Autodesk University 2012

You Can Do That?!
Scripting with Autodesk PLM 360 Part 1

Jared Sund - Autodesk

PL3655-P In this class, you will discover how to take your Autodesk PLM 360 integration to the next level. Using server-side JavaScript, you will learn how to use and modify the existing scripts within Autodesk PLM 360 for workflow transitions and behaviors. For more advanced scripting, follow this class with "The Answer Is Yes: Scripting with Autodesk® PLM 360, Part 2."

Learning Objectives

At the end of this class, you will be able to:

- Use condition scripts on workflow transitions
- Take advantage of validation scripts on workflow actions
- Explain how action scripts return and set values in workspaces
- Debug and test scripts in Autodesk PLM 360

About the Speaker

Jared is an Autodesk PLM 360 Product Manager residing in the Lake Oswego, Oregon office. Jared's main focus areas within PLM 360 include: the scripting engine, REST API, Enterprise Integrations, Administration/Configuration, along with other areas of the tool. In addition to managing aspects of the PLM 360 product, Jared is also called upon for his technical knowledge in developing different customer solutions. Jared came to Autodesk in 2011 after spending 10+ years supporting and implementing PLM, PDM, CAD, CAM, and analytical engineering tools for companies that include: Xerox Corp., L3 Communications, Raytheon, and Tektronix. He also holds a BS in Computer Science from Portland State University.

Email: jared.sund@autodesk.com

PLM 360's server-side scripting engine provides tools and functionality to take your PLM 360 solutions to the next level. While scripting is not required to build solutions in PLM 360, the functionality and tools offered here can increase the capability and automation of your solutions. Through this course we examine the basic to PLM 360 scripting, and provide a strong foundation for the PLM 360 Script 2 class.

Server-Side Scripting Overview

PLM 360's scripting engine offers a server-side JavaScript feature to extent your business solutions through advanced workflow and data management capabilities. There are two topics worth noting before we dive into PLM 360 scripting: JavaScript the language, and server-side programming.

A wealth of knowledge about JavaScript is widely available in books and on the Internet, so I will not go into great detail about the language or its roots here. What is worth noting about the language is that it is not Java... While JavaScript's language syntax and base libraries were based on Java, the similarities stop there. Unlike Java, JavaScript is not a strong typed language. This makes creating simple or even complex JavaScript scripts much easier. The JavaScript engine interprets the variable type based on each of the variables contents and the operations performed on them. For example, in JavaScript the arithmetic operator (+, addition) is the same as the string concatenate (+) operator.

Arithmetic Operator *(number plus number)*

```
var a = 2;  //number
var b = 2;  //number
var c = a+b;  //the content of c is 4 = 2+2, a number
```

String Operator *(number plus string)*

```
var a = 2;  //number
var b = '2';  //string
var c = a + b // the content of c is '22' = 2 + '2', a string
```

As you can see from the example above, the + operator either performs simple addition when all values are numbers or concatenates (adds) all values together to form a single string. In strongly typed languages, such as Java, the programmer is required to maintain operations based on similar data types. JavaScript's weakly typed convention removes the headaches for us the programmers to ensure that we're either converting values into similar types before operations, or worrying about the type of value all together. Thus, making writing scripts much easier to write and manage.

Another major difference between Java and JavaScript is the object oriented styles of the two languages. JavaScript implements what is called a prototype based object-oriented

programming style. We could fill the whole class discussing object-oriented programming and the styles implemented by JavaScript or Java, but I'll leave that for another class.

In the table below, I've outlined some simple and common syntax of the JavaScript language.

Comments	//Single line comment	Comments are a great way to document your scripts and are ignored by the scripting engine
	/* Multiple line comments Multiple line comments */	A multiline or block comment is great for creating a header at the top of your script. Everything inside /* …… */ is ignored by the scripting engine
Variables	var instructor = 'Jared Sund';	The variable instructor contains the string – Jared Sund
	var qty = 7;	The variable qty holds the integer number 7
	var price = 3.25;	The variable price hold the decimal number 3.25
	var subTotal = qty * price;	The variable subTotal holds the decimal number 22.75 (7*3.25)
Array	var students = []; students[0] = 'Tom'; students[1] = 'Jane';	An array is a list of items

Conditions	If(subTotal > 150)
	{
	tax = .05;
	}
		Conditions are questions. When the question is true (the value in the variable subTotal is greater than the number 150), perform the instructions with the body ({ …… })

Loops	for(var i =0; i < students.lengh; i++){
	println(students[i]);
	}
		Loops are used to iterate through arrays (lists), or can be used to iterate until a condition (question) is true (see while loops)

To learn more about JavaScript, I've supplied a couple of links below that I like to frequent:

