Accela Civic Platform®

Scripting Guide

Accela

Accela Civic Platform Scripting Guide

Version 9.0.0
March 2017

Corporate Headquarters

2633 Camino Ramon
Suite 500
Bishop Ranch 3
San Ramon, CA 94583

Tel: (888) 722-2352
Fax: (925) 659-3201

www.accela.com

Contents

Introduction

Civic Platform scripting offers an extensive amount of customization opportunities across the Civic Platform user interface, workflows, and processes. The Civic Platform scripting architecture consists of the following components:

- **Custom scripts and custom functions** that interact with the master scripts and EMSE API to implement custom business logic

- **JavaScript scripting framework** which allows script developers to use standard JavaScript syntax, code repository, testing and deployment tools

- **Standard Choice script controls** which comprise a legacy mechanism for customizing the master scripts. Accela recommends the use of the JavaScript framework instead of the Standard choice script controls.

- **Master Scripts** that provide an environment in which the JavaScript or Standard Choice scripts execute. Civic Platform provides master scripts for most comment events.

- **Event Manager Scripting Engine (EMSE) and EMSE API** are at the heart of the Civic Platform scripting platform that provide the scripting interface with Civic Platform objects. Civic Platform stores the master script files, written in JavaScript, in the Civic Platform database. Civic Platform uses the Rhino open source JavaScript engine to convert scripts into Java classes that Civic Platform executes through the EMSE API.

- **Expression Builder** interface for scripting form-based interactions (auto-populating data fields based on user-selected values, for example) that occur before you trigger an event activity.
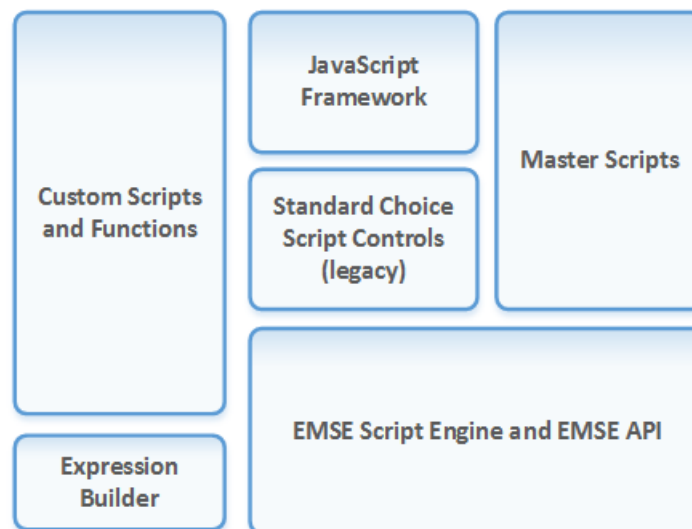
Figure 1: Civic Platform Scripting

Civic Platform launches scripts when the events that you associate with the script occur. You can use these event-triggered scripts to:

- Automate business processes

- Help save mouse-clicks

- Assess fees

- Update workflow

- Enforce business rules

- Custom data validation

- Confirm event pre-requisites

- Communicate

- Send event driven email

- Support Event / Batch driven data collection

- Communicate/access web services, email, and interact with the file system

Master Script Flow of Execution shows the flow when you trigger an event with an associated master script.
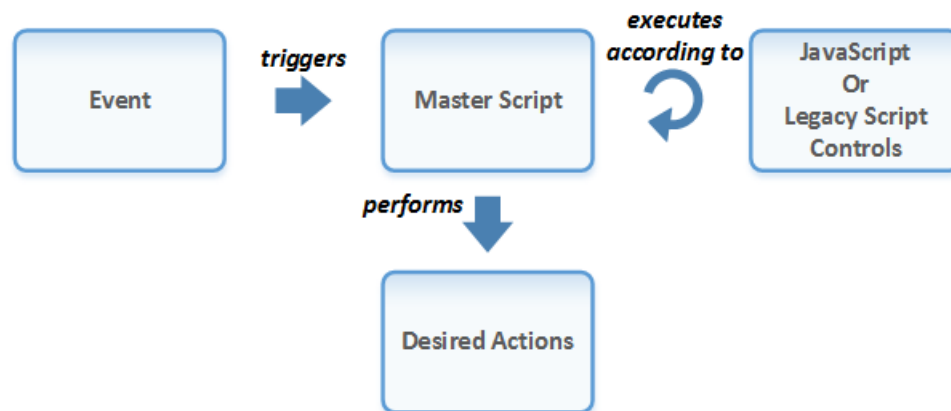
Figure 2: Master Script Flow of Execution

Related Information

Understanding Events

Understanding Master Scripts

Understanding Standard Choice Script Controls

Understanding Expression Builder Scripting

Understanding Events

An action that a user performs through the Civic Platform user interface, clicking the **Submit** button to create a new record for example, constitutes an event (Launching a Civic Platform Event). These events initiate some sort of reaction that may affect other parts of your system. For example, when you create a new record and save it, Civic Platform updates information on your system, as required.
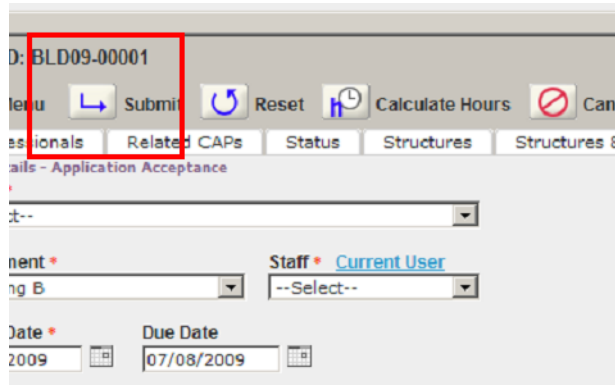
Figure 3: Launching a Civic Platform Event

Other possible events include finding a record, assessing a fee, scheduling an inspection, and so forth. Civic Platform provides more than 200 events with which you can associate scripts. You cannot create new events, but you can choose the events to set up for your agency and disable the events that you do not use.
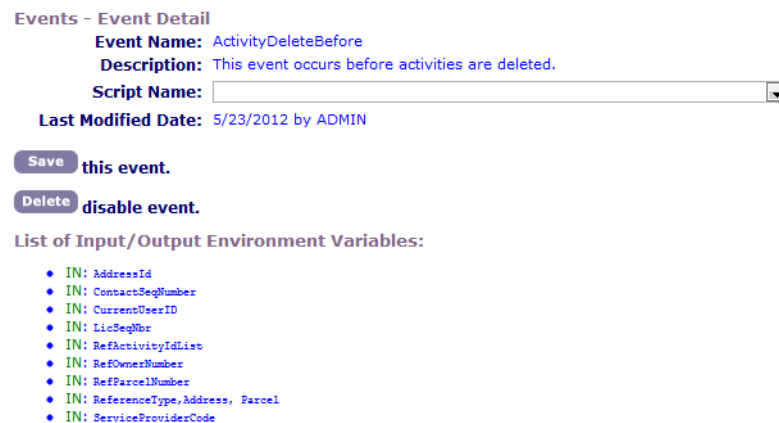
Figure 4: Scriptable Event and Event Variables

Each of the events includes a predefined set of variables that contain values about the event trigger. The associated master script can access these values.

You can trigger events from Civic Platform clients, such as Mobile Office, IVR, and Citizen Access, or from integrated third-party products.

Civic Platform provides before and after event types (Triggered event process flow).

Figure 5: Triggered event process flow

A before event occurs before you save any data to the database. Scripts associated with before event types typically validate data to ensure the process saves clean and accurate data to the database. Civic Platform provides the word "before" in the suffix of before event names.

An after event occurs directly after Civic Platform saves submitted data to the database. Scripts associated with an after event implement automation of an action for the user. Civic Platform provides the word "after" in the suffix of after event names.

Example Use Cases

The following provides example use cases for scripting.

Scheduling

The following provides some example use cases that relate to scheduling. You can:

- Automatically update the task status in an inspection workflow when you schedule an inspection.

- Schedule an investigation inspection for the next business day after filing of a complaint.

- Check to ensure that all required inspections have passed, before scheduling a final inspection.

Assessing Fees

The following provides some example use cases for assessing fees. You can:

- Assess and invoice standard fees or assess and invoice application dependent fees.

- Check to ensure that the balance due for a record (permit or license, for example) is less than or equal to zero before issuance

Processing Documents

The following provides some example use cases that relate to document processing. You can:

- Email a PDF copy of a license, to the license holder, upon issuance or renewal.

- Check to ensure submission of all required documents, before processing an application.

Initiating and Configuring Communications

The following provides some example use cases that relate to initiating communications and configuring communications. You can:

- Initiate any kind of communication with events and scripts.

- Configure the title/subject and content of communication, by the event or action that initiates the communication.

- Configure each communication with a different title/subject and with different content according to the event or action that initiate the communication.

- Configure communications to have a different title/subject and content according to the recipients.

- Configure communications to have a different title/subject and content according to the type of communication (i.e. email, text message, or Civic Platform/Citizen Access announcement).

- Configure communications to have a different title/subject and content according to a custom set of agency-defined criteria. Examples include:

  - Record type (4 level hierarchy or alias)

  - Standard fields

  - Template fields

  - ASI

  - Property information

Attaching Communications to Records

Civic Platform automatically attaches any outgoing emails to the license or case from which it originated. Civic Platform tracks the date and user that sent the correspondence along with the comments.

You can define whether the message subject, message body, bcc field, cc field, etc. dictates which emails Civic Platform retrieves, stores in the database, and attaches to corresponding records.

Configuring Communication Recipients

You can configure who receives communications, based on any of the following:

- Configuring communication recipients based on initiating event

  You can configure communications to have different recipients according to the event or action that initiates the communication.

- Configuring recipients based on type of communication

  You can configure communications to have different recipients according to the type of communication (i.e. email, text message, or Civic Platform/Citizen Access announcement).

- Configuring communication recipients based on Civic Platform user profiles

  You configure scripts to send communications to one or more recipients based on their user profile in Civic Platform, including:

  - Agency

  - Organization (agency > bureau > division > section > group > office (department alias)

  - User Group

  - Individual users

  - Users with inspector status enabled (vs. disabled)

- Configuring communication recipients based on APO owners

  You can configure scripts to send communications to one or more recipients based on being in the reference APO database as an owner.

- Configuring communication recipients based on APO owners on a record

  You can configure scripts to send communications to one or more recipients based on being a property owner on a record (including a Work Order).

- Configuring communication recipients based on reference contacts

  You can configure scripts to send communications to one or more recipients based on their contact type as a reference contact.

- Configuring communication recipients based on transaction contacts

  You can configure scripts to send communications to one or more recipients based on their contact type as a contact on a record (including a Work Order).

  Civic Platform prompts the user to email recipient(s) from a list of contacts associated with the license or case record.

- Configuring communication recipients based on reference licensed professionals

  You can configure scripts to send communications to one or more recipients based on their professional license:

  - All licensed professionals

  - Licensed professionals of one or more licensed professional types

- Configuring communication recipients based on licensed professionals associated with a record

  You can configure scripts to send communications to one or more recipients based on being a licensed professional on a record (including a Work Order).

- Configuring communication recipients based on Citizen Access public user permissions

  You can configure scripts to send communications to one or more recipients based on their Citizen Access public user permissions:

  - All public users

  - Anonymous public users

  - Registered public users

  - Record creator

  - Contact

  - Owner

  - Licensed Professional (any or specific)

- Configuring communication recipients based on their association with an inspection

  You can configure scripts to send communications to one or more recipients based on their association with an inspection:

  - Requestor

  - Contact

  - Inspector

- Configuring recipients based on their association with a workflow task

  You can configure scripts to send communications to one or more recipients based on their association to a workflow task:

- Action By Department

- Action By User

- Assigned to Department

- Assigned to User

- Configuring recipients based on their association with a condition assessment

  You can configure scripts to send communications to one or more recipients based on their association with a condition assessment:

  - Department

  - Inspector

- Configuring recipients based on their assignment to an activity

  You can configure communications to be send to one or more recipients assigned to an activity.

- Configuring recipients of the communication by agency-defined criteria (i.e. set)

  You can configure scripts to send communications to a set of recipients according to agency-defined criteria.

  Examples of criteria that you can use to create a set include:

  - All contacts on records of a designated record type (4 level hierarchy or alias)

  - All licensed professionals associated with records that contain designated values in standard fields, template fields or ASI fields

  - All owners of property according to some selection criteria such as range of addresses or proximity to a location

  - Any other set of recipients as defined by the agency

Preventing Duplicate Communications

When you properly configure the communication event script, for each type of communication with the same subject and same content, a single person can receive only one of each type of communication, even if they are members of more than one group of recipients.

For example: A person may be part of an agency organization (agency > bureau > division > section > group > office) and also part of an agency group (building clerk).

- If you configure an email to announce scheduled maintenance to members of this organization and also this group, the person only receives one email.

- If you configure an email and a text message for members of this organization and also this group, the person receives one email and one text message.

Configuring Email and Text Message Sent-from Values

You can configure emails and text messages to have different "from" values, according to the initiating event/action, type of communication, or other agency-defined criteria.

Understanding Master Scripts

Master scripts are out-of-the-box scripts that provide custom scripts the framework for extending standard event processing. For the most common events, Civic Platform provides a master script file unique to that event. For the other events, Civic Platform provides a universal master script that you can use as a template for development of a custom event script. Civic Platform also provides global master script files that contain the standard, global master script functions. To see a list of the master scripts, see Master Scripts.

Do not modify the master scripts. Customizations are implemented in custom scripts and custom functions.

Note:
Civic Platform does not support changes to or overrides of master script files, especially the functions that three global master scripts include.

