next, each).

For more information, see [Business Intelligence Connector](#). See also [OData Explorer](#) to browse available OData resources and get started with a sample code, and [NSOA.listview.data\(listviewId\)](#) to read published list view data.

Note: The OpenAir Business Intelligence Connector must be enabled for your account to use NSOA.listview and NSOA.report functions. OpenAir Business Intelligence Connector is a licensed add-on. To enable this feature, contact your OpenAir account manager.

For more information about publishing list views and reports to the OpenAir OData service, see the [OpenAir Business Intelligence Connector Guide](#).

Parameters

- reportId — the ID number of the report (integer).
- (optional) optionalParameters — an object with the following properties:
 - (optional) select — the list of columns to return (string array).
 - (optional) filter — a logical expression defining the criteria items must match to be included in the response (string).

For information about column names and filter expression syntax and guidelines, see [OpenAir Business Intelligence Connector Guide](#).

Returns

- A specialized report data iterator (length, index, next, each).
 - length — number of items
 - index — index of last returned item
 - next — returns next item from iterator or undefined when iterator is done
 - each — calls specified function for each item in the iterator

Units Limit

- 10 units for each 1000-item page loaded into iterator on-demand. Consumes 10 units for the first fetch even when the page is empty.

Since

- October 13, 2018

Example

```

1 // get the iterator for report data; it has following members
2 // it consumes 10 units for each 1000-item page loaded into iterator on-demand
3 // * 'length' - number of items
4 // * 'index' - index of last returned item
5 // * 'next' - returns next item from iterator or undefined when iterator is done
6 // * 'each' - calls specified function for each item in the iterator
7 //
8 // optionalParameters:
9 // * 'select' - list of field names to be returned

```

```

10 // * 'filter' - limiting condition
11 var iterator = NSOA.report.data(
12     7,
13     {
14         select: ["Name", "Remaining Budget"],
15         filter: "Name eq 'Nathan Brown'"
16     }
17 );
18
19 // get number of records published
20 var row_count = iterator.length;
21
22 // grab first two records
23 var first = iterator.next();
24 var second = iterator.next();
25
26 // process rest of the report
27 iterator.each(function(record, index) {
28
29     // search for particular name
30     if (record.Name === "Nathan Brown") {
31
32         // set the field value
33         NSOA.form.setValue("remaining_budget__c", record["Remaining Budget"]);
34
35         // stop iterating
36         return false;
37     }
38 });

```

NSOA.report.list()

Use this function to read the list of reports published using OpenAir Business Intelligence Connector. The list contains the same data as the "list" report available in your business intelligence tool.

For more information, see [Business Intelligence Connector](#). See also [NSOA.listview.list\(\)](#) to read the list of published list views.



Note: The OpenAir Business Intelligence Connector must be enabled for your account to use NSOA.listview and NSOA.report functions. OpenAir Business Intelligence Connector is a licensed add-on. To enable this feature, contact your OpenAir account manager.

For more information about publishing list views and reports to the OpenAir OData service, see the [OpenAir Business Intelligence Connector Guide](#).

Parameters

N/A

Returns

- The list of published reports. Each item in the list has the following properties:
 - ID — The report ID
 - Name — The name of the report
 - Rows — The number of rows of data in the report
 - PublishType — The scope of use specified for the published report.
 - Last published — The date the report was last published

Important: The Last published property will be Last_published with an underscore instead of the space if the optional feature **Replace Non-Alphanumeric Characters with Underscores in Column Titles and Metadata** is enabled for your account. Accommodate both possibilities in your scripts to ensure your scripts continue work whether the feature is enabled or not.

Units Limit

- 1 unit

Since

- October 13, 2018

Example

```

1 // get the list of published reports
2 var reports = NSOA.report.list();
3
4 // each item in the list has following properties
5 // * 'ID'
6 // * 'Name'
7 // * 'Rows'
8 // * 'Last published'
9
10 // loop through all published reports and find given report ID
11 var i;
12 var reportId=0;
13 for (i = 0; i < reports.length; i++) {
14     if (reports[i].Name === 'My Approved Bookings') {
15         reportId = reports[i].ID;
16         break;
17     }
18 }
19
20 // if report ID was found get its data
21 if (reportId > 0) {
22     var rows = NSOA.report.data(reportId);
23
24     // process all report rows
25 }

```

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.

Note: For more information about the SOAP API (Web Services), see [OpenAir XML API & SOAP API 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.

```

1 // Define a category object to create in OpenAir
2 var category = new NSOA.record.oaCategory();
3 category.name = "New Category";
4 category.cost_centerid = "123";
5 category.currency = "USD";
6
7 // Invoke the add call
8 var results = NSOA.wsapi.add( [category] );
9
10 // Get the new ID
11 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.approve(approveRequest)

Use this function to approve bookings, 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.

```

1 // Create the approval object
2 var approvalObj = new NSOA.record.oaApproval();
3 approvalObj.notes = "approve from scripting";
4
5 // Prepare the record for approve
6 var timesheetToProcess = new NSOA.record.oaTimesheet();
7 timesheetToProcess.id = 45;
8
9 // Define the approve requests
10 var requests = [{
11     approve: timesheetToProcess,
12     attributes: [], // approve attributes are optional
13     approval: approvalObj
14 }];
15
16 // Invoke the action call
17 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.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 about the SOAP API (Web Services), see [OpenAir XML API & SOAP API 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.

```

1 // Delete customer with internal ID 66
2 var customer = new NSOA.record.oaCustomer();
3 customer.id = 66;
4
5 // Invoke the delete call
6 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 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.

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

- Enable user filterset on .wsapi requests.

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

Note: This the default OpenAir behavior, that is, user filter sets enabled.

- Return the Boolean state (without changing setting)

```
1 | if( NSOA.wsapi.disableFilterSet() ) {
2 |     // The user filter sets are disabled
3 | }
```

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.

Script Deployment Messages			
Severity	Timestamp	Generated by	Message
Debug	2014-01-13 10:37:34	User	API Request: <?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Env
Debug	2014-01-13 10:37:34	User	API Response: <?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:En

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 and 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 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.

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

- Disable wsapi logging.

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

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

- Returns the Boolean state (without changing setting)

```
1 | if( NSOA.wsapi.enableLog() ) {
2 |     // wsapi logging is enabled
3 | }
```

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 need to specify the internal ID for each object passed, as well as the properties you want to modify.

Note: For more information about the SOAP API (Web Services), see [OpenAir XML API & SOAP API 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.

```

1 // Modify customer's email address
2 var customer = new NSOA.record.oaCustomer();
3 customer.id = 37;
4 customer.addr_email = "newest@example.com";
5
6 // Not attributes required
7 var attributes = [];
8
9 // Invoke the modify call
10 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. The function returns an error if the response would exceed 1000 objects.

Note: For more information about the SOAP API (Web Services), see [OpenAir XML API & SOAP API Guide](#).

See also [Making SOAP Calls](#).

Parameters

- readRequest* {object} [required] — Array of [ReadRequest](#) objects.

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.

```

1 // Create the issue object
2 var issue = new NSOA.record.oaIssue();
3 issue.project_id = NSOA.form.getValue('id');
4 issue.issue_stage_id = 1;
5
6 // Define the read request
7 var readRequest = {
8   type : "Issue",
9   method : "equal to", // return only records that match search criteria
10  fields : "id, date", // specify fields to be returned
11  attributes : [ // Limit attribute is required; type is Attribute
12    {
13      name : "limit",
14      value : "10"
15    }
16  ],
17  objects : [ // One object with search criteria; type implied by rr 'type'
18    issue
19  ]
20 };
21
22 // Invoke the read call
23 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.reject(rejectRequest)

Use this function to reject bookings, 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.

```

1 // Create the approval object
2 var approvalObj = new NSOA.record.oaApproval();
3 approvalObj.notes = "reject from scripting";
4
5 // Prepare the record for reject
6 var timesheetToProcess = new NSOA.record.oaTimesheet();
7 timesheetToProcess.id = 45;
8
9 // Define the reject requests
10 var requests = [{
11     reject: timesheetToProcess,
12     attributes: [], // reject attributes are optional
13     approval: approvalObj
14 }];
15
16 // Invoke the action call
17 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.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.

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

See also [NSOA.context.remainingTime\(\)](#).

For more information see [Scripting Governance](#).

NSOA.wsapi.submit(submitRequest)

Use this function to submit bookings, 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.

```
1 | // Create the approval object
2 | var approvalObj = new NSOA.record.ooApproval();
3 | approvalObj.notes = "submit from scripting";
4 |
5 | // Prepare the record for submit
```

```

6 var timesheetToProcess = new NSOA.record.oaTimesheet();
7 timesheetToProcess.id = 45;
8
9 // Define the submit requests
10 var requests = [{
11     submit: timesheetToProcess,
12     attributes: [], // submit attributes are optional
13     approval: approvalObj
14 }];
15
16 // Invoke the action call
17 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.unapprove(unapproveRequest)

Use this function to unapprove bookings, 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.

```

1 // Create the approval object
2 var approvalObj = new NSOA.record.oaApproval();
3 approvalObj.notes = "unapprove from scripting";
4
5 // Prepare the record for unapprove
6 var timesheetToProcess = new NSOA.record.oaTimesheet();
7 timesheetToProcess.id = 45;
8
9 // Define the unapprove requests

```

```

10 var requests = [{
11     unapprove: timesheetToProcess,
12     attributes: [], // unapprove attributes are optional
13     approval: approvalObj
14 }];
15
16 // Invoke the action call
17 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.

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 about the SOAP API (Web Services), see [OpenAir XML API & SOAP API 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.<complexType>\(\[id\]\)](#).

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.

```

1 | //Define a category object to create/update in OpenAir

```

```

2   var category = new NSOA.record.oaCategory();
3   category.name = "Updated Category";
4   category.externalid = "555";
5
6   // Specify that the lookup is done by external_id and not by (default) internal ID
7   var attribute = {
8     name : "lookup",
9     value : "externalid"
10  };
11
12  // Invoke the upsert call
13  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 about the SOAP API (Web Services), see [OpenAir XML API & SOAP API 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.

```

1   function logUser() {
2     var user = NSOA.wsapi.whoami();
3     NSOA.meta.alert( "User ID " + user.id + " saved this record");

```

```
4 | }
```

Note: This simple example does not show error checking, see [Handling SOAP 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](#)
- [Outbound Calling — SOAP Call Using HTTPS POST](#)
- [Outbound Calling — Post a Slack Message](#)
- [Outbound Calling — HTTPS GET with Authorization](#)

See also [Real World Use Cases](#).

Comparing Date Fields

```
1 // compare two date fields on a receipt
2 function validateTravelDates() {
3   var receiptDate = NSOA.form.getValue('date');
4   var travelDate = NSOA.form.getValue('TravelDate__c');
5
6   if ( receiptDate < travelDate ) {
7     NSOA.form.error('TravelDate__c', 'The travel date cannot be after the receipt date!');
8   }
9 }
```

See also:

- [NSOA.form.getValue\(field\)](#)
- [NSOA.form.error\(field, message\)](#)

Validating Numeric Fields

```
1 // validate a number entered into a custom numeric field
2 function projectRating() {
3   var rating = NSOA.form.getValue('ProjectRating__c');
4 }
```

```

5 |   if ( rating < 1 || rating > 5 ) {
6 |       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

```

1 | // require notes on all airfare exceeding $1,000 dollars
2 | function airfareCost() {
3 |     var cost = NSOA.form.getValue('cost');
4 |     var notes = NSOA.form.getValue('notes');
5 |     var item = NSOA.form.getValue('item_id');
6 |
7 |     if ( cost > 1000 && notes.length < 1 && item == '4' ) {
8 |
9 |         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

```

1 | // add an error log entry to the validateTravelDates() function above
2 | function validateTravelDates() {
3 |     var receiptDate = NSOA.form.getValue('date');
4 |     var travelDate = NSOA.form.getValue('TravelDate__c');
5 |
6 |     if ( receiptDate < travelDate ) {
7 |         NSOA.form.error('TravelDate__c', 'The travel date cannot be after the receipt date!');
```

See also:

- [NSOA.form.getValue\(field\)](#)
- [NSOA.form.error\(field, message\)](#)
- [NSOA.meta.log\(severity, message\)](#)

Sending email

```

1 | function sendAlert() {
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 OpenAir</i>"
14  };
15
16  NSOA.meta.sendMail(msg);
17  }

```

See also:

- [NSOA.meta.sendMail\(message\)](#)

SOAP API — Prevent closing a project with an open issue

```

1  function test_prevent_project_close_with_open_issue() {
2      var project_stage_id = NSOA.form.getValue('project_stage_id');
3      if (project_stage_id != 4) // if new stage is not closed, skip check
4          return;
5
6      //Read request
7      var issue = new NSOA.record.oaIssue();
8      issue.project_id = NSOA.form.getValue('id');
9      issue.issue_stage_id = 1;
10     var readRequest = {
11         type : "Issue",
12         method : "equal to", // return only records that match search criteria
13         fields : "id, date", // specify fields to be returned
14         attributes : [ // Limit attribute is required; type is Attribute
15             {
16                 name : "limit",
17                 value : "10"
18             }
19         ],
20         objects : [ // One object with search criteria; type implied by rr 'type'
21             issue
22         ]
23     };
24     var arrayOfreadResult = NSOA.wsapi.read(readRequest);
25     if (!arrayOfreadResult || !arrayOfreadResult[0])
26         NSOA.form.error('', "Internal error analyzing project. Contact account administrator.");
27     else if (arrayOfreadResult[0].errors === null && arrayOfreadResult[0].objects)
28         arrayOfreadResult[0].objects.forEach(
29             function(o) {
30                 NSOA.form.error('', "Can't close project with open issues.");
31             }
32         );
33 }

```

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

```

1  // This is called on the "After save" event for the Project form

```



```

2 function append_to_project_notes() {
3     var newr = NSOA.form.getNewRecord();
4
5     var project = new NSOA.record.oaProject();
6     project.id = newr.id;
7     project.notes = newr.notes + "\nAppended to notes: " + (new Date()).toString();
8
9     NSOA.wsapi.disableFilterSet(true);
10    var arrayOfupdateResult = NSOA.wsapi.modify([], [project]);
11    NSOA.meta.alert("Got modify status: " + arrayOfupdateResult[0].status);
12 }

```

See also:

- [NSOA.form.getNewRecord\(\)](#)
- [NSOA.record.<complex type>\(\[id\] \)](#)
- [NSOA.wsapi.modify\(attributes, objects\)](#)
- [NSOA.meta.alert\(message\)](#)

SOAP API — Require task assignment