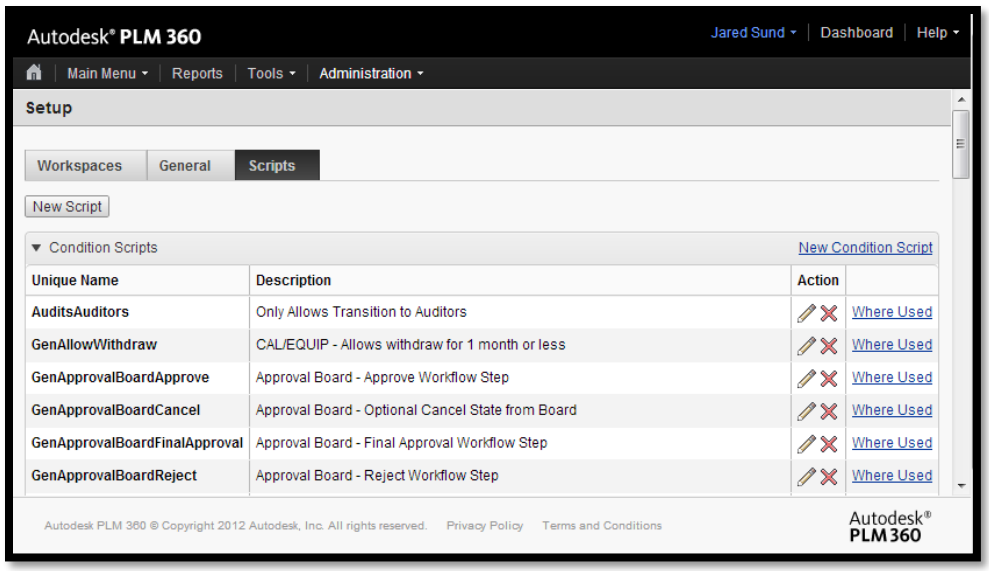
- http://en.wikipedia.org/wiki/JavaScript - General history and overview
- http://www.w3schools.com/js/default.asp - A great tutorial and reference for the language.

Also, the O'Reilly, JavaScript Pocket Reference is a great inexpensive reference book to accompany you on your JavaScript journey. I have a copy, and commonly have it sitting on my desk for quick access!

As you start exploring websites and books about JavaScript you'll notice a great deal of information regarding the HTML DOM and client side scripting. You can simply ignore these sections. The scripting engine in PLM 360 is server-side, which means you do not have access to client objects such as: navigator, event, window, and document. As we explore PLM 360 scripting, you will see that our scripts run on the PLM 360 server against the PLM 360 standard objects, not in our client-side browser. For a complete review of the language specification, please see ECMA-262 3rd edition:

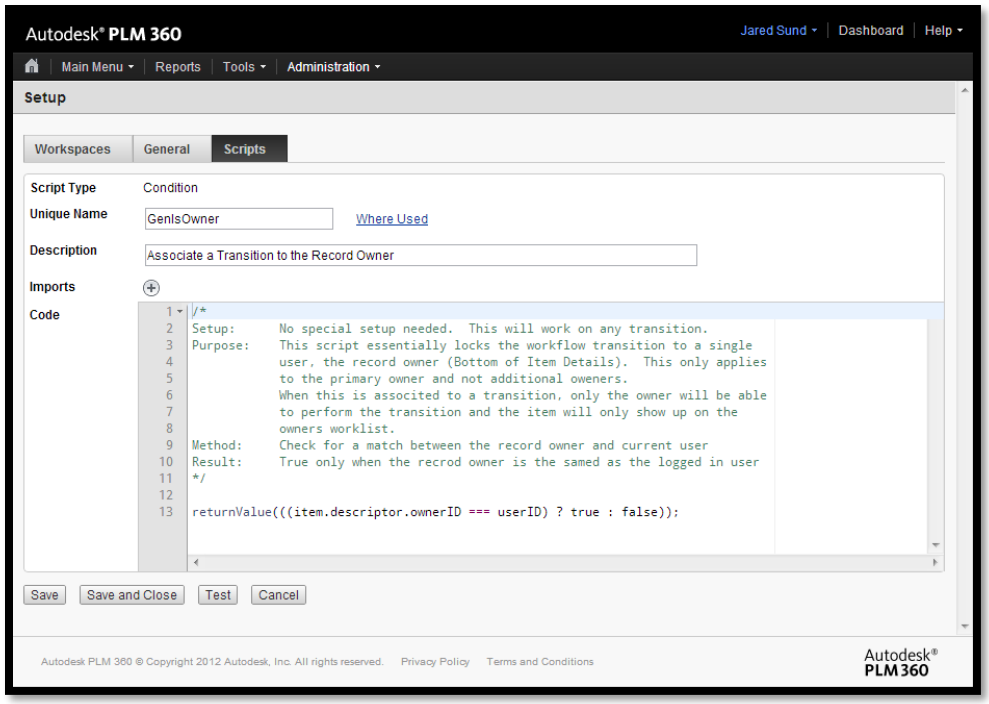
http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf

Enough about JavaScript, let's start discussing the topic at hand, PLM 360 Scripting. All of your PLM 360 scripts are available in the **Administration**, **Setup**, **Scripts** screen. From this single location, you have all the tools you need to create and manage your PLM 360 scripts.