You can trigger a script from:

- Events - see Event and Script Setup

- Citizen Access page flow - see Citizen Access Page Flow Scripts

- Batch jobs - see Batch Job Scripting

- Script tests - see Script Testing

- Set scripts - see *Civic Platform Administration Guide* > Extending Set Functionality

Understanding JavaScript Scripting Framework

Starting with Civic Platform 8.0.1, master scripts (version 3.0) allow customizations using a JavaScript-based framework instead of the legacy Standard Choice Script Controls. With the JavaScript framework, script developers can leverage the following features:

- Use of standard JavaScript syntax

  - Faster development times

  - Decreased learning curve for new Accela developers

  - Cleaner, more readable code

  - Support for IDE tools and syntax highlighting

- Improved error-handling and debugging

  - Errors reference appropriate lines in code

  - Improved script tester code

  - Use of try/catch for error handling

  - Debugging is 10x faster

- EMSE Tool integration

  - EMSE Tool integrates with version control systems (SVN and Git)

  - Enables standards based configuration management of script code

  - Custom functions and event scripts can be contained in individual files for better organization

  - Easy migration of scripts between environmentsBasic set of scripting variables, standard functions, flow of execution

The following diagram illustrates a typical script development process using the JavaScript framework with the master scripts:

Test within
Civic Platform

Modify
script in text
editor

Leverage
Script
Tester to
unit test
change

Commit
change to
SVN or Git
with comments

Push the
"build" into
Civic Platform

Understanding Standard Choice Script Controls

> **Note:** The use of Standard Choice Script Controls is a legacy scripting mechanism for customizing and interacting with the master scripts. Instead of using Standard Choice Script Controls, Accela recommends the use of the JavaScript Scripting Framework. It simplifies script development by using the standard JavaScript syntax without the restrictions of Standard Choice script controls.

You connect an event to a script through a comparably named Standard Choice script control. The script control calls functions from the global master script files, included in each script file, and passes parameters to these functions to control how the script interacts with the event (Standard Choice Script Control).

Standard Choices Item - Edit

Use this form to set up a Standard Choices Item.

Standard Choices Item Name: ApplicationSubmitAfter

Description:
(250 char max)

Status: ⦿ Enable ◯ Disable
Type: ◯ System Switch ◯ Shared drop-down ⦿ EMSE ◯ Business Configuration

Standard Choices Value(Default)	Standard Choices Value	Value Desc	Active	
01	01	true ^ showDebug = false ; showMessage = false	☑	Delete
02	02	appMatch("Licenses/Environmental Health/Food Safety/Restaurant") && {Type of Construction} ==	☑	Delete
03	03	appMatch("Licenses/Environmental Health/Food Safety/Restaurant") && {Type of Construction} != "	☑	Delete
04	04	appMatch("Building/Water Heater/NA/NA") ^ addFee("WH10","WHATERVHEATE","FINAL",1,"Y")	☑	Delete
05	05	appMatch("Enforcement/Complaint/Code Violation/NA") && {Abandoned Vehicle} == "Yes" ^ sched	☑	Delete
06	06	appMatch("Enforcement/Complaint/Code Violation/NA") && {Weeds & Debris} == "Yes" ^ schedule	☑	Delete

Update   Add   Cancel

Figure 6: Standard Choice Script Control

Understanding Expression Builder Scripting

> **Note:**
> Civic Platform uses the Event Manager and Scripting Engine (EMSE) to handle default form and portlet data fields.

Expression Builder provides an interface to script client side interactions (or expressions) before triggering an event type. For example, you can use Expression Builder to define expressions that trigger when a form loads, or when a user selects or enters a value in an individual form field. You can use expressions to perform calculations, provide drop-down lists, or auto-populate data fields based on user-selected values.

For information about how to use Expression Builder, see *Civic Platform Administration Guide* > Configuring Agency Business Objects > Using Expression Builder.

Example Use Case

A user selects a value from a drop-down list in ASI. You create a script for an expression that makes the selected value affect other fields in the form to:

- Mark fields as required.

- Mark fields as read-only or hidden as they are no longer required.

- Pre-populate them based on a calculation or lookup table.

- Trigger an alert pop-up window or alert message next to other fields.

Example Use Case

A user enters a permit number or license number in an ASI text field. Civic Platform provides a message about the validity of the permit or license number before the user submits the form.

Expressions implement business rules that require users to receive immediate feedback in the user interface before they submit a form. You can use "before" events in the master script framework to perform a similar type of form validation. However, with the master script framework, the user must complete the entire form and submit it before receiving feedback. With expressions, the user receives feedback immediately upon completing an individual field on the form.

Expression Builder provides a wizard to create expressions (see *Civic Platform Administrator's Guide*) and Civic Platform generates scripts to implement the expressions that you create through the wizard. You can view and edit the generated expression scripts in an Expression Builder window when you toggle Expression Builder from wizard mode to script mode (Expression Builder Portlet). You can select whether to execute the expressions for Civic Platform only, Citizen Access only, or both.

Figure 7: Expression Builder Portlet

You can use a combination of field level and form level data validation, depending on your business needs. You can trigger an master script from an expression. See the discussion thread on Accela Community for more information about Expression Builder.

You can configure an expression script to populate form data from an external data source, through an external web services. When connected to an external web service, administrators can generate expressions that use data elements, from an external web service, as variables or data items.

Example Use Case

An agency administrator uses Expression Builder to build and execute an expression for the License Professional portlet. The script interacts with an external web service, such as the State Licensing Board, to check for the current status of a license and whether the Licensed Professional selected in a new application is valid.

Event and Script Setup

Related Information

Listing of Events and Master Scripts

Working with Events

Working with Scripts

Associating Events with Scripts

Listing of Events and Master Scripts

The Events Manager on the Civic Platform Administration's **Events** portal shows the Civic Platform events and their associated scripts:

Events - Event List

Edit	Event	Associated Script
•	AAAddressUpdateAfter	
•	ActivityDeleteAfter	
•	ActivityDeleteBefore	
•	ActivityInsertAfter	
•	ActivityInsertBefore	
•	ActivityUpdateAfter	
•	ActivityUpdateBefore	
•	AddressLookUpAfter	AddressLookUpAfter
•	AddressLookUpBefore	amendmentcopyscript
•	ApplicationConditionAddAfter	AppConditionAddAfter
•	ApplicationConditionAddBefore	AppConditionAddAfter
•	ApplicationConditionDeleteAfter	ApplicationConditionDeleteAfter
•	ApplicationConditionDeleteBefore	ApplicationConditionDeleteAfter
•	ApplicationConditionUpdateAfter	ApplicationConditionUpdateAfter
•	ApplicationConditionUpdateBefore	ApplicationConditionUpdateAfter

Figure 8: Events and Associated Scripts

The following table lists the scriptable Civic Platform events and their associated master scripts:

Table 1: Event and Master Script List

Event Name	Description	Master Script
AAAddressUpdateAfter	The after event for when a user updates a daily address.	
AAAddressUpdateBefore	The before event for when a user updates a daily address.	
AddContractLicenseAfter	The after event for when agency administrators associate a licensed professional with the public user account for an external inspector.	
AddContractLicenseBefore	The before event for when agency administrators associate a licensed professional with the public user account for an external inspector.	
AAOwnerUpdateAfter	The after event for when a user updates a daily owner.	
AAOwnerUpdateBefore	The before event for when a user updates a daily owner.	
ActivityDeleteAfter	The after event for when a user deletes an activity.	
ActivityDeleteBefore	The before event for when a user deletes an activity.	
ActivityInsertAfter	The after event for when a user inserts an activity.	
ActivityInsertBefore	The before event for when a user inserts an activity.	

Event Name	Description	Master Script
ActivityNewBefore	The before event for when a user creates an activity.	
ActivityNewAfter	The after event for when a user creates an activity.	
ActivityUpdateAfter	The after event for when a user updates an activity.	
ActivityUpdateBefore	The before event for when a user updates an activity.	
AddAgendaAfter	The after event for when a user adds an agenda.	
AddAgendaBefore	The before event for when a user adds an agenda.	
AdditionalInfoUpdateAfter	The after event for when a user updates additional information.	AdditionalInfoUpdateAfter
AdditionalInfoUpdateBefore	The before event for when a user updates additional information.	AdditionalInfoUpdateBefore
AddLicenseToPublicUserAfter4ACA	Citizen Access - The after event for when a user adds a license to a public user.	
AddLicenseValidation4ACA	Citizen Access - The after event for when a user adds a license to a user account.	
AddressAddAfter	The after event for when a user creates an address.	
AddressAddBefore	The before event for when a user creates an address.	
AddressConditionAddAfter	The after event for when a user adds a condition to an address.	
AddressLookUpAfter	The after event for when a user creates a reference address after looking up an address from reference.	
AddressLookUpBefore	The before event for when a user creates a reference address after looking up an address from reference.	
AddressRemoveAfter	The after event for when a user removes an address from the daily side.	
AddressRemoveBefore	The before event for when a user removes an address from the daily side.	
AddressSelectOnSpearFormAfter	The after event for when a user attaches selected addresses on the ref addresses look up result list portlet.	
AddressSelectOnSpearFormBefore	The before event for when a user attaches selected addresses on the ref addresses look up result list portlet.	
AddressSetDetailUserExecuteAfter	The after event for when a user executes an address set script.	
AddressUpdateAfter	The after event for when a user updates an address.	
AddressUpdateBefore	The before event for when a user updates an address.	
AppHierarchyAddAfter		
AppHierarchyAddBefore		

Event Name	Description	Master Script
AppHierarchyDeleteAfter		
AppHierarchyDeleteBefore		
ApplicationConditionAddAfter	The after event for when a user adds an application condition task	ApplicationConditionAddAfter
ApplicationConditionAddBefore	The before event for when a user adds an application condition task.	
ApplicationConditionBatchUpdateAfter	The after event for when a user updates conditions of approvals.	
ApplicationConditionDeleteAfter	The after event for when a user deletes an application condition task.	
ApplicationConditionDeleteBefore	The before event for when a user deletes an application condition task.	ApplicationConditionDeleteBefore
ApplicationConditionUpdateAfter	The after event for when a user updates an application condition task.	ApplicationConditionUpdateAfter
ApplicationConditionUpdateBefore	The before event for when a user updates an application condition task.	ApplicationConditionUpdateBefore
ApplicationDeleteAfter	The after event for when a user deletes a record.	
ApplicationDeleteBefore	The before event for when a user deletes a record.	
ApplicationDetailNewAfter	The after event for when a user creates an application detail.	
ApplicationDetailNewBefore	The before event for when a user creates an application detail.	
ApplicationDetailUpdateAfter	The after event for when a user updates an application detail.	
ApplicationDetailUpdateBefore	The before event for when a user updates an application detail.	
ApplicationGISGovXMLSubmitAfter	The after event for when a user creates an application through GIS GovXML.	
ApplicationSelectAfter	The after event for when a user selects an application.	
ApplicationSelectBefore	The before event for when a user selects an application.	
ApplicationSpecificInfoUpdateAfter	The after event for when a user updates application specific information.	ApplicationSpecificInfoUpdateAfter
ApplicationSpecificInfoUpdateBefore	The before event for when a user updates application specific information.	ApplicationSpecificInfoUpdateBefore
ApplicationStatusUpdateAfter	The after event, that adds a history record, when a user updates application status.	ApplicationStatusUpdateAfter
ApplicationStatusUpdateBefore	The before event for when a user updates application status.	ApplicationStatusUpdateBefore
ApplicationSubmitAfter	The after event for when a user creates a record according to the following scenarios:The createCap web service operationThe initiateCAP GovXML operationThe user interface of Civic Platform, Citizen Access, Mobile Office, Accela Wireless, and IVR	ApplicationSubmitAfter

Event Name	Description	Master Script
ApplicationSubmitBefore	The before event for when a user creates a record in the following scenarios: The createCap web service operationThe initiateCAP GovXML operationThe user interface of Civic Platform, Citizen Access, Mobile Office, Accela Wireless, and IVR	ApplicationSubmitBefore
AssetSubmitAfter	The after event for when a user creates an asset.	
AssetSubmitBefore	The before event for when a user updates an asset.	
AssetUpdateAfter	The after event for when a user updates an asset.	
AssetUpdateBefore	The before event for when a user updates an asset.	
AssociateAssetToWorkOrderAfter	The after event for when a user associates an asset to a work order.	
AssociateAssetToWorkOrderBefore	The before event for when a user associates an asset to a work order.	
AuditSetDetailUserExecuteAfter	The after event for when a user executes a script on a random audit set.	
AutoApproveExams AutoApproveCEHs AutoApproveEducations AutoApproveRequiredCEHs AutoApproveRequiredExam	Enables auto-approval of exams, continuing education entries, and education entries in a given record. Also, enables auto-association of existing education or exam data with a record or a contact during license renewal.	
AutoPayAfter	The after event for when a user submits an auto payment.	
AutoPayBefore	The before event for when a user submits an auto payment.	
BatchResultInspectionByCSVAfter	The after event for when external inspectors upload CSV files that contain inspection results or when agency users update a batch of inspection results according to the inspection result CSV file that a contract inspector or a self-certified inspector submits.This event triggers in both Civic Platform and Citizen Access.	
BatchResultInspectionByCSVBefore	The before event for when external inspectors upload CSV files that contain inspection results or when agency users update a batch of inspection results according to the inspection result CSV file that a contract inspector or a self-certified inspector submits.This event triggers in both Civic Platform and Citizen Access.	
CAEConditionAddAfter	The after event for when a user adds a condition to a CAE.	
CapSetDetailUserExecuteAfter	Occurs after the record set script executes.	
CapSetProcessing	For processing a set of records.	CapSetProcessing
CommunicationReceivingEmailBefore	The before event for when Civic Platform receives an email from the email server.	