```

1 // Add form error if user is not assigned to project task to which they're about to be booked.
2 function require_task_assignment() {
3
4     // Prepare read query
5     var pta = new NSOA.record.oaProjecttaskassign();
6     pta.projecttaskid = NSOA.form.getValue('project_task_id');
7     pta.userid = NSOA.form.getValue('user_id');
8
9     var readRequest = {
10        type : "Projecttaskassign",
11        method : "equal to",           // return only records that match search criteria
12        fields : "id",                // specify fields to be returned
13        attributes : [                // Limit attribute is required; type is Attribute
14            {
15                name : "limit",
16                value : "1"
17            }
18        ],
19        objects : [                    // One object with search criteria
20            pta
21        ]
22    };
23
24    // Run query
25    NSOA.wsapi.disableFilterSet(true); // disable the current user's filter for read query
26    var result = NSOA.wsapi.read(readRequest);
27
28    // Check query results
29    if (!result || !result[0])
30        NSOA.form.error('', "Internal error analyzing booking. Contact account administrator.");
31    else if (result[0].errors !== null || result[0].objects === null || result[0].objects.length === 0)
32        NSOA.form.error('_user_id', "Can't book this user without being assigned to selected project task.");
33 }

```

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.

Submission rules			
Active	Rule	Hours/Percent	Action
<input checked="" type="checkbox"/>	Minimum number of hours required on the timesheet	Fixed hours <input type="text" value="10"/>	Warn
<input checked="" type="checkbox"/>	Maximum number of hours allowed on the timesheet	Fixed hours <input type="text" value="168"/>	Error
<input type="checkbox"/>	Minimum number of hours per day required on the timesheet	Fixed hours <input type="text" value=""/>	Error
<input checked="" type="checkbox"/>	Maximum number of hours per day allowed on the timesheet	Fixed hours <input type="text" value="24"/>	Error
<input checked="" type="checkbox"/>	Minimum leave accrual balance	<input type="text" value="8"/>	Warn

```

1 function main(type) {
2
3     // Create the approval object
4     var approvalObj = new NSOA.record.oaApproval();
5     approvalObj.notes = "submit from scripting";
6
7     // Prepare the record for submit
8     var timesheetToProcess = new NSOA.record.oaTimesheet();
9     timesheetToProcess.id = 45;
10
11    // Ignore any warnings
12    var ignore_warnings = {
13        name : "submit_warning",
14        value : "1"
15    };
16
17    // Define the submit requests
18    var requests = [{
19        submit: timesheetToProcess,
20        attributes: [ignore_warnings],
21        approval: approvalObj
22    }];
23
24    // Invoke the action call
25    var results = NSOA.wsapi.submit(requests);
26
27 }

```

See also:

- [NSOA.record.<complex type>\(\[id\]\)](#)
- [NSOA.wsapi.submit\(submitRequest\)](#)

Outbound Calling — SOAP Call Using HTTPS POST

In this code sample, the **NSOA.https.post** function is used to make a SOAP call. An object is created with url, headers and body properties. The object is then passed to the NSOA.https.post function and the response returned.

Note: Refer to the API you are calling for details on the expected request and response formats.

```

1 /**
2  * SOAP call to get data center URLs using HTTPS POST method.

```

```

3  * @param {Str}   accountID   ID of a NetSuite account.
4  * @return {Obj}   An https.post response object.
5  */
6  function getDataCenterUrls(accountID){
7
8
9     var url = 'https://webservices.netsuite.com/services/NetSuitePort_2017_1';
10
11    var headers = {
12      'SOAPAction': 'getDataCenterUrls',
13      'Content-type': 'application/javascript'
14    };
15
16
17    var body = '<?xml version="1.0" encoding="UTF-8"?>' +
18      '<soapenv:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
19      xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:platformMsgs="urn:messages_2017_1.platform.webservices.netsuit
20      e.com">' +
21      '  <soapenv:Body>' +
22      '    <getDataCenterUrls xsi:type="platformMsgs:GetDataCenterUrlsRequest">' +
23      '      <account xsi:type="xsd:string">' + accountID + '</account>' +
24      '    </getDataCenterUrls>' +
25      '  </soapenv:Body>' +
26      '</soapenv:Envelope>';
27
28    var request = {
29      url: url,
30      headers: headers,
31      body: body
32    };
33
34    var response = NSOA.https.post(request);
35
36    return response;
37 }

```

See also:

- [NSOA.https.post\(request\)](#)

Outbound Calling — Post a Slack Message

In this code sample, the **NSOA.https.post** function is used to post a message on Slack using a webhook URL. An object is created with url, headers and body properties. The body is defined, including any attachments, and the headers property is used to specify the content type. The object is then passed to the NSOA.https.post function and the response returned.

```

1  /**
2  * Post a message to slack using a webhook URL.
3  * @param {Str}   text       Text to display on message (required).
4  * @param {Array} attachments Array of attachment objects (optional).
5  * @return {Obj}   An https.post response object.
6  */
7  function postSlackMessage(url, text, attachments) {
8
9     // Check that url parameter has a value, otherwise return
10    url = url || '';
11    if (!url || url.length === 0) { return null; }
12
13    // Check that text parameter has a value, otherwise return
14    text = text || '';
15    if (!text || text.length === 0) { return null; }
16
17    var body = {
18      text: text
19    };
20

```

```

21 // If attachments param is provided, and it is of type Array (isArray method isn't supported...)
22 if (attachments && Object.prototype.toString.call(attachments) === '[object Array]') { body.attachments = attachments; }
23
24 NSOA.meta.log('debug', 'post.body -> ' + JSON.stringify(body));
25
26 var headers = {
27   'Content-type': 'application/json'
28 };
29
30 var response = NSOA.https.post({
31   url: url,
32   body: body,
33   headers: headers
34 });
35
36 return response;
37 }

```

See also:

- [NSOA.https.post\(request\)](#)
- [NSOA.meta.log\(severity, message\)](#)

Outbound Calling — HTTPS GET with Authorization

In this code sample, the **NSOA.https.get** function is used to get a protected resource. A **Password script parameter** is used to store the API token securely. The headers are created with the API token and then passed with the URL to the **NSOA.https.get** function.

```

1  /**
2  * Get a protected resource from a URL.
3  * @param {Str} url    API URL (required)
4  * @return {Obj}      An https.get response object.
5  */
6  function getProtectedResource(url) {
7    // Check that url parameter has a value, otherwise return
8    url = url || '';
9    if (!url || url.length === 0) { return null; }
10
11    // Retrieve API auth token from script parameter
12    api_token = NSOA.context.getParameter('api_token');
13
14    var headers = {
15      'Authorization': 'Bearer ' + api_token
16    };
17
18    var response = NSOA.https.get({
19      url: url,
20      headers: headers
21    });
22
23    return response;
24 }

```

See also:


- [NSOA.context.getParameter\(name\)](#)
- [NSOA.https.get\(request\)](#)

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.

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


- JavaScript is case sensitive.

 **Important:** The following variables are NOT the same!

```
1 | var receiptDate;
2 | var ReceiptDate;
```

- JavaScript supports single and multiline comments. Use comments to make your scripts maintainable!


```
1 | // This is a single line comment
2 |
3 | /*
4 |   This is a multiline comment
5 | */
```


 **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.

```
1 | 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.

```
1 | price = 500;
```

A variable can be assigned a value when it is declared.


```
1 | var price = 500;
```

A variable can be emptied by setting its value to [null](#).


```
1 | price = null;
```

If you re-declare a variable, the variable will not lose its value.

```
1 | var travelType = "Car";
2 | 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 to 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.

```
1 | var travelType;           // travelType is undefined
2 | var travelType = 5;      // travelType is a Number
3 | 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 ".

```
1 | var name = "John Smith";
2 | var type = 'customer';
```

You can use quotes inside a string, as long as they don't match the quotes surrounding the string.

```
1 | var responseText = "It was paid to 'John Smith'";
2 | var responseText = 'It was paid to "John Smith"';
```

You can put a quote inside a string using the \ character.

```
1 | var responseText = 'It\'s okay.';
```

You can access a character in a string by its zero-based position index.

```
1 | var name = "John Smith";
2 | 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.

```

1 | var pi = 3.14;
2 | var amount = 314e5; // 31400000
3 | 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.

```

1 | var x=0377; // This is 255 in decimal
2 | 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**.

```

1 | var sent = true;
2 | 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**.

```

1 | travelType = null;

```

undefined

This is a special keyword denoting an undefined value.

Before a variable is assigned a value it is undefined.

```

1 | var travelType; // variable is undefined

```

Arrays

In JavaScript an array is created as follows:

```

1 | // Creating an array
2 | var priority = new Array();
3 | priority[0] = "Low";
4 | priority[1] = "Normal";
5 | priority[2] = "High";
6 |
7 | // Literal array

```



```
8 | var priority = ["Low", "Normal", "High"];
```

Note: JavaScript Arrays are zero base.

An element is accessed in the array by **index** number:

```
1 | // To access the first element
2 | var level = priority[0];
3 |
4 | // To modify the first element
5 | priority[0] = "Not required";
```

You can have different types in an array:

```
1 | var entry = new Array();
2 | entry[0] = Date.now;
3 | entry[1] = "Book";
4 | 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.

```
1 | // To create an associative array
2 | contacts = {
3 |   firstname : 'John',
4 |   lastname  : 'Smith'
5 | };
```

Note: Notice the similarity to [Objects](#).

An associative array is accessed by a key name.

```
1 | // Use the key to access an entry
2 | var value = contacts['firstname']; // value is 'John'
3 |
4 | // You can also use dot notation
5 | 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.

```
1 | // Get all the values on the fields on the form
2 | var allValues = NSOA.form.getAllValues();
3 |
4 | //Loop through all the values
5 | for( var key in allValues ) {
6 |   NSOA.meta.alert(key + ' has value ' + allValues[key]);
```

```
7 | }
```

See also:

- [NSOA.form.getAllValues\(\)](#)
- [NSOA.meta.alert\(message\)](#)

You can change the value using assignment to a property.

```
1 | // Using the array notation
2 | contacts['firstname'] = 'Joe';
3 |
4 | // Using dot notation
5 | contacts.firstname = 'Joe';
```

You can add a new key/value pair by assigning to a property that doesn't exist.

```
1 | // Using the array notation
2 | contacts['company'] = 'NetSuite';
3 |
4 | // Using dot notation
5 | 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.

```
1 | // To declare an object
2 | var person={ firstname : "John", lastname : "Smith", age: 25};
3 |
4 | // Use spaces and line breaks to make your definition clearer
5 | var person={
6 |     firstname : "John",
7 |     lastname  : "Smith",
8 |     age       : 25
9 | };
10 |
11 | // You can access object properties in two way
12 | var age = person.age;
13 | 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:

```
1 | objectName.propertyName
```

This example uses the length property of the [String](#) object to find the length of a string:

```
1 | var message = "Hello World!";
2 | 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:

```
1 | objectName.methodName()
```

This example uses the toUpperCase() method of the [String](#) object, to convert a text to uppercase:

```
1 | var message="Hello world!";
2 | 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](#).

```
1 | // Declaring a function
2 | function calcSum (x,y) {
3 |     return x + y;
4 | }
5 |
6 | // Calling a function
7 | 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.

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

You can pass as many arguments as you like separated by commas.

```
1 | function calcSum (x,y,z) {
2 |     var result = x + y + z;
3 |     return result;
```

```
4 | }
```

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.

```
1 | function logFormError(message) {
2 |     NSOA.meta.log('error', 'Form error - ' + message);
3 | }
```

See also:

- [NSOA.meta.log\(severity, message\)](#)

Use the **return** statement to return a variable from a function.

```
1 | function calcProduct (x,y) {
2 |     var result = x * y;
3 |     return result;
4 | }
```

You can use the **return** statement to immediately exit a function. The return value is optional.

```
1 | function reduceValue (x,y) {
2 |     if ( x < y ) {
3 |         return; // exit function
4 |     }
5 |
6 |     var result = x - y;
7 |     return result;
8 | }
```

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

```
1 for (initialization; condition; increment) {
2   // statements
3 }
```

Example

```
1 for (var i = 0; i < 5; i++) {
2   x = x + "The number is " + i;
3 }
```

for in

The **for in** loop is for iterating through the enumerable properties of an object. See [Objects](#) and [Associative Array](#).

Syntax

```
1 for (variable in object) {
2   // code to be executed
3 }
```

Example

```
1 var person={firstName:"John",lastName:"Smith",age:21};
2
3 for (i in person) {
4   s = s + person[i];
5 }
```

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.

```
1 var a = ["a", "b", "c"];
2 a.forEach(function(o) {
3   NSOA.meta.alert(o);
4 });
```

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

```
1 do {
2   // statements
3 }
4 while (condition)
```

Example

```
1 var i=0;
2 do {
3   x = x + "The number is " + i;
4   i++;
5 }
6 while ( i < 5 )
```

while

The **while** loop iterates through a block of code as long as a specified condition is true.

Syntax

```
1 while (condition) {
2   // statements
3 }
```

Example

```
1 var i = 0;
2 while ( i < 5 ) {
3   x = x + "The number is " + i;
4   i++;
5 }
```

Conditional Statements

JavaScript supports [if ... else](#) and [switch](#) conditional statements.

if ... else syntax

```
1 if (condition) {
2   statements_1
3 } else {
4   statements_2
5 }
```

switch syntax

```

1 switch (expression) {
2   case label_1:
3     statements_1
4     [break;]
5   case label_2:
6     statements_2
7     [break;]
8   default:
9     statements_n
10    [break;]
11 }

```

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.

```

1 var endDate = NSOA.form.getValue('end_date');
2 var today = new Date();
3 if (today < endDate) {
4   // statements to execute if we haven't started this yet
5 }


```

If the expression is false (i.e. `today < endDate` is not true) the optional block following **else** is executed.

```

1 var endDate = NSOA.form.getValue('end_date');
2 var today = new Date();
3 if (today < endDate) {
4   // statements to execute if we haven't started this yet
5 } else {
6   // statements to execute if we have started
7 }

```


 **Note:** 'else' is optional, but 'if' must be present


You can chain together as many **if ... else** statements as required.

```

1 var type = NSOA.form.getValue('type');
2 if (type == 0) {
3   // statements to handle type 0
4 } else if (type == 1) {
5   // statements to handle type 1
6 } else if (type == 2) {
7   // statements to handle type 2
8 } else {
9   // statements to handle all other types
10 }

```

 **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.

```

1 var type = NSOA.form.getValue('type');
2 switch (type) {
3   case 0:
4     // statements to handle type 0
5     break;
6   case 1:
7     // statements to handle type 1
8     break;
9   case 2:
10    // statements to handle type 2
11    break;
12  default:
13    // statements to handle all other types
14 }

```

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

```

1 try {
2   // The code to run
3 }
4 catch(err) {
5   // Code to handle any errors
6 }

```

Example

```

1 try {
2   var receiptDate = NSOA.form.getValue('date');
3   var travelDate = NSOA.form.getValue('TravelDate__c');
4
5   if ( receiptDate < travelDate ) {
6     NSOA.form.error('TravelDate__c', 'The travel date cannot be after the receipt date!');
7   }
8 }
9 catch(err) {
10  NSOA.meta.log('error', err.message);
11 }

```

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.

