POLARION 2404

Scripting Guide & Examples

Polarion 2404

About Siemens Digital Industries Software

Siemens Digital Industries Software is a leading global provider of product life cycle management (PLM) software and services with 7 million licensed seats and 71,000 customers worldwide. Headquartered in Plano, Texas, Siemens Digital Industries Software works collaboratively with companies to deliver open solutions that help them turn more ideas into successful products. For more information on Siemens Digital Industries Software products and services, visit https://www.siemens.com/plm

Support Center: https://www.support.sw.siemens.com

Send Feedback on Documentation: https://www.support.sw.siemens.com/doc_feedback_form

Welcome to the **Polarion Scripting SDK Guide**. This document is for developers and others who may need to write custom scripts for workflow conditions or functions and jobs. It covers supported scripting languages and versions, specific use cases (with examples), script configuration, debugging, and implementation. The guide concludes with some Frequently Asked Questions (FAQs).

If you have custom scripting needs but do not have the needed resources, Polarion's Professional Services team has broad and deep experience with custom scripting and many other Polarion customizations and integrations in a variety of industries. For more information, visit https://polarion.plm.automation.siemens.com/training-and-consulting or contact your company's Siemens sales representative.

Tip: Download the **SDK Scripting Guide Example files** archive from **Support Center** for more script examples.

Supported Languages and Scripting Setup

Supported Languages

You can create Polarion custom scripts for Workflows and Jobs using the Groovy or JavaScript languages.

JavaScript

Polarion 2304

Starting with Polarion 2304, the default JavaScript engine is GraalVM. If you have any scripts that were written for previous versions of Polarion for Nashorn or Rhino, you will have to upgrade them to GraalVM. See the migration guides to GraalVM and the "*FAQ*" section in this document to adapt your scripts.

GraalVM has limitations on multithreading in the same js context. This means that it cannot be used. For more details, see: https://www.graalvm.org/22.3/reference-manual/js/Multithreading/

Polarion 22 R2

Polarion 22 R2 is the last version of Polarion, where Nashorn is the default JavaScript engine. Starting with 2304, the GraalVM engine should be used to write scripts in JavaScript, because it will be the default engine in future Polarion releases. To enable GraalVM as the default JavaScript engine, set the following property to "`true`" in the **polarion.properties** file: `com.polarion.scripting.useGraalJsEngine=true`.

GraalVM has limitations on multithreading in the same js context. This means that it cannot be used. For more details, see: https://www.graalvm.org/22.3/reference-manual/js/Multithreading/

All JavaScript examples in this guide have been written and tested with GraalVM.

Polarion 21 R2 and previous versions

The default JavaScript engine used in Polarion 21 R2 and previous versions is Nashorn. You can also enable Rhino as the default engine if you have older scripts (by setting the `com.polarion.scripting.useRhinoJsEngine` property to **"true"** in the **polarion.properties** file).

If you are moving to Polarion 22 R1 and later, you should review your scripts and adapt them to GraaVM. *Polarion 21 R2 is the last version that supports Rhino.* See the migration guides to GraalVM and the "*FAQ*" section in this document to adapt your scripts.

Documentation

The GraalVM JavaScript compatibility guide: https://www.graalvm.org/reference-manual/js/JavaScriptCompatibility/

*Please note that Polarion does **not** use the Nahshorn Compatibility Mode when GraalVM is enabled.*

If you want to migrate scripts that were written for older JavaScript engines to GraalVM:

- Migration guide from Rhino: https://www.graalvm.org/reference-manual/js/RhinoMigrationGuide/
- Migration guide from Nashorn: https://www.graalvm.org/reference-manual/js/NashornMigrationGuide/

Groovy

Polarion 22 R1

Starting with Polarion 22 R1, scripts written in Groovy v2.4.21 are supported.

All Groovy examples in this guide have been tested with Groovy v2.4.21.

Documentation

The official Groovy v2.4.21 documentation can be found at:
https://docs.groovy-lang.org/docs/groovy-2.4.21/html/documentation/

Polarion 21 R2 and previous versions

Polarion 21 R2 and previous versions support scripts written in Groovy v1.5.7.

Polarion SDK

You can find the Polarion SDK content in the **$POLARION_HOME$/polarion/sdk** folder of your installation's file system. Documentation is located in the doc subfolder.

The **[POLARION_HOME]/polarion/sdk/index.html** file contains links to all SDK resources.

You can access the SDK index page via your web browser by appending **polarion/sdk/** to the URL of your Polarion server.

Location of scripts

Scripts must be stored in the **scripts** directory: (If the directory does not yet exist, create it.)

- Windows: `C:\Polarion\scripts`
- Linux: `/opt/polarion/scripts`

Note: The above root paths are the Polarion installation default paths. An installing administrator can specify a different installation root directory.

Workflow Scripts

You can extend the Work Item, Document, and Test Run Workflows with custom scripts. These scripts can check if a specific condition is met before allowing a workflow action (for example, are there still any unresolved comments) or execute a function when a workflow action is taken. (For example, increment the version field of the Document).

ScriptCondition

A `ScriptCondition` is a type of script used to check if a specific condition is met before allowing a workflow action. Polarion executes the script associated with a `ScriptCondition` when a user initiates a change to the **Status** field in the Polarion UI and will generate a list of possible actions the user can take.

For example, suppose a Document workflow has a transition from **In Review** to **Reviewed** via a workflow action configured with a condition script that checks that the Document doesn't have any unresolved comments. If so, then Polarion disables the transition action if the Document has unresolved comments:
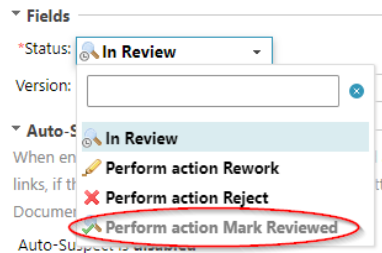
Script Implementation

A `ScriptCondition` is a script that returns a Boolean (true/false) value that indicates to Polarion whether or not the condition is met.

The script can also return a String that Polarion then shows to the user as a reason why the transition is not allowed. (Returning a String is equivalent to returning false.)

For example, there are still unresolved comments and the String returned by the script is: *Unresolved comment found. Make sure to resolve all of your comments.*
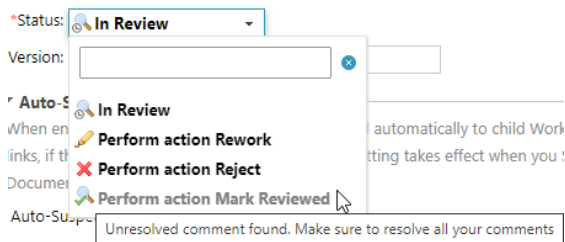
Available Objects

`ScriptCondition` scripts have access to the following Polarion Open API Interfaces:

- `com.polarion.alm.tracker.workflow.ICallContext` (Used for getting the Work Item, Document or Test Run object where the workflow is executed.)
- `com.polarion.alm.tracker.workflow.IArguments` (Used for getting custom configuration settings of the script.)

In the script, you have access to the following variables:

- `workflowContext`: Object of type ICallContext
- `arguments`: Object of type IArguments

Start of a ScriptCondition

Typically, a `ScriptCondition` starts by getting the object that the workflow script is executed from. This is done by using the `workflowContext` object and executing this code:

JavaScript:

```
var baseObject = workflowContext.getTarget();
```

Groovy:

```
def baseObject = workflowContext.getTarget();
```

Depending on whether the workflow is executed from a Work Item, a Document, or a Test Run, the `workflowContext.getTarget();` call can return the following object types:

- `com.polarion.alm.tracker.model.IWorkItem` (If executed from a Work Item.)
- `com.polarion.alm.tracker.model.IModule` (If executed from a Document.)
- `com.polarion.alm.tracker.model.ITestRun` (If executed from a Test Run.)

If you are using custom configuration arguments in your Script Configuration (see next section), then those are available using the `arguments` object.

For example, if you have a custom argument called `versionFieldName`, then you can access its value by executing this code:

JavaScript:

```
// Getting the Version Field Name configured in the Workflow. If no value is provided in the configuration, the default value
will be "version")
var versionFieldID = arguments.getAsString("versionFieldName", "version");
```

Groovy:

```
// Getting the Version Field Name configured in the Workflow. If no value is provided in the configuration, the default value
will be "version")
def versionFieldID = arguments.getAsString("versionFieldName", "version");
```

End of a ScriptCondition

A `ScriptCondition` must end with a Boolean value or a String. This tells Polarion whether or not to enable the Workflow Action on which the script executes.

Please note that you cannot make use of a "return" at the end of the script. To get Polarion to understand the Boolean value, the last statement of the script must literally be the Boolean value or String.