Event Name	Description	Master Script
CommunicationReceivingEmailAfter	The after event for when Civic Platform receives an email from the email server.	
CommunicationSendingEmailBefore	The before event for when Civic Platform sends an email.	
CommunicationSendingEmailAfter	The after event for when Civic Platform sends an email.	
ConditionAssessmentSubmitAfter	The after event for when a user creates a condition assessment.	
ConditionAssessmentSubmitBefore	The before event for when a user creates a condition assessment.	
ConditionAssessmentUpdateAfter	The after event for when a user updates a condition assessment.	
ConditionAssessmentUpdateBefore	The before event for when a user updates a condition assessment.	
ContactAddAfter	The after event for when a user adds a contact.	ContactAddAfter
ContactAddBefore	The before event for when a user adds a contact.	ContactAddBefore
ContactAddressDeactivateAfter	The after event for when a user deactivates a contact address.	
ContactAddressDeactivateBefore	The before event for when a user deactivates a contact address.	
ContactAddressEditAfter	The after event for when a user edits a contact address.	
ContactAddressEditBefore	The before event for when a user edits a contact address.	
ContactAddressLookUpAfter	The after event for when a user looks up a contact address.	
ContactAddressLookUpBefore	The before event for when a user looks up a contact address.	
ContactAddressNewAfter	The after event for when a user adds a contact address.	
ContactAddressNewBefore	The before event for when a user adds a contact address.	
ContactEditAfter	The after event for when a user edits a contact.	ContactEditAfter
ContactEditBefore	The before event for when a user edits a contact.	ContactEditBefore
ContactLookUpAfter	The after event for when a user adds a reference contact to a record.	
ContactLookUpBefore	The before event for when a user adds a reference contact to a record.	
ContactRelatedByPublicAfter	The after event for when a public user submits a request to associate a contact to their account.	
ContactRemoveAfter	The after event for when a user removes a contact.	ContactRemoveAfter
ContactRemoveBefore	The before event for when a user removes a contact.	ContactRemoveBefore

Event Name	Description	Master Script
ContactUpdateAfter	The before event for when a user updates a contact.	
ContactUpdateBefore	The before event for when a user updates a contact.	
ContinuingEducationUpdateAfter	The after event for when a user commits continuing education.	
ConvertToRealCAPAfter	Citizen Access - The after event for converting a partial record ID to a real record ID.	ConvertToRealCapAfter
ConvertToRealCAPBefore	Citizen Access - The before event for converting a partial record ID to a real record ID.	
DeleteContractLicenseAfter	The after event for when agency administrators disassociate a licensed professional with the public user account for an external inspector.	
DeleteContractLicenseBefore	The before event for when agency administrators disassociate a licensed professional with the public user account for an external inspector.	
DocumentDeleteAfter	The after event for when a user deletes one or more documents.	
DocumentDeleteBefore	The before event for when a user deletes one or more documents.	
DocumentReviewAddAfter	The after event for when a user assigns one or more document reviewers.	
DocumentReviewAddBefore	The before event for when a user assigns one or more document reviewers.	
DocumentReviewDeleteAfter	The after event for when a user deletes one or more document reviewers.	
DocumentReviewDeleteBefore	The before event for when a user deletes one or more document reviewers.	
DocumentReviewUpdateAfter	The after event for when a user updates a document review. An agency can use the DocumentReviewUpdateAfter event to automatically update the document status to 'Approved' or 'Rejected' accordingly when all reviewers have finished their document review.	
DocumentReviewUpdateBefore	The before event for when a user updates a document review.	
DocumentUpdateAfter	The after event for when a user updates document information.	DocumentUploadAfter
DocumentUpdateBefore	The before event for when a user updates document information.	DocumentUploadBefore
DocumentUploadAfter	Citizen Access - The after event for when a user uploads a document or when an external inspector uploads a CSV file containing inspection results.	
DocumentUploadBefore	Citizen Access - The before event for when a user uploads a document.	

Event Name	Description	Master Script
EstablishmentAddAfter	The after event for when a user adds an establishment.	
EstablishmentAddBefore	The before event for when a user adds an establishment.	
EstablishmentUpdateAfter	The after event for when a user updates an establishment.	
EstablishmentUpdateBefore	The after event for when a user updates an establishment.	
ExaminationBatchUpdateByCSVAfter	Citizen Access - The after event for when a user uploads a CSV file to batch update examination.	
ExaminationRosterUpdateAfter	The after event for when a user updates the examination roster (register roster, reschedule roster, delete roster, update score).	
ExaminationSiteUpdateAfter	The after event for when a user updates an examination site.	
ExaminationUpdateAfter	The after event for when a user updates an examination.	
ExaminationUpdateBefore	The before event for when a user updates an examination.	
FeeAssessAfter	The after event for when a user assesses an application fee.	FeeAssessAfter
ExternalDocReviewCompleted	The after event when a third-party document review application calls the Check-In Document Review Construct API to check-in a reviewed record document. For an example of how to use this event, see Example External Document Review Script.	
FeeAssessBefore	The before event for when a user assesses an application fee.	FeeAssessBefore
FeeEstimate4PlanReviewBefore	The before event for when a user creates a fee estimate for plan review.	
FeeEstimateAfter	The after event for when a user creates a fee estimate in an application intake form.	
FeeEstimateAfter4ACA	Citizen Access - The after event for when a user creates a fee estimate in the fee item list page.	FeeEstimateAfter4ACA
FundTransferAfter	The after event for when a user transfers a fund.	
FundTransferBefore	The before event for when a user transfers a fund.	
GuidesheetUpdateAfter	The after event for when a user updates a guidesheet.	
GuidesheetUpdateBefore	The before event for when a user updates a guidesheet.	
InspectionAssignAfter	The after event for when a user assigns an inspection.	

Event Name	Description	Master Script
InspectionAssignBefore	The before event for when a user assigns an inspection.	
InspectionCancelAfter	The after event for when a user cancels one or more inspections.	
InspectionCancelBefore	The before event for when a user cancels one or more inspections.	
InspectionMultipleScheduleAfter	The after event for when a user schedules one or more inspections for manage inspection.	InspectionMultipleScheduleAfter
InspectionMultipleScheduleBefore	The before event for when a user schedules one or more inspections for manage inspection.	InspectionMultipleScheduleBefore
InspectionResultModifyAfter	The after event for when a user modifies an inspection result.	InspectionResultModifyAfter
InspectionResultModifyBefore	The after event for when a user modifies an inspection result.	InspectionResultModifyBefore
InspectionResultSubmitAfter	The after event for when a user submits an inspection result.	InspectionResultSubmitAfter
InspectionResultSubmitBefore	The before event for when a user submits an inspection result.	InspectionResultSubmitBefore
InspectionScheduleAfter	Citizen Access - The after event for when a user schedules one or more inspections.	InspectionScheduleAfter
InspectionScheduleBefore	Citizen Access - The before event for when a user schedules one or more inspections.	InspectionScheduleBefore
InvoiceFeeAfter	The after event for when a user invoices a fee (manually or automatically)	InvoiceFeeAfter
LicenseProfessionalRemoveAfter	The after event for when a user removes a licensed professional.	
LicenseProfessionalRemoveBefore	The before event for when a user removes a licensed professional.	
LicProfAddAfter	The after event for when a user adds a licensed professional.	
LicProfAddBefore	The before event for when a user adds a licensed professional.	
LicProfLookUpSubmitAfter	The after event for when a user adds a reference license to a record.	LicProfLookupSubmitAfter
LicProfLookUpSubmitBefore	The before event for when a user adds a reference license to a record.	LicProfLookupSubmitBefore
LicProfUpdateAfter	The after event for when a user updates a licensed professional.	LicProfUpdateAfter
LicProfUpdateBefore	The before event for when a user updates a licensed professional.	LicProfUpdateBefore
MeetingAddAfter	The after event for when a user creates a meeting.Replaces EventAddAfter.	
MeetingAddBefore	The before event for when a user creates a meeting.Replaces EventAddBefore.	
MeetingCancelAfter	The after event for when a user cancels a meeting.	

Event Name	Description	Master Script
MeetingCancelBefore	The before event for when a user cancels a meeting.	
MeetingRemoveAfter	The after event for when a user removes a meeting.Replaces EventRemoveAfter.	
MeetingRemoveBefore	The before event for when a user removes a meeting.Replaces EventRemoveBefore.	
MeetingRescheduleAfter	The after event for when a user reschedules a meeting.Replaces EventRescheduleAfter.	
MeetingRescheduleBefore	The before event for when a user reschedules a meeting.Replaces EventRescheduleBefore.	
MeetingScheduleAfter	The after event for when a user schedules a meeting.	
MeetingScheduleBefore	The before event for when a user schedules a meeting.	
MeetingUpdateAfter	The after event for when a user updates calendar event information.	
MeetingUpdateBefore	The before event for when a user updates a meeting.Replaces EventUpdateBefore.	
MoveAgendaAfter	The after event for when a user moves an agenda to another meeting.	
MoveAgendaBefore	The before event for when a user moves an agenda to another meeting.	
OnlinePaymentPost	Citizen Access - Occurs after a user posts an online payment.	
OnlinePaymentRegister	Citizen Access - Occurs after a user submits a payment.	
OnLoginEventAfter4ACA	Citizen Access - Occurs after the login validation.	
OnLoginEventBefore4ACA	Citizen Access - Occurs before the login validation.	
OwnerLookUpAfter	The after event for when a user creates a reference owner after looking up the owner from reference.	
OwnerLookUpBefore	The before event for when a user creates a reference owner after looking up the owner from reference.	
OwnerRemoveAfter	The after event for when a daily user removes an owner.	
OwnerRemoveBefore	The before event for when a daily user removes an owner.	
OwnerSelectOnSpearFormAfter	The after event for when a user attaches selected addresses on the reference addresses look up result list portlet.	
OwnerSelectOnSpearFormBefore	The before event for when a user attaches selected addresses on the reference addresses look up result list portlet.	
ParcelAddAfter	The after event for when a user creates a parcel.	ParcelAddAfter

Event Name	Description	Master Script
ParcelAddBefore	The before event for when a user creates a parcel.	ParcelAddBefore
ParcelConditionAddAfter	The after event for when a user adds a condition to a parcel.	
ParcelLookUpBefore	The before event for when a user creates a reference parcel after looking up the parcel from reference.	
ParcelMergeAfter	The after event for when a user merges parcels.	
ParcelMergeBefore	The before event for when a user merges parcels.	
ParcelRemoveAfter	The after event for when a daily user removes a parcel.	
ParcelRemoveBefore	The before event for when a daily user removes a parcel.	
ParcelSelectOnSpearFormAfter	The after event for when a user attaches selected parcels on the reference parcels look up result list portlet.	
ParcelSelectOnSpearFormBefore	The before event for when a user attaches selected parcels on the reference parcels look up result list portlet.	
ParcelSetDetailUserExecuteAfter	The after event for the parcel set execute script.	
ParcelSplitAfter	The after event for when a user splits parcels.	
ParcelSplitBefore	The before event for when a user splits parcels.	
ParcelUpdateAfter	The after event for when a user updates a parcel.	ParcelUpdateAfter
ParcelUpdateBefore	The before event for when a user updates a parcel.	
PartTransactionSubmitAfter	The after event for when a user creates a part transaction.	
PartTransactionSubmitBefore	The before event for when a user creates a part transaction.	
PartTransactionUpdateAfter	The after event for when a user updates a part transaction.	
PartTransactionUpdateBefore	The before event for when a user updates a part transaction.	
PaymentProcessingAfter	The after event for when a user indicates "do pay" in the payment processing portlet.	PaymentProcessingAfter
PaymentProcessingBefore	The before event for when a user indicates "do pay" in the payment processing portlet.	PaymentProcessingBefore
PaymentReceiveAfter	Citizen Access - The after event for when Citizen Access records payment allocation.	PaymentReceiveAfter
PaymentReceiveBefore	Citizen Access - The before event for when Citizen Access records payment allocation.	PaymentReceiveBefore

Event Name	Description	Master Script
PaymentRefundAfter	The after event for a payment processing/ POS/record payment refund.	
PaymentRefundBefore	The before event for a payment processing/POS/record payment refund.	
PaymentRefundSubmitBefore	The before event for when a user submit the request to refund a payment.	
PermitIssueAfter	The after event for when a user creates a permit printout.	
PermitIssueBefore	The before event for when a user creates a permit printout.	
ProctorAssignedAfter	The after even when proctors receive an assignment to one or more examination sessions	
ProctorUnassignedAfter	The after event when a proctor is unassigned (deleted) from an examination session	
ProfessionalSetDetailUserExecuteAfter	The after event for the professional set execute script.	
ProximityAlertBefore	The before event for when a user issues a workflow proximity alert.	
PublishCommentsAfter	The after event for when a user publishes comments.	
PublishCommentsBefore	The before event for when a user publishes comments.	
RefContactEditAfter	The after event for when a user updates a contact in reference side.	
RefContactEditBefore	The before event for when a user updates a contact in reference side..	
RefContactNewAfter	The after event for when a user creates a contact in reference side.	
RefContactNewBefore	The before event for when a user creates a contact in reference side.	
RefExamUpdateAfter	The after event for when a user updates a reference examination.	
RefExamUpdateBefore	The before event for when a user updates a reference examination.	
RefLicProfAddAfter	The after event for when a user adds a reference licensed professional.	
RefLicProfAddBefore	The before event for when a user adds a reference licensed professional.	
RefLicProfUpdateAfter	The after event for when a user updates a reference licensed professional.	
RefLicProfUpdateBefore	The before event for when a user updates a reference licensed professional.	
RegistrationSubmitAfter	Citizen Access - The after event for when a public user submits a registration or when a public user associates a licensed professional with his user account.	
RegistrationSubmitBefore	Citizen Access - The before event for when a user submits a registration.	