```

1 // Function that throws a custom "Divide by zero" error
2 function divide(x,y) {
3     if( y == 0 ) {
4         throw( "Divide by zero" );
5     } else {
6         return x / y;
7     }
8 }
9
10 //Function that catches the custom error as a string
11 function test() {
12     try {
13         return divide(10,0);
14     }
15     catch(err) {
16         // err == "Divide by zero"
17     }
18 }

```



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.

```

1 // Creating an array
2 var priority = new Array();
3 priority[0] = "Low";
4 priority[1] = "Normal";
5 priority[2] = "High";
6
7 // To access the first element
8 var level = priority[0];
9
10 // To modify the first element
11 priority[0] = "Not required";
12
13
14 // To find the length of an array

```

```

15 | var x = priority.length
16 |
17 | // To find the index position of an element in the array
18 | 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**).

```

1 | var bool = new Boolean();

```

Boolean Properties

Property	Description
constructor	Returns the function that created the Boolean object's prototype.

Property	Description
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"). 1 <code>bool.toString()</code>
valueOf()	Returns the primitive value of a Boolean object (either true or false). 1 <code>bool.valueOf()</code>

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:

```
1 | var dt = new Date();
2 | var dt = new Date(milliseconds);
3 | var dt = new Date(dateString);
4 | var dt = new Date(year, month, day, hours, minutes, seconds, milliseconds);
```

Example of setting a date

```
1 | var startDate = new Date();
2 | startDate.setFullYear(2013,0,14); // startDate == "Jan 14, 2013"
```

 **Note:** month is zero-based i.e. 0 == 'January'

Example of comparing two dates

```
1 | var startDate = new Date();
2 | startDate.setFullYear(2013,0,14);
3 | var today = new Date();
4 | if (startDate > today) {
5 |     // startDate later than today's date
6 | } else {
7 |     // startDate is on or before today's date
8 | }
```

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
<code>getDate()</code>	Returns the day of the month (from 1-31).
<code>getDay()</code>	Returns the day of the week (from 0-6).
<code>getFullYear()</code>	Returns the year (four digits)
<code>getHours()</code>	Returns the hour (from 0-23).
<code>getMilliseconds()</code>	Returns the milliseconds (from 0-999).
<code>getMinutes()</code>	Returns the minutes (from 0-59).
<code>getMonth()</code>	Returns the month (from 0-11).
<code>getSeconds()</code>	Returns the seconds (from 0-59).
<code>getTime()</code>	Returns the number of milliseconds since midnight Jan 1, 1970.
<code>getTimezoneOffset()</code>	Returns the time difference between UTC time and local time, in minutes.
<code>getUTCDate()</code>	Returns the day of the month, according to universal time (from 1-31).
<code>getUTCDay()</code>	Returns the day of the week, according to universal time (from 0-6).
<code>getUTCFullYear()</code>	Returns the year, according to universal time (four digits).
<code>getUTCHours()</code>	Returns the hour, according to universal time (from 0-23).
<code>getUTCMilliseconds()</code>	Returns the milliseconds, according to universal time (from 0-999).
<code>getUTCMinutes()</code>	Returns the minutes, according to universal time (from 0-59).
<code>getUTCMonth()</code>	Returns the month, according to universal time (from 0-11).
<code>getUTCSeconds()</code>	Returns the seconds, according to universal time (from 0-59).
<code>getYear()</code>	Deprecated. Use the <code>getFullYear()</code> method instead.
<code>parse()</code>	Parses a date string and returns the number of milliseconds since midnight of January 1, 1970.
<code>setDate()</code>	Sets the day of the month of a date object.
<code>setFullYear()</code>	Sets the year (four digits) of a date object.
<code>setHours()</code>	Sets the hour of a date object .
<code>setMilliseconds()</code>	Sets the milliseconds of a date object.
<code>setMinutes()</code>	Set the minutes of a date object.
<code>setMonth()</code>	Sets the month of a date object.
<code>setSeconds()</code>	Sets the seconds of a date object.
<code>setTime()</code>	Sets a date and time by adding or subtracting a specified number of milliseconds to/from midnight January 1, 1970.
<code>setUTCDate()</code>	Sets the day of the month of a date object, according to universal time.
<code>setUTCFullYear()</code>	Sets the year of a date object, according to universal time (four digits).

Method	Description
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, formatted 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.

```
1 | var pi = Math.PI; // Returns PI value
2 | 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).

Property	Description
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.

```
1 | 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.

Property	Description
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.

```

1 | var s = new String("Hello world!");
2 | // or just:
3 | var s = "Hello world!";
4 |
5 | // Finding the length of a string
6 | var message = "Hello World!";
7 | var x = message.length; // x is 12
8 |
9 | // Converting a string to uppercase
10 | var message="Hello world!";
11 | var x=message.toUpperCase(); // x is "HELLO WORLD!"
12 |
13 | /* The indexOf()method returns the position (as a number) of the
14 | first found occurrence of a specified text inside a string */
15 | var str="Hello world, welcome to OpenAir.";
16 | 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.

```

1  /* If the quantity ordered is more than 1000
2  * then the packing cost is for a large packet,
3  * otherwise the packing cost is for a small packet.
4  */

```



```
5 | 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
===	exactly equal to (value and type)
!=	not equal
!==	not exactly equal (different value or type)

Operator	Description
>	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 is 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 is \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 and OpenAir Mobile](#).
- If you are using NetSuite Connector to import Project records from NetSuite, review NetSuite Connector configuration options. Depending on the integration configuration, form scripts triggered by the “Before save” or “After save” events may run for each imported project record. See [Scripting and OpenAir NetSuite Connector](#).
- 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 OpenAir. 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: Oracle may provide sample code in SuiteAnswers, OpenAir Help Center, User Guides, or elsewhere through help links. All such sample code is provided "as is" and "as available", for use only with an authorized OpenAir Service account, and is made available as a SuiteCloud Technology subject to the SuiteCloud Terms of Service at www.netsuite.com/tos where the term "Service" shall mean the OpenAir Service.

Oracle 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
 - Send a Slack notification when issues are created or (re)assigned

Note: Find these examples on the **Platform Solutions** catalog page in OpenAir Help Center. See [Platform Solutions Catalog](#).


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](#).













Platform Solutions Catalog

Find the real world use case scripting examples on the **Platform Solutions** catalog page in OpenAir Help Center.

To view the platform solutions catalog, go to User Center > Help Center > Platform Solutions. The page lists all the real world user scripting examples described in this guide. Solutions are color coded by category — Validation, Automation, Workflow — with a visual representation and a short summary to indicate the purpose of each solution.

- To save the solution XML file, click the **DOWNLOAD** link. You can then import the solution file from the Scripting Center in OpenAir.
- To read the help topic describing the solution, click the **LEARN MORE** link.

Platform Solutions

 (Validation) Commission	 (Validation) Overtime Note	 (Validation) Ticket Attachment	 (Validation) Time Check	
<p>Ensure value of multiple commission fields equals 100%.</p> <p>LEARN MORE DOWNLOAD</p>	<p>Require notes field to be populated on time entries when more than 8 hours in a day.</p> <p>LEARN MORE DOWNLOAD</p>	<p>When submitting an expense report, validate each ticket has an attachment (e.g. scanned receipt).</p> <p>LEARN MORE DOWNLOAD</p>	<p>Ensure resource time entry matches booking planning and project worked hours.</p> <p>LEARN MORE DOWNLOAD</p>	
 (Automation) Project PO	 (Automation) Auto Timesheet	 (Automation) Budget Hours		
<p>Optionally create a new Customer PO when editing a project.</p> <p>LEARN MORE DOWNLOAD</p>	<p>Create time entries from task assignments when the user creates a new timesheet.</p> <p>LEARN MORE DOWNLOAD</p>	<p>Control budgeted hours for a project using the project budget feature and a custom hours field.</p> <p>LEARN MORE DOWNLOAD</p>		
 (Workflow) Time Off	 (Workflow) Open Issues	 (Workflow) Risk Issues	 (Workflow) Email Alert	 (Workflow) Slack Notification
<p>Prevent a booking from being created if the selected resource has approved time off during the booking period.</p> <p>LEARN MORE DOWNLOAD</p>	<p>Prevent closing a project that has open issues.</p> <p>LEARN MORE DOWNLOAD</p>	<p>Automatically create a new issue when project stage is "at risk" and prevent project stage from changing until this issue is resolved.</p> <p>LEARN MORE DOWNLOAD</p>	<p>Send an alert email when a scheduled script completes.</p> <p>LEARN MORE DOWNLOAD</p>	<p>Send a Slack notification when issues are created or (re)assigned.</p> <p>LEARN MORE DOWNLOAD</p>

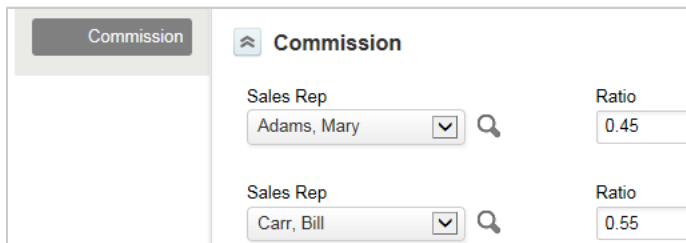
Validation

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.



Commission	
Sales Rep <input type="text" value="Adams, Mary"/>	Ratio <input type="text" value="0.45"/>
Sales Rep <input type="text" value="Carr, Bill"/>	Ratio <input type="text" value="0.55"/>

Follow the steps below or [download the solutions file](#), see [Creating Solutions](#) for details.

Setup

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](#).

▼ Scripting Studio

Association
Project

Employee
Collins, Marc

Execution displays internal form script deployment log error debug detail for this user

References

ALL 4 | SELECTED 0

Event
Before save

Entrance function
checkCommish

No log mess:

```

1 //
2 var
3
4
5 ];
6
7 fun
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

```

- 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 first custom field to **Commission** to place the custom fields in their own section.

Position	Display name	Name	Association	Field type	Active
		P	Project	- All -	- All -
12	Sales Rep	prj_sales_rep_1	Project	Pick List	✓
14	Sales Rep	prj_sales_rep_2	Project	Pick List	✓
13	Ratio	prj_sales_rep_ratio_1	Project	Ratio	✓
15	Ratio	prj_sales_rep_ratio_2	Project	Ratio	✓

- 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.

View Log

```

1 // ADD YOUR REP AND RATIO CUSTOM FORM FIELD N
2 var repCompFlds = [
3 'prj_sales_rep_1_c', 'prj_sales_rep_ratio_1_c'
4 'prj_sales_rep_2_c', 'prj_sales_rep_ratio_2_c'
5 ];
6
7 function checkCommish(type) {
8
9     try {
10         var len = repCompFlds.length,
11             Skip over sales rep name
12             = 0;
13         <_len {
14             Comp += parseFloat(NSOA.form
15             // Skip over sales rep nam
16
17         form.error(
18             'The total sales commission '
19             (' + round(totalCompPercent

```

▼ Form schema

```

prj_sales_rep_1_c (custom_28) [Sales Rep] <String>
prj_sales_rep_2_c (custom_30) [Sales Rep] <String>
prj_sales_rep_ratio_1_c (custom_29) [Ratio] <Number>
prj_sales_rep_ratio_2_c (custom_31) [Ratio] <Number>
project_stage_id [Project stage] <Number>
start_date [Start date] <Date>
ta_approver_choice [Project timesheets approved by] <String>
tax_location_id [Tax location] <Number>
tb_approver_choice [Project invoices approved by] <String>
te_approver_choice [Project expense reports approved by] <String>

```

✓ **Tip:** If the new custom fields are not listed in the **Form schema**, navigate to Projects, open a project form (this will refresh the custom field list), and then open the Scripting Studio again.

Program Listing

```

1 // ADD YOUR REP AND RATIO CUSTOM FORM FIELD NAMES TO THE ARRAY BELOW
2 var repCompFlds = [
3   'prj_sales_rep_1__c', 'prj_sales_rep_ratio_1__c', // Use Form schema to find param names
4   'prj_sales_rep_2__c', 'prj_sales_rep_ratio_2__c'
5 ];
6
7 function checkCommish(type) {
8
9   try {
10    var _len = repCompFlds.length,
11        _i = 1, // Skip over sales rep name
12        _j = 0,
13        totalComp = 0;
14
15    while (_i < _len) {
16      totalComp += parseFloat(NSOA.form.getValue(repCompFlds[_i]));
17      _i += 2; // Skip over sales rep name
18    }
19
20    var totalCompRound = round(totalComp, 2),
21        totalCompPercent = totalCompRound * 100;
22    if (totalCompRound !== 0 && totalCompRound !== 1) {
23
24      NSOA.form.error(
25        '',
26        'The total sales commission ' + totalCompRound +
27        ' (' + round(totalCompPercent, 2) + '%' + ') must equal 100%!';
28      );
29
30      for (_j; _j < _len; _j++) {
31        NSOA.form.error(repCompFlds[_j], 'Please check and re-save.');
```

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)

Follow the steps below or [download the solutions file](#), see [Creating Solutions](#) for details.

Setup

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**.

The screenshot shows the Scripting Studio interface for a 'Timesheet [Edit]' form script deployment. The interface includes the following elements:

- Association:** Timesheet [Edit] (highlighted with a red circle '1')
- Employee:** Collins, Marc
- References:** A table with 30 rows. The first row contains '1' and 'fun'.
- Event:** Before approval (highlighted with a red circle '2')
- Entrance function:** require_timeentry_notes_daily_overtime (highlighted with a red circle '3')

Program Listing

```

1 function require_timeentry_notes_daily_overtime() {
2
3     // Load task data
4     var task = new NSOA.record.oaTask();
5     task.timesheetid = NSOA.form.getOldRecord().id;
6     NSOA.meta.log('debug', "got ts ID " + task.timesheetid);
7
8     var readRequest = {
9         type: "Task",
10        fields: "id, date, decimal_hours, notes",
11        method: "equal to",
12        objects: [task],
13        attributes: [{
14            name: "limit",
15            value: "1000"
16        }]
17    };
18
19    var arrayOfreadResult = NSOA.wsapi.read(readRequest);
20
21    // Analyze the timesheet

```

```


22 | var ts_total_by_day = {};
23 | var task_no_notes_by_day = {};
24 | if (!arrayOfreadResult || !arrayOfreadResult[0])
25 |     NSOA.form.error('', "Internal error loading timesheet details.");
26 |
27 | else if (arrayOfreadResult[0].errors === null && arrayOfreadResult[0].objects)
28 |     arrayOfreadResult[0].objects.forEach(
29 |         function(o) {
30 |             // Get yyyy-mm-dd part
31 |             var date = o.date.substr(0, 10);
32 |
33 |             // Track total hours for this day
34 |             if (ts_total_by_day[date] === undefined)
35 |                 ts_total_by_day[date] = Number(o.decimal_hours);
36 |             else if (ts_total_by_day[date] <= 8)
37 |                 ts_total_by_day[date] += Number(o.decimal_hours);
38 |             else
39 |                 return; // Already reported form error if we got here
40 |             NSOA.meta.log('debug', date + " -> " + ts_total_by_day[date]);
41 |
42 |             // Track time entries with no notes
43 |             if (task_no_notes_by_day[date] === undefined)
44 |                 task_no_notes_by_day[date] = [];
45 |             if (o.notes === undefined || o.notes.length === 0)
46 |                 task_no_notes_by_day[date].push(o.id);
47 |
48 |             // Check the policy
49 |             if (ts_total_by_day[date] > 8 && task_no_notes_by_day[date] !== undefined &&
50 |                 task_no_notes_by_day[date].length > 0) {
51 |                 NSOA.meta.log('trace', ts_total_by_day[date] + " -> " + task_no_notes_by_day[date].length);
52 |                 NSOA.form.error('', "Notes are required on all " + date +
53 |                     " time entries: total reported time that day is more than 8 hours.");
54 |             }
55 |         }
56 |     );
57 | }

```

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.