For example, if you are testing that the **Severity** field value is "**should_have**", then the last line of the script should be:

JavaScript/Groovy:

```
baseObject.getSeverity().getId().equals("should_have");
```

In some cases, you can also store the Boolean Value in a variable and just add it at the end of the script:

JavaScript/Groovy:

```
if(ABC)
{
    (...)
    myResult = true;
}
else
{
    myResult = false;
}
// Last Line of the script
myResult;
```

If you want to display a message to users, then you can return a String instead of a Boolean value. If a String is returned, Polarion treats it as Boolean "`False`".

JavaScript/Groovy:

```
if(ABC)
{
    (...)
    myResult = true;
}
else
{
    myResult = "Unresolved comment found. Make sure to resolve all of your comments.";
}
// Last Line of the script
myResult;
```

Script Configuration

Step 1: Enter one of the following Administration pages, depending on the type of workflow you wish to configure:

- Work Items: 🔧**Administration** ➡ ⚙**Work Items** ➡ ⚙**Workflow**
- Documents: 🔧**Administration** ➡ 📄 **Documents & Pages** ➡ 📄**Document Workflow**
- Test Runs: 🔧 **Administration** ➡ ⚗**Testing** ➡ 📋**Test Run Workflow**

Step 2: Click 📝**Edit** beside the artifact to which you'd like to add the script condition.

Step 3: Click ✏ **Edit** beside the **Action** you to which you want to add the script condition.

Step 4: Select **ScriptCondition** from the **Conditions** drop-down list click ➕.

Step 5: Click ✏ **Edit** beside the newly created **Condition.**

Step 6: Define the following required parameters:

- **engine**: The name of the scripting engine to use. Groovy (.groovy) and JavaScript/ECMAScript (.js) file types are supported.
  Valid values: **groovy** for the Groovy engine, or **js** for JavaScript/ECMAScript.
- **script**: The name of the script in the scripts folder to run as the **Condition**.

For example, a JavaScript called *CheckDocumentCommentsStatus.js:*
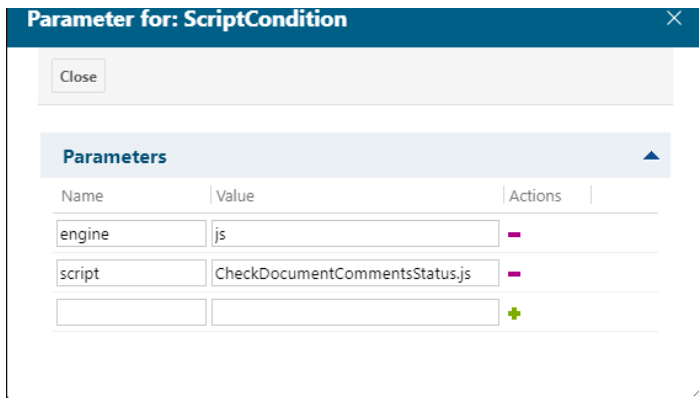
If you are using custom arguments, you can just add new parameters where **Name** is the name of your custom parameter and **Value** is the value read in your script.

Let's say that the script checks if there are more than five unresolved comments and you want to make this number configurable via the `numberOfComments` parameter.
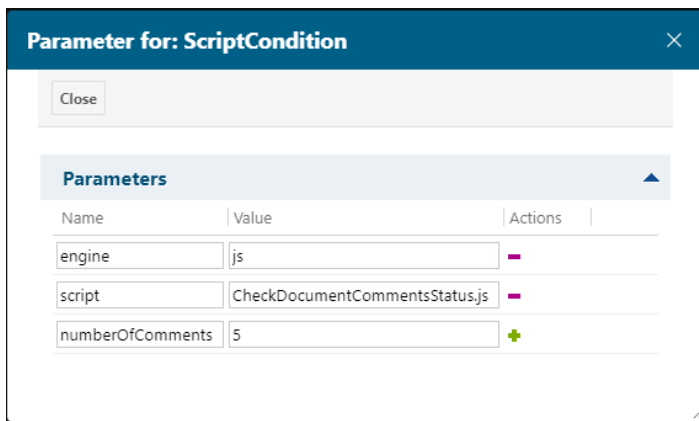
In the script, this would be the code used to read the value of the parameter:

JavaScript:

```javascript
// Get the Number of Comments configured in the Workflow. If no value is provided in the configuration, default value will be 0)
var numberComments = arguments.getAsInteger("numberOfComments", 0);
```

Groovy:

```groovy
// Get the Number of Comments configured in the Workflow. If no value is provided in the configuration, default value will be 0)
def numberComments = arguments.getAsInteger("numberOfComments", 0);
```

Step 7: Click ✕ to close the **Parameters** dialog box and ✕ again to close the **Details for Action** dialog box.

The new **Condition** appears for the **Action** you defined it for.

Step 8: Click **Save** at the top.

Script Debugging

No debugger is available for `ScriptCondition`. The method typically used for debugging is by using custom log files that can be read from the Polarion log folder Path: `C:\Polarion\data\main\logs` (Windows) and `/opt/polarion/data/main/logs` (Linux).

The idea is to get the script to create a log file that can be used to display the content of variables in your script. To get the log file populated by your log entries, you must then do what an end user would do in the Polarion UI to get the `ScriptCondition` to execute, which is clicking on the **Status** field of a Work Item, Document, or Test Run and selecting the Workflow Action that your script is used for.

To create a log file for your script, you can execute:

JavaScript:

```javascript
// Creating the log file
var FileWriter = Java.type('java.io.FileWriter');
var outFile = new FileWriter("./logs/main/NAME_OF_YOUR_SCRIPT.log");
var BufferedWriter = Java.type('java.io.BufferedWriter');
var out = new BufferedWriter(outFile);
```

Groovy:

```groovy
// Creating the log file
def logFile = new File('./logs/main/NAME_OF_YOUR_SCRIPT.log').newOutputStream();
```

Then in your script, you can simply create log entries for anything you want to track:

JavaScript:

```javascript
out.write("Running NAME_OF_YOUR_SCRIPT.js script"); out.newLine(); out.flush();
```

Groovy:

```groovy
logFile << "Running NAME_OF_YOUR_SCRIPT.groovy script \n";
```

At the end of the script, make sure to always close the log file:

JavaScript:

```javascript
// Closing the Log File
out.close();
```

Groovy:

```groovy
// Closing the Log File
logFile.close();
```

Script Examples

CheckDocumentCommentsStatus.js - Returns Boolean Example

This example will return the false if it finds at least one Document comment that is unresolved.

```javascript
// When no comments are found, the script will return true
// -------------------------------------------------------
var returnValue = true;

// Instantiate the Document object from the workflow context constructor
// ---------------------------------------------------------------------
var document = workflowContext.getTarget();

// Retrieve all Comments in the Document
// See: com.polarion.alm.tracker.model.IModule for a list of all Documents Methods
// -------------------------------------------------------------------------------
var allComments = null;
allComments = document.getComments();

// Check the status of each comment retrieved from the Document
// -----------------------------------------------------------
for (i = 0; i < allComments.size(); i++)
{
  var comment = allComments.get(i);

  // Check if comment is NOT resolved
  // See: com.polarion.alm.tracker.model.IModuleComment
  // for the list of all Document Comments Methods
  // -------------------------------------------------
  if(!comment.isResolvedComment())
  {
    // As soon as we find an unresolved comment, then we will return FALSE
    // ------------------------------------------------------------------
    returnValue = false;
    break;
  }
}

// Last line of the script is Boolean. No messages will be displayed for the user
// ------------------------------------------------------------------------------
returnValue;
```

CheckDocumentCommentsStatus.js - Returns String Example

This example will return a String to the user if it finds at least one Document comment that is unresolved.

```javascript
// Variable that tracks if an unresolved comment was found
// -------------------------------------------------------
var isUnresolvedCommentFound = false;

// Instantiate the Document object from the workflow context constructor
```

```
// ------------------------------------------------------------------
var document = workflowContext.getTarget();

// Retrieve all Comments in the Document
// See: com.polarion.alm.tracker.model.IModule for a list of all Documents Methods
// ------------------------------------------------------------------------------
var allComments = null;
allComments = document.getComments();

// Check the status of each comment retrieved from the Document
// --------------------------------------------------------------
for (i = 0; i < allComments.size(); i++)
{
  var comment = allComments.get(i);

  // Check if comment is NOT resolved
  // See: com.polarion.alm.tracker.model.IModuleComment
  // for a list of all Document Comments Methods
  // -----------------------------------------------
  if(!comment.isResolvedComment())
  {
    // As soon as we find an unresolved comment, we set the variable to true
    // --------------------------------------------------------------------
    isUnresolvedCommentFound = true;
    break;
  }
}

// When no comments are found, the script will return true
// -------------------------------------------------------
var returnValue = true;
if (isUnresolvedCommentFound)
{
  returnValue = "Unresolved comment found. Make sure to resolve all your comments"
}


// Last line of the script will return true when no comments are found and the workflow
// action will be enabled.
// When unresolved comments are found, it will return a string that will displayed to
// the user and it will deactivate the workflow action
// ------------------------------------------------------------------------------
returnValue;
```

ScriptFunction

A `ScriptFunction` is the type of script used to execute a function or functions when a workflow action is initiated by a user changing the **Status** field of a Work Item, Document, or Test Run. Polarion will invoke the script associated with a `ScriptFunction` during the **Save** action after a user has changed the **Status** of a workflow (via a workflow action).