Event Name	Description	Master Script
RelatedCapUpdateAfter	The after event for when a user updates related records.	
RelatedCapUpdateBefore	The before event for when a user updates related records.	
RemoveAgendaAfter	The after event for when a user removes an agenda from a meeting.	
RemoveAgendaBefore	The before event for when a user removes an agenda from a meeting.	
RenewalInfoUpdateAfter	The after event for when a user creates a permit printout.	RenewalInfoUpdateAfter
ReportServiceRunAfter	The after event for when a user runs a report service.	
ReportServiceRunBefore	The before event for when a user runs a report service.	
SaveAndResumeAfter4ACA	Citizen Access - The after event for when a user saves and resume .	
SearchMultiSeriveAfter	The after event for when a user searches a service.	
SelectLicenseValidation4ACA	Citizen Access - Occurs when the user selects a license by the license drop-down list.	
StrucEstLookUpAfter	The after event for when a user creates a reference structure or establishment, after looking up the owner from reference.	
StrucEstLookUpBefore	The before event for when a user creates a reference structure or establishment after looking up the owner from reference.	
StrucEstRemoveAfter	The after event for when a daily user removes a structure or establishment.	
StrucEstRemoveBefore	The before event for when a daily user removes a structure or establishment.	
StructureAddAfter	The after event for when a user adds a structure.	
StructureAddBefore	The before event for when a user adds a structure.	
StructureUpdateAfter	The after event for when a user updates a structure.	
StructureUpdateBefore	The before event for when a user updates a structure.	
taskEditActionFormBefore	The before event for when a user updates a workflow task.	
TimeAccountingAddAfter	Executes when time accounting entries are about to be added.	TimeAccountingAddAfter **Note:** The script can synchronize the cost items in time accountings into the costs list in the related entities, such as work orders.

Event Name	Description	Master Script
TimeAccountingAddBefore	Executes when time accounting entries are about to be added.	
TimeAccountingDeleteAfter	Executes when time accounting entries are about to be removed.	TimeAccountingDeleteAfter 📝 **Note:** The script can synchronize the cost items in time accountings into the costs list in the related entities, such as work orders.
TimeAccountingDeleteBefore	Executes when time accounting entries are about to be removed.	
TimeAccountingUpdateAfter	Executes when time accounting entries are about to be updated.	TimeAccountingUpdateAfter 📝 **Note:** The script can synchronize the cost items in time accountings into the costs list in the related entities, such as work orders.
TimeAccountingUpdateBefore	Executes when time accounting entries are about to be updated.	
V360InspectionResultSubmitAfter	The after event for when a user results an inspection.	V360InspectionResultSubmitAfter
V360InspectionResultSubmitBefore	The before event for when a user results an inspection	V360InspectionResultSubmitBefore
V360ParcelAddAfter		V360ParcelAddAfter
VoidFeeAfter	The after event for when a user voids a (manually or automatically)	
VoidFeeBefore	The before event for when a user voids a fee (manually or automatically)	
VoidPaymentAfter	The after event for when a user voids a payment.	VoidPaymentAfter
VoidPaymentBefore	The before event for when a user voids a payment.	VoidPaymentBefore
WorkflowAdhocTaskAddAfter	The after event for when a user adds a workflow task.	
WorkflowAdhocTaskAddBefore	The before event for when a user adds a workflow task.	
WorkflowAdhocTaskUpdateAfter	The after event for when a user updates an adhoc workflow task.	
WorkflowAdhocTaskUpdateBefore	The before event for when a user updates an adhoc workflow task.	
WorkflowTaskUpdateAfter	The after event for when a user updates a workflow task.	WorkflowTaskUpdateAfter
WorkflowTaskUpdateBefore	The before event for when a user updates a workflow task.	WorkflowTaskUpdateBefore
XRefContactAddressEditAfter	The after event for when a user edits the cross reference contact address.	

Event Name	Description	Master Script
XRefContactAddressEditBefore	The before event for when a user edits the cross reference contact address.	
XRefContactAddressLookUpAfter	The after event for when a user looks up a cross reference contact address.	
XRefContactAddressLookUpBefore	The before event for when a user looks up a cross reference contact address.	
XRefContactAddressNewAfter	The after event for when a user creates a cross reference contact address.	
XRefContactAddressNewBefore	The before event for when a user creates a cross reference contact address.	
XRefContactAddressRemoveAfter	The after event for when a user removes a cross reference contact address.	
XRefContactAddressRemoveBefore	The before event for when a user removes a cross reference contact address.	

Managing Events

This section provides basic topics for working with events.

Topics:

*   Searching for an Active Event

*   Viewing the Full List of Available Events

*   Enabling an Event

*   Disabling an Event
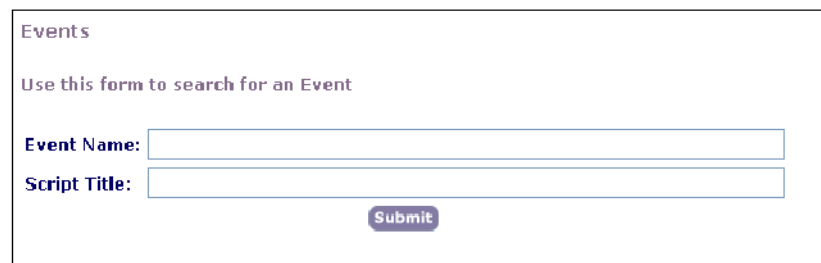
Searching for an Active Event

Before you can view or edit an active event, you must first locate it. You must also search for an event to associate a script with it. You can search for any enabled event.

To search for an event

1. Choose **AA Classic** > **Admin Tools** > **Events** > **Events**.

Civic Platform displays the Event Search window.

2. Complete these fields:

Event Name	Enter the name of the event that you want to find.
Script Title	Enter the name or title of the script associated with the event that you want to find.

3. Click **Submit**.

Civic Platform displays the Event List window.

4. Click the red dot that appears to the left of the event that you want.

Civic Platform displays the Event Detail window.

Viewing the Full List of Available Events

You can view the entire list of available events, including active events and disabled events.

To view the full list of Civic Platform events

1. Choose **AA Classic** > **Admin Tools** > **Events** > **Events**.

Civic Platform displays the Event Search window.

2. Leave the fields blank and click **Submit**.

Civic Platform displays the Event List window.

3. Click **Add**.

   Civic Platform displays the Add New Event window.

4. Click the drop-down menu to expand the list. This list contains all of available events.

Enabling an Event

Before you can use an event, you must enable it for your agency. Depending on the events that you choose to enable and the script that you associate with each event, you can customize Civic Platform to automatically perform various transactions.

To enable an event

1. Choose **AA Classic** > **Admin Tools** > **Events** > **Events**.

Civic Platform displays the Event Search window.

Events

Use this form to search for an Event

Event Name: [   ]

Script Title: [   ]

<center>Submit</center>

2. Click **Submit** to see a list of events enabled for your agency.

Civic Platform displays the Event List window.

Events - Event List

Edit	Event	Associated Script
●	AAAddressUpdateAfter	
●	ActivityDeleteAfter	
●	ActivityDeleteBefore	
●	ActivityInsertAfter	
●	ActivityInsertBefore	
●	ActivityUpdateAfter	
●	ActivityUpdateBefore	
●	AddressLookUpAfter	AddressLookUpAfter
●	AddressLookUpBefore	amendmentcopyscript
●	ApplicationConditionAddAfter	AppConditionAddAfter
●	ApplicationConditionAddBefore	AppConditionAddAfter
●	ApplicationConditionDeleteAfter	ApplicationConditionDeleteAfter
●	ApplicationConditionDeleteBefore	ApplicationConditionDeleteAfter
●	ApplicationConditionUpdateAfter	ApplicationConditionUpdateAfter
●	ApplicationConditionUpdateBefore	ApplicationConditionUpdateAfter

3. Click **Add**.

Civic Platform displays the Add New Event window.

4. Use the drop-down list to choose from the available events.

5. Click **Add**.

6. Associate a script with the event. For details, see Associating Events with Scripts.

Disabling an Event

You can disable any currently enabled event. When you disable an event, Event Manager no longer tracks the event or executes any script associated with it.

To disable an event for your agency

1. Choose **AA Classic** > **Admin Tools** > **Events** > **Events**.

   Civic Platform displays the Event Search window.

2. Search for the event that you want or click the **Submit** button to see a list of events enabled for your agency.

   Civic Platform displays the Event List.

3. Click the red dot that appears next to the event you want to disable.

   Civic Platform displays the Event Detail window.

Events - Event Detail

Event Name: ActivityDeleteBefore
Description: This event occurs before activities are deleted.
Script Name: [_____] ▼
Last Modified Date: 5/23/2012 by ADMIN

[Save] this event.

[Delete] disable event.

List of Input/Output Environment Variables:

- IN: AddressId
- IN: ContactSeqNumber
- IN: CurrentUserID
- IN: LicSeqNbr
- IN: RefActivityIdList
- IN: RefOwnerNumber
- IN: RefParcelNumber
- IN: ReferenceType, Address, Parcel
- IN: ServiceProviderCode

4. Click **Delete**.

5. Click **OK** to confirm your choice.
Civic Platform disables the event.

Triggering Events

This section provides details on before and after event triggers.

Topics

- Triggering Meeting Agenda Events

- Triggering Meeting Schedule Events

Triggering Meeting Agenda Events

Civic Platform provides six events related to meeting agendas (records).

- AddAgendaBefore

- AddAgendaAfter

- MoveAgendaBefore

- MoveAgendaAfter

- RemoveAgendaBefore

- RemoveAgendaAfter

The same user action triggers the before and after version of an event.

- Click **Select** to trigger the AddAgendaBefore and AddAgendaAfter events.

- Click **Submit** to trigger the MoveAgendaBefore and MoveAgendaAfter events.

- Click **Remove** to trigger the RemoveAgendaBefore and RemoveAgendaAfter events.

To trigger a before or after agenda-related event

1. Use one of the five Civic Platform portlets to access meeting details

    - Select a meeting calendar as an administrator (**Admin** > Setup > **Calendars** > **Calendar** > **Calendar by Type** > **Meeting** > select a meeting calendar > select a meeting).

    - Select a meeting calendar as a daily user (**Calendars** > **Calendar by Type** > **Meeting** > select a meeting calendar > select a meeting).

    - Select a meeting calendar from the MyTasks portlet (**My Tasks** > **Meetings** > select a meeting).

    - Select a meeting calendar from the Task Management portlet (**Task Management** > select a record of the meeting task type).

    - Select a meeting calendar from the Record portlet (**Record** > **Calendar** tab > select a meeting in calendar view).

2. Click the **Agenda & Vote** tab.

3. Trigger a before or after AddAgenda event.

    a. Click **Add**.

    b. Enter search criteria for the record(s) to add and click **Submit**.

    c. Select the record(s) you want to add and click **Select** (should be Submit according to IST) to trigger the event.

4. Trigger a before or after MoveAgenda event.

    a. Select one or more records to move.

    b. Click **Move**.

    c. Enter search criteria for the meeting to which you want to move the records and click **Submit**.

    d. Select the meeting to which you want to move the agenda and click **Submit** to trigger the event.

5. Trigger a before or after add remove event.

    a. Select one or more records to remove and click **Remove** to trigger the event.

Triggering Meeting Schedule Events

Civic Platform provides four events related to meeting schedules.

- MeetingScheduleBefore

- MeetingScheduleAfter

- MeetingCancelBefore

- MeetingCancelAfter

The same user action triggers the before and after version of an event.

- Click **Submit** to trigger the MeetingScheduleBefore and MeetingScheduleAfter events.

- Click **Cancel** to trigger the MeetingCancelBefore and MeetingCancelAfter events.

To trigger a before or after schedule-related event

1. Access the Records portlet.

2. Select a record for which you want to trigger the schedule-related event.

3. Click the **Meetings** tab.

4. Select the meeting you want to schedule or cancel.

- Trigger a before or after MeetingCancel event by clicking the **Manage Meeting** > **Cancel Meeting** submenu.

- Trigger a before or after MeetingSchedule event.

    1. Click the **Manage Meeting** > **Schedule Meeting** submenu.

    2. Enter search criteria for the meeting to which you want to schedule the record and click **Submit**.

    3. Select the meeting for which you want to schedule the record and click **Submit** to trigger the event.

Managing Scripts

This section provides general topics for working with custom scripts.

Topics:

- Adding a Script

- Searching for a Script

- Editing a Script

- Deleting a Script

Adding a Script

Scripts allow you to make specific changes to your database based on the event that occurs. For each pre-defined and enabled event, you can determine the script that you want to run for that event. In addition to associating standard scripts with standard events, you can write custom scripts that you want to assign to certain events.

To add a new script

1. Choose **AA Classic** > **Admin Tools** > **Events** > **Script**.

   Civic Platform displays the Scripts search window.

2. Click **Submit** to see a list of scripts enabled for your agency.

3. Click **Add**.

   Civic Platform displays the Add a new script page.

Scripts - Add a new script

Please enter name and content of the new script.

Script Code * :

Script Title * :

Script Initializer:

Script Content:

Add

4. Complete the necessary fields as described in Script Details.

5. Click **Add**.

Table 2: Script Details

Script Code	Enter the code or abbreviation that identifies the script.
Script Title	Enter the name or title of the script.
Script Initializer	If the script requires an initializer, enter it here. The initializer may be necessary to start certain scripts and contain certain input parameters.
Script Content/Text	Enter the script text here. You can also copy and paste the script into this text area.

Searching for a Script

You can search for a script to view or edit it.

To search for a script

1. Choose **AA Classic** > **Admin Tools** > **Events** > **Script**.

2. Complete the necessary fields as described in Script Details.

3. Click **Submit**.

4. Click the red dot that appears to the left of the script that you want.

Editing a Script

For each pre-defined and enabled event, you can determine the script that you want to run for that event. Accela provides several standard scripts. In addition to writing original scripts, you can modify standard scripts. You can make changes to any existing script that is currently on your system.