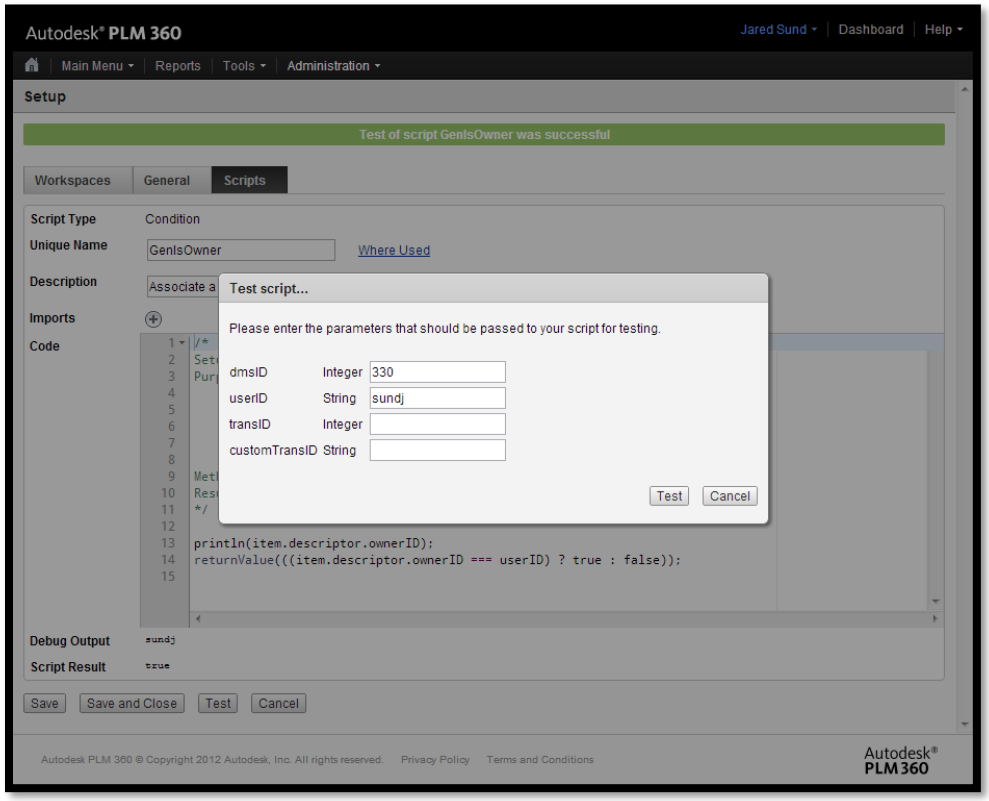
This page offers tools to perform the following functions: create new scripts, edit and delete existing scripts, and view where each particular script is currently used.

PLM 360 also provides an easy to use ACE (http://ace.ajax.org) embedded script editor, directly within the PLM 360 web interface. There are no additional tools needed to begin adding scripts to your PLM 360 solution, they are all delivered to your web browser from the PLM 360 cloud.



The PLM 360 editor provides syntax highlighting to make scripts easier to read and edit, along with helpful warning and error messages to aid you in working with and developing PLM 360 scripts.

Along with tools to help you manage and create scripts, PLM 360 offers tools to help you test and debug your scripts. While creating scripts, you can test your work as part of the development process. When you select Test from the script editor, you will be presented with a "Test Script" dialog to add information to test your script against. One of the main system variables you will likely set in the "Test Script" dialog is a dmsID (record ID) to test your script against. We will cover dmsID in more detail as we continue through this class.



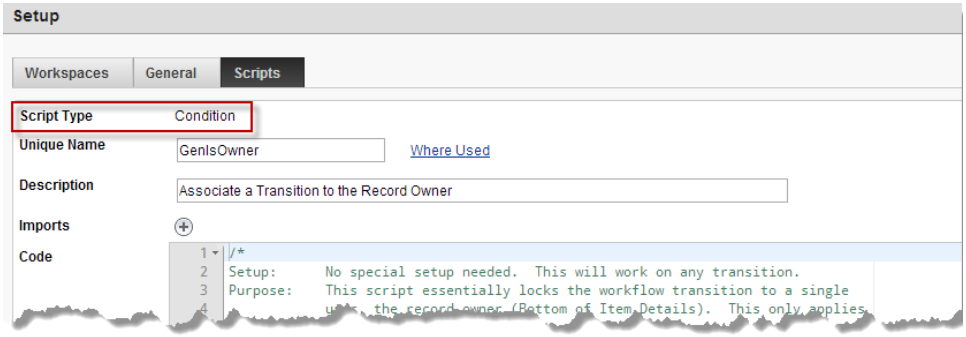
To help with debugging and testing scripts, the scripting engine provides print() and println() functions. When used during testing, these functions can be used to write information to the "Debug Output" section of the script editor screen. As an example, notice in the image in the screen print above that we have issued the instruction,

`println(item.descriptor.ownerID);` , which prints the value of (`item.descriptor.ownerID`) to the Debug Output. The use of println() and print() in conjunction with testing your script will be an important part of your PLM 360 scripting experience. For more details on testing and debugging your PLM 360 scripts, see:

http://wikihelp.autodesk.com/PLM_360/enu/Help/Help/0086-Develope86/0087-Server-S87/0090-Testing_90

Script Types and Events

Each PLM 360 script is created as one of four scripting types: Condition, Validation, Action, or Library scripts.



We will discuss these four scripting types briefly, and an in-depth review of each can be found in the About Scripting page located here: http://wikihelp.autodesk.com/PLM_360/enu/Help/Help/0086-Develope86/0087-Server-S87/0088-About_Sc88

Condition Scripts: These scripts are used to block or allow access to workflow transitions. Condition scripts are set and triggered from workflow transitions. They have a return of true or false. If a condition script is set as a pre-condition for a transition, a return value of true allows the access to the transition, and a return value of false blocks access to the transition.