For example, if a Document workflow has a transition from **Approved** to **Draft** via a workflow action configured with a script function that increments the version number stored in a specific Document custom field, then Polarion runs the script to increment the version number when users execute the workflow action and clicks **Save on** the Document. When the save action completes, they will see the new version number.
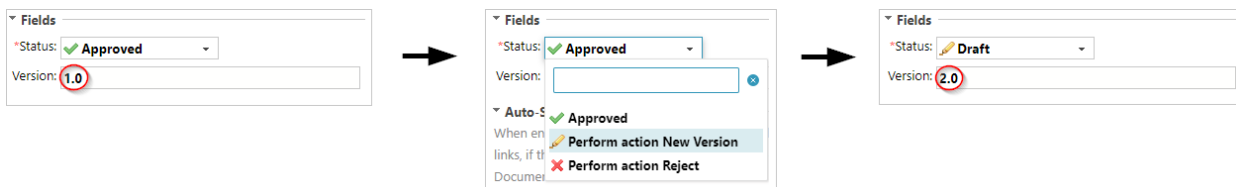
Script Implementation

A `ScriptFunction` is a script that is invoked during the save action of a Work Item, Document or Test Run when a user takes a specific workflow action.

Available Objects

A `ScriptFunction` script has access to the following Polarion Open API Interfaces:

- `com.polarion.alm.tracker.workflow.ICallContext` (Used to get the Work Item, Document or Test Run object where the workflow is executed.)
- `com.polarion.alm.tracker.workflow.IArguments` (Used to get the custom configuration settings of the script.)

In the script, you have access to the following variables:

- `workflowContext`: Object of type ICallContext
- `arguments`: Object of type IArguments

Start of a ScriptFunction

Typically, a `ScriptFunction` starts by getting the object from which the workflow script is being executed. This is done by using the `workflowContext` object and executing this code:

JavaScript:

```
var baseObject = workflowContext.getTarget();
```

Groovy:

```
def baseObject = workflowContext.getTarget();
```

Depending on if you execute the workflow from a Work Item, a Document or a Test Run, the `workflowContext.getTarget();` call returns the following object types:

- `com.polarion.alm.tracker.model.IWorkItem` (If you are executing from a Work Item.)
- `com.polarion.alm.tracker.model.IModule` (If you are executing from a Document.)
- `com.polarion.alm.tracker.model.ITestRun` (If you are executing from a Test Run.)

If you are using custom configuration arguments in your Script Configuration (see next section), then they will be available with the use of the `arguments` object. For example, if you have a custom argument called `versionFieldName`, then you can get access to its value by executing this code:

JavaScript:

```
// Getting the Version Field Name configured in the Workflow. If no value provided, the default value will be "version"
var versionFieldName = arguments.getAsString("versionFieldName", "version");
```

Groovy:

```
// Getting the Version Field Name configured in the Workflow. If no value provided, the default value will be "version"
def versionFieldName = arguments.getAsString("versionFieldName", "version");
```

End of a ScriptFunction

A `ScriptFunction` does not require any special handling or return value at the end of the script. If you have opened any file handle (see "*Script Debugging*" section) or allocated any memory, make sure to close the handle and free the memory.

Script Configuration

Step 1: Enter one of the following Administration pages, depending on the type of workflow you wish to configure:

- Work Items: 🔧**Administration** ➡ ⚙**Work Items** ➡ ⚙**Workflow**
- Documents: 🔧**Administration** ➡ 📄 **Documents & Pages** ➡ 📄**Document Workflow**
- Test Runs: 🔧 **Administration** ➡ 🛠**Testing** ➡ 🧪**Test Run Workflow**

Step 2: Click 📝**Edit** beside the artifact to which you'd like to add the script function.

Step 3: Click ✏ **Edit** beside the **Action** you to which you want to add the script function.

Step 4: Select **ScriptFunction** from the **Functions** drop-down list click ➕.

Step 5: Click ✏ **Edit** beside the newly created **Function**

Step 6: Define the following required parameters:

- **engine**: The name of the scripting engine to use. Groovy (.groovy) and JavaScript/ECMAScript (.js) file types are supported.
  Valid values: **groovy** for the Groovy engine, or **js** for JavaScript/ECMAScript.
- **script**: The name of the script in the scripts folder to run as the **Condition**

For example, a JavaScript called **incrementVersion.js**

If you use custom arguments, you can add new parameters where the **Name** is the name of your custom parameter and **Value**, the value that will be read in your script.

For example, let's say the script can be configured with the **ID** of the custom field used to contain the version number and you want to make this configurable via the **versionFieldID** parameter.
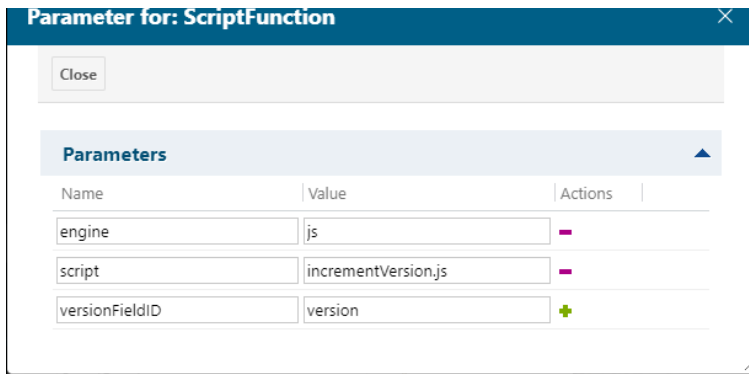
In the script, this would be the code used to read the value of the parameter:

JavaScript:

```
// Getting the ID of the Version Custom Field used to store the version. If no value is provided in the configuration, the
default value will be "version"
var versionCustomFieldID = arguments.getAsString("versionFieldID", "version");
```

Groovy:

```
// Getting the ID of the Version Custom Field used to store the version. If no value is provided in the configuration, the
default value will be "version"
def versionCustomFieldID = arguments.getAsString("versionFieldID", "version");
```

Step 7: Click ✖ to close the **Parameters** dialog box and ✖ again to close the **Details for Action** dialog box.

The new **Function** appears for the **Action** you defined it for.

Step 8: Click **Save** at the top.

Script Debugging

No debugger is available for `ScriptFunction`. The method typically used for debugging is by using the custom log files read from the Polarion log folder. Path: `C:\Polarion\data\main\logs` (**Windows**) and `/opt/polarion/data/main/logs` (**Linux**).

The idea is to get the script to create a log file that can be used to display the content of variables in your script. To populate the log file with your log entries, you must then execute the workflow action that implements the `ScriptFunction` by using the **Status** field of your Work Item, Document or Test Run and then hit **Save**.

To create a log file for your script, you can execute:

JavaScript:

```
// Creating the log file
var FileWriter = Java.type('java.io.FileWriter');
var outFile = new FileWriter("./logs/main/NAME_OF_YOUR_SCRIPT.log");
var BufferedWriter = Java.type('java.io.BufferedWriter');
var out = new BufferedWriter(outFile);
```

Groovy:

```
// Creating the log file
def logFile = new File('./logs/main/NAME_OF_YOUR_SCRIPT.log').newOutputStream();
```

Then in your script, you can simply create log entries for anything you want to track:

JavaScript:

```
out.write("Running NAME_OF_YOUR_SCRIPT.js script"); out.newLine(); out.flush();
```

Groovy:

```
logFile << "Running NAME_OF_YOUR_SCRIPT.groovy script \n";
```

At the end of the script, make sure to always close the log file:

JavaScript:

```
// Closing the Log File
out.close();
```

Groovy:

```
// Closing the Log File
logFile.close();
```

Use of throw

It is possible to use `throw` statements in your `ScriptFunction` code to generate a user message in the Polarion UI. Generating a user message via a `throw` stops the current saving action. This means that if the condition that generates the `throw` cannot be controlled by the user, then the save action may never complete. Typically, you should only use `throw` statements in development or in very specific cases.