Follow the steps below or [download the solutions file](#), see [Creating Solutions](#) for details.

Setup

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**.

▼ Scripting Studio

Association
Expense report

Employee
Collins, Marc

Execution displays internal form script deployment log error debug detail for this user

References

ALL 4 SELECTED 0

Select all Clear all

Event
Before approval

Entrance function
check_receipt_has_attachments

No log mess

1 fun

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

Program Listing

```

1 function check_receipt_has_attachments(type) {
2
3     // return if not an approve_request
4     if (type !== 'approve_request')
5         return;
6
7     // Load receipt data
8     var envelope = NSOA.form.getOldRecord();
9     var ticket = new NSOA.record.oaTicket();
10    ticket.envelopeid = envelope.id;
11
12    var readRequest = {
13        type: "Ticket",
14        fields: "id, attachmentid, reference_number, missing_receipt",
15        method: "equal to",
16        objects: [ticket],
17        attributes: [{
18            name: "limit",
19            value: "250"
20        }]
21    };
22
23    var arrayOfreadResult = NSOA.wsapi.read(readRequest);
24    var missingAttachment = [];
25    if (!arrayOfreadResult || !arrayOfreadResult[0])
26        NSOA.form.error('', "Internal error reading envelope receipts.");
27
28    else if (arrayOfreadResult[0].errors === null && arrayOfreadResult[0].objects)
29        arrayOfreadResult[0].objects.forEach(
30            function(o) {
31                if (o.attachmentid === '0' && o.missing_receipt !== '1')
32                    missingAttachment.push(o.reference_number);
33            }
34        );
35
36    if (missingAttachment.length > 0) {
37        NSOA.form.error('',

```

```

38 | "The following receipts (by reference number) are missing an attachment: " +
39 |     missingAttachment.join(", ");
40 | }
41 | }

```

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)

Follow the steps below or [download the solutions file](#), see [Creating Solutions](#) for details.

Setup

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

```

1 | function verify_timeentry_policy(type) {
2 |     var timesheet = NSOA.form.getOldRecord();
3 |
4 |     // Only check on approval request and if current user is the timesheet owner
5 |     if (type != 'approve_request' || timesheet.userid != NSOA.wsapi.whoami().id)
6 |         return;
7 |
8 |     // Load task data

```

```

9   var taskFilter = new NSOA.record.oaTask();
10  taskFilter.timesheetid = timesheet.id;
11
12  // disable current user's filter for this script
13  NSOA.wsapi.disableFilterSet(true);
14
15  // Analyze tasks to load related records
16  var task_readRequest = {
17    type: "Task",
18    fields: "id, date, projecttaskid, decimal_hours",
19    method: "equal to",
20    objects: [taskFilter],
21    attributes: [{
22      name: "limit",
23      value: "1000"
24    }, {
25      name: "filter",
26      value: "current-user"
27    }]
28  };
29  var task_arrayOfreadResult = NSOA.wsapi.read(task_readRequest);
30
31  var tasks_by_uniqueKey = {};
32  var ts_pta_worked_hours = {};
33  var ptaFilters = {};
34  var bookingFilters = {};
35  if (!task_arrayOfreadResult || !task_arrayOfreadResult[0])
36    NSOA.form.error('', "Internal error loading %task% details.");
37  else if (task_arrayOfreadResult[0].errors === null && task_arrayOfreadResult[0].objects)
38    task_arrayOfreadResult[0].objects.forEach(function(task) {
39      NSOA.meta.log('debug', "Got task: " + JSON.stringify(task));
40
41      // Only consider project task assignments
42      if (!task.projecttaskid)
43        return;
44
45      // Correlate booking <=> project_task_assignment via task tuple(project_task_id,user_id)
46      var uniqueKey = task.projecttaskid;
47
48      // Store information about this time entry
49      if (tasks_by_uniqueKey[uniqueKey])
50        tasks_by_uniqueKey[uniqueKey].push(task);
51      else
52        tasks_by_uniqueKey[uniqueKey] = [task];
53      ts_pta_worked_hours[uniqueKey] += parseFloat(task.decimal_hours);
54
55      // Prepare related booking filters
56      var bookingFilter = new NSOA.record.oaBooking();
57      bookingFilter.project_taskid = task.projecttaskid;
58      bookingFilters[uniqueKey] = bookingFilter; // elimiate duplicates
59
60      // Prepare related project_task_assign filters
61      var ptaFilter = new NSOA.record.oaProjecttaskassign();
62      ptaFilter.projecttaskid = task.projecttaskid;
63      ptaFilters[uniqueKey] = ptaFilter; // elimiate duplicates
64
65    });
66  else
67    return; // assume no data found
68
69  // Now load and analyze project task assignments (one read request)
70  if (Object.keys(ptaFilters).length > 0) {
71    var equalTo = [];
72    for (var i = 0; i < Object.keys(ptaFilters).length; i++)
73      equalTo.push("equal to");
74    var ptaFilter = [];
75    Object.keys(ptaFilters).forEach(function(k) {
76      ptaFilter.push(ptaFilters[k]);
77    });
78    var pta_readRequest = {
79      type: "Projecttaskassign",
80      fields: "id, planned_hours, userid, projecttaskid",
81      method: equalTo.join(', or '),

```



```

82     objects: ptaFilter,
83     attributes: [{
84         name: "limit",
85         value: "1000"
86     }, {
87         name: "filter",
88         value: "current-user"
89     }]
90 };
91 NSOA.meta.log('debug', "pta_readRequest=" + JSON.stringify(pta_readRequest));
92 var pta_arrayOfreadResult = NSOA.wsapi.read(pta_readRequest);
93 NSOA.meta.log('debug', "pta_arrayOfreadResult=" + JSON.stringify(pta_arrayOfreadResult));
94
95 var pta_planned_hours = {};
96 var pta_worked_hours = {};
97 if (!pta_arrayOfreadResult || !pta_arrayOfreadResult[0])
98     NSOA.form.error('', "Internal error loading %project_task% assignment details.");
99 else if (pta_arrayOfreadResult[0].errors === null && pta_arrayOfreadResult[0].objects)
100     pta_arrayOfreadResult[0].objects.forEach(function(pta) {
101         var uniqueKey = pta.projecttaskid;
102         var planned_hours = parseFloat(pta.planned_hours);
103
104         // Skip assignment if no planned hours
105         if (!planned_hours)
106             return;
107
108         // Compute worked hours for current user's assignment
109         var taskFilter = new NSOA.record.oaTask();
110         taskFilter.projecttaskid = pta.projecttaskid;
111         taskFilter.userid = pta.userid;
112         var task_readRequest = {
113             type: "Task",
114             fields: "id, decimal_hours",
115             method: "equal to",
116             objects: [taskFilter],
117             attributes: [{
118                 name: "limit",
119                 value: "1000"
120             }, {
121                 name: "filter",
122                 value: "current-user"
123             }]
124         };
125         var task_arrayOfreadResult = NSOA.wsapi.read(task_readRequest);
126
127         var worked_hours = 0;
128         if (!task_arrayOfreadResult || !task_arrayOfreadResult[0])
129             NSOA.form.error('', "Internal error loading %timeentry% assignment details.");
130         else if (task_arrayOfreadResult[0].errors === null && task_arrayOfreadResult[0].objects)
131             task_arrayOfreadResult[0].objects.forEach(function(task) {
132                 worked_hours += parseFloat(task.decimal_hours);
133             });
134
135         // Verify user's worked hours haven't exceeded as a result of this timesheet
136         NSOA.meta.log('debug', "worked=" + worked_hours + ",planned=" + planned_hours);
137         if (worked_hours && worked_hours > planned_hours) {
138             var pt = NSOA.record.oaProjecttask(pta.projecttaskid);
139             var error = "Worked %hours% (" + worked_hours + ") including %timeentry% on this %timesheet% exceeds your
planned %hours% (" + planned_hours + ") for %project% '" + NSOA.record.oaProject(pt.projectid).name + "' %project_task% '" +
pt.name + "'.";
140             if (ts_pta_worked_hours[uniqueKey]) {
141                 error += "This %timesheet% adds " + ts_pta_worked_hours[uniqueKey] + " %hours%.";
142                 var worked_excluding_ts = worked_hours - ts_pta_worked_hours[uniqueKey];
143                 if (worked_excluding_ts <= planned_hours)
144                     error += "Please reduce your worked %hours% by " + (worked_hours - planned_hours) + ".";
145             }
146             NSOA.form.error('', error);
147         }
148     });
149 }
150
151 // Now load and analyze bookings
152 var df = require('lib_date_format');
```

```

153 |     if (Object.keys(bookingFilters).length > 0) {
154 |         var equalTo = [];
155 |         for (var i = 0; i < Object.keys(bookingFilters).length; i++)
156 |             equalTo.push("equal to");
157 |         var bookingFilter = [];
158 |         Object.keys(bookingFilters).forEach(function(k) {
159 |             bookingFilter.push(bookingFilters[k]);
160 |         });
161 |         var booking_readRequest = {
162 |             type: "Booking",
163 |             fields: "id, enddate, userid, project_taskid, projectid",
164 |             method: equalTo.join(', or '),
165 |             objects: bookingFilter,
166 |             attributes: [{
167 |                 name: "limit",
168 |                 value: "1000"
169 |             }, {
170 |                 name: "filter",
171 |                 value: "current-user"
172 |             }]
173 |         };
174 |         NSOA.meta.log('debug', "booking_readRequest=" + JSON.stringify(booking_readRequest));
175 |         var booking_arrayOfreadResult = NSOA.wsapi.read(booking_readRequest);
176 |         NSOA.meta.log('debug', "booking_arrayOfreadResult=" + JSON.stringify(booking_arrayOfreadResult));
177 |
178 |         if (!booking_arrayOfreadResult || !booking_arrayOfreadResult[0])
179 |             NSOA.form.error('', "Internal error loading %project_task% assignment details.");
180 |         else if (booking_arrayOfreadResult[0].errors === null && booking_arrayOfreadResult[0].objects)
181 |             booking_arrayOfreadResult[0].objects.forEach(function(booking) {
182 |                 var uniqueKey = booking.project_taskid;
183 |                 NSOA.meta.log('debug', uniqueKey + ", " + JSON.stringify(tasks_by_uniqueKey));
184 |                 var tasks = tasks_by_uniqueKey[uniqueKey];
185 |                 if (!tasks)
186 |                     return;
187 |                 tasks.forEach(function(task) {
188 |                     var taskDate = new Date(task.date.substr(0, 10));
189 |                     taskDate.setDate(taskDate.getDate() + 1);
190 |                     NSOA.meta.log('debug', JSON.stringify(task));
191 |                     var bookingDate = new Date(booking.enddate.substr(0, 10));
192 |                     NSOA.meta.log('debug', "Check: " + taskDate + '>' + bookingDate);
193 |                     if (taskDate && bookingDate) {
194 |                         if (taskDate > bookingDate) {
195 |                             var pt = NSOA.record.oaProjecttask(booking.project_taskid);
196 |                             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 + "'.");
197 |                             return;
198 |                         }
199 |                     }
200 |                 });
201 |             });
202 |         }
203 |     }

```

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 screenshot shows the 'Properties' tab of a project form for 'Project: Cloud connector'. The 'Sales Information' section is expanded, and the 'Quick Customer PO' sub-section is active. It contains the following fields and options:

- Customer PO Number:** Input field with value 'PO-2023-123'. Below it: 'Enter the Customer PO number'.
- Customer PO Amount:** Input field with value '250000'. Below it: 'Enter an amount if different from project budget'.
- Customer PO Date (DD-MM-YYYY):** Input field with value '27-03-2023' and a calendar icon. Below it: 'Enter a date if different from the project start date'.
- Create Customer PO:** A checked checkbox with the label 'Create Customer PO' and a sub-label 'Check to generate a new Customer PO after saving your project'.

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.

The screenshot shows the 'Financials: Customer PO' table. The table has the following columns: Customer PO, Number, Customer, Date, Total (money), and Active. A single record is displayed with the following values:

Customer PO	Number	Customer	Date	Total (money)	Active
PO-2023-123	PO-2023-123	Global Information	23-Mar-27	250,000 USD	✓

Follow the steps below or [download the solutions file](#), see [Creating Solutions](#) for details.

Setup

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 first custom field to **Quick Customer PO** to place the custom fields in their own section.