Validation Scripts: Validation scripts validate information and provide end user feedback if a validation does not pass the criteria listed within your script. Validations, like condition scripts are also triggered from workflow transitions. When linked to a transition as a validation, this script type will return an empty array (all validations passed), or an array with one or more messages (strings) to prompt the user of needed changes. If the return value array is empty the transition will succeed, otherwise it will not succeed and the end user will be notified of the pending errors.

Action Scripts: Action scripts unlike condition and validation scripts can both read and write information. Like the name suggests, action scripts are used to perform some action. These scripts have no return type and can be linked to workflow transitions or workspace behaviors (add item, and edit item

details).

Library Scripts: Library scripts are used to help make scripts more manageable. They hold one or more JavaScript functions or objects. They are linked (imports) to condition, validations, actions, or other library scripts and allow you to centralize scripting functionality that can be reused. Creating and using libraries is a common programming method to make your scripts more manageable and easier to use.

For all PLM 360 scripts, we must be aware that scripts DO NOT maintain PLM 360 permissions. This means that scripts can perform more tasks than the user who is performing the action that triggers the script. So, a user may have access to perform a workflow action (transition) that may read and/or write information to a location that user doesn't not have permission to perform. In short, a PLM 360 script's trigger (event that fires the script, such as a workflow action, or workspace behavior) controls whether or not the user can run the script. Once the script is running, it may modify, read, or create PLM 360 information that the user does not directly have permission to perform. This scripting permission system allows greater flexibility for creating scripts, but it is something we should be mindful of when we're creating scripts.

Scripting Object and Functions

Before writing our own and editing existing scripts, we need to examine how we access PLM 360 information within our scripts. When each script is executed, it is supplied a loaded object (item) for the record this script is acting upon. To better understand, if a validation script is fired from a user performing a workflow action (workflow transition), then the validation script will be pre-loaded with the item object.

For more details, see the "item" object and Access to Item Fields section in the About Scripting page for more details:

http://wikihelp.autodesk.com/PLM_360/enu/Help/Help/0086-Develope86/0087-Server-S87/0088-About_Sc88

This item object contains properties and functions that allow our scripts to interact with PLM 360 Workspace records. The table below illustrates many of the widely used properties and functions associated with each item object.

Property	Description	Examples
descriptor	Properties for the item object (meta-fields)	• item.descriptor.workflowState • item.descriptor.ownerID
fields	Read/write access to custom fields added to item details or the grid tab	• item.TITLE • item.QTY
grid	Read/write access to the rows and columns of a grid[row][column]	• item.grid[1].QTY • item.grid[2].TITLE • item.grid.addRow(row) • item.grid[1].remove()
milestones	Read/write access to items in the milestones tab	• item.milestones[1].progress • item.milestones[1].milestoneDate
attachments	Read access to items in the attachments tab	• item.attachments[1].fileStatus • item.attachments[1].fileSize
workflowActions	Read only access to workflow actions history (first is the most recent)	• item.workflowActions[0].userID • item.workflowActions[0].timeStamp
functions	Additional operations that can be performed on an item	• item.performWorkflowTransition • item.addMilestone • item.deleteItem

In addition to accessing a records item object, the PLM 360 scripting engine also has access to specific PLM 360 functions. The following table outlines scripting functions available to use without our scripts.

function	Description	Example
createItem	Creates a new record in a given workspace	• var newItem = createItem(workspace ID);
getPrintView	Returns the rendered html body of an Advanced Print View	• var body = getPrintView(APV name);
loadItem	Returns an existing item by dmsID	• var relatedItem = loadItem(dmsID);
security	A set of functions to return user/group/role information	• var user = Security.loadUser(userID); • var inGroup = Security.inGroup(group name);
Email	Create and send emails from scripts	• var email = new Email(); • email.to = 'jared.sund@autodesk.com'; • email.subject = 'This is a test email'; • email.body = getPrintView('Item Details'); • email.send();
print/println	Used when testing scripts, writes the debug section	• println(item.descriptor.workflowState);
Logger	Writes to the item's Change Log	• Logger.log('Owned by: ' + item.descriptor.ownerID);
XMLHttpRequest	Access external web services	Come to Scripting part 2 – to see this new functionality!

For a complete listing of PLM 360's scripting object and functions, please see the Scripting Reference page at:

http://wikihelp.autodesk.com/enu?adskContextId=PLM360_HELPID_DG_SCRIPTING_REF&language=enu&product=PLM_360

"Hello World" Scripts

Now that we have a basic understanding of server-side JavaScript, PLM 360 scripting interface, scripting types/events, and the PLM 360 object and functions, let's start writing and modifying scripts.

For our first scripts, we'll create or modify each of the four scripting types (condition, validation, action, and library). I have setup each of these examples in a method that will be very similar to

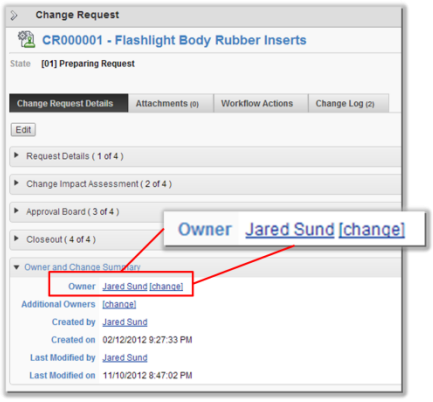
how you will provide solutions using PLM 360's scripting engine. First we examine a problem, determine our objective, create a plan, and identify the resources required to develop our scripts.