JavaScript:

```
throw "USER MESSAGE GENERATED BY A THROW STATEMENT";
```

Groovy:

```
throw new Exception("USER MESSAGE GENERATED BY A THROW STATEMENT");
```

The code above generates this user message in the Polarion UI:
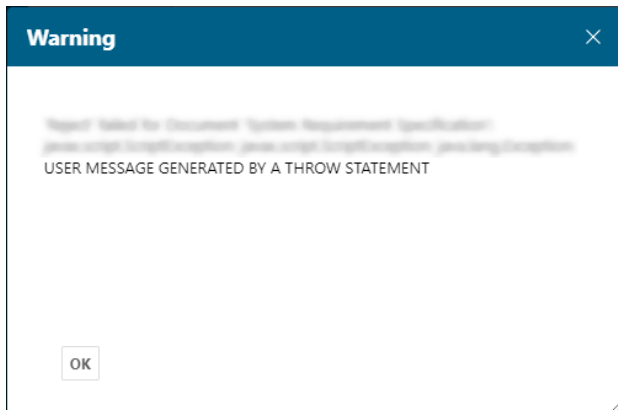
Script Examples

IncrementVersion.js - Using 2 Configuration Parameters and debug information

This example increments a String type custom field (Parameter: `versionFieldID` with a default value: `version`).
The increment value can be configured (Parameter: `incrementValue` with a default of 1.0).

The script also includes a log file that can be used for debugging.

This script works for Work Items, Documents and Test Runs because they all inherit from the same `Custom Field` Interface:

- `com.polarion.platform.persistence.model.IHasCustomValues` (Used for `getCustomField` and `setCustomField`)

```
// Creating the log file
// ---------------------
var FileWriter = Java.type('java.io.FileWriter');
var outFile = new FileWriter("./logs/main/incrementDocVersionWF.log");
var BufferedWriter = Java.type('java.io.BufferedWriter');
var out = new BufferedWriter(outFile);

// Evaluate the incoming workflow action parameters
// For versionFieldID, default value: "version"
// For incrementValue, default value: "1.0"
// --------------------------------------------
var versionCustomFieldID = arguments.getAsString("versionFieldID", "version");
out.write("Version Field Id: " + versionCustomFieldID); out.newLine(); out.flush();
var incrementValue = arguments.getAsString("incrementValue", "1.0");
out.write("Increment Value: " + incrementValue); out.newLine(); out.flush();

// Instantiate the base object from the workflow context constructor
// ----------------------------------------------------------------
var baseObject = workflowContext.getTarget();

// Get the custom field value from the version custom field (ID: versionCustomFieldID)
// We are expecting a custom field of type string
// --------------------------------------------------------------------------
var oldVersion = baseObject.getCustomField(versionCustomFieldID);
var newVersion = "";

out.write("Values:"); out.newLine(); out.flush();
out.write("Old Version: " + oldVersion); out.newLine(); out.flush();

// If no value is defined in the Version Field, we set it to the value
// of the incrementValue
```

```
// ----------------------------------------------------------------
if (oldVersion == "" || oldVersion == null || oldVersion == "NaN")
{
  newVersion = incrementValue;
  baseObject.setCustomField(versionCustomFieldID, newVersion);
  out.write("New Version: " + newVersion); out.newLine(); out.flush();
}
// The version field already has a value, so we just increment it by incrementValue
// ------------------------------------------------------------------------------
else
{
  newVersion = parseFloat(oldVersion) + parseFloat(incrementValue);
  baseObject.setCustomField(versionCustomFieldID, newVersion.toFixed(1).toString());
  out.write("New Version: " + newVersion.toFixed(1).toString()); out.newLine(); out.flush();
}

// Closing the Log File
// --------------------
out.close();
```

The `ScriptFunction` workflow configuration should be:

Parameter for: ScriptFunction   ✕

Close

Parameters   ▲

Name	Value	Actions
engine	js	▬
script	incrementVersion.js	▬
versionFieldID	myVersionField	▬
incrementValue	1.0	✚
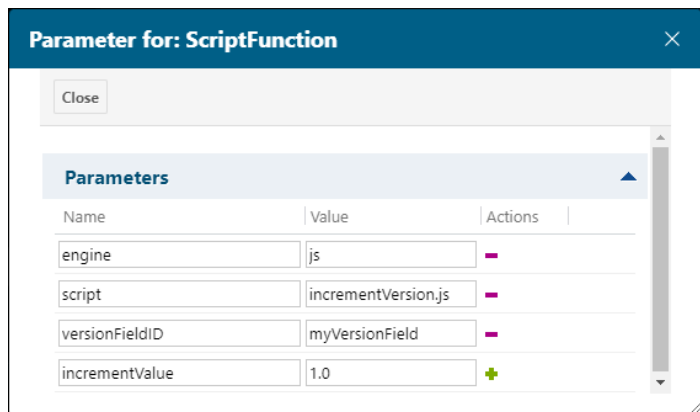
CreateDocumentBaseline.js

This script creates a Document baseline for the current Document it is executed on. The document baseline will be associated with the target revision. For example, if the `ScriptFunction` is used for an action between **In Review** and **Approved** statuses, then the Document baseline is created for the revision associated with the **Approved** status.

Approved

Type:	⚑ **Document Baseline**
Document:	⏱ 📄 **Generic Document**
*Base Revision:	**760**
Author:	**System Administrator**
Description:	**Created automatically by a workflow transition**
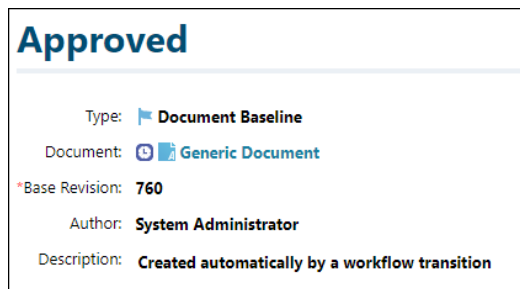
In this example, we are creating a new object that is not part of the current Document and the workflow it is executing on. (So we will need to save the baseline.)

```
// Instantiate the base object from the workflow context constructor
// ----------------------------------------------------------------
var baseObject = workflowContext.getTarget();

// Create a new baseline for the target revision using createObjectBaselineForChange
// via the com.polarion.alm.tracker.IBaselinesManager
// --------------------------------------------------------------------------------
var baseline = baseObject.getProject().getBaselinesManager().createObjectBaselineForChange(baseObject);

// Set the Name and Description of the Baseline
// See com.polarion.alm.tracker.model.IBaseline
// --------------------------------------------
baseline.setName(createName());
baseline.setDescription(com.polarion.core.util.types.Text.plain(createDescription()));

// Save the Baseline
// Since we are creating a new object that is not part of the current Document, we must save it
// --------------------------------------------------------------------------------------------
baseline.save();
```

```javascript
// Defines name for the created baseline
// ------------------------------------
function createName()
{
  return baseObject.getEnumerationOptionForField("status", workflowContext.getTargetStatusId()).getName();
}

// Defines description for the created baseline
// -------------------------------------------
function createDescription()
{
  return "Created automatically by a workflow transition";
}
```

Job Scripts

Polarion comes with a scheduler that executes jobs. Many of these jobs are included by default, and new ones can be created via Java extensions. One of the job types included is a `script.job` that can execute a custom Groovy or JavaScript/ECMAScript. (See the "*Scripting Setup and Supported Languages*" section for supported versions.)

These job scripts can have access to all the Polarion data and perform data modification automation activities (when permissions are set up for them), export data in a specific format in a specific location, and send custom email notifications, etc.

Script Configuration

To create a `script.job`, a new entry must be created in the Polarion ⊙ **Scheduler** (**Global Administration** - ⊙ **Scheduler**).

As with any Polarion Job, these attributes can be used in the XML entry to configure the job:

- **name**: (REQUIRED) Name of the job used when viewed in the ⌁ **Monitor**.
- **id**: (REQUIRED) Identifier used by Polarion to find the job. In the case of a job script, the identifier **must** be: `script.job`.
- **scope**: (REQUIRED) Scope of the job. Must be one of:
    - `project:<project_id>`
    - `path:<repository_path>`
    - `system`
- **cronExpression**: (REQUIRED only if the job needs to be automatically executed). UNIX cron-like specification of points in time when the job should be started. See **Quick Help** in the **Scheduler** topic in the **Global Administration** for examples.
- **disabled**: (OPTIONAL - Default value is "`false`"). When set to "`false`", then the job is automatically executed by the scheduler as per the cron expression. When set to "`true`", the job can only be executed manually via the ⌁ **Monitor**.
- **node**: (OPTIONAL) Set the cluster node where job will be executed. Possible values are:
    - unspecified - scheduler will choose one of the nodes
    - node="*" - will be executed on all nodes in a cluster
    - node="`<nodeId>`" - will be executed only on specified node

A `script.job` entry must use the value "`script.job`" as the **ID**.