Position	Display name	Name	Association	Field type	Active
		P	Project	- All -	- All -
11	Create Customer PO	prj_create_po	Project	Checkbox	✓
9	Customer PO Amount	prj_custpo_amt	Project	Text	✓
10	Customer PO Date (MM/DD/YY)	prj_custpo_date	Project	Date	✓
8	Customer PO Number	prj_custpo_num	Project	Text	✓

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

```

1 function createCustomerPO(type) {
2   try {
3     var FLD_CUSTPO_NUM = 'prj_custpo_num_c',
4         FLD_CUSTPO_AMT = 'prj_custpo_amt_c',
5         FLD_CUSTPO_DATE = 'prj_custpo_date_c',

```

```

6      FLD_CREATE_PO = 'prj_create_po_c';
7
8      // get updated project record fields
9      var updPrj = NSOA.form.getNewRecord();
10
11     // if the "Create PO" checkbox is checked and a PO number is entered, create a PO
12     if (updPrj[FLD_CREATE_PO] == '1' && updPrj[FLD_CUSTPO_NUM]) {
13
14         var recCustPO = new NSOA.record.oaCustomerpo();
15         recCustPO.number = updPrj[FLD_CUSTPO_NUM];
16         recCustPO.name = updPrj[FLD_CUSTPO_NUM] + ' ' + updPrj.name;
17
18         // use the PO date if available, otherwise use project start date
19         if (updPrj[FLD_CUSTPO_DATE] != '0000-00-00') {
20             recCustPO.date = updPrj[FLD_CUSTPO_DATE];
21         } else {
22             recCustPO.date = updPrj.start_date;
23         }
24
25         // currency custom fields return ISO-#.##; remove the ISO code and dash
26         var cleanAmt = updPrj[FLD_CUSTPO_AMT].replace(/-\w{3}/, '');
27
28         // use the PO amt if available, otherwise use project budget
29         if (cleanAmt && cleanAmt != '0.00') {
30             recCustPO.total = cleanAmt;
31         } else if (updPrj.budget && updPrj.budget > 0.00) {
32             recCustPO.total = updPrj.budget;
33         }
34
35         recCustPO.currency = updPrj.currency;
36         recCustPO.customerid = updPrj.customerid;
37         recCustPO.active = 1;
38
39         // disable the current user's filter set while script runs
40         NSOA.wsapi.disableFilterSet(true);
41
42         // add the new customer po to the project
43         var custPOResults = NSOA.wsapi.add(
44             [recCustPO]
45         );
46
47         if (!custPOResults || !custPOResults[0]) {
48             NSOA.meta.log('error', 'Unexpected error! Customer PO was not created.');
```

```

49         } else if (custPOResults[0].errors) {
50             custPOResults[0].errors.forEach(function(err) {
51                 NSOA.meta.log('error', 'Error: ' + err.code + ' - ' + err.comment);
52             });
53         } else {
54             // new customer po to project link object
55             var recCustPOtoProj = new NSOA.record.oaCustomerpo_to_project();
56             recCustPOtoProj.customerpoid = custPOResults[0].id;
57             recCustPOtoProj.customerid = updPrj.customerid;
58             recCustPOtoProj.projectid = updPrj.id;
59             recCustPOtoProj.active = '1';
60
61             // disable the current user's filter set while script runs
62             NSOA.wsapi.disableFilterSet(true);
63
64             // add the new customer po to the project
65             var custPOtoProjResults = NSOA.wsapi.add(
66                 [recCustPOtoProj]
67             );
68
69             if (!custPOtoProjResults || !custPOtoProjResults[0]) {
70                 NSOA.meta.log('error',
71                     'Unexpected error! Customer PO was not linked to the project.');
```

```

72             } else if (custPOtoProjResults[0].errors) {
73                 custPOtoProjResults[0].errors.forEach(function(err) {
74                     NSOA.meta.log('error', 'Error: ' + err.code + ' - ' + err.comment);
75                 });
76             }
77         }
78     }

```

```

79
80 // create project object to hold update information and clear quick po
81 var recProj = new NSOA.record.oaProject(updPrj.id);
82 recProj[FLD_CUSTPO_NUM] = '';
83 recProj[FLD_CREATE_PO] = '';
84 recProj[FLD_CUSTPO_AMT] = '';
85 recProj[FLD_CUSTPO_DATE] = '';
86
87 // disable the current user's filter set while script runs
88 NSOA.wsapi.disableFilterSet(true);
89
90 // enable custom field editing
91 var update_custom = {
92   name: 'update_custom',
93   value: '1'
94 };
95
96 // update the project to clear quick customer po create information
97 var projResults = NSOA.wsapi.modify(
98   [update_custom], [recProj]
99 );
100
101 if (!projResults || !projResults[0]) {
102   NSOA.meta.log('error', 'Unexpected error! Project was not updated.');
```

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).

New timesheet

Cancel **SAVE**

General

Attachments

Timesheet starting date
08/01/14

Notes

Allow overlapping timesheets

Prefill from task assignments

Follow the steps below or [download the solutions file](#), see [Creating Solutions](#) for details.

Setup

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 **populate_ts_from_assignments** as the **Entrance Function**.

Scripting Studio

Association
Timesheet [New]

Employee
Collins, Marc

References

ALL 4 SELECTED 0

Event
After save

Entrance function
populate_ts_from_assignments

No log messa

```

1 var
2 fun
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25 }
26
27
28 //f
29 fun
30
31
32
33
34
35
36
37
38

```

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.

Name	Association	Field type	Display name	Active
All	Timesheet	All		All
ts_prefill_from_task_assignments	Timesheet	Checkbox	Prefill from task assignments	✓

- 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.

No log messages

1 var CUST_FIELD = 'ts_prefill_from_task_assignments__c';

2 function prepopulate_ts_from_assignments(type) {

3

4 var ts = NSOA.form.getValue(CUST_FIELD);

5

6 // if the checkbox is not ticket, exit

7 if (ts === false) {

▼ **Form schema**

Fields [\[View by param\]](#)

Accounting date [acct_date] <Date>

Default Client : Project [customer_project] <String>

Default Task [project_task_id] <Number>

Default Time type [timetype] <String>

Notes [notes] <String>

Prefill from task assignments [ts_prefill_from_task_assignments__c (custom_33)] <Boolean>

the current user

NSOA.wsapi.whoami();

✓ **Tip:** If the new custom field is not listed in the **Form schema**, navigate to Timesheets, create a new Timesheet form without saving (this will refresh the custom field list), and then open the Scripting Studio again.

Program Listing

```

1 var CUST_FIELD = 'ts_prefill_from_task_assignments__c'; // Use Form schema to find param name
2 function prepopulate_ts_from_assignments(type) {
3
4     var ts = NSOA.form.getValue(CUST_FIELD);
5
6     // if the checkbox is not ticked, exit
7     if (ts === false) {
8         return;
9     }
10
11     // retrieve the current user
12     var user = NSOA.wsapi.whoami();
13
14     //look for current assignments for this user
15     var defaultPerRow = find_assignments(user.id);
16
17     // retrieve the new object
18     var newr = NSOA.form.getNewRecord();
19     var timesheet = new NSOA.record.oaTimesheet();
20
21     timesheet.id = newr.id;
22     timesheet.default_per_row = defaultPerRow;
23
24     var result = NSOA.wsapi.modify([], [timesheet]);
25 }
26
27
28 //find the assignments for this user and return a string for timesheet.default_per_row
29 function find_assignments(userid) {
30
31     // fetch a list of task assignments for the current user
32     var taskAssign = new NSOA.record.oaProjecttaskassign();
33     taskAssign.userid = userid;
34
35     var readTasksAssign = {
36         type: "Projecttaskassign",
37         method: "equal to",

```



```

38     fields: "projecttaskid",
39     attributes: [{
40         name: "limit",
41         value: "0,1000"
42     }],
43     objects: [taskAssign]
44 };
45
46 var CSV = {
47     pt_id: [],
48     cp_id: []
49 };
50 var resultTaskAssign = NSOA.wsapi.read(readTasksAssign);
51
52 // iterate through all the task assignments and filter only current ones
53 // retrieve all tasks that belong to user, started in the past and percent_complete < 100
54 if (resultTaskAssign[0].errors === null && resultTaskAssign[0].objects) {
55
56     for (var i = 0; i < resultTaskAssign[0].objects.length; i++) {
57
58         var projectTask = new NSOA.record.oaProjecttask();
59         projectTask.id = resultTaskAssign[0].objects[i].projecttaskid;
60
61         var readTask = {
62             type: "Projecttask",
63             method: "equal to",
64             fields: "calculated_starts,starts,percent_complete,customerid,id,projectid",
65             attributes: [{
66                 name: "limit",
67                 value: "0,1000"
68             }],
69             objects: [projectTask]
70         };
71
72         var resultTask = NSOA.wsapi.read(readTask);
73
74         // do we have results?
75         if (resultTask[0].errors === null && resultTask[0].objects) {
76             for (var k = 0; k < resultTask[0].objects.length; k++) {
77
78                 var ptStartDate = resultTask[0].objects[k].starts.substring(0, 10);
79
80                 // optimization: skip blank dates
81                 if (ptStartDate === '0000-00-00') {
82                     ptStartDate = resultTask[0].objects[k].calculated_starts.substring(0, 10);
83                     if (ptStartDate === '0000-00-00') {
84                         continue;
85                     }
86                 }
87
88                 var currentDate = new Date();
89                 var starts = new Date(ptStartDate);
90
91                 // do we have a date?
92                 // NSOA.meta.alert('currentdate=' + currentDate + ' starts='+starts);
93
94                 if (starts <= currentDate &&
95                     parseInt(resultTask[0].objects[k].percent_complete, 10) < 100) {
96                     CSV.pt_id.push(resultTask[0].objects[k].id);
97                     CSV.cp_id.push(resultTask[0].objects[k].customerid + ":" +
98                         resultTask[0].objects[k].projectid);
99                 }
100             }
101         }
102     }
103 }
104
105 var retval = "pt_id," + CSV.pt_id.join(",") + "\n";
106 retval = retval + "cp_id," + CSV.cp_id.join(",");
107 return retval;
108
109 }

```

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

Follow the steps below or [download the solutions file](#), see [Creating Solutions](#) for details.

Setup

- Create a new **Budget form script deployment**.
- Enter a **Filename** and click **SAVE**. The extension '.js' is automatically appended if not supplied.
- Click on the script link to launch the **Scripting Studio**.
- (1) Copy the **Program Listing** below into the editor, (2) set the **After save** event, and set **updateProjectBudgetHours** as the **Entrance Function**.

The screenshot shows the Scripting Studio interface for the 'Budget' form. The interface includes the following elements:

- Scripting Studio** header with a red circle '1' next to it.
- Association Budget** section.
- Employee** dropdown menu showing 'Collins, Marc' with a search icon.
- References** section with 'ALL' (4 items) and 'SELECTED' (0 items) tabs, a search bar, and 'Select all' and 'Clear all' links.
- Event** dropdown menu set to 'After save' with a red circle '2' next to it.
- Entrance function** dropdown menu set to 'updateProjectBudgetHours' with a red circle '3' next to it.
- A vertical scroll bar on the right side of the interface showing a list of lines from 1 to 38. Line 1 contains the text 'fun'.

- Set up an **Hours** custom field for **Project** and an **Hours** custom field for **Budget**.

Position	Display name	Name	Association	Field type	Active
1	Budget hours	budget_hours	Budget	Hours	✓
16	Project budget hours	prj_budget_time	Project	Hours	✓

Program Listing

```

1 function updateProjectBudgetHours(type) {
2   try {
3     // DEBUG: Uncomment next line to enable XML logging
4     // var wsLog = NSOA.wsapi.enableLog(true);
5     // list all fields used in script
6     var FLD_PRJ_ID = 'id',
7         FLD_PRJ_BUD_HRS = 'prj_budget_time__c',
8         FLD_BUD_PID = 'projectid';
9
10    // store newly saved budget record
11    var updBudget = NSOA.form.getNewRecord();
12
13    // create new budget object to store information
14    var budRec = new NSOA.record.aaBudget();
15    budRec.projectid = updBudget.projectid;
16    var budRequest = {
17      type: "Budget",
18      method: "equal to",
19      fields: "id,budget_hours__c", // budget_hours__c is a custom hours field
20      attributes: [{
21        name: "limit",
22        value: "100"
23      }],
24      objects: [budRec]
25    };
26
27    // disable the current user's filter set while script runs
28    NSOA.wsapi.disableFilterSet(true);
29
30    // search for all budget transactions with current projectid
31    var budResults = NSOA.wsapi.read(budRequest);
32    if (!budResults || !budResults[0]) {
33      NSOA.meta.log('error', 'Unexpected error! Could not return project budgets. ');
34      return;
35    } else if (budResults[0].errors !== null && budResults[0].errors.length > 0) {
36      prjResults[0].errors.forEach(function(err) {
37        var fullError = err.code + ' - ' + err.comment + ' ' + err.text;
38        NSOA.meta.log('error', 'Error: ' + fullError);
39      });
40      return;
41    }
42    var b,
43        totalBudHrs = 0,
44        budObj = budResults[0].objects,
45        budObjLen = budObj.length;
46    for (b = 0; b < budObjLen; b++) {
47      var budHrs = parseInt(budObj[b].budget_hours__c, 10);
48      totalBudHrs += budHrs; // add all hours together
49    }
50    // create new project object to store information
51    var prjRec = new NSOA.record.aaProject();
52    prjRec[FLD_PRJ_ID] = updBudget[FLD_BUD_PID];
53    prjRec[FLD_PRJ_BUD_HRS] = totalBudHrs;
54
55    // disable the current user's filter set while script runs
56    NSOA.wsapi.disableFilterSet(true);
57
58    // update the project budget
59    var prjResults = NSOA.wsapi.modify(

```

```

60     [{
61         name: "update_custom",
62         value: "1"
63     }], [prjRec]
64     );
65     if (!prjResults || !prjResults[0]) {
66         NSOA.meta.log('error', 'Unexpected error! Project was not updated.');
```

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

Follow the steps below or [download the solutions file](#), see [Creating Solutions](#) for details.

Setup

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**.

- (1) Copy the **Program Listing** below into the editor, (2) set the **Before save** event, and set **validate_vacation** as the **Entrance Function**.

- Set up a **Checkbox** and a **Text Area** custom field for **Booking**.

Position	Display name	Name	Association	Field type	Active
1	Override booking vacation re...	bk_override_vacation	Booking	Checkbox	✓
2	Override reasons	bk_override_vacation_notes	Booking	Text Area	✓

- 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 will refresh the custom field list), and then open the Scripting Studio again.