To edit a script

1. Choose **Administrator Tools** > **Events** > **Script**.

2. Search for the script that you want.

3. Complete the necessary fields as described in Script Details.

4. If you want to test the script, click the **Script Test** button.

5. Click **Save**.

Deleting a Script

You can delete any script.

To delete a script

1. Choose **Administrator Tools** > **Events** > **Script**.

2. Search for the script that you want.

3. Click **Delete**.

4. Click **OK** to confirm your choice.

Associating Events with Scripts

After you enable an event and add a script to your system, you can associate a script with an event. Associating a script with an event allows Civic Platform to execute or run the script when the event occurs.

To associate an event with a script, the script must already exist. For information on adding a script to your system, see Working with Scripts.

Example Use Case

Someone applies for a permit and you want Civic Platform to check the license expiration date to confirm that the license has not expired. You select an event such as ApplicationSubmitBefore and then associate a script that compares license expiration dates with the current date.

To associate an event with a script

1. Choose **Administrator Tools** > **Events** > **Script**.

2. Search for the event that you want. For details, see Searching for an Active Event.

3. Use the **Script Name** drop-down list to choose the script that you want to associate with this event.

4. Click **Save**.

Master Scripts

Civic Platform provides some Out-Of-The-Box master scripts. For most of these master scripts, Civic Platform defines a 1-1 relationship between the master script and the event which triggers master script execution. Civic Platform uses the same base name for the master script and the associated trigger event. The following lists these master scripts.

AdditionalInfoUpdateAfter	AdditionalInfoUpdateBefore
ApplicationConditionAddAfter	ApplicationConditionDeleteBefore
ApplicationConditionUpdateAfter	ApplicationConditionUpdateBefore
ApplicationSpecificInfoUpdateAfter	ApplicationSpecificInfoUpdateBefore
ApplicationStatusUpdateAfter	ApplicationStatusUpdateBefore
ApplicationSubmitAfter	ApplicationSubmitBefore
CapSetProcessing	ContactAddAfter
ContactAddBefore	ContactEditAfter
ContactEditBefore	ContactRemoveAfter
ContactRemoveBefore	ConvertToRealCapAfter
DocumentUploadAfter	DocumentUploadBefore
FeeAssessAfter	FeeAssessBefore
FeeEstimateAfter4ACA	InspectionMultipleScheduleAfter
InspectionMultipleScheduleBefore	InspectionResultModifyAfter
InspectionResultModifyBefore	InspectionResultSubmitAfter
InspectionResultSubmitBefore	InspectionScheduleAfter
InspectionScheduleBefore	InvoiceFeeAfter
LicProfLookupSubmitAfter	LicProfLookupSubmitBefore
LicProfUpdateAfter	LicProfUpdateBefore
ParcelAddAfter	ParcelAddBefore
ParcelUpdateAfter	PaymentProcessingAfter
PaymentProcessingBefore	PaymentReceiveAfter
PaymentReceiveBefore	RenewalInfoUpdateAfter
TimeAccountingAddAfter	TimeAccountingUpdateAfter
V360InspectionResultSubmitAfter	V360InspectionResultSubmitBefore
V360ParcelAddAfter	VoidPaymentAfter
VoidPaymentBefore	WorkflowTaskUpdateAfter
WorkflowTaskUpdateBefore	

In addition to event-specific master scripts, Civic Platform provides the following master script files:

UniversalMasterScript	Provides a template for creating additional event-specific master scripts.
ScriptTester	Enables you to test script controls without triggering an event from the user interface.
INCLUDES_ACCELA_FUNCTIONS	Included by each master script during runtime. Contains all global functions provided by Accela. Do not modify this file outside of official Accela master script releases.
INCLUDES_ACCELA_FUNCTIONS_ASB	Included by each master script during runtime. Similar to INCLUDES_ACCELA_FUNCTIONS but contains Accela provided functions specific to the ApplicationSubmitBefore event in which a current record is not available.

INCLUDES_ACCELA_ GLOBALS	Included by each master script during runtime. Contains global flags that are responsible for the setup the EMSE master script environment. Each master script file, from previous framework versions, set these flags individually in the master script file. Some examples of these global settings are enableVariableBranching, showDebug, showMessage, and useAppSpecificGroupName.
INCLUDES_CUSTOM	Contains customizations made to the master script framework. Every executed master script evaluates the script code in this file. Segregation of customizations in this file enables you to upgrade and maintain the EMSE master script framework without an impact to your customizations.
	Accela recommends the use of the custom function JavaScript files within the JavaScript framework to implement customizations to the master scripts. See JavaScript Scripting Framework.

Related Information

Understanding the EMSE Execution Path

Creating a New Script

Configuring the Universal Script

Configuring Global Variables

Adding Custom Functions

Viewing Master Scripts

Civic Platform provides the Master Scripts and Custom Script administration tools as part of the Event administration tools. These tools enable you to view available master scripts, and to view or edit the custom script.

The custom script, INCLUDES_CUSTOM, is a placeholder master script that contains custom functions. Instead of modifying standard functions provided by Accela, create custom functions and save them in INCLUDES_CUSTOM. At runtime, the master scripts can call the custom functions in INCLUDES_CUSTOM.

To view a master script

1. Choose **AA Classic** > **Admin Tools** > **Events** > **Master Scripts**.

   Civic Platform displays the Master Scripts search window.

2. In the **Master Script Version** drop-down list, select the Master Script Version you want to view.

   > 📝 **Note:**
   >
   > You can upgrade the master script version when you upgrade Civic Platform. Civic Platform makes all versions of the master scripts available at the same time. Administrators can set the Standard Choice MASTER_SCRIPT_DEFAULT_VERSION to continuously apply a specific version of master scripts, regardless of the master script upgrades.

3. Click the **Submit** button to see the list of master scripts provided in the version.

   Civic Platform displays the master script list.

   For the complete master script list, see Viewing Master Scripts.

4. Click the red dot that appears next to the master script you want to view.

   Civic Platform displays the master script detail.

To view and edit a custom script

1. Choose **AA Classic** > **Admin Tools** > **Events** > **Custom Script**.

   Civic Platform displays the custom script detail.

   The script name of the custom script is **INCLUDES_CUSTOM**.

2. Edit the script code of the custom script in the **Master Script Text** field.

   For more information on editing custom script, see Adding Custom Functions.

Understanding the EMSE Execution Path

EMSE Execution Path shows that the master script execution process leverages four script include files.

Event level globals that control pre-execute and event entry points

ApplicationSubmitBefore utilizes a special set of functions due to uninitialized data available from the database yet.

Master Script Functions are now loaded and available for use once the include is completed

Master Script globals are now loaded and logged via debugging. Used to setup defaults for debugging, version checks, variable branching, and other master script flags.

Custom include file is loaded and available for use in scripting

Event Specific Parameters are loaded and available for use

Use JavaScript or legacy Script Controls?

JavaScript or script controls are invoked

Figure 9: EMSE Execution Path

Creating a New Script

Civic Platform provides master scripts for most of the events. You use the UniversalMasterScript as a template to create scripts for the remaining events.

Civic Platform requires a separate script per event to:

*   Identify the entry point Standard Choice that contains the script controls for that event's desired actions (for scripts using the legacy Standard Choice script controls)

*   Identify the scripted business logic for a module, application type, and sub-type (for scripts using the JavaScript framework).

*   To create and populate event-specific variables needed for each specific event (eg. wfTask, inspType)

A traditional practice to create custom scripts is to use the legacy Standard Choice script controls, as described in Using the Legacy Standard Choice Script Controls. A better way to create custom scripts is by Using the JavaScript Framework.

Using the JavaScript Framework

Instead of using Standard Choices and script controls in your custom scripts, Accela recommends the use of the JavaScript Scripting Framework. It simplifies script development by using the standard JavaScript syntax without the restrictions of Standard Choice script controls. The following summarizes the process:

1. Setup a script repository, which includes a scripts folder organized by trigger mechanisms such as events, batch, expressions, etc.

2. Copy the contents of the UniversalMasterScript file and paste the contents into your script editor.

3. Modify the configurable parameters, as described in Configuring the Universal Script.

4. Implement your business logic using JavaScript syntax.

    > **Note:** A custom script using the JavaScript framework no longer requires the legacy Standard Choice script controls. See Comparing the JavaScript Framework with Legacy Script Controls for details.

5. Save the JavaScript file using the script file naming convention.

6. Deploy the custom script using the EMSE tool.

Using the Legacy Standard Choice Script Controls

To create a new script:

1. Copy the contents of the UniversalMasterScript file and paste the contents into your script development environment (text editor or IDE).

2. Save the new script file with the same base name as the event to which you plan to associate the new script.

3. Create a new standard choice with the same name as the event. This standard choice becomes the entry point standard choice for this event (Script Controls).

4. Modify the new script file as required (Configuring the Universal Script).

5. Install the script file (Event and Script Setup).

Configuring the Universal Script

When you create a new script file, you copy the contents of the UniversalMasterScript file into your new script file (Creating a New Script). You then need to modify this copied content to configure it for your particular application.

To configure the universal script

1. Locate the `START Configurable Parameters` section of the master script.

2. Set the value of the `controlString` variable to the name of the Standard Choice. The Standard Choice name must match the name of the event for which the Standard Choice contains the script controls.

   ```
   var controlString = "<Standard
   Choice>"
   ```

3. Set the value of the `preExecute` variable to indicate whether to trigger the script before or after the event.

   ```
   var preExecute = "<before
   or after>"
   ```

   where:

   <before or after> is `PreExecuteForBeforeEvents` for before events and `PreExecuteForAfterEvents`for after events.

4. Set the `documentOnly` variable to specify whether or not to display the hierarchy of standard choice steps.

   ```
   var documentOnly = false
   ```

5. Configure the remaining sections as required.

   - The following section of the master script configures the internal version of the master script file and the global master scripts to include during runtime.

     ```
     var SCRIPT_VERSION = 2.0
     eval(getScriptText("INCLUDES_ACCELA_FUNCTIONS"));
     eval(getScriptText("INCLUDES_ACCELA_GLOBALS"));
     eval(getScriptText("INCLUDES_CUSTOM"));
     ```

     📑 **Note:**

     The ApplicationSubmitBefore event includes the INCLUDES_ACCELA_FUNCTIONS_ASB master script instead of the INCLUDES_ACCELA_FUNCTIONS master script

     📑 **Note:**

     The master script files include the INCLUDES_CUSTOM master script as a placeholder to incorporate customizations to the master script. Civic Platform does not provide the INCLUDES_CUSTOM master script so as not to overwrite existing master script customizations during system upgrades.

   - This section includes the scripting to evaluate the value of the `documentOnly` variable configured in the previous section.

     ```
     if (documentOnly) {
         doStandardChoiceActions(controlString,false,0);
         aa.env.setValue("ScriptReturnCode", "0");
     ```

```
    aa.env.setValue("ScriptReturnMessage", "Documentation Successful. No
  actions executed.");
    aa.abortScript();
    }
```

- The `BEGIN Event Specific Variables` section loads the values for the variables of the associated event (AdditionalInfoUpdateAfter Variables).

Events - Event Detail

Event Name: AdditionalInfoUpdateAfter
Description: This event occurs after a Additional Info is updated.
Script Name:
Last Modified Date: 5/23/2012 by ADMIN

Save this event.

Delete disable event.

List of Input/Output Environment Variables:

- IN: AdditionalInfoBuildingCount
- IN: AdditionalInfoConstructionTypeCode
- IN: AdditionalInfoHouseCount
- IN: AdditionalInfoPublicOwnedFlag
- IN: AdditionalInfoValuation
- IN: CurrentUserID
- IN: PermitId1
- IN: PermitId2
- IN: PermitId3

Figure 10: AdditionalInfoUpdateAfter Variables

For example, Civic Platform uses the following variables for the AdditionalInfoUpdateAfter event.

```
var aiBuildingCount = aa.env.getValue("AdditionalInfoBuildingCount");
var aiConstructionTypeCode =
 aa.env.getValue("AdditionalInfoConstructionTypeCode");
var aiHouseCount = aa.env.getValue("AdditionalInfoHouseCount");
var aiPublicOwnedFlag = aa.env.getValue("AdditionalInfoPublicOwnedFlag");
var aiValuation = aa.env.getValue("AdditionalInfoValuation");
This variable list corresponds to the default set of variables defined for
 the event.
```

📝 **Note:**

The INCLUDES_ACCELA_FUNCTIONS master scripts resolves the CurrentUserID, PermitId1, PermitId2, and PermitId3 global variables.

- After logging event specific variable, the master script executes the `Main Loop` by performing the actions prescribed by the applicable Standard Choice script controls.

```
doStandardChoiceActions(controlString,true,0);
```

Configuring Global Variables

The following table provides parameters you can configure in the INCLUDES_ACCELA_GLOBALS file.

Table 3: Configurable Global Parameters

Parameter Name	Default Value	Description
showMessage	false	Controls whether or not to show the messages added by the comment() function.
showDebug	false	Controls whether to show the debug messages during script execution.
documentOnly	false	Controls whether to spool out standard choices to the debug window.
disableTokens	false	Enables or disabled the token substitution
useAppSpecificGroupName	false	Enables or disables use of group name when populating and referring to ASI. When enabled, the ASI subgroup name prepends to all ASI field names, which ensures the uniqueness of ASI field names required by script controls.
useTaskSpecificGroupName	false	Enables or disables use of group name when populating and referring to TSI
enableVariableBranching	true	Enables the use of variable branching in the branch function
maxEntries	99	Specifies the maximum number of script controls in a single standard choice branch

Adding Custom Functions

Civic Platform master scripts provide a placeholder to include the INCLUDES_CUSTOM master script file.

```
eval(getScriptText("INCLUDES_CUSTOM"));
```

Custom functions are stored in the INCLUDES_CUSTOM.js file. The practice used with the legacy Standard Choice script controls was to manually add the custom functions into the INCLUDES_CUSTOM script, as described in Manually Updating INCLUDES_CUSTOM. A better way to create custom functions is by Using the JavaScript Framework to Create Custom Functions.