Then, the job has the following required parameters:

- **scriptName**: The name of the script to run. (It must reside in in the **scripts** folder.)
- **scriptEngine**: The name of the scripting engine to use. Groovy (.groovy) and JavaScript/ECMAScript (.js) file types are supported. Valid values: **groovy** for the Groovy engine, or **js** for JavaScript/ECMAScript.

A Scheduler `script.job` entry example for a JavaScript job that automatically imports test results from an XML file every 10 minutes between 8:00 and 18:00 every day:

```
<job name="Test results importer via XML" id="script.job" cronExpression="0 0/10 8-17 * * ?" disabled="false" scope="system">
    <scriptName>TestResultsImporter.js</scriptName>
    <scriptEngine>js</scriptEngine>
</job>
```

Custom properties that can be read by the `script.job` can also be added. These are defined inside the `<properties>` element of the job. The property in the example below would be accessible as variable with the name `importFolder` containing the value `c:\importResults`. Here's a groovy script example:

```
<job name="Test results importer via XML" id="script.job" cronExpression="0 0/10 8-17 * * ?" disabled="false" scope="system">
    <scriptName>TestResultsImporter.groovy</scriptName>
    <scriptEngine>groovy</scriptEngine>
    <properties>
        <importFolder>c:\importResults</importFolder>
    </properties>
</job>
```

In the script, the name of the custom property becomes a variable, and its content the property value:

JavaScript and Groovy

```
// Getting the property value configured for the <importFolder> is done by using the variable importFolder
logger.info("------------- Parameters -------------");
logger.info("The value of the parameter --importFolder-- is " + importFolder);
```

Script Implementation

A `script.job` is a script that can be executed manually using the Polarion ⌁ **Monitor** topic or executed automatically via the Polarion ⊙ **Scheduler**. (Available by default in any Polarion Project, or added as a topic via the ⊹ **Portal** topic in Administration. See the **Configure the Portal** and the **Configure the Scheduler** topics in Help.)

Available Objects

In the script you will be able to access the following variables:

`script.job` scripts have access to the following Polarion Open API Interfaces via their corresponding variables:

- logger: A `com.polarion.platform.jobs.ILogger` you can use to log messages for the job
- scope: A `com.polarion.platform.context.IContext` that represents the scope of the job.
- workDir: A `java.io.File` pointing to the working directory of the job.
- jobUnit: The `com.polarion.platform.jobs.IJobUnit` that runs the script.
- trackerService: The `com.polarion.alm.tracker.ITrackerService` service that gives access to the main entry point for tracker-related functionalities and the APIs of any Polarion artifacts.
- projectService: The `com.polarion.alm.projects.IProjectService` service that gives access to the main entry point for project-related functionalities and APIs.
- result: A Boolean variable that can be used to display the status of the job execution in the log file of the job.

Content of a script.job

How to use the scope

The scope object contains the scope of the `script.job` as defined in its XML configuration entry in the **Scheduler**. The `script.job` defines whether or not the scope is used.

One way to use the scope is to specify a `projectID` corresponding to the project that the `script.job` should be executed on.

In that case, you could read and make use of the scope value as follows:

JavaScript:

```javascript
// Read the scope value (using the scope object)
var jobScope = scope.getId().getContextName();
var tProjects = [];
if(jobScope != null)
{
    tProjects = projectService.searchProjects(jobScope);
}
```

Groovy:

```groovy
// Read the scope value (using the scope object)
def jobScope = scope.getId().getContextName();
def tProjects = [];
if(jobScope != null)
{
    tProjects = projectService.searchProjects(jobScope);
}
```

How to read script properties

As it is described in the "*Script Configuration*" section, the XML configuration entry of the `script.job` can define custom properties that can be read from the script. A good practice is to read these custom properties at the beginning of the script, log their value and act on them if they do not contain the expected value. (For example, if they are empty.)

If we have a `script.job` XML configuration that defines:

```xml
<properties>
    <importFolder>c:\importResults</importFolder>
</properties>
```

Then in the `script.job`, you could use a combination of try/catch.

JavaScript:

```javascript
var sImportFolder = "";
try
{
    // Check that the importFolder property is defined, if not we exit the job with an error
    logger.info("-------------- Parameters --------------");
    if( importFolder == null || importFolder.trim().length() == 0 )
    {
        throw "Error: Missing value for parameter 'importFolder'";
    }
    else
    {
        logger.info("The value of the parameter --importFolder-- is " + importFolder);
        sImportFolder = importFolder;
    }

    if(!sImportFolder.equals(""))
    {
        // Here, this is where the actual code of the job would start
        //(...)
    }
}
```

```
catch (err)
{
    // Error message is added to the log
    logger.error(err);
}
```

Groovy:

```
def sImportFolder = "";
try
{
    // Check that the importFolder property is defined, if not we exit the job with an error
    logger.info("------------- Parameters -------------");
    if( importFolder == null || importFolder.trim().length() == 0 )
    {
        throw "Error: Missing value for parameter 'importFolder'";
    }
    else
    {
        logger.info("The value of the parameter --importFolder-- is " + importFolder);
        sImportFolder = importFolder;
    }

    if(!sImportFolder.equals(""))
    {
        // Here, this is where the actual code of the job would start
        //(...)
    }
}
catch (err)
{
    // Error message is added to the log
    logger.error(err);
}
```

How to use the workDir variable

The `script.job` makes the working directory of the job available as a `java.io.File` object.

Each execution of the job will create a distinct `workDir` under the main job working directory.

The main job working directory is located under:

Path: `C:\Polarion\path\workspace\polarion-data\jobs\` (Windows) and `/opt/polarion/workspace/polarion-data/jobs/` (Linux).

This can be useful if you need to manipulate files in the working directory of the job.

For example:

JavaScript:

```
var workDirPath = workDir.getAbsolutePath();
logger.info("The job executes in the folder: " + workDirPath);
```

Groovy:

```
def workDirPath = workDir.getAbsolutePath();
logger.info("The job executes in the folder: " + workDirPath);
```

How to start and commit a transaction

When a `script.job` modifies Polarion content (Work Items, Documents, Test Runs, etc.), then it must use a transaction. The idea is to execute the modifications within the context of a transaction that needs to be started and committed (where rollbacks are also possible) via the `ITransactionService`. (`com.polarion.platform.ITransactionService`).

A transaction gets committed in the name of the user manually executing the `script.job` or the **polarion** user if the `script.job` is executed automatically by the Polarion **Scheduler** (See the "*Permissions of a script.job*" section to make sure you have the right permissions to execute the `script.job`.)

For example:

JavaScript:

```
// Retrieving the Transaction Service
var txService =
com.polarion.platform.core.PlatformContext.getPlatform().lookupService(com.polarion.platform.ITransactionService.class);

// Start a new transaction as users might get deactivated
txService.beginTx();
```

```
// Code to modify Polarion Content
// Do not forget to call ".save()" on any Polarion object that gets modified before committing the transaction
// (...)

// Commit Transaction
// endTx(false) will commit the transaction
// endTx(true) will rollback the changes done during the transaction
txService.endTx(false);
```

Groovy:

```
// Retrieving the Transaction Service
def txService =
com.polarion.platform.core.PlatformContext.getPlatform().lookupService(com.polarion.platform.ITransactionService.class);

// Start a new transaction because users might get deactivated
txService.beginTx();

// Code to modify Polarion Content
// Do not forget to call ".save()" on any Polarion object that gets modified before committing the transaction
// (...)

// Commit Transaction
// endTx(false) will commit the transaction
// endTx(true) will rollback the changes done during the transaction
txService.endTx(false);
```

Running parts of a script.job as another user with doAsUser

Note: Starting with **Polarion 2304**, `doAsUser` is disabled by default. See section on *Default API Security* to enable it.

The method `doAsUser` (`com.polarion.platform.security.ISecurityService`) allows you to run functions defined in the `script.job` as another user. This can be useful when a `script.job` is modifying or creating new Polarion data and is meant to be run automatically by the **Scheduler** (that uses the **polarion** user that only has a read-only access to the Polarion data.)

To run parts of your `script.job` as another user with `doAsUser`, you must first identify (or create) a Polarion user that will have the right level of permissions to access and modify the Polarion data you want modified by the `script.job`. You must then create a **User Account Vault** entry for this user.