Program Listing

```

1 // timetype_id depends on account settings
2 var CUST_FIELD = 'bk_override_vacation__c';
3 var CUST_FIELD_NOTES = 'bk_override_vacation_notes__c';
4
5 function validate_vacation() {
6 // To support exception situations where booking should be allowed over scheduled timeoff,
7 // new checkbox custom field with associated notes field has been added to Booking form.
8 // When that field is checked, we want the notes field to be required, so we validate the
9 // custom fields settings at the start.
10 var override = NSOA.form.getValue(CUST_FIELD);
11 var req_notes = NSOA.form.getValue(CUST_FIELD_NOTES);
12
13 // If we are overriding the booking vacation restrictions
14 if (override) {
15 // And the notes field is not set
16 if (!req_notes) { // This is a basic has-a-value check, typically check should be
17 // more extensive, i.e. not blank spaces, of certain length, etc.
18 // Set custom field error message to indicate required, and prevent form saving
19 NSOA.form.error(CUST_FIELD,
20 'When overriding vacation restrictions, notes are required');
21 }
22 return; // Stop, as no further checks are required
23 }
24
25 // getValue returns JS Date objects for Date type fields
26 var start = NSOA.form.getValue('startdate'); // While adding/changing a script,
27 var end = NSOA.form.getValue('enddate'); // the Form Schema section provides a list
28 // of available form fields and the expected
29 // return values of those fields
30
31 // Create the oaSchedulerequest object for the WSAPI read search
32 // Information on available records can be found in the user scripting guide
33 // Note the form field is user_id but the SOAP API field is userid
34 var approvedSchedReq = new NSOA.record.oaSchedulerequest();
35 approvedSchedReq.userid = NSOA.form.getValue('user_id');
36 approvedSchedReq.approval_status = 'A'; // (A)pproved, (O)pen, (S)ubmitted, (R)ejected
37 approvedSchedReq.timetypeid = '5'; // Personal time is timetypeid 5
38
39 // Pull the start and end dates for Schedulerequests that match our criteria
40 var aPTO = NSOA.wsapi.read({
41 type: 'Schedulerequest', // The SOAP API complex type
42 method: 'equal to',
43 fields: 'startdate,enddate', // start & end fields for Schedulerequest complex type
44 attributes: [{
45 name: 'limit', // ReadRequest objects must have a limit specified
46 value: '100' // '100' returns up to 100, '50,100' returns 50 - 100
47 }],
48 objects: [approvedSchedReq] // The previously created search object
49 });
50
51 // NSOA.wsapi.read() returns an array of objects with error and objects properties
52 for (x = 0; x < aPTO.length; x++) {
53 // If there were errors, notify the user and stop
54 if (aPTO[x].errors) {
55 var errorMsg = '';
56 for (i = 0; i < aPTO[x].errors.length; i++) {
57 errorMsg += 'SOAP error [' + aPTO[x].errors[i].code + ']:';
58 errorMsg += aPTO[x].errors[i].text + ' - ';
59 errorMsg += aPTO[x].errors[i].comment + "\n";
60 }
61 NSOA.form.error('', errorMsg); // Set the main form error message with the details
62 return;

```

```

63     }
64
65     // If there were approved personal Schedulerequest objects found
66     if (aPTO[x].objects) {
67         NSOA.meta.alert(aPTO[x].objects.length);
68         // Loop through and validate the time off doesn't overlap booking request period
69         for (i = 0; i < aPTO[x].objects.length; i++) {
70             var thisStart = convertToDate(aPTO[x].objects[i].startdate);
71             var thisEnd = convertToDate(aPTO[x].objects[i].enddate);
72
73             // If the PTO overlaps the start of the period
74             if ((thisStart <= start && thisEnd <= end && thisEnd >= start) ||
75                 (thisStart <= start && thisEnd >= end) || // Or overlaps whole period
76                 (thisStart >= start && thisEnd <= end) || // Or is wrapped by the period
77                 (thisStart >= start && thisStart <= end && thisEnd >= end)) { // Or end
78                 var malDate;
79                 if (thisStart.getTime() == thisEnd.getTime()) { // If the is a single day
80                     malDate = thisStart.toDateString(); // Only display one date
81                 } else { // Else start/end range
82                     malDate = thisStart.toDateString() + ' - ' + thisEnd.toDateString();
83                 }
84
85                 // Set the form startdate error message accordingly, then stop.
86                 NSOA.form.error('startdate', 'The requested resource has approved personal time off' + ' during the selected
booking period: ' + malDate);
87                 return;
88             }
89         }
90     }
91 }
92 }
93
94 // Helper function for converting date strings to JS Date objects
95 function convertToDate(vDate) {
96     // Expected date format is a string: YYYY-MM-DD 0:0:0
97     var aYMD = vDate.split(' ');
98     aYMD = aYMD[0].split('-');
99     return new Date(aYMD[0], aYMD[1] - 1, aYMD[2]);
100 }

```

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

Follow the steps below or [download the solutions file](#), see [Creating Solutions](#) for details.

Setup

- Create two parameters, see [Creating Parameters](#).

Script deployments		
Form	Scheduled	Library
Parameters		
All <input type="button" value="v"/>		
Name	Description	Type
All <input type="button" value="v"/>		All <input type="button" value="v"/>
ProjectClosedStage	Project closed stage	Pick List
IssueOpenStage	Issue open stage	Pick List

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

```

1 // project_stage_id and issue_stage_id depend on account settings
2 function test_prevent_project_close_with_open_issue() {
3
4     // return if new stage is not closed
5     if (NSOA.fozm.getValue('project_stage_id') !=
6         NSOA.context.getParameter('ProjectClosedStage'))
7         return;
8
9     // Load issue data
10    var issue = new NSOA.record.oaIssue();
11    issue.project_id = NSOA.fozm.getValue('id');
12    issue.issue_stage_id = NSOA.context.getParameter('IssueOpenStage');
13
14    var readRequest = {
15        type: "Issue",
16        fields: "id, date",

```



```

17     method: "equal to",
18     objects: [issue],
19     attributes: [{
20         name: "limit",
21         value: "1"
22     }]
23 };
24
25 var arrayOfreadResult = NSOA.wsapi.read(readRequest);
26
27 if (!arrayOfreadResult || !arrayOfreadResult[0])
28     NSOA.form.error('', "Internal error analyzing project issues.");
29
30 else if (arrayOfreadResult[0].errors === null && arrayOfreadResult[0].objects)
31     arrayOfreadResult[0].objects.forEach(
32         function(o) {
33             NSOA.form.error('', "Can't close project with open issues.");
34         }
35     );
36 }


```

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 Administrator Guide](#) for more details on Project Stages.

Setup 1 — Custom Field

1. Set up a **Checkbox** and a **Text Area** custom field for **Issue**.

Position	Display name	Name	Association	Field type	Active
8	Project at risk	for_at_risk_project	Project	Checkbox	✓

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

```

1 function proj_at_risk_aftersave() {
2   var PROJECT_STAGE_AT_RISK = NSOA.context.getParameter('ProjectAtRiskStage');
3   var ISSUE_STAGE_OPEN = NSOA.context.getParameter('IssueOpenStage');
4
5   // return if new stage is changed and "at risk"
6   var proj = NSOA.form.getNewRecord();
7   var old_stage = NSOA.form.getOldRecord().project_stageid;
8   var current_stage = proj.project_stageid;
9   NSOA.meta.log("debug", "old=" + old_stage + ", new=" + current_stage);
10  if (old_stage == current_stage || current_stage != PROJECT_STAGE_AT_RISK)
11    return;
12
13  // Check for an existing at-risk event
14  var issue = new NSOA.record.oaIssue();
15  issue.project_id = proj.id;
16  issue.for_at_risk_project_c = '1';
17
18  var readRequest = {
19    type: "Issue",
20    fields: "id, name, date",

```

```

21     method: "equal to",
22     objects: [issue],
23     attributes: [{
24         name: "limit",
25         value: "1"
26     }]
27 };
28
29 var arrayOfreadResult = NSOA.wsapi.read(readRequest);
30 if (!arrayOfreadResult || !arrayOfreadResult[0])
31     NSOA.form.error('', "Internal error analyzing project issues.");
32
33 else if (arrayOfreadResult[0].errors === null &&
34     (!arrayOfreadResult[0].objects || arrayOfreadResult[0].objects.length === 0)) {
35     issue.owner_id = NSOA.wsapi.whoami().id;
36     issue.description = "Projected reported at risk";
37     issue.issue_status_id = 1; // Unassigned
38     issue.issue_stage_id = ISSUE_STAGE_OPEN;
39     issue.date = (new Date()).toISOString().slice(0, 10);
40     NSOA.meta.log('debug', JSON.stringify(issue));
41     NSOA.wsapi.add(issue);
42 }
43 }

```

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**.

Scripting Studio

Association
Project

Employee
Collins, Marc

Execution displays internal form script deployment log error debug detail for this user

References
ALL 4 SELECTED 0

Event
Before save

Entrance function
proj_at_risk_beforesave_validate

No log mess

1 fun
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38

Program Listing for Setup 3

```

1 function proj_at_risk_beforesave_validate() {
2   var PROJECT_STAGE_AT_RISK = NSOA.context.getParameter('ProjectAtRiskStage');
3   var ISSUE_STAGE_OPEN = NSOA.context.getParameter('IssueOpenStage');
4
5   // return if new stage is not changing from "at risk"
6   var current_stage = NSOA.form.getOldRecord().project_stageid;
7   var new_stage = NSOA.form.getValue('project_stage_id');
8   if (!(current_stage == PROJECT_STAGE_AT_RISK && new_stage != PROJECT_STAGE_AT_RISK))
9     return;
10
11  // Load issue data
12  var issue = new NSOA.record.oaIssue();
13  issue.project_id = NSOA.form.getValue('id');
14  issue.issue_stage_id = ISSUE_STAGE_OPEN;
15  issue.for_at_risk_project_c = '1';
16
17  var readRequest = {
18    type: "Issue",
19    fields: "id, name, date",
20    method: "equal to",
21    objects: [issue],
22    attributes: [{
23      name: "limit",
24      value: "1"
25    }]
26  };
27
28  var arrayOfreadResult = NSOA.wsapi.read(readRequest);
29
30  if (!arrayOfreadResult || !arrayOfreadResult[0])
31    NSOA.form.error('', "Internal error analyzing project issues.");
32
33  else if (arrayOfreadResult[0].errors === null && arrayOfreadResult[0].objects)
34    arrayOfreadResult[0].objects.forEach(
35      function(o) {
36        NSOA.form.error('', "Can't change project stage until " +
37          "the following issue is resolved: " + o.name);
38      }
39    );
40 }

```

Send an alert email when a scheduled script completes

This script informs a user when a scheduled script completes successfully.

Follow the steps below or [download the solutions file](#), see [Creating Solutions](#) for details.

Setup

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**.

The screenshot shows the Scripting Studio interface. At the top left, there is a dropdown menu for 'Employee' with 'Collins, Marc' selected. Below it, a search icon and a note: 'Execution displays internal scheduled script deployment log error debug detail for this user'. Under 'References', there are two tabs: 'ALL' (with a count of 4) and 'SELECTED' (with a count of 0). A search bar and 'Select all' / 'Clear all' links are present. At the bottom left, there are two dropdown menus: 'Event' set to 'Schedule' and 'Entrance function' set to 'main'. On the right, a log viewer shows a list of lines from 1 to 14. Line 1 contains 'fun' and line 14 contains '}'. A red circle with the number '1' is placed over the top right corner of the interface. Another red circle with the number '2' is placed over the 'Schedule' dropdown, and a third red circle with the number '3' is placed over the 'main' dropdown.

Program Listing

```

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
16    NSOA.meta.sendMail(msg);
17 }

```

Send a Slack notification when issues are created or (re)assigned

This script sends a notification on a specified Slack channel whenever an issue is created or modified.

OpenAir Scripting API ▼ 🔔
 ● Marc Collins

Threads

Channels +

- # general
- # random
- # slack-updates

+ Add a channel

Direct Messages +

- ♥ Slackbot
- Marc Collins (you)
- Ed Ellis
- Marie Porter

+ Invite people

Apps +

- OpenAir Notifier

#general 👤 3 🔒 0 | Company-wide announcement 📞 ℹ️ ⚙️ @ ☆ ⋮

Today

OpenAir Notifier APP 9:03 AM

An issue has been created on **Altima Technology : ERP deployment**.

- New Issue
- Issue**
IS-42: Project delayed beyond tolerance
- Customer : Project**
Altima Technology : ERP deployment
- Severity** **Stage**
S1 Open
- Assigned to**
Collins, Marc
- Notes**
🔗 OpenAir User Scripting API | Today at 9:03 AM

Issue **IS-36** has been modified.

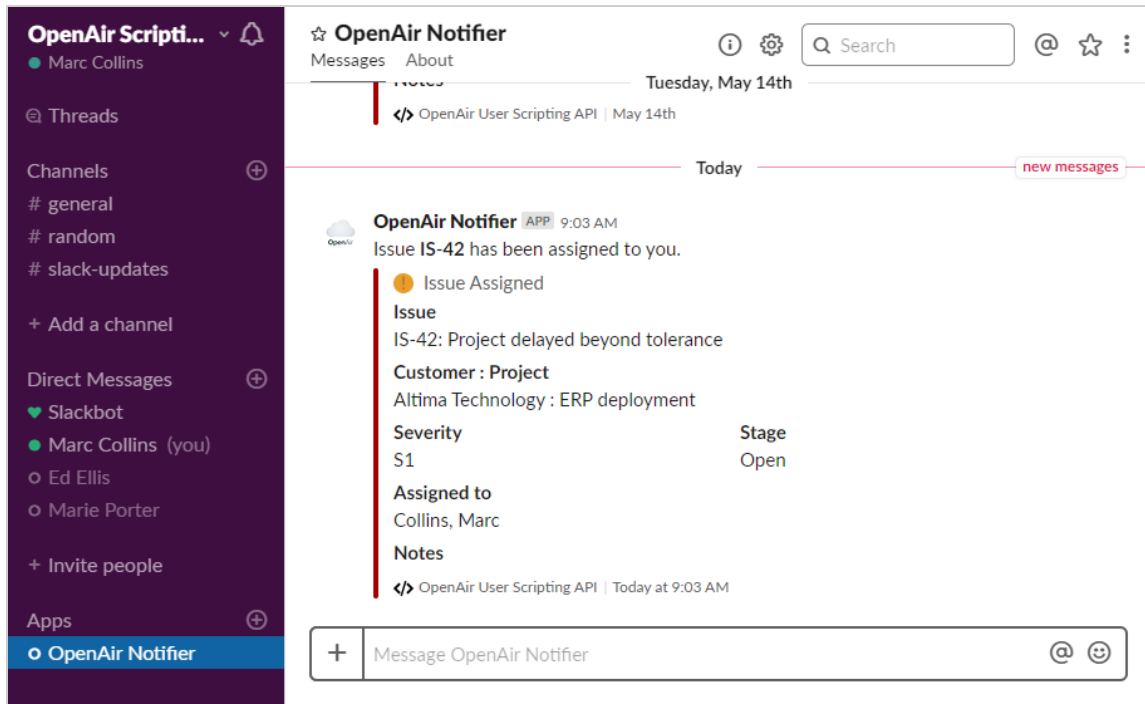
- Issue Updated
- Issue**
IS-36: Resources redeployment
- Customer : Project**
Altima Technology : ERP deployment
- Severity** **Stage**
S1 Open
- Assigned to**
Adams, Mary
- Notes**
🔗 OpenAir User Scripting API | Today at 9:05 AM

Issue **IS-36** is Resolved.

- ✔ Issue Resolved
- Issue**
IS-36: Resources redeployment
- Customer : Project**
Altima Technology : ERP deployment
- Severity** **Stage**
S1 Resolved
- Assigned to**
Adams, Mary
- Notes**
🔗 OpenAir User Scripting API | Today at 9:05 AM

+ Message #general @ 😊

The script also sends a direct message from the Slack app bot to notify employees whenever an issue has been assigned to them.