"Hello World" Condition Script:

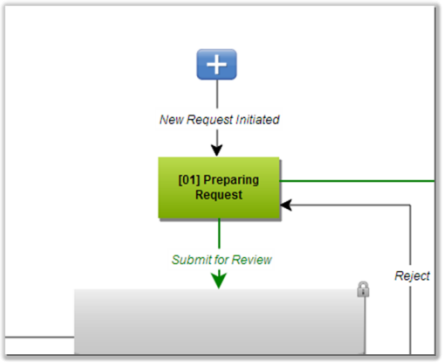
Problem: In our PLM 360 Change Orders, the general permission to create and submit Change Orders allows users to submit change orders which they do not own.

Objective: We would like to only allow the Owner of a change order to be able to submit their Change Order for review and approval.

Plan: Use a condition script to block the submission workflow action for a Change Order to all permitted users, except for the Change Order's Owner.



Block submit transition for all users except for the record Owner



Resources: Within our condition script we will need access to the Change Order's Owner (`item.descriptor.ownerID`), and the current user accessing the workflow transitions (`userID`).

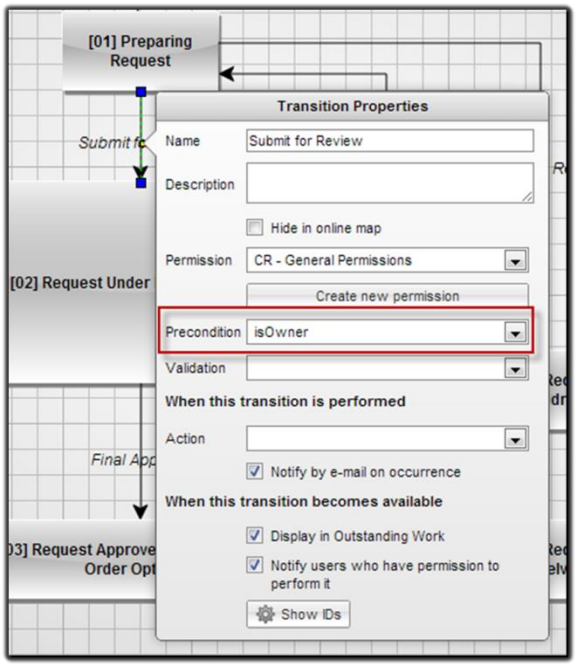
Condition Script (*isOwner*):

```
/*
Purpose:    Blocks a transition for all, other than the record owner
Method:     Check for a match between the record owner and current user
Result:     True only when the record owner is the same as the logged
in user
*/
var returnVar = false;  //boolean variable for the return assignment
if(item.descriptor.ownerID === userID){
    returnVar = true;
}
//If returns true, then the transition will be available
returnValue(returnVar);
```

Event:

Our condition script, isOwner, will be used as the Precondition for the Submit for Review Workflow Transition.

This will block all users from this transition, which are not the Change Order's listed Owner.

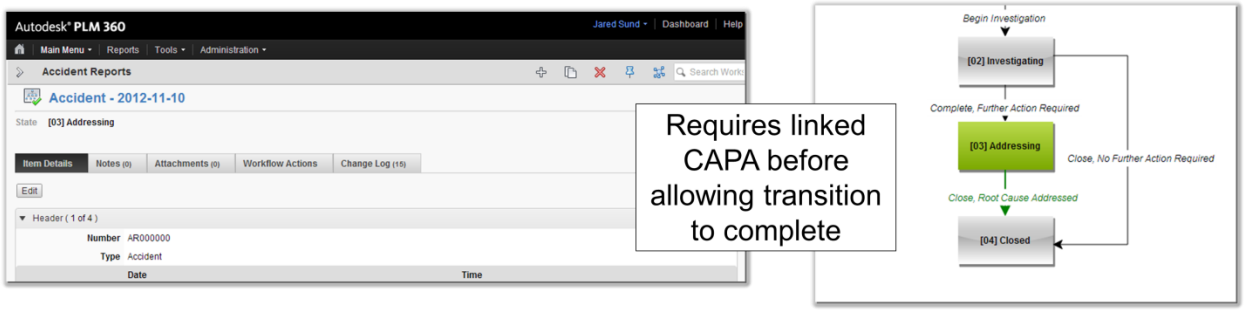


"Hello World" Validation Script:

Problem: Our company has a new HR policy that mandates a CAPA (corrective/preventative action) must be created and linked to any accident report that is of type accident.

Objective: To ensure that all accident reports of type accident have a linked CAPA associated.

Plan: Use a validation script to validate that a CAPA is linked to an accident report of type accident, before the "Close, Root Cause Addressed" transition will succeed.