To create a new **User Account Vault** Entry:

Step 1: Log on with administrator permissions for the repository.
Step 2: Open the 🗄 **Repository** and enter 🔧**Administration**.
Step 3: Expand 👥 **User Management** and select 👤 **User Account Vault**.

A **User Account Vault** table of existing **Keys** appears.

Step 4: ➕**Add or** ➖ **Remove a Key:**

Add a key:
Enter any **Key** name, the Polarion **User Name** (of the user that will be used to access the data), its **Password**, then re-enter the **Password**, then click ➕.

Step 5: Click 💾 **Save**.

In your `script.job`, a dedicated function that will be run as another user should be created. This function will contain the code that requires access to the target Polarion data. For example in the function, you might be starting a transaction, modifying or adding Polarion content, and then committing the transaction. Then, using `doAsUser`, you will tell Polarion to execute the function in the context of the user (with all their permissions) via the key that you added to the **User Account Vault**.

For example:

JavaScript (using GraalVM):

```
// This function requires that it be executed by a user with read/write access to the Polarion repository
function MyFunctionWithRW()
{
    logger.info("MyFunctionWithRW - Code doing a modification on the Polarion data (Requires RW Access)")
}

// Login with a user with RW access that was added to the Polarion Account Vault
// and identified with the key "rProject.key"
var securityService = trackerService.getDataService().getSecurityService();
var userWithRWAccess = securityService.loginUserFromVault("rProject.key", "");

// Create a Privileged Action object that will be run by the doAsUser function
// with our user with RW access
var privilegedAction = Java.extend(Java.type('java.security.PrivilegedAction'));
var privilegedActionImpl = new privilegedAction({ run: function () {
{
    logger.info('User used by doAsUser to execute this call is: ' + securityService.getCurrentSubject().toString());
```

```
    // Our call to the function requiring RW access
    MyFunctionWithRW();


    return null;
}});

// Using doAsUser to call the privilegedActionImpl function on behalf of the user from userWithRWAccess
securityService.doAsUser(userWithRWAccess, privilegedActionImpl);
```

JavaScript (using Nashorn):

```
// This function requires that it is executed by a user with read/write access to the Polarion repository
function MyFunctionWithRW()
{
    logger.info("MyFunctionWithRW - Code doing a modification on the Polarion data (Requires RW Access)")
}

// Login with a user with RW access that was added to the Polarion Account Vault
// and identified with the key "rProject.key"
var securityService = trackerService.getDataService().getSecurityService();
var userWithRWAccess = securityService.loginUserFromVault("rProject.key", "");

// Create a Priviledged Action object that will be run by the doAsUser function
// with our user with RW access
var privilegedActionImpl = new Object();
privilegedActionImpl.run = function ()
{
    logger.info('User used by doAsUser to execute this call is: ' + securityService.getCurrentSubject().toString());

    // Our call to the function requiring RW access
    MyFunctionWithRW();


    return null;
}

// Using doAsUser to call the privilegedActionImpl function on behalf of the user from userWithRWAccess
securityService.doAsUser(userWithRWAccess, new java.security.PrivilegedAction(privilegedActionImpl));
```

Groovy:

```
// This function must be executed by a user with read/write access to the Polarion repository
void MyFunctionWithRW()
{
    logger.info("MyFunctionWithRW - Code doing a modification on the Polarion data (Requires RW Access)")
}

// Login with a user with RW access that was added to the Polarion Account Vault
// and identified with the key "rProject.key"
def securityService = trackerService.getDataService().getSecurityService();
def userWithRWAccess = securityService.loginUserFromVault("rProject.key", "");

// Using doAsUser to call the MyFunctionWithRW() function on behalf of the user from userWithRWAccess
def foo = securityService.doAsUser(
    userWithRWAccess,
    {
        logger.info('User used by doAsUser to execute this call is: ' + securityService.getCurrentSubject().toString());
        MyFunctionWithRW()
    } as java.security.PrivilegedAction
);
```

Script Execution

> ⬥ **Warning:**
> Depending on a job's scope and type, it can consume system resources and significantly slow down the system for other users.
> It's a good idea to bear this in mind when deciding to explicitly run a job.

Manual script.job execution

The ⅍ **Monitor** topic provides access to information about currently-running and scheduled jobs. The topic is available when working in a project or in the global (Repository) scope.

From the topic, a user can select any of the available jobs by selecting their associated check box, then clicking **Execute now**.
Note: A job is only listed in the ⅍ **Monitor** if it is added to the ◷ **Scheduler** (See the "*Script Configuration*" section.)

Scheduled Jobs

Automatic script.job execution

A `script.job` can be scheduled to run automatically by Polarion at any time interval. See the "*Script Configuration"* section for details.

Permissions of a script.job

When executing a `script.job` manually via the Polarion ⋀•**Monitor** topic, the `script.job` executes on behalf of the user executing it. (It can only access the data that the user has permission to access.) This is important to understand because a `script.job` might have been created and tested by a user with full administrator permissions, but the `script.job` can unexpectedly fail because **the user executing it doesn't have the necessary permissions to access or modify the data.**

A `script.job` can be executed automatically via the Polarion ⊙**Scheduler**. When this happens, Polarion executes the script on behalf of the **polarion** user. This system user has read-only access to all the Polarion data. This is important to understand because if the `script.job` is expected to modify any data, then the automatic execution via the ⊙**Scheduler** will fail. The best practice is to implement the parts of the `script.job` that require write access to the data using the `doAsUser()` method as described in the "*Running parts of a script.job as another user with doAsUser"* section.

Note: Starting with **Polarion 2304**, `doAsUser` is disabled by default. See section on *Default API Security* to enable it.

Script Debugging

No debugger is available for `script.job`. The method typically used for debugging is by using the job logger. The job logger allows you to write to the job log associated with the running job, and can be accessed from the job monitor by clicking on the **log** link associated with the job.

The idea is to use the job log to populate it with any information you need for debugging your script. A job log is generated each time the job is executed manually (via the ⋀•**Monitor**) or automatically via the ⊙**Scheduler**. To use the logger, you need to use the `logger` object, for which different log message levels can be used:

JavaScript and Groovy

```
logger.info("This will generate an Info type message in the job log");
logger.error("This will generate an Error type message in the job log");
logger.warn("This will generate an Warn type message in the job log");
logger.fatal("This will generate an Fatal type message in the job log");
```

These generate these entries in the log:

```
2021-12-15 15:53:12,943 [Worker-0: My Script Job | u:admin | job: script.job] INFO  root  - This will generate an Info type message in the job log
2021-12-15 15:53:12,950 [Worker-0: My Script Job | u:admin | job: script.job] ERROR root  - This will generate an Error type message in the job log
2021-12-15 15:53:12,957 [Worker-0: My Script Job | u:admin | job: script.job] WARN  root  - This will generate an Warn type message in the job log
2021-12-15 15:53:12,963 [Worker-0: My Script Job | u:admin | job: script.job] FATAL root  - This will generate an Fatal type message in the job log
```

Script Examples

emailDigestJob.js

This job example sends a mail to all users in the repository (who have not turned off notifications) that are invited to sign a Document but have not signed it yet.

The scheduler configuration should be as follows: (Here, it is configured to execute every working day of the week at 6 am.)

```xml
<job name="Email Digest" id="script.job" cronExpression="0 00 6 ? * MON-FRI" disabled="false" scope="system">
    <scriptName>emailDigestJob.js</scriptName>
    <scriptEngine>js</scriptEngine>
    <properties>
        <subjectPrefix>[MY POLARION SERVER]</subjectPrefix>
        <senderAddress>any_email@polarion.com</senderAddress>
    </properties>
</job>
```