Two different methods are used to post the messages. An incoming webhook is used to post messages on a specific Slack channel. Slack API methods `conversations.open` and `chat.postMessage` are used to post direct messages to users from the app bot user. The script also uses the Slack API method `users.lookupByEmail` to identify the Slack users direct messages should be posted to.

Follow the steps below or [download the solutions file](#), see [Creating Solutions](#) for details.

Note: You will need to create, configure and install a Slack app associated to your workspace. See [Setup 1 — Create, Configure and Install a Slack App](#).

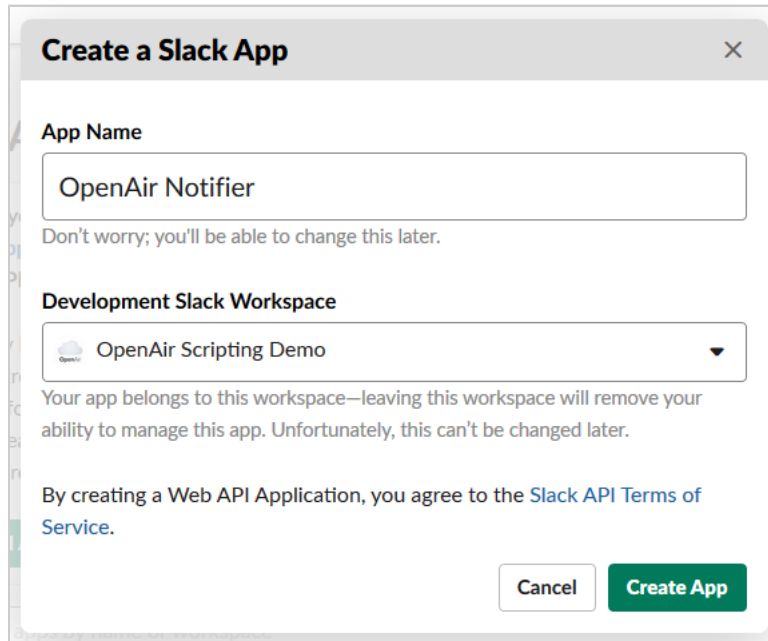
Four setup steps are required for this example

- Configure the Slack App used by the scripts to send notifications. See [Setup 1 — Create, Configure and Install a Slack App](#).
- Define and set 3 parameters used by the library script. See [Setup 2 — Script Parameters](#).
- Create a library script to be used by the two form scripts. See [Setup 3 — Slack Notification Library Script](#).
- Create scripts associated to the Issue and the Project Issue forms. See [Setup 4 — Issue After Save / Project Issue After Save](#)

Setup 1 — Create, Configure and Install a Slack App

1. Open the Slack desktop application and login to your Slack workspace using an Administrator account.
2. Click your workspace name in the top left.
3. Select **Administration > Manage apps**. This will open the app directory for your workspace on a new tab in your default web browser.
4. Click **Build** on the top right. This will redirect you to the Slack API documentation web page.

5. Click **Start Building**. The Create a Slack App dialog appears.
6. Enter the name of your application and the workspace associated with the app.



Create a Slack App ✕

App Name

OpenAir Notifier

Don't worry; you'll be able to change this later.

Development Slack Workspace

OpenAir Scripting Demo

Your app belongs to this workspace—leaving this workspace will remove your ability to manage this app. Unfortunately, this can't be changed later.

By creating a Web API Application, you agree to the [Slack API Terms of Service](#).

7. Click **Create App**. You will be redirected to the Basic Information screen for your Slack app.
8. Click **Incoming Webhooks** under **Add Features and Functionality**. The Incoming Webhooks screen will display.
9. Enable the **Activate Incoming Webhooks** toggle.
10. Navigate back to the Basic Information page.
11. Click **Bots** under **Add Features and Functionality**. The Bot User screen will display.
12. Click **Add a Bot User** and enter a Display name and a Default username for your bot. Optionally, enable the **Always Show My Bot as Online** toggle.

Bot User

You can bundle a bot user with your app to interact with users in a more conversational manner. Learn more about [how bot users work](#).

Display name

Names must be shorter than 80 characters, and can't use punctuation (other than apostrophes and periods).

Default username

If this username isn't available on any workspace that tries to install it, we will slightly change it to make it work. Usernames must be all lowercase. They cannot be longer than 21 characters and can only contain letters, numbers, periods, hyphens, and underscores.

Always Show My Bot as Online On

When this is off, Slack automatically displays whether your bot is online based on usage of the RTM API.

Add Bot User

13. Click **Add Bot User**. A confirmation message displays.
14. Navigate back to the Basic Information page and click **Permissions** under **Add Features and Functionality**. The OAuth & Permissions screen will display.
15. Scroll down to the Scopes section. The following permission scopes should already be listed:
 - Post to a specific channel in Slack (incoming-webhook)
 - Add a bot user with the username `@yourbot_default_username` (bot)
16. Use the **Select permission scopes** dropdown to add the following permission scopes:
 - Send messages as `Yourbot_Display_Name` (chat:write:bot)
 - Access your workspace's profile information (users:read)
 - View email addresses of people on this workspace (users:read.email)

Scopes

Scopes define the [API methods](#) this app is allowed to call, and thus which information and capabilities are available on a workspace it's installed on. Many scopes are restricted to specific [resources](#) like channels or files.

If your app is submitted to the Slack App Directory, we'll review your reasons for requesting each scope. After your app is listed in the Directory, it will only be able to use permission scopes Slack has approved.

Select Permission Scopes

Add permission by scope or API method...

CONVERSATIONS

- Send messages as OpenAir Notifier
chat:write:bot
- Post to specific channels in Slack
incoming-webhook

INTERACTIVITY

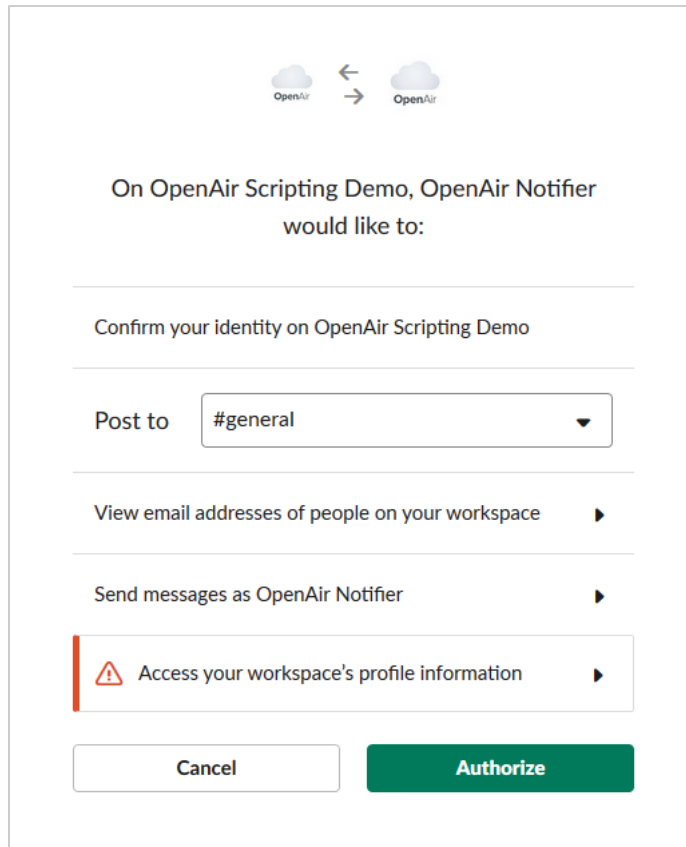
- Add a bot user with the username @oanotifier
bot

USERS

- Access your workspace's profile information
users:read
- View email addresses of people on this workspace
users:read.email

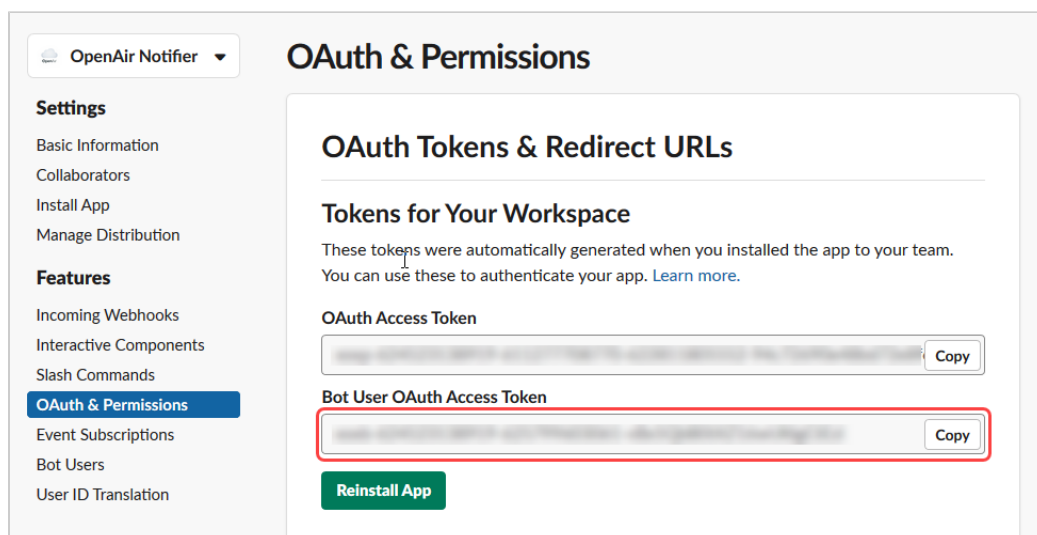
Save Changes

- Click **Save Changes**.
- Scroll back to the top of the page and click **Install App to Workspace** under OAuth Tokens & Redirect URLs. A new screen will display.
- Review the permission scopes for the app you are about to install and select the channel messages will be posted to using the Incoming Webhook.



In this example, Slack users are identified using their email address. This requires adding the permission scopes `users:read` and `users:read.email`. The `users:read` permission scope enables the app to access profile information for all users on the Slack workspace. If this is not desirable, an alternative method to identify users for sending direct messages would be to use a custom field in OpenAir to store a Slack user ID in the User records.

20. Click **Authorize**. You will be redirected back to the OAuth & Permissions screen.
21. Take a note of the **Bot User OAuth Access Token** under OAuth Tokens & Redirect URLs.



- Click the **Incoming Webhook** link on the left and take a note of the **Webhook URL**. You will need this when setting the script parameters in [Setup 2 — Script Parameters](#).

Incoming Webhooks

Settings

- Basic Information
- Collaborators
- Install App
- Manage Distribution

Features

- Incoming Webhooks**
- Interactive Components
- Slash Commands
- OAuth & Permissions
- Event Subscriptions
- Bot Users
- User ID Translation

Slack ♥
Help
Contact
Policies
Our Blog

Activate Incoming Webhooks On

Incoming webhooks are a simple way to post messages from external sources into Slack. They make use of normal HTTP requests with a JSON payload, which includes the message and a few other optional details. You can include [message attachments](#) to display richly-formatted messages.

Each time your app is installed, a new Webhook URL will be generated.

If you deactivate incoming webhooks, new Webhook URLs will not be generated when your app is installed to your team. If you'd like to remove access to existing Webhook URLs, you will need to [Revoke All OAuth Tokens](#).

Webhook URLs for Your Workspace

To dispatch messages with your webhook URL, send your [message](#) in JSON as the body of an `application/json` POST request.

Add this webhook to your workspace below to activate this curl example.

Sample curl request to post to a channel:

```
curl -X POST -H 'Content-type: application/json' --data '{"text":"Hello, World!"}' https://hooks.slack.com/services/...
```

Webhook URL **Channel** **Added By**

Webhook URL	Channel	Added By
https://hooks.slack.com/services/... Copy	#general	Marc Collins May 4, 2019 🗑️

[Add New Webhook to Workspace](#)

Posting messages on a specified Slack channel as an app bot user can be done using a **Webhook URL**. In order to post direct messages to Slack users as an app bot user, a **Bot User OAuth Access Token** is required. You will need to set these as script parameters in [Setup 2 — Script Parameters](#).

Setup 2 — Script Parameters

- Log in to OpenAir as an Administrator and go to the OpenAir Scripting Center.
- Create and set the following Password parameters:
 - SlackBotOAuthAccessToken** — Set the value for this parameter to the Bot User OAuth Access Token you noted in [Setup 1 — Create, Configure and Install a Slack App](#).
 - SlackWebhookUrlForIssuesNotifications** — Set the value for this parameter to the Webhook URL you noted in [Setup 1 — Create, Configure and Install a Slack App](#).

For more information about creating parameters, see [Creating Parameters](#).

- Create and set the following Text parameter:
 - SlackUrl** — Set the value for this parameter to your Slack workspace URL (e.g. `https://myslackworkspace.slack.com`).

Description	Name	Type	Value
	S	All	
Webhook URL for the Slack channel Issues Notifications will be posted to	SlackWebhookUrlForIssuesNotifications	Password	[encrypted]
Your Slack Workspace URL (e.g. https://myslackworkspace.slack.com)	SlackUrl	Text	https://openairscripdingdemo.slack.com
Authentication token for your Slack Application Bot	SlackBotOAuthAccessToken	Password	[encrypted]

The parameters created will be referenced in the library script created in [Setup 3 — Slack Notification Library Script](#).