Resources: We will need to create a linked picklist in our Accident Report to a CAPA record (fieldID: CORRECTIVE_PREVENTATIVE_ACTION), which we can test to make sure it is populated. We will also need to test the existing type field (fieldID: TYPE), to see if the

accident report is of type “Accident”. Since our solution already has a validation script (AccidentReportsRequiredFields), we can simply add our new validation test to the existing script. We will also need to include a test in our existing script to only run this portion of the validation for a specific transition (164) in our Accident Reports workflow.

Validation Script (*AccidentReportsRequiredFields*):

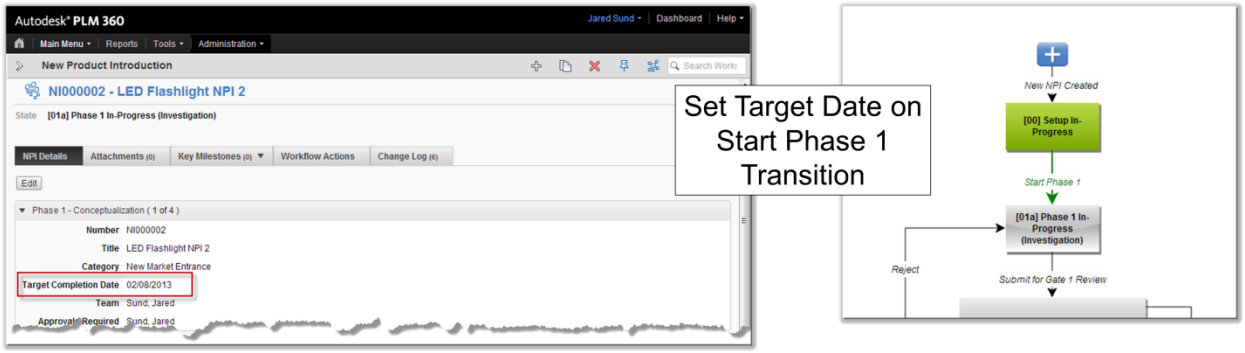
```
/*
Setup:      Fields in the Incident Description section are needed
Purpose:    This script tests that the incident description fields are
populated
Method:     Test each field to not be empty or null
Result:     Returns "messages" and they're presented to the user when
items are empty
*/

//Create an array for us to place the messages for the errors if
present.
var messages = [];

if(item.DEPARTMENT_OF_OCCURENCE === null){
    messages.push('Department of Occurrences is required to complete
this action');
}

.
.
.// existing code

if( transID === 164){
    if(item.TYPE === 'Accident' &&
      item.CORRECTIVE_PREVENTATIVE_ACTION === null){
        messages.push('A CAPA  must be assigned before this transition
can be completed');
    }
}

returnValue(messages);
```

“Hello World” Action and Library Script:

Problem: Our current NPI (New Product Introduction) projects do not have a standard target for the duration of each project.

Objective: To create a default duration of 90 days for each NPI project that is started.

Plan: Once a NPI project is submitted, we will use an action script to set a target completion date 90 days from the date of submission.



Resources: Create a new field in our NPI workspace item details to hold the target completion date (fieldID: TARGET_COMPLETION_DATE). Create a JavaScript function (getDateFromNow) to calculate 90 days from the current date, and add this to a new library script for general date functions (DateFunctions).

Library Script (*DateFunctions*):

```
//provides a date that is some number of days from today
function getDateFomNow(daysOffset){

    var targetDate = new Date();
    targetDate.setDate(targetDate.getDate()+ daysOffset);

    return targetDate;
}
```

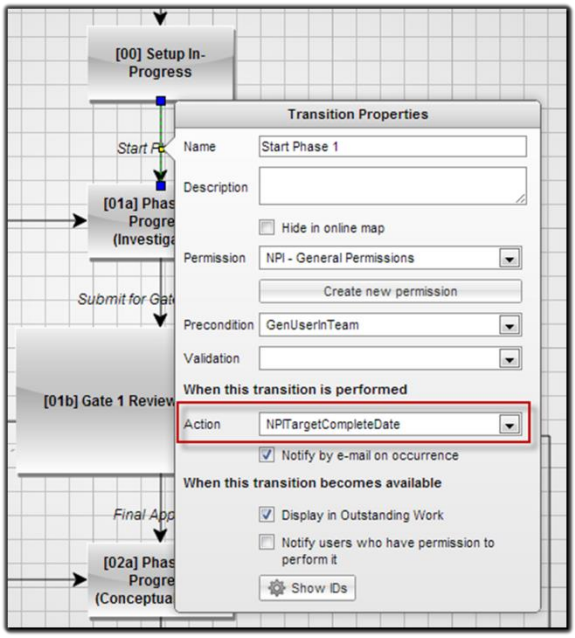
Action Script (*NPITargetCompleteDate*):

```
/*
Purpose:    Set the Target Completion Date for an new NPI item.
Method:     Add the specified number of days to the creation date.
Result:     NPI item's target completion date will be automatically
set.
*/

var daysOffset = 90;
var targetCompletionDate = getDateFomNow(daysOffset);
item.TARGET_COMPLETION_DATE = targetCompletionDate;
```

Event:

Add the newly created NPITargetCompleteDate action script to our Start Phase 1 NPI Workflow transition.

When the workflow action for this transition is invoked, a target completion date will be created 90 days from the current date.