Using the JavaScript Framework to Create Custom Functions

Instead of manually adding all custom functions into the INCLUDES_CUSTOM file, Accela recommends the use of separate custom function JavaScript files within the JavaScript Scripting Framework to implement customizations to the master scripts. The use of separate JavaScript files facilitates easier creating, storing, and managing of the custom functions. The following summarizes the process:

1. Setup a script repository, which includes a folder named INCLUDES_CUSTOM.

2. For each custom function, create a JavaScript file in the INCLUDES_CUSTOM folder. Use the name of the function as the JavaScript filename.

   > **Note:** A custom function using the JavaScript framework no longer requires the branch() function used with the legacy Standard Choice script controls. See Comparing the JavaScript Framework with Legacy Script Controls for details.

3. Build and deploy the INCLUDES_CUSTOM script using the EMSE tool.

Manually Updating INCLUDES_CUSTOM

This section describes the legacy process of adding new custom functions by manually updating the INCLUDES_CUSTOM file. To create new functions, save your customizations in a file named INCLUDES_CUSTOM.js.

> **Note:**
>
> To prevent the Civic Platform installer from overwriting existing customizations during a product upgrade, Civic Platform does not provide the INCLUDES_CUSTOM master script file as part of the Civic Platform installer.

If the INCLUDES_ACCELA_FUNCTIONS and INCLUDES_CUSTOM contain a function with the same name, the function in the INCLUDES_CUSTOM file overwrites the function in the INCLUDES_ACCELA_FUNCTIONS file.

Do not modify functions in the INCLUDES_ACCELA_FUNCTIONS file. If you want to modify a function from the INCLUDES_ACCELA_FUNCTIONS file, create a same named function with the different functionality in the INCLUDES_CUSTOM file.

As a best practice, use a commenting structure in your INCLUDES_CUSTOM file to keep it organized and easy to interpret. The following provides and example.

```
/**************** Modified Accela Standard Functions (Start)
 **************/
//All modified Accela standard functions will live here
/**************** Modified Accela Standard Functions (End)
 ****************/
/**************** Custom Building Functions (Start) **************/
//All custom building functions will live here
```

```
/**************** Custom Building Functions (End) ****************/
/**************** Custom Licensing Functions (Start) **************/
//All custom licensing functions will live here
/**************** Custom licensing Functions (End) ****************/
/**************** Custom Planning Functions (Start) **************/
//All custom planning functions will live here
/**************** Custom planning Functions (End) ****************/
/* Start a new section for each logical group */
```

JavaScript Scripting Framework

Starting with Civic Platform 8.0.1, master scripts (version 3.0) allow customizations using a JavaScript-based framework instead of the legacy Standard Choice Script Controls. With the JavaScript framework, script developers can leverage the following features:

- Use of standard JavaScript syntax

  - Faster development times

  - Decreased learning curve for new Accela developers

  - Cleaner, more readable code

  - Support for IDE tools and syntax highlighting

- Improved error-handling and debugging

  - Errors reference appropriate lines in code

  - Improved script tester code

  - Use of try/catch for error handling

  - Debugging is 10x faster

- EMSE Tool integration

  - EMSE Tool integrates with version control systems (SVN and Git)

  - Enables standards based configuration management of script code

  - Custom functions and event scripts can be contained in individual files for better organization

  - Easy migration of scripts between environmentsBasic set of scripting variables, standard functions, flow of execution

The following diagram illustrates a typical script development process using the JavaScript framework with the master scripts:

Comparing the JavaScript Framework with Legacy Script Controls

The JavaScript framework eliminates the legacy coding mechanism required by the legacy Standard Choice script controls:

No more Standard Choice script controls: Earlier Master Scripts relied on several shortcuts, primarily in an attempt to make the code easier to read in the compressed space in Standard Choices. The JavaScript scripting framework does away with these shortcuts and uses pure JavaScript instead.

No more Standard Choice line numbers: Scripts using the JavaScript framework are stored in the scripts area instead of Standard Choices. There is no longer a need to indicate the order of execution with a line number. Instead the JavaScript code is evaluated top-to-bottom.

No more "Carets" as if/then/else: Probably the most prominent difference is the elimination of the caret ^ to separate if/then/else clauses. JavaScript has a different purpose for the caret, so this notation is no longer allowed.

- Using legacy Script Controls:

```
wfStatus== "Issued" ^ editAppStatus("Issued",""); ^ editAppStatus("Not
 Issued","");
```

- Using JavaScript:

```
if (wfStatus== "Issued") {
  editAppStatus("Issued", "");
} else {
  editAppStatus("Not Issued", "");
}
```

No more branch() function: The legacy branch() function was used to shift code execution to a different Standard Choice, much like a subroutine in other programming languages. Instead of calling the branch function, call a custom JavaScript function.

- Using legacy Script Controls:

```
branch("EMSE:AddBuildingFees");
```

- Using JavaScript:

```
addBuildingFees();
```

No more curly braces around custom ("ASI") fields: Earlier master scripts used curly braces { } surrounding an ASI / TSI / Parcel Attribute label to substitute the value of that field. Instead, place the custom field name in a JavaScript string array called AInfo.

- Using legacy Script Controls:

```
{Square Footage} > 10
```

- Using JavaScript:

```
AInfo["Square Footage"] > 10
```

If you are new with JavaScript, see JavaScript Primer or any JavaScript tutorial and reference on the internet.

Configuration

Topics:

- Standard Choice Configuration

- Custom Global Variables

- Linking Events to Master Scripts 3.0

Standard Choice Configuration

To enable the JavaScript scripting framework with Master Scripts 3.0, configure the following standard choices:

- **MASTER_SCRIPT_DEFAULT_VERSION** - Defines the version of master scripts to apply in the scripts.

Standard Choice Value	Value Desc	Description
8.0.1.0.0	<leave blank>	Civic Platform 8.0.1.0.0 provides Master Scripts 3.0 which supports the JavaScript framework.

- **EMSE_EXECUTE_OPTIONS** - Specifies whether to invoke JavaScript or Script Control scripts.

Standard Choice Value	Value Desc	Description
SCRIPT	<leave blank>	Invokes scripts using JavaScript. Mark the **Active** checkbox.
STD_CHOICE	<leave blank>	Invokes scripts using Standard Choice script controls. Although Accela recommends the use of JavaScript, If you intend to maintain scripts using the Standard Choice script controls as well as JavaScript, you can add the STD_CHOICE value. >>QUESTION: What happens if both SCRIPT and STD_CHOICE are configured and activated?

- **EMSE_VARIABLE_BRANCH_PREFIX** - Specifies a list of events with their corresponding abbreviation prefixes. Use the specified abbreviation to prefix the event's script filename.

Standard Choice Value	Value Desc	Description
<eventName> For example, ApplicationConditionAddAfter	*<abbreviationPrefix>* For example, ACAA	The Value Desc specifies the prefix used by the JavaScript filename. For details about the file naming convention, see Event Script File Naming Convention.

- **MULTI_SERVICE_SETTINGS** - Applies to multi-agency administration.

Standard Choice Value	Value Desc	Description
SUPER_AGENCY_FOR_EMSE	*<superAgencyName>*	The name of the super-agency which is used by EMSE scripts.
SUPER_AGENCY_INCLUDE_SCRIPT	INCLUDES_CUSTOM_ENTERPRISE	Optional includes file for all agencies within a super-agency.

Custom Global Variables

To store custom global variable declarations, in **Civic Platform Admin Tool > Events > Scripts**, add the INCLUDES_CUSTOM_GLOBALS script. For example:

Linking Events to Master Scripts 3.0

Master Scripts 3.0 are included with Civic Platform release 8.0.1.0.0 and later. To link a Civic Platform event to one of the Master Scripts 3.0:

1. Go to Civic Platform Admin Tools > Events > Events.

2. Search for the event you want to configure.

3. Select the event to edit it.

4. On the Script Name dropdown list, scroll down to find its corresponding Master Script - 8.0.1.0.0.

   The following example shows the ContactAddBefore event linked to the ContactAddBefore Master Script 8.0.1.0.0:

Events – Event Detail

Event Name: ContactAddBefore
Description: ContactAddBefore
Script Name: ContactAddBefore (Master Script - 8.0.1.0.0) ▼
Last Modified Date: 12/9/2013 by ADMIN

Save this event.

Delete disable event.

List of Input/Output Environment Variables:

- IN: CapId
- IN: Contact
- IN: ContactAttribute
- IN: CurrentUserID

Script Repository Setup

Topics:

- Script Directory Structure

- Event Script File Naming Convention

- Custom Function Files in the INCLUDES_CUSTOM folder

- Managed Code Repository

Script Directory Structure

To help organize your JavaScript files in a source code repository, Accela recommends the following directory structure:

```
Sample Repository Structure
  Project 1              - For an environment with multiple projects
    Agency 1             - For a multi-agency environment
      Documentation      - For any related documentation
      INCLUDES_CUSTOM    - Stores custom function JavaScript files
      Scripts            - Stores JavaScript files, by trigger mechanism
        Batch
        Event
        Expression
        Interface
        Misc
        Pageflow
        Set
    Agency 2
  Project 2
```
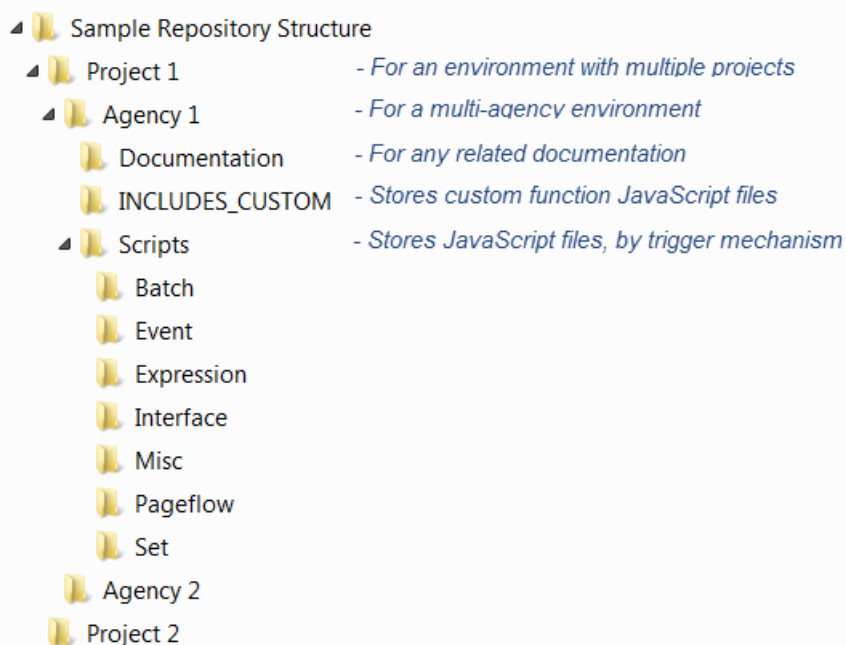
Event Script File Naming Convention

To easily manage and and allow EMSE to locate event scripts, use the following naming convention for the JavaScript filenames:

```
AAAA;B!C!D!E.js
```

Where (use capital letters):

- `AAAA` = Event Prefix (see EMSE_VARIABLE_BRANCH_PREFIX standard choice, examples are WTUA, CRCA, ASA)

- `B` = Module (e.g., LICENSES)

- `C` = Application Type, or tilde (~) for wildcard

- `D` = Sub-type, or tilde (~) for wildcard

For example:

```
ASA;CASEMANAGEMENT!CASE!~!~.js
ASA;LICENSES!AMENDMENT!AE BARB BUSINESS!CLOSE BUSINESS.js
ASIUA;CASEMANAGEMENT!CONSUMER COMPLAINT!~!~.js
RIUA;LICENSES!~!~!~.js
```

When the script files are deployed from your local repository into Civic Platform using the EMSE tool, the script filenames are converted into script names that show variable branching. For example:
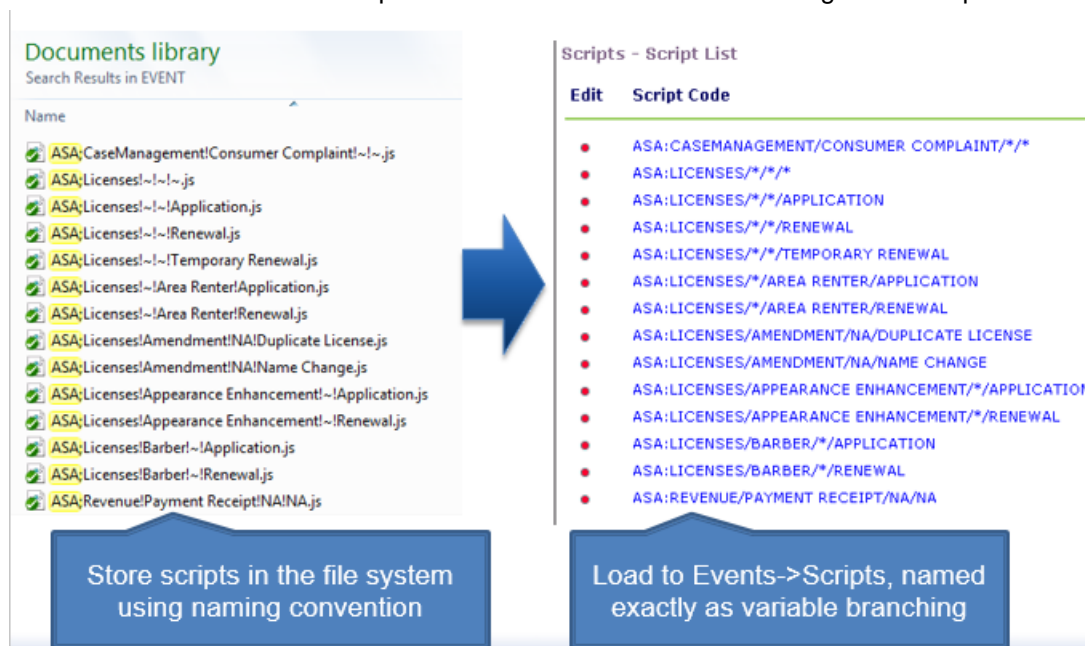
Custom Function Files in the INCLUDES_CUSTOM folder

Instead of storing multiple custom functions in one INCLUDES_CUSTOM file, Master Scripts 3.0 allows you to easily create and manage custom functions in individual JavaScript files. The only requirement is that you store the JavaScript files in a folder named **INCLUDES_CUSTOM**. Use the custom JavaScript function name as the filename. For example:
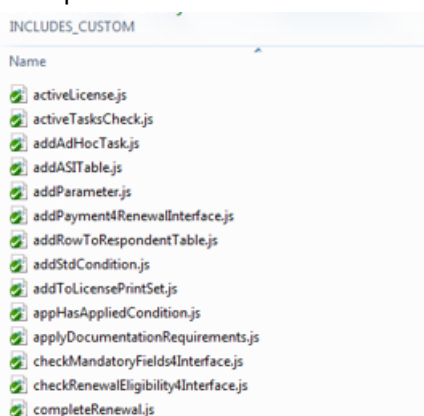
When the custom function files are ready for deployment into Civic Platform, use the EMSE Tool to build them into the INCLUDES_CUSTOM script and upload it into Civic Platform. See Deploying Custom Functions.