Setup 3 — Slack Notification Library Script

1. Create a new Library script deployment with the filename SlackMessageReIssues.js
For more information about creating library scripts, see [Creating Library Scripts](#).
2. Locate and open the library script you have just created.
3. Reference the three parameters created in [Setup 2 — Script Parameters](#).
4. Copy the script below and paste it in the Scripting Studio editor.

```

1  /*
2
3  LIBRARY SCRIPT USED FOR ISSUE AND PROJECT ISSUE FORM SCRIPTS
4
5  SLACK MESSAGING REFERENCE
6  > https://api.slack.com/messaging
7
8  SLACK MESSAGE ATTACHMENT REFERENCE
9  > https://api.slack.com/docs/message-attachments
10
11 */
12
13 /**
14  * Post a message to slack after creating or modifying an issue.
15  * @param {Str} type Standard entrance function type.
16  */
17
18 function postIssuesOnSlack(type) {
19
20     // Retrieve parameters required to post to the Slack workspace or channel
21
22     var slackBotAuth = NSOA.context.getParameter('SlackBotOAuthAccessToken');
23     var slackUrl = NSOA.context.getParameter('SlackUrl');
24     var slackWebhook = NSOA.context.getParameter('SlackWebhookUrlForIssuesNotifications');
25
26     // Only proceed if all the required parameters have been set
27
28     if (!slackBotAuth || slackBotAuth.length === 0 || !slackUrl || slackUrl.length === 0 || !slackWebhook || slackWebhook.length === 0) { return; }
29
30     // Only proceed if the issue record is new or has been modified
31     var issue = NSOA.form.getNewRecord();
32     var issueReAssigned = false;
33     if (type !== 'new') {
34         // Get issue record from database with the newly saved values and the previous values
35         var issueOld = NSOA.form.getOldRecord();
36         if (issue.updated === issueOld.updated) {return;}
37         if (issue.user_id !== issueOld.user_id) issueReAssigned = true;
38     }
39
40     // Record the SOAP API requests and responses in the log
41     NSOA.wsapi.enableLog(true);
42
43     // Execute the script independently of user filter sets
44     NSOA.wsapi.disableFilterSet(true);
45
46     // Get user, project, issue severity and issue stage records associated with the issue

```

```

47 var user = NSOA.record.oaUser(issue.user_id);
48 var project = NSOA.record.oaProject(issue.project_id);
49 var issueseverity = NSOA.record.oaIssueSeverity(issue.issue_severity_id);
50 var issuestage = NSOA.record.oaIssueStage(issue.issue_stage_id);
51
52 // Construct Slack messages content and attachments
53
54 var issName = issue.name;
55 var issDescr = issue.description;
56 var issSeverity = issueseverity.name;
57 var issStage = issuestage.name;
58 var issNotes = issue.issue_notes;
59 var issAssignee = user.name;
60 var prjName = project.name;
61 var prjCustName = project.customer_name;
62 var issUpdated = issue.updated;
63 var issCreated = issue.created;
64
65 var createdEpoch = convertTimestampToEpoch(issUpdated);
66
67 var attachmenticon = 'https://www.pngrepo.com/download/30662/warning.png';
68 var messagetext = 'Issue *' + issue.name + '* has been modified.';
69 var attachmenttitle = 'Issue Updated';
70 var attachmentcolor = 'warning';
71
72 if (type === 'new') {
73   attachmenttitle = 'New Issue';
74   messagetext = 'An issue has been created on *' + prjCustName + ': ' + prjName + '*.';
75   attachmentcolor = 'danger';
76   createdEpoch = convertTimestampToEpoch(issCreated);
77 }
78
79 if (issuestage.considered_closed) {
80   messagetext = 'Issue *' + issue.name + '* is ' + issuestage.name + '.';
81   attachmenttitle = 'Issue ' + issuestage.name;
82   attachmentcolor = 'good';
83   issNotes = issue.resolution_notes;
84   attachmenticon = 'https://www.pngrepo.com/download/167754/checked.png';
85 }
86
87 var fields = [
88   {
89     title: 'Issue',
90     value: issName + ': ' + issDescr,
91     short: false
92   },
93   {
94     title: 'Customer : Project',
95     value: prjCustName + ': ' + prjName,
96     short: false
97   },
98   {
99     title: 'Severity',
100    value: issSeverity,
101    short: true
102  },
103  {
104    title: 'Stage',
105    value: issStage,
106    short: true
107  },
108  {
109    title: 'Assigned to',
110    value: issAssignee,
111    short: false
112  },
113  {
114    title: 'Notes',
115    value: issNotes,
116    short: false
117  }
118 ];
119

```

```

120     var issueattachment = {
121         fallback: messagetext,
122         color: attachmentcolor,
123         author_name: attachmenttitle,
124         author_icon: attachmenticon,
125         fields: fields,
126         footer: 'OpenAir User Scripting API',
127         footer_icon: 'https://www.pngrepo.com/download/36709/programming-code-signs.png',
128         ts: createdEpoch
129     };
130
131     // Post message onto slack channel using Webhook URL
132     var response = postMessageToSlackChannel(slackWebhook, messagetext, [issueattachment]);
133
134     // Post direct message to assignee if issue newly (re)assigned
135     if (((type === 'new' && issue.user_id) || issueReAssigned) && slackBotAuth) {
136
137         var assignedmessagetext = 'Issue *' + issue.name + '* has been assigned to you.';
138
139         var issueassignedattachment = {
140             fallback: assignedmessagetext,
141             color: 'danger',
142             author_name: 'Issue Assigned',
143             author_icon: 'https://www.pngrepo.com/download/30662/warning.png',
144             fields: fields,
145             footer: 'OpenAir User Scripting API',
146             footer_icon: 'https://www.pngrepo.com/download/36709/programming-code-signs.png',
147             ts: createdEpoch
148         };
149
150         response = postSlackDirectMessage (slackUrl, slackBotAuth, assignedmessagetext, user.addr_email, [issueassigned
151         dattachment]);
152     }
153 }
154 exports.postIssuesOnSlack = postIssuesOnSlack;
155
156 /**
157  * Post a message to a slack channel using a webhook URL.
158  * @param {Str} url      The webhook url to post a message on a specific channel (required).
159  * @param {Str} text     Text to display on message (required).
160  * @param {Array} attachments Array of attachment objects - must be provided if text param empty (optional).
161  * @return {Obj}        An https.post response object.
162  */
163 function postMessageToSlackChannel (url, text, attachments) {
164
165     // Check that url parameter has a value, otherwise return
166     url = url || '';
167     if (!url || url.length === 0) { return null; }
168
169     // Check there's something to post, otherwise return
170     text = text || '';
171     if (!text || text.length === 0 || !(attachments && Object.prototype.toString.call(attachments) === '[object
172     Array]')) { return null; }
173
174     var body = {};
175
176     //If text param is provided and not empty
177     if (text && text.length!==0) { body.text = text; }
178
179     // If attachments param is provided, and it is of type Array (isArray method isn't supported...)
180     if (attachments && Object.prototype.toString.call(attachments) === '[object Array]') { body.attachments = attach
181     ments; }
182
183     var headers = {
184         'Content-Type': 'application/json'
185     };
186
187     var response = NSOA.https.post({
188         url: url,
189         body: body,
190         headers: headers
191     });

```

```

190     return response;
191 }
192 }
193
194 /**
195  * Post a Direct Message on Slack as bot using the Slack Web API.
196  * @param {Str}    url        The url for the slack workspace (required).
197  * @param {Str}    auth       Authorization token (required)
198  * @param {Str}    text       Text to display on message (required).
199  * @param {Str}    recipient  The email address of the recipient (required).
200  * @param {Array} attachments Array of attachment objects - must be provided if text param empty (optional).
201  * @return {Obj}           An https.post response object.
202  */
203
204 function postSlackDirectMessage (url, auth, text, recipient, attachments) {
205
206     // Check there's a message to post, otherwise return
207     text = text || '';
208     if (!text || text.length === 0 || !(attachments && Object.prototype.toString.call(attachments) === '[object Array]')) { return null; }
209
210     // Get recipient Slack User ID if found, otherwise return
211     var recipientId = getSlackUserId (url, auth, recipient);
212
213     if (!recipientId) { return null; }
214
215     //Construct headers
216     var headers = {
217         'Content-Type': 'application/json',
218         'Authorization': 'Bearer ' + auth
219     };
220
221     // Open Conversation and get Slack Channel ID - return if Slack Channel not identified
222     var request = {
223         url : url + '/api/conversations.open',
224         body: { users: recipientId },
225         headers: headers
226     };
227
228     var response = NSOA.https.post(request);
229
230     if (!response.body.channel.id) { return null; }
231
232     var channelId = response.body.channel.id;
233
234     //Construct body
235     var body = {channel: channelId};
236
237     //If text param is provided and not empty, append to body
238     if (text && text.length!="" ) { body.text = text; }
239
240     // If attachments param is provided, and it is of type Array (isArray method isn't supported...), append to body
241     if (attachments && Object.prototype.toString.call(attachments) === '[object Array]') { body.attachments = attachments; }
242
243     request = {
244         url: url + '/api/chat.postMessage',
245         body: body,
246         headers: headers
247     };
248
249     response = NSOA.https.post(request);
250
251     return response;
252 }
253
254 /**
255  * Lookup Slack user ID by email.
256  * @param {Str}    url        The url for the slack workspace (required).
257  * @param {Str}    auth       Authorization token (required)
258  * @param {Str}    email      The email address of the user (required).
259  * @return {Str}           A Slack user ID.
260  */

```



```

261 function getSlackUserId (url, auth, email) {
262
263     // Check that url parameter has a value, otherwise return
264     url = url || '';
265     if (!url || url.length === 0) { return null; }
266
267     // Check that auth parameter has a value, otherwise return
268     auth = auth || '';
269     if (!auth || auth.length === 0) { return null; }
270
271     // Check that email parameter has a value, otherwise return
272     email = email || '';
273     if (!email || email.length === 0) { return null; }
274
275     // Get recipient Slack User ID if found, otherwise return
276
277     var request = {
278         url: url + '/api/users.lookupByEmail?token=' + auth + '&email=' + email,
279         headers: {'Content-type': 'application/x-www-form-urlencoded'}
280     };
281
282     var response = NSOA.https.get(request);
283
284     if (response.body.ok) {return response.body.user.id;}
285     else return null;
286 }
287
288 /**
289  * Converts an OpenAir datetime string into a javascript date object.
290  * @private
291  * @param {Str} dateStr Datetime string.
292  * @return {Obj}      Date object.
293  */
294 function _convertStringToDateParts(dateStr) {
295     var regEx = /^(\d{4})-(\d{2})-(\d{2}) (\d{1,2}):(\d{1,2}):(\d{1,2})$/;
296     var match = regEx.exec(dateStr);
297
298     var year    = match[1];
299     var month   = match[2] - 1;
300     var day     = match[3];
301     var hours   = match[4];
302     var minutes = match[5];
303     var seconds = match[6];
304
305     var d = new Date(year, month, day, hours, minutes, seconds);
306
307     return d;
308 }
309
310 /**
311  * Converts an OpenAir datetime string to epoch time.
312  * @param {Str} dateStr An OpenAir datetime string.
313  * @return {Int}      An epoch date value.
314  */
315 function convertTimestampToEpoch(dateStr) {
316     var d = _convertStringToDateParts(dateStr);
317     return d.getTime() / 1000;
318 }
319
320 /**
321  * Converts Names from Surname, Fistname format to Firstname Surname.
322  * @private
323  * @param {Str} name Full name formatted Surname, Firstname.
324  * @return {Str}      Full name formatted Firstname Surname.
325  */
326 function _surCommaFirstToFirstSpaceSur(name) {
327     var regEx = /^(^,)+, (.+)$/g;
328     return name.replace(regEx, '$2 $1');
329 }
330

```

5. Click **Save**.

The library script will be referenced by the two form scripts created in [Setup 4 — Issue After Save / Project Issue After Save](#).

Setup 4 — Issue After Save / Project Issue After Save

1. Create a new Issue form script deployment and give it a filename.
For more information about creating form scripts, see [Creating Form Scripts](#).
2. Reference the library script **SlackMessageReIssues.js** created in [Setup 3 — Slack Notification Library Script](#).
3. Copy the script below and paste it in the Scripting Studio editor.

```
1 function afterSaveIssue(type) {  
2   var SlackMessageReIssues = require('SlackMessageReIssues');  
3   SlackMessageReIssues.postIssuesOnSlack(type);  
4 }
```

4. Click **Save & continue editing**.
5. Set the event triggering the execution of the script to **After Save**.
6. Set the Entrance Function to **afterSaveIssue**.
7. Click **Save**.
8. Create a new project Issue form script deployment and give it a filename.
9. Repeat steps 2 — 7.

User Scripting Release History

Here you can find all changes made to OpenAir user scripting by release.

April 13, 2024

—

October 7, 2024

—

April 15, 2023

- Updated [Logs](#) — Sort log messages by internal ID, and view system-generated log messages for script changes.
- Updated [View history](#) — View when each script revision was deployed and by whom in the script deployment history.

October 8, 2022

- Added [Clear Log Entries for a Specific Script](#) — Clear all log entries for a specific script from the Scripting Center.
- Updated [NSOA.meta.sendMail\(message\)](#) — Increase the maximum body length of email messages that can be sent from your form or scheduled scripts.

April 9, 2022

- Added [OData Explorer](#) — Browse OData resources and columns available in these resources when creating your scripts in the scripting studio.
- Updated [NSOA.report.data\(reportId,optionalParameters\)](#) — Use select and filter query options to target exactly the information you need from published OpenAir reports.
- Added [NSOA.context.getLanguage\(\)](#) — Get the user's display language preference from your form scripts.

October 9, 2021

- Added [NSOA.NSConnector.integrateWorkflowGroup\(name\)](#)
- Updated [NSOA.report.list\(\)](#) — Each item in the returned list has an additional property: PublishType — The scope of use specified for the published report.

April 18, 2020

- Added `NSOA.listview.data(listviewId)`
- Added `NSOA.listview.list()`
- Added [Platform Solutions Catalog](#) — Find real world use case scripting examples in the OpenAir Help Center.

October 12, 2019

- Added [Real World Use Cases](#):
 - [Send a Slack notification when issues are created or \(re\)assigned](#)
- Updated `NSOA.form.setValue(field, value)`— Now supports dropdown, dropdown and text, pick list and radio group field types.
- Added `NSOA.https.delete(request)`
- Added `NSOA.https.patch(request)`
- Added `NSOA.https.put(request)`

April 13, 2019

- Added [Outbound Calling](#).
- Added `NSOA.https.get(request)`.
- Added `NSOA.https.post(request)`.
- Added [Code Samples](#):
 - [Outbound Calling — SOAP Call Using HTTPS POST](#)
 - [Outbound Calling — Post a Slack Message](#)
 - [Outbound Calling — HTTPS GET with Authorization](#)

October 13, 2018

- Added [Business Intelligence Connector](#).
- Added `NSOA.report.data(reportId,optionalParameters)`.
- Added `NSOA.report.list()`.

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](#).

April 15, 2017

- SetValue on Submit. See [NSOA.form.setValue\(field, value\)](#).
- Script Deployment Logs. See [View Log](#).
- Added [Trace Level Logs](#).

October 15, 2016

- Added approval functions. See [Scripting Approvals](#).

April 16, 2016

- Enhanced Scripting Studio. See [Scripting Studio](#).
 - Editor dynamically resizes to maximize the use of the available area and scrolls independently of the tool area.
 - Customize the Scripting Studio according to your personal preferences with new display options.
 - Search and replace code in scripts using simple or regexp search expressions.
 - Jump directly to a script line number to quickly resolve script errors.
 - Create scripts that are ready to run with a default entrance function and event preselected.
- Custom Field Protection.
- Platform Role Permissions.
- Execute as User.
- Parameter Values.
- Unapprove Event.
- Connector API.
- Enhanced Platform Solutions. See [Creating Solutions](#).
 - Create custom fields as part of applying a Platform Solution to an account.
 - See the referencing solutions when viewing the Form, Scheduled and Parameters screens.
 - Use the new solution form to create Platform Solutions which include multiple scripts.
 - Link Platform Solutions to supporting documentation.
 - Create scripts that are ready to run with a default entrance function and event preselected.

October 17, 2015

- Platform Solutions.
- Email Support.
- Confirmation and Warning Messages.