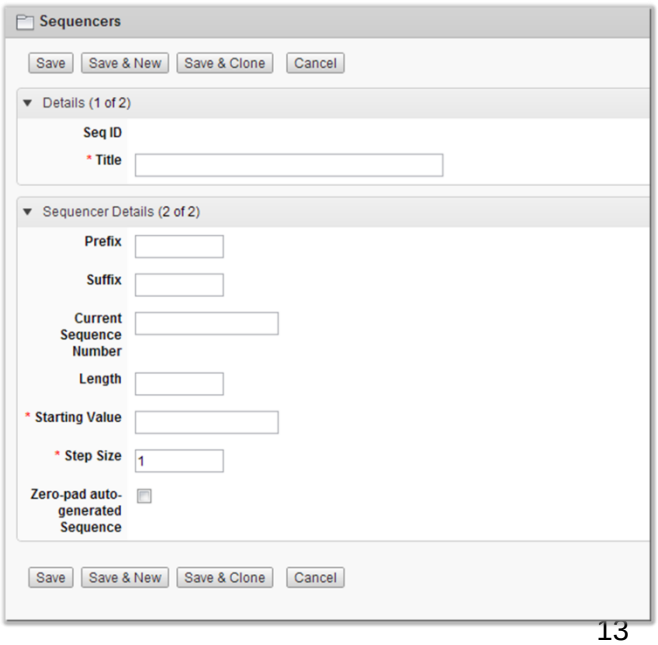


Standard Scripts Review

To complete our PLM 360 Script 1 class, we will review an Advanced Sequence Generator, and review properties in a grid object to determine whether an Inspection has passed or failed.

Advanced Sequence Generator (ASG):

PLM 360 provides a field type, Auto Number, to quickly create basic sequencing operations. However, there may be situations that require more advanced sequence generation. The PLM 360 standard tenant contains a solution that provides a sequence generator to include prefix, suffix, a numeric starting point, a defined length for the number of digits in the numeric sequence, step size, and the option to zero pad the numeric sequence.



To create the ASG, we only require four PLM 360 objects: A workspace to house the definition and values for each ASG, a library script to control the logic of the ASG, an action script to perform the tasks of the ASG, and an event to trigger the scripts.

Using a workspace for the ASG provides a location to create as many generators as we need. Not only will each record in the ASG workspace hold individual definitions, each record also provides a storage location for the Current Sequence Number.

The sequenceOperator library contains two functions: nextSeqNumber, and zerofill. The nextSeqNumber function takes the ASG record id (dmsID) as an input, which loads the ASG record into an item record (seqGenerator). This record is loaded using the loadItem(dmsID) function, with provides our script access to the definition and current sequence number for our ASG. Notice that this loaded seqGenerator object works in the manor as our normal item object. As you review the code below, you'll notice that the function is building a return value of adjustedNo. Each property with the ASG record loaded is reviewed and appended to our return value. The zerofill function is in the case that zero padding was selected within our ASG record definition. This function prepends 0's to match the length requirement specified by the ASG.

```
function nextSeqNumber(SEQID){
    var seqGenerator = loadItem(SEQID);
    if(seqGenerator === null)
    {
        return null;
    }
    var prefix = seqGenerator.PREFIX;
    var sufix = seqGenerator.SUFIX;
    var stepSize = seqGenerator.STEP_SIZE;
    var includePadding = seqGenerator.ZEROPAD_AUTOGENERATED_SEQUENCE;

    var sequenceNo = parseFloat(seqGenerator.CURRENT_SEQUENCE_NUMBER);
    seqGenerator.CURRENT_SEQUENCE_NUMBER = ''+(sequenceNo+stepSize);

    var adjustedNo = '';
    if(includePadding === true){
        adjustedNo = '' + zeroFill(sequenceNo,padding, '0');
    }
    else{
        adjustedNo = '' + sequenceNo;
    }

    if(prefix !== null){adjustedNo = prefix + adjustedNo;}
    if(sufix !== null){adjustedNo = adjustedNo + sufix;}

    return adjustedNo;
}
```

```
function zeroFill(number, width, pChar)
{
  width -= ('' + number).length;
  if ( width > 0 )
  {
    return new Array( width + (/\./.test( number ) ? 2 : 1) ).join(
pChar ) + number;
  }
  return number;
}
```

For an example use of the ASG, let's review the RFQSeqGen action script. The design intent for sequence generation is that each new RFQ receives a new RFQ number, and cloned RFQ records maintain the original RFQ number. So, our action script needs to first test if the RFQ_NUMBER field is empty(null). If the field is empty, then this is a new record and requires a new RFQ number. If the RFQ_NUMBER is not empty, then we do not need a new RFQ number.