Managed Code Repository

After you have created JavaScript files in your local file structure, Accela recommends that you upload the file structure into a managed code repository to help you manage and share the scripts across your team or across environments (for example, Development to Production environments). Open-source cloud code repository systems are available such as Github or Assembla. You can also use traditional source code control systems such as Subversion SVN.

The following procedure summarizes how you would setup your scripts in a code repository:

- Create a repository in an online or enterprise code repository system such as Github or SVN.

- Synchronize the repository with your local folder.

- If not done yet, populate the folder with the script file structure, as described in Script Directory Structure.

- Commit or check-in the initial revision into the repository.

Script Development and Testing

Topics:

* Converting Legacy Scripts

* Testing Scripts

Converting Legacy Scripts

To leverage existing business logic in legacy scripts, you can use a conversion utility called **Bizdomain to Script Converter.js**, which helps you convert Standard Choice script controls to JavaScript.

The **Bizdomain to Script Converter.js** conversion utility is available on Accela Community. The **Bizdomain to Script Converter.js** file is located in the Misc folder within the **Master Script Enterprise Scripts release distribution.zip** download file.

The conversion process involves the following general steps:

* Run the contents of **Bizdomain to Script Converter.js** in **Civic Platform Admin Tool > Events > Script Test**

* Review each script output

* Convert existing business rules

* Convert branch() statements to functions

* Convert existing functions

The Upgrading to Master Scripts 3.0 and the EMSE Tool Webinar on Accela Community includes a walkthrough of converting a legacy script using the Script Converter.

Testing Scripts

To debug and test your scripts, you can use a script test utility called **Script Tester.js**, which allows you to call and test your scripts within the context of EMSE.

The **Script Tester.js** test utility is available on Accela Community. The **Script Tester.js** file is located in the Misc folder within the **Master Script Enterprise Scripts release distribution.zip** download file.

The following diagram illustrates how to use the Script Tester:

```
1  var myCapId = "1";
2  var myUserId = "ADMIN";
3
4  /* ASA  */  var eventName = "ApplicationSubmitAfter";
5  /* WTUA */  //var eventName = "WorkflowTaskUpdateAfter";  wfTask =
6  /* IRSA */  //var eventName = "InspectionResultSubmitAfter" ; insp
7  /* ISA  */  //var eventName = "InspectionScheduleAfter" ; inspType
8  /* PRA  */  //var eventName = "PaymentReceiveAfter";
9
10 var useProductScript = false;  // set to true to use the "producti
11 var runEvent = true; // set to true to simulate the event and run all std choices/scripts fc
12
13 /* master script code don't touch */ aa.env.setValue("EventName",e
14
15 //
16 // User code goes here
17 //
18
19 try {
20     showDebug = true;
21
22
23     }
24 catch (err) {
25     logDebug("A JavaScript Error occured: " + err.message);
26     }
27 // end user code
28 aa.env.setValue("ScriptReturnCode", "1");   aa.env.setValue("ScriptReturnMessage", debug)
29
30
```

Record ID (AltID) to use for testing

User ID to use when simulating event

Event to simulate. Uncomment or input manually.

Set to true to use the productized master scripts.

Set to true to simulate the event and run all std choices / scripts...

...or add your own code to test here

The Upgrading to Master Scripts 3.0 and the EMSE Tool Webinar on Accela Community includes a walkthrough of using the Script Tester.

Script Deployment Using the EMSE Tool

Topics:

- Overview

- Configuration

- Pulling Scripts from Civic Platform

- Deploying Scripts

- Deploying Custom Functions

- Comparing Scripts

Overview

The EMSE Tool leverages third-party Source Code Control Systems (SCCS) such as Subversion and Github to facilitate easier storage, management, and deployment of EMSE scripts. Script developers can use their choice of script editor and SCCS to edit and manage their scripts, and use the EMSE Tool to deploy them into Civic Platform. The EMSE Tool provides the following capabilities:

- Connects to the SCCS via the Accela Gateway

- Supports two popular SCCS: Github and Subversion (SVN)

- Compares scripts between the SCCS and Civic Platform

- Pushes scripts into Civic Platform from the SCCS, builds, and deploys them. Building a script consolidates multiple custom scripts within the INCLUDES_CUSTOM directory into a single script file in Civic Platform. Deploying a script replaces the current script entry in the Civic Platform database with the new script from the SCCS.

- Pulls scripts from Civic Platform into the SCCS. This capability can be used to begin a new repository or refresh scripts that may have been corrupted in the SCCS.

- Validates script syntax before building or deploying a script

- Integrates with EMSE scripts and Expression Builder scripts

- Applies Accela's standard naming and file structure conventions when deploying scripts to Civic Platform

Configuration

An administrator must configure the EMSE Tool to connect to the appropriate SCCS repository. Although the EMSE Tool supports both Github and TortoiseSVN SCCS repositories, it only supports one SCCS per configuration.

To configure the SCCS repository and connection settings, configure the following standard choice:

- **EMSEToolsConfig** - Specifies source code control settings.

Standard Choice Value	Value Desc	Description
`agency_repo_username`	`<aUserid>`	An agency user's username for the SCCS repository containing agency scripts. Civic Platform connects to the SCCS repository using this username. 📝 **Note:** The specified username must have permission to access the agency script SCCS repository.
`agency_repo_password`	`<aPassword>`	An agency user's password for the SCCS repository containing agency scripts. Civic Platform connects to the SCCS repository using this password.
`agency_url_svn`	`<https://svnUrl>`	**For SVN**: The SVN URL for the agency script repository. You can get the SVN URL from your repository's SVN folder properties. 📝 **Note:** If you specified agency_url_svn, do not specify agency_url_git.
`agency_url_git`	`<https://gitUrl>`	**For Github**: The Github URL for the agency script repository. You can get the Github URL from the HTTPS Clone URL on your agency script repository's Github page. 📝 **Note:** If you specified agency_url_git, do not specify agency_url_svn.
`master_repo_username`	`<aUserid>`	An agency user's username for the SCCS repository containing Civic Platform master scripts. Civic Platform connects to the SCCS repository using this username. 📝 **Note:** The specified username must have permission to access the master script SCCS repository.
`master_repo_password`	`<aPassword>`	An agency user's password for the SCCS repository containing Civic Platform master scripts. Civic Platform connects to the SCCS repository using this password.
`master_url_svn`	`<https://svnUrl>`	**For SVN**: The SVN URL for the master script repository. You can get the SVN URL from your repository's SVN folder properties. 📝 **Note:** If you specified master_url_svn, do not specify master_url_git.
`master_url_git`	`<https://gitUrl>`	**For Github**: The Github URL for the master script repository. You can get the Github URL from the HTTPS Clone URL on your master script repository's Github page. 📝 **Note:** If you specified master_url_git, do not specify master_url_svn.

Ensure that the **Active** checkbox is enabled for each of the Standard Choice values you entered.

The EMSE Tool Portlet

To access the EMSE Tool via Civic Platform Administration, click the **EMSE Tool** main link from the Civic Platform home page.

> 📝 **Note:** To provide agency users and EMSE script developers access to the EMSE Tool within Civic Platform Administration, an administrator must configure a main link that opens the EMSE Tool portlet. For information about how to add a main link in Civic Platform, see "Adding a Main Link" in *Civic Platform Administrator Guide*.

The EMSE Tool portlet shows a tree structure of the Master scripts and Agency scripts from the SCCS repository which have not been synchronized with the scripts in the Civic Platform database. The following example shows the Agency scripts in the SCCS repository that have not been deployed into or synchronized with Civic Platform:
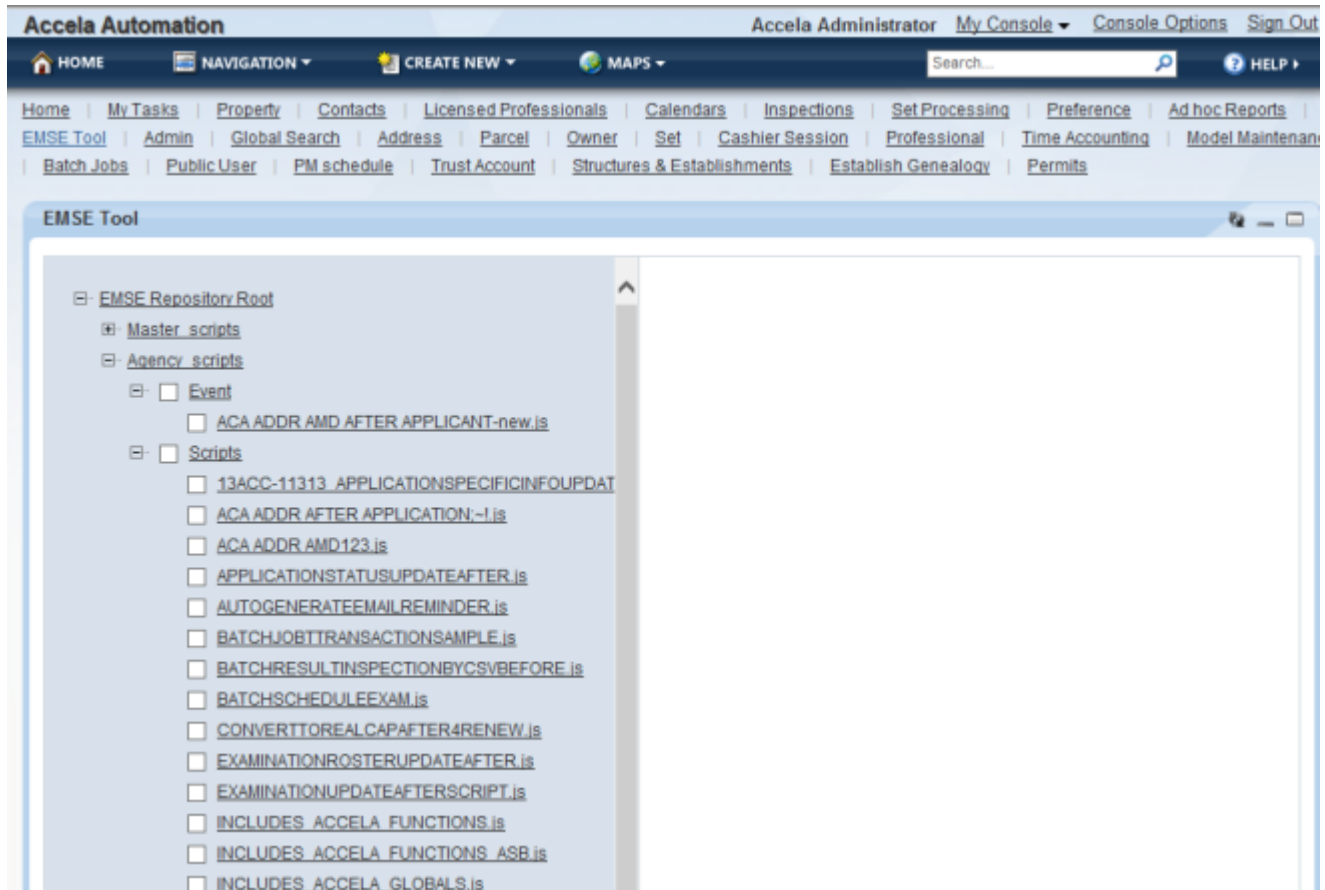
Pulling Scripts from Civic Platform

The EMSE Tool enables an agency user to initially populate or refresh an SCCS repository with the scripts stored in Civic Platform.

Pulling a repository

The EMSE Tool automatically detects if a configured master or agency repository in the SCCS is empty. When you click the EMSE Tool link while the agency or master repository is empty, the EMSE Tool manager prompts you to pull all the scripts. Click **OK** to confirm that you want to pull all the scripts from the Civic Platform repository into your SCCS repository. After the EMSE Tool has pulled all scripts, both repositories are synchronized.