> ◆ **Warning:**
> For this job to work, your Polarion server must be properly configured with an SMTP server and be able to send email notifications.
> See the installation documentation for your operating system for details.

```javascript
/**
 ** Script:      emailDigestJob.js
 **
 ** Purpose:     This job will get a list of all users in the repository and notify them if:
 **                - Documents are awaiting their signatures
 **
 ** Polarion Job Parameters - these variables, while never defined are created by the script extension
 **    senderAddress : Senders (From) Address to send the send from
 **    subjectPrefix : Subject prefix for all outgoing emails.
 **
 **  Example Configuration
 **      This configuration will execute the job every working day in the week at 6am.
 **      <job cronExpression="0 00 6 ? * MON-FRI" disabled="false" id="script.job" name="Email Digest" scope="system">
 **        <scriptName>emailDigestJob.js</scriptName>
 **        <scriptEngine>js</scriptEngine>
 **        <properties>
 **          <subjectPrefix>[Polarion Server DEV]</subjectPrefix>
 **          <senderAddress>any_email@polarion.com</senderAddress>
 **        </properties>
 **      </job>
 **/

logger.info(" ");
logger.info("------------- Parameters -------------");
logger.info("subjectPrefix        = " + subjectPrefix);
logger.info("senderAddress        = " + senderAddress);

// Function sendEmail
// subject: Takes a string to use as the subject line of the email to be sent
// senderAddress: Takes a string to use as the sender address of the email to be sent
// emails: an array of emails addresses as strings --> string emails
// message: a string representing the HTML body of the emails
// ----------------------------------------------------------------------------
function sendEmail(subject, senderAddress, emails, message)
{
  // Sending an email notification by using the Polarion Announcement Service
  // ------------------------------------------------------------------
  var receiver = [emails];
  var headerString = "From: "+ senderAddress + "\n";
  headerString = headerString + "To: " + emails + "\n";
  headerString = headerString + "Subject: " + subject + "\n\n";
  var announcerSrv = com.polarion.platform.core.PlatformContext.getPlatform().lookupService(com.polarion.platform.announce.IAnnouncerService.class);
  var errMsg = "";
  if (announcerSrv !== 'undefined')
  {
    var Announcement = Java.type('com.polarion.platform.announce.Announcement');
    var announcement = new Announcement();
    announcement.setSender(senderAddress);
    announcement.setReceivers(receiver);
    announcement.setContentType("text/html");
    announcement.setSubject(subject);
    announcement.setContent(message);
```

```javascript
      try
      {
         announcerSrv.sendAnnouncement("smtp", announcement);
      }
      catch(ex)
      {
         errMsg = "Could not send email notification. Please, check mail settings for Polarion. \n\n" + ex;
         logger.error(errMsg);
      }
   }
   else
   {
     logger.info("We are not able to get the Announcer Service");
   }

   logger.info("\n-------------------------------- eMail notification -------------------------------- \n");
   logger.info(headerString + message + "\n\n");
   if(errMsg != "")
   {
     logger.error("Error Message: " + errMsg + "\n\n");
   }
}

// Constructor EmailMessage
// subject: Takes a string that will be the subject of the email message you are writing
// addresses: Takes an array of strings that are the email address to send this email to
// body: Takes a string that will be sent as the email body. Do not include beginning and ending <html> tags
//
// void addBodyLine(string): adds a new line to the body of the email
// string getBodyHTML(): returns the body of the email wrapped in <html> tags
// string createHTMLLink(string, string): returns a valid HTML link pointing to the URL with text text
// ---------------------------------------------------------------------------------------------------------
function EmailMessage(subject, addresses, body)
{
   this.subject = subject;
   this.addresses = addresses;
   this.body = body;

   this.addBodyLine = function(newLine) {
   this.body = this.body + "<br>\n" + newLine;
   }

   this.addBodyLineNoBR = function(newLine) {
   this.body = this.body + "\n" + newLine;
   }

   this.getBodyHTML = function() {
   return "<html>\n" + this.body + "\n</html>";
   }
}

function createHTMLLink(url, text)
{
    return "<a href=\"" + url + "\">" + text + "</a>"
}

// -------------------------------   MAIN CODE -------------------------------- //
try
{
   var dataService = trackerService.getDataService();
   var projectsService = trackerService.getProjectsService();
   var myUsers = projectsService.getUsers()

   for (i = 0; i < myUsers.size(); i++)
   {
     // Getting info on the users
     // -------------------------
     var myUser = myUsers.get(i);
     var myUserId = myUser.getId();
     var isUserDisabled = myUser.isDisabled();
     var myUserEmailAddress = "" + myUser.getEmail() + "";
     var isMyNotificationDisabled = myUser.hasDisabledNotifications();

     // Check if the user is disabled or if user notification is disabled or user mail id is null
     // -----------------------------------------------------------------------------------------
     if( isUserDisabled || isMyNotificationDisabled || myUserEmailAddress == "null" || myUserEmailAddress == "")
```

```
      {
        if(isUserDisabled)
        {
          // Skip processing this user because their account is disabled
          // -------------------------------------------------
          logger.info("Skip disabled user: " + myUserId + " email ID :" + myUserEmailAddress + "\n");
        }
        else if(isMyNotificationDisabled)
        {
          // Skip any further processing for this user (Because the user's email notifications are disabled)
          // ----------------------------------------------------------------------------------
          logger.info("skip processing user: " + myUserId + " Disable Notification status : " + isMyNotificationDisabled + "\n");
        }
        else if(myUserEmailAddress == "null" || myUserEmailAddress == "")
        {
          // Skip any further processing for this user (Because the user's email id is not set)
          // ---------------------------------------------------------------------
          logger.info("skip processing user: " + myUserId + " email ID is NOT SET, email ID: " + myUserEmailAddress + "\n");
        }
        continue;
      }

      // Query to find all documents a specific user has to sign
      // ----------------------------------------------------
      var documentsToSign = dataService.searchInstances("Module", "signatures:invited=" + myUserId, null);

      // No Documents to sign
      // --------------------
      if (documentsToSign.size() == 0)
      {
        logger.info("No signatures for: " + myUserId + "\n");
      }
      else
      {
        logger.info("Preparing for user: " + myUserId + "\n");

        // Creating the email message with the subject, the user email address and the start of the email
        // ---------------------------------------------------------------------------------------
        var message = new EmailMessage(subjectPrefix + " Polarion Awaiting Document Signatures", myUserEmailAddress, "<head><title>Polarion Awaiting
Signatures Digest</title></head><h2>Awaiting Your Document Signatures</h2>");

        // If the user is assigned to sign some Documents, then a table will be created in the email and every Document will be listed
        // ----------------------------------------------------------------------------------------------------------------
        if (documentsToSign.size() > 0)
        {
          message.addBodyLineNoBR("<table cellpadding=4 ><tr bgcolor=\"#E1F0FE\"><th>Title</th><th>Space</th><th>Project</th></tr>");

          for (j = 0; j < documentsToSign.size(); j++)
          {
            var myDoc = documentsToSign.get(j);
            var pId = myDoc.getProject().getId();
            myFullDocName = myDoc.getModuleFolder() + "/" + myDoc.getModuleName();
            message.addBodyLineNoBR('<tr><td>' + myFullDocName + '</td><td>' + myDoc.getModuleFolder() + '</td><td>' + myDoc.getProject().getName() +
'</td></tr>');
          }
          message.addBodyLineNoBR("</table>");
        }
        else
        {
          message.addBodyLineNoBR("<h4>No Awaiting Signature</h4>");
        }

        // Uncomment this to print the message body in the logger (for debugging purpose)
        // --------------------------------------------------------------------------
        // logger.info(message.getBodyHTML());

        // Send email (you can comment this line to not send any emails during debugging sessions)
        // --------------------------------------------------------------------------
        sendEmail(message.subject, senderAddress, message.addresses, message.getBodyHTML());
      }
  }
  result = true;  // This will add this message in the log file -> Script returned: true
}
catch (e)
{
```

```
  logger.error("Job generated an error:" + e);
  result = false; // This will add this message in the log file -> Script returned: false
}
```

Default API Security

Starting with Polarion 2304, we have restricted the list of `com.polarion.platform.security.ISecurityService` methods that can be invoked by default by `ScriptCondition`, `ScriptFunction` and `script.job`. Polarion prevents default usage of methods which are used for user management purposes (typically by administrators) to increase Polarion script security by preventing unintended elevation of user privileges.

Note that a script always runs with the privileges of the user on whose behalf the script is running.

(The user executing it, or a specific impersonated user via the `doAsUser` method).

> ⚠ **Warning:**
> - If you see strong reasons to leverage these methods blocked by default in `ScriptConditions`, `ScriptFunctions` and `script.jobs`, you can allow usage of specific methods by listing them via the below mentioned polarion.properties.
> - Polarion administrators need to be aware that they open the system to potential malicious scripts that can change user permissions (and more) if executed by a user with elevated privileges (i.e. administrator).
> - So extra precautions should be taken.
>   (Scripts deployed to Polarion must only come from trusted users, or be subject to code review before enabling them in Polarion.)

To unblock methods for `ScriptCondition` and `ScriptFunction`, add the following property to the polarion.properties file:

- `com.polarion.scripting.allowedSecurityServiceMethods`
  - For example: `com.polarion.scripting.allowedSecurityServiceMethods=doAsUser,removeGlobalRoleFromUser`

To unblock methods for `script.job`, add the following property to the polarion.properties file:

- `com.polarion.scripting.job.allowedSecurityServiceMethods`
  - For example: `com.polarion.scripting.job.allowedSecurityServiceMethods=doAsUser`

JavaScript Scripts compatibility with GraalVM

This section lists GraalVM compatibility issues for scripts that were written for Rhino or Nashorn.