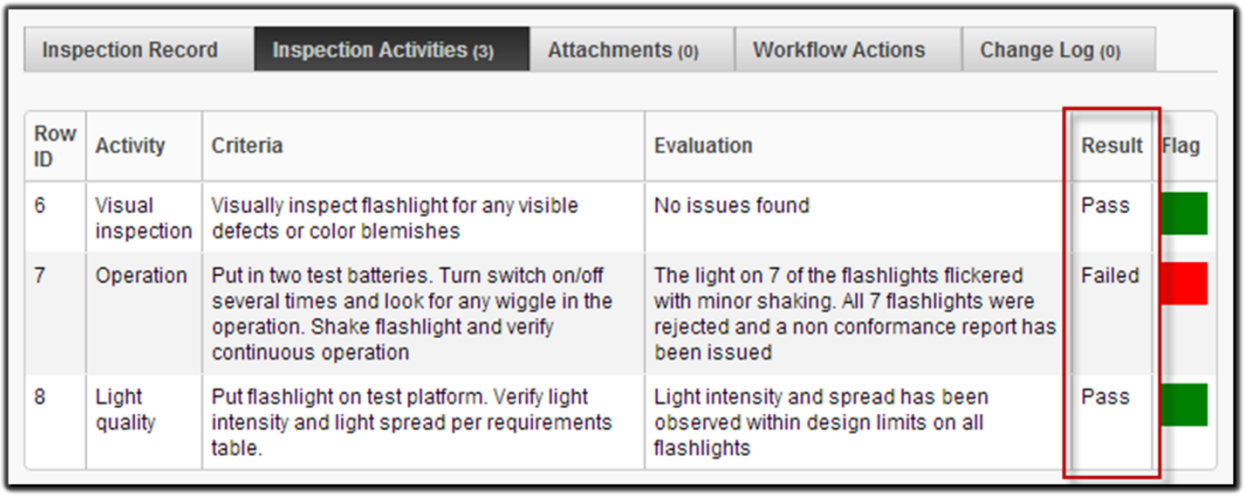
```
/*
Setup:      Needs RFQ_NUMBER field to write to
Purpose:    creates a new RFQ number, only for new, not cloned
Method:     If existing field is null, generate new number
Result:
*/
var RFQSeqID = '332';
if(item.RFQ_NUMBER === null)
{
  //set the new number
  var newRFQNumber = nextSeqNumber(RFQSeqID);

  if(newRFQNumber !== null){
      item.RFQ_NUMBER = newRFQNumber;
  }
}
```

The RFQSeqGen action script is triggered using the Workspace behavior "Script to run at item creation".

Through the use of a workspace to hold many different ASG definition, and library scripts, we easily implement this behavior with a simple action script to any workspace we need.

InspectionPassedFailed (Grid):

For our final topic, let's review how a condition script evaluates the Inspection Activities (grid) of an inspection. The standard solution for inspections contains a grid tab to hold all inspection and inspection results for a particular inspection operation.



Inspection Record | Inspection Activities (3) | Attachments (0) | Workflow Actions | Change Log (0)

Row ID	Activity	Criteria	Evaluation	Result	Flag
6	Visual inspection	Visually inspect flashlight for any visible defects or color blemishes	No issues found	Pass	
7	Operation	Put in two test batteries. Turn switch on/off several times and look for any wiggle in the operation. Shake flashlight and verify continuous operation	The light on 7 of the flashlights flickered with minor shaking. All 7 flashlights were rejected and a non conformance report has been issued	Failed	
8	Light quality	Put flashlight on test platform. Verify light intensity and light spread per requirements table.	Light intensity and spread has been observed within design limits on all flashlights	Pass	

The intent of our script is to look through each row within our grid to look for "Failed" or empty(null) results. Since our Inspection Activities grid could contain 0 or more activities, our code needs to be able to review any number of rows that are added to the grid object.

```
//Transition ID used in this code.  This transition ID are required for
//this script to operate
var CLOSEDFAILED_TRANSID = 40;      //Close - Failed

//This is an identical script to the InspectionPassed script other than
//the return value (last line of script)
var grid = item.grid;
var InsPassed = true;

//Loop through the item grid and look for any inspections that are
//either Failed or not reported (null)
//If such inspection lines are found, change InsPassed variable to
false
for (var index in grid){
    var gridRow = grid[index];
     if (gridRow.RESULT == "Failed" || gridRow.RESULT === null) {
      InsPassed = false;
     }
}
```

```
if(transID ===  CLOSEDFAILED_TRANSID){
     //Return the opposite of what InspectionPassed script returns
     returnValue (!InsPassed);
}
else{
     //Return the value of what InspectionPassed script returns
     returnValue (InsPassed);
}
```

You will notice that the grid for an item is accessed through item.grid, and we've added this item's grid to a variable var grid = item.grid to make the script easier to read. Also notice that a grid item is comprised of rows and columns (grid[rowIndex][column]). The rowIndex is a numeric value that indicates each row (i.e. – var gridRow1 = item.grid[1]). Each column in each row is identified by the grid column index (numeric) or by the fieldID. With a for loop, we're able to iterate (loop) through each row while testing specific columns for matching values ("Failed", or null).

We will cover temporal and value based conditions in more detail in PLM 360 scripting 2, so we will only focus on how we access grid elements in this review.

This concludes the PLM 360 script 1 class. In this class we've examined:

- ✓ An overview of PLM 360's server-side scripting with JavaScript
- ✓ PLM 360 scripting types and events
- ✓ PLM 360 scripting item object and functions
- ✓ Condition, Validation, Action, and Library "Hello World" scripts
- ✓ Standard Script Review:
    - ✓ Advanced Sequence Generator (ASG)
    - ✓ InspectionPassedFailed Condition Script's access to the grid

In PLM 360 Script 2 class we'll expand on the topics covered here as we explore in more detail:

- ✓ Spawning items from Scripts
- ✓ Workflow Approval Board
- ✓ Temporal and Value based Transitions
- ✓ Integration