Pulling a script

If a script has been updated within Civic Platform, you can select that script in the EMSE Tool to see the highlighted modifications on the AA script pane and pull the modified script into your SCCS repository. To pull the script from Civic Platform into your SCCS repository, click the **Pull** button under the AA script pane. After the EMSE Tool has pulled the script from Civic Platform into the SCCS repository, both repositories are synchronized and the script is no longer displayed under the EMSE repository tree structure.

Deploying Scripts

When you deploy one or more scripts, the EMSE Tool validates each script and uploads the script(s) to the Civic Platform database.

To deploy one or more scripts:

1. Select the script(s) from the EMSE Repository tree structure.

2. Right-click, and choose **Deploy**.

The EMSE Tool uses the JavaScript engine to validate script syntax such as missing keyword, incomplete loop or branch, undefined variable, unmatched symbol pairs, and others. For each script validation failure, the EMSE Tool displays the script line number where the syntax error occurred. Fix the error in your script editor, check-in your modifications in your SCCS repository, and deploy the script again in EMSE Tool. After a script has been deployed, it is stored in the Civic Platform database, and is no longer displayed in the EMSE Repository tree structure.

Deploying Custom Functions

Deploying custom functions consists of:

• Building the INCLUDES_CUSTOM script by concatenating all custom function JavaScript files in the INCLUDES_CUSTOM folder into one INCLUDES_CUSTOM.js file.

• Deploying the INCLUDES_CUSTOM script into Civic Platform

To build the INCLUDES_CUSTOM script:

1. Select the INCLUDES_CUSTOM folder from the EMSE Repository tree structure.
The left script pane lists the custom scripts in the Includes Custom folder.

2. Click **Build**.
The right script pane displays the Includes Custom script file containing all custom scripts.

To deploy the Includes Custom script into Civic Platform:

1. Select the INCLUDES_CUSTOM folder from the EMSE Repository tree structure.
The left script pane lists the custom scripts in the Includes Custom folder.

2. Click **Deploy**.
The right script pane displays the Includes Custom script file containing all custom scripts.

To find the INCLUDES_CUSTOM script in Civic Platform:

1. Go to Civic Platform Admin Tools.

2. Select **Events > Custom Scripts**.
The following shows a sample INCLUDES_CUSTOM script in Civic Platform:

Comparing Scripts

You can compare scripts in your SCCS repository with the deployed scripts in Civic Platform to determine what has been modified.

To compare a script between Civic Platform and SCCS:

1. Select the script in the EMSE Repository tree structure.

2. Right-click, and choose **Diff**.

   The EMSE Tool displays the script from the SCCS repository on the left pane and the script from Civic Platform on the right pane, counts the number of differences, and highlights the differences. For example:

New Functions in Master Scripts 3.0

Master Scripts 3.0 includes the following new functions in INCLUDES_ACCELA_FUNCTIONS and INCLUDES_ACCELA_FUNCTIONS_ASB:

- activeLicense

- activeTasksCheck

- addAddressDistrict

- addAdHocTask

- addFeeByDate

- addFeeWithExtraDataByDat

- addGuideSheet

- addParameter

- addPublicUserLPsToRecord

- asiTableValObj

- associateRefContactAddressToRecordContact

- CallWebService

- checkRequiredASIFields

- comparePeopleStandard

- contactObj

- convertContactAddressModelArr

- copyAppSpecific4ACA

- copyEducation

- createLicense

- createRecord

- createRefContactAddressFromAddress

- createRefLP4Lookup

- dateDiff

- deactivateActiveTasks

- decode64

- deleteLicensedProfessional

- describe

- docWrite

- doScriptActions

- doStandardChoiceActions

- editAppSpecific4ACA

- editCapConditionStatus

- editConstTypeCode

- editCreatedBy

- editEstimatedJobValue

- editFirstIssuedDate

- editTaskACAVisibility

- encode64

- externalLP_CA_3_2

- feeBalanceFromDate

- generateReport

- generateReport4Workflow

- genericTemplateObject

- getACADocDownloadParam4Notification

- getACADocumentDownloadUrl

- getACARecordParam4Notification

- getACARecordURL

- getACAUrl

- getAddressCountyByAddressType

- getAddressLineByAddressType

- getAppConditions

- getContactArrayBefore

- getContactByType

- getContactObj

- getContactObjBySeqNbr

- getContactObjs

- getContactObjsByCap

- getContactParams4Notification

- getDateDiff

- getDocumentList

- getParentByCapId

- getParentCapID4Renewal

- getParentLicenseCapID

- getPartialCapID

- getPeople

- getPrimaryAddressLineParam4Notification

- getPrimaryOwnerParams4Notification

- getProp

- getRecordParams4Notification

- getRenewalCapByParentCapIDForReview

- getRootNode

- getRoots

- getScriptAction

- getScriptAction_v_1_6

- getScriptText

- getURLToNewRecord

- getUserEmail

- getUserFullName

- guideSheetObject

- handleError

- htmlEncode

- include

- insertTask

- isBlank

- isEmpty

- isMatchPeople

- isReadyRenew

- isRenewProcess

- isSameNode

- licenseObject

- licenseProfObject

- linkPublicUserToContact

- loadAddressAttributes4ACA

- loadAppSpecific

- loadAppSpecific4Contact

- loadASITables

- loadASITables4ACA

- loadASITablesBefore

- logGlobals

- logMessage

- lpSet

- pairObj

- paymentByTrustAccount_V5

- prepareRenewal

- removeRefContactAddressFromRecordContact

- replaceNode

- runEvent

- runReport4Email

- runReport4EmailOrPrint

- runReportAttach

- scheduleInspect

- sendNotificatio

- setContactTypeFlagByType

- setLicExpirationDate

- stringPrototypes

- token

- transferFeesAndPayments

- transferReceiptAndApply

- updateEnfOfficer

- updateFeeItemInvoiceFlag

- updateGuidesheetASIField

- updateGuidesheetID

- verhoeff

Script Controls

Related Information

Understanding Script Controls

Understanding Script Control Syntax

Understanding Criteria (the If Clause)

Understanding Actions (the Then Clause)

Specifying Script Controls as Standard Choices

Understanding Script Control Branching

Naming Inspection Result Events

Exploring an Object

Understanding Script Controls

This section describes the legacy process of using Standard Choice script controls to instruct Civic Platform how to perform before and after event activities.

📝 **Note:** For more efficient script development and deployment, Accela recommends the use of the JavaScript framework (provided in Civic Platform 8.0.1) to implement customizations to the master scripts. See JavaScript Scripting Framework.

Each script control provides parameters to master script functions (Master Script Function List) within a framework of conditional (if-then-else) expressions. A single Standard Choice can contain multiple script controls. The master script evaluates the script controls in the order that the Standard Choice numbering specifies.

Script controls use the caret (^) symbol to delimit the if clause (predicate) from the then clause (consequent) and the else clause (alternative) in a single conditional expression. Civic Platform interprets the first clause as the if clause, the second clause as the then clause, and the third (optional) clause as the else clause.

Each clause in a script control calls a master script function and provides parameter values required by that function. The variables associated with the scriptable event () determine the scope of possible variables that the script control provides to the master script function.

Enclose master script function parameters in parenthesis. Use a comma to delimit master script function parameters. Enclose string parameters in double straight quotes. JavaScript Primer provides additional Javascript syntax elements you can use in script controls.

Example Use Case

Script Control Syntax shows a single script control. This script control says, "If the current record type is not a Building/Reroof type, then assess but do not invoice all of the fees from the fee schedule called BLDCR05." The master script function that the script control calls,

```
appMatch
```

for example, provides a return value, in this case true or false, to determine whether to perform the function in the then clause.
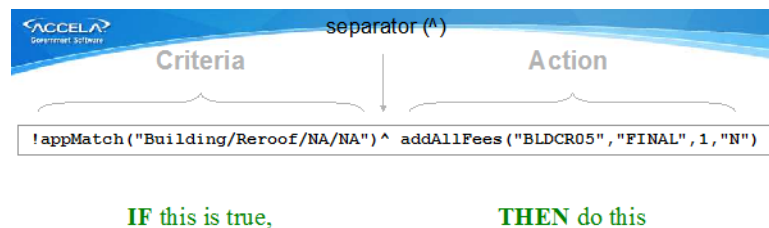
Figure 11: Script Control Syntax

Example Use Case

Script Control Structure (if/then/else) says, "If Acres Disturbed is less than 5 then assess but do not invoice the SMALLACRE fee, else assess but do not invoice the BIGACRE fee."
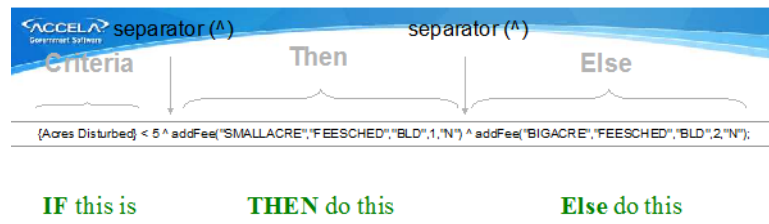
Figure 12: Script Control Structure (if/then/else)

Understanding Script Control Syntax

The following topics address the syntax for script controls.

Topics:

- Understanding Case Sensitivity

- Understanding Variable and Function Names

- Understanding Curly Brackets

- Understanding Argument Types

Understanding Case Sensitivity

The master scripts and underlying JavaScript require case sensitivity for function calls or when referring to a variable. For example in Script Control Structure (if/then/else) you see the function addFee called in both the then and else action. If you write the same script control but call the function AddFee, the script returns an error that the function AddFee does not exist. The script considers addFee and AddFee two different function names.

Understanding Variable and Function Names

Variables and function names in the master scripts follow the camelCase practice. For example totalSquareFeet, taxiNumber, addFee(), etc. Always be aware of case sensitivity as it many times could be the culprit of causing script errors.

Understanding Curly Brackets

We already saw the usage of the caret (^) to form conditional statements, another master script specific syntax is the usage of curly brackets { }. When a user triggers an event, Civic Platform calls the associated master script. Before EMSE evaluates the first line of script controls, the master script does some pre-work to initialize and set the value of several global and event-specific variables that the script controls can reference. Some of this pre-work loads application information, task information, and parcel attributes into individual variables. The script control encloses each of these variables between two curly brackets (Script Control Structure (if/then/else)). For example, {Acres Disturbed} in the script control condition indicates an application specific Information field.

Understanding Argument Types

Always enclose strings in double quotes. For example:

- the criteria -- {Land Use} == "Farming"

- setting the value of a variable -- layerName = "Zoning"

- a function call that accepts string parameters addFee("AppFee","BLD_11","FINAL",1,"Y")

Do not enclose numeric fields in double quotes.

Script controls must be valid JavaScript. If a script control deviates from JavaScript syntax, outside of that which is unique to the master scripts, syntax errors occur.

Understanding Criteria (the If Clause)

Criteria must always evaluate to either true or false. A criteria statement can contain logical operators, such as ==, >, >=, <, <=, or != to evaluate if a statement is true or false, and can also call functions that return true or false (Criteria Examples with Single Operators).

Table 4: Criteria Examples with Single Operators

Criteria	Description
true	If you use a true as the if clause, the specified action always executes.
`appMatch("Building/Commercial/*/*")`	The appMatch function returns true or false depending on whether the current record type matches the record type in the function parameter. The asterisks (*) indicate a wildcard. In this example, any record types that start with Building/Commercial return a true.

`!appMatch("Building/Commercial/*/*")`
`appMatch("Building/Commercial/*/*") !=`
` true`
	These two examples mean the same thing, with different syntax.Both say, if the current record type is not under Building/Commercial do the action.

`inspType == "Final Inspection"`	Use double equals (==) check whether a value equals another variable or a string. In the example, if the value for the `inspType` variable of the triggered event equals "Final Inspection" then execute the associated action.
`{STRUCTURE DETAILS.Total Square Feet} >=` ` 2000`	You can use criteria to test the value of an Application Specific Information. In the example, if the value of the ASI field name Total Square Feet within the ASI subgroup STRUCTURE DETAILS equals or is greater than 2000, then execute the action.A period delimits the ASI subgroup name which precedes the ASI field name. 📝 **Note:** You can configure a global variable to precede all ASI field names with the ASI subgroup name.
`{ParcelAttribute.Neighborhood} ==` ` "Downtown Area"}`	Similar to ASI fields, enclose a parcel attribute in curly brackets, and prepend it with ParcelAttribute and a period separator. In the example, if the parcel attribute Neighborhood equals Downtown Area then execute the associated action.
`proximity("GIS","Schools",parseInt({Number` ` of feet}));`	Similar to the appMatch function example, the function proximity returns true or false. The function checks to see if the parcel for the current record falls within a buffered distance on a layer within GIS. The example checks whether the current record's parcel is within a certain number of feet (a value specified in an ASI field).
`!taskStatus("Permit Issuance","Issued");`	The taskStatus checks to see if a workflow task currently has a particular status. The example checks to see if the status of the permit issuance task updated to issued. You can use this type of check to prevent inspection scheduling before permit issuance.

Topics:

- Understanding Criteria with Multiple Conditional Statements

Understanding Criteria with Multiple Conditional Statements

Criteria (the if clause) can contain multiple conditional statements separated by the logical "and" operator (&&) and/or the logical "or" operator (||). All "and" conditions must be true in order for the criteria to be true. Only one "or" condition needs to be true in order for the criteria to be to true.

You can use as many logical operators in your criteria as you need to satisfy your business rules. You use parenthesis to specify the evaluation order of criteria with multiple conditions and multiple operators (Criteria Examples with Multiple Operators).

Table 5: Criteria Examples with Multiple Operators

Criteria	Description
`inspType == "Final Inspection" && !isScheduled("Electrical")