1- When executing a JavaScript with the GraalVM engine, I am getting this type of error: `TypeError: (...) : Message not supported.`

Answer: This error is due to variables being declared using their fully qualified Java class name.

For example:

```
var currentDate = new java.util.Date();
```

With GraalVM, you must use `Java.type(typename)`

For example:

```
var Date = Java.type('java.util.Date');

var currentDate = new Date();
```

2- When executing a JavaScript with the GraalVM engine, I am getting this type of error: `ReferenceError: importPackage is not defined.`

Answer: This error is due to the use of an unsupported "`importPackage`" statement. (Probably from a script written for Rhino.)

For example:

```
importPackage(com.polarion.platform);
importPackage(com.polarion.platform.core);
importPackage(com.polarion.platform.context);
```

To fix it, first make sure you need this package. If not, delete it.

If you need it, for example:

```
importPackage(java.io);

(...)

var outFile = new FileWriter("./logs/main/NAME_OF_YOUR_SCRIPT.log");
```

With GraalVM, make use of `Java.type(typename)` to access specific classes:

```
var FileWriter = Java.type('java.io.FileWriter');

var outFile = new FileWriter("./logs/main/NAME_OF_YOUR_SCRIPT.log");
```

3- When executing a JavaScript with the GraalVM engine, I am getting this type of error: `Illegal char <:> at index X: nashorn:mozilla_compat.js`

Answer: This error is due to the use of an unsupported "`load("nashorn:mozilla_compat.js");`" statement. (Probably from a script converted from Rhino to Nashorn.)

To fix it, delete the statement completely. You might need to rewrite how you are including packages. See "*FAQ Question #6 and #7*" for more information.

4- When executing a JavaScript with the GraalVM engine, I am getting this type of error: `ReferenceError: JavaImporter is not defined.`

Answer: This error is due to the use of an unsupported "`JavaImporter`" statement. (Probably from a script written for Nashorn.)

For example:

```
var JavaPackages = new JavaImporter(java.io,
                                    java.text,
                                    java.lang
                        );
```

With GraalVM, you should define a variable for every class of a specific type you want to access by making use of `Java.type(typename)`.

If you need it, for example:

```
var JavaPackages = new JavaImporter(java.io);

with( JavaPackages )
{
    var outFile = new FileWriter("./logs/main/NAME_OF_YOUR_SCRIPT.log");
}
```

With GraalVM, it should be written as:

```
var FileWriter = Java.type('java.io.FileWriter');

var outFile = new FileWriter("./logs/main/NAME_OF_YOUR_SCRIPT.log");
```

5- When executing a JavaScript with the GraalVM engine, I cannot use java.lang.String methods on Strings.

Answer: GraalVM treats strings as JavaScript Strings and not Java Strings.

For example:

```javascript
var MyString = "This is my String";

if(MyString.equals("This is my String")
{
  // The strings are equal!
}
```

Will generate this error: *TypeError: MyString.equals is not a function*

With GraalVM, it should be written as:

```javascript
var MyString = "This is my String";

if(MyString.valueOf() == "This is my String")
{
  // The strings are equal!
}
```

The list of JavaScript Strings can be found here: https://www.w3schools.com/jsref/jsref_obj_string.asp

The Nashorn scripting engine treats strings as `java.lang.String` by default. Because of this, scripts written for Nashorn that use strings may need to be adapted for GraalVM.

6- When executing a JavaScript with the GraalVM engine, I am getting this type of error: `Cannot convert 'XX.YY'(...) to Java type 'float': Invalid or lossy primitive coercion.`

Answer: Setting Float type fields with GraalVM engine needs to be done explicitly.

For example:

```javascript
var priority = 7.505;

workItem.setPriority(workItem.createPriorityOpt(priority.toFixed(2)));
```

Will generate this error: *Cannot convert '7.505' (...) to Java type 'float': Invalid or lossy primitive coercion.*

With GraalVM, it should be written as:

```javascript
var priority = 7.505;

var Float = Java.type('java.lang.Float');

workItem.setPriority(workItem.createPriorityOpt(Float.valueOf(priority.toFixed(2))));
```

7- When executing a JavaScript with the GraalVM engine, I am getting this type of error: `'X.Y' is not instance of java.lang.Float.`

Answer: GraalVM treats conversion from String to Float differently than previous JavaScript engines.

To do so, you can use this method:

```javascript
var floatValue = "2.0";

workItem.setCustomField("myFloatFieldID", parseFloat(floatValue));
```

or this method:

```javascript
var Float = Java.type('java.lang.Float');

var floatValue = new Float('2.0');

workItem.setCustomField("myFloatFieldID", floatValue);
```

8- When executing JavaScript with the GraalVM engine, some properties suddenly become 'undefined'.

Answer: When using Nashorn, directly accessing an object property using the 'object.property' syntax automatically invokes the 'object.getProperty()' method. In contrast, GraalVM JS requires explicit use of the getter method.

9- When executing a JavaScript with the GraalVM engine, I am getting this type of error: `Cannot convert '1'(language: Java, type: java.lang.Integer) to Java type 'java.lang.String': Invalid or lossy primitive coercion.`

Answer: To fix this error, you need to explicitly convert the Integer value to a String value before using it in a context where a String is expected.

For example:

```javascript
var str = String.valueOf(1);
or
var str = Integer.toString(1);
or
(1+1).toString();
```

FAQ

1- Where can I go to get help with scripts and Polarion API?

Answer: The best place to ask questions and get answers from other Polarion users and experts is on the Polarion Community website.

2- Do we need to call `.save()` on the object (`workflowContext.getTarget()`) from which the workflow execute in a `ScriptCondition`?

Answer: You should never call `.save()` on the object that executes the workflow in a `ScriptCondition`. (It is not a script type meant for modifying Polarion Data.)

3- Do we need to call `.save()` on the object (`workflowContext.getTarget()`) from which the workflow execute in a `ScriptFunction`?

Answer: You should never call the `.save()` on the object that executes the workflow in a `ScriptFunction` because the script is already executing when Polarion saves the base object. If your `ScriptFunction` modifies another object (for example, a Work Item *linked* to the current Work Item that the workflow executes on), then yes, `.save()` should be called on this other object.

4- I have updated the Scheduler entry for my `script.job` but when I try to run it manually in the Monitor, it does not pick up my modifications.

Answer: If you are updating the ⏱ **Scheduler** entry of your job, make sure to refresh the ⋀ **Monitor** page to pick up the changes.

5- My `script.job` is failing when executed automatically via the scheduler but not when I run it manually.

Answer: This means that the job does not have the same access to the Polarion data when it is executed automatically by the ⏱ **Scheduler** because it uses the **polarion** user. (That only has read-only access to the data.) To execute it successfully, you must use the `doAsUser` method.
See the "*Running parts of a script.job as another user with doAsUser*" section for details.

6- When executing a Groovy script an error is generated when trying to retrieve the current Groovy version.

Answer: The error is due to the use of an unsupported `getVersion()` call, probably from a Groovy script that was written for v1.5.7.

For example:

```
org.codehaus.groovy.runtime.InvokerHelper.getVersion();
```

With Groovy 2.4.21, it should be written as:

```
GroovySystem.getVersion();
```

7- When executing a `script.job`, I am getting this type of error: `Method (...) cannot be executed from a job script.`

Answer: Starting with Polarion 2304, some methods from `com.polarion.platform.security.ISecurityService` were deactivated by default. You can unblock methods for `script.job` that are blocked by default by defining them in the following property to the polarion.properties file:
`com.polarion.scripting.job.allowedSecurityServiceMethods`

For example, if you want to unblock `addGlobalRoleToUser()` and `addContextRoleToUser()`:

```
com.polarion.scripting.job.allowedSecurityServiceMethods=addGlobalRoleToUser,addContextRoleToUser
```

For more information on the property, consult the *Default API Security* section of this guide and *Polarion System Configuration Properties Reference* guide on Support Center.

8- When executing a `ScriptFunction` or `ScriptCondition`, I am getting this type of error: `Method (...) cannot be executed from a script.`

Answer: Starting with Polarion 2304, some methods from `com.polarion.platform.security.ISecurityService` were deactivated by default. You can unblock methods for `ScriptFunction` and `ScriptCondition` that are blocked by default by defining them in the following property to the polarion.properties file `com.polarion.scripting.allowedSecurityServiceMethods`.

For example, if you want to unblock `addGlobalRoleToUser()` and `addContextRoleToUser()`:

```
com.polarion.scripting.allowedSecurityServiceMethods=addGlobalRoleToUser,addContextRoleToUser
```

For more information on the property, consult the *Default API Security* section of this guide and *Polarion System Configuration Properties Reference* guide on Support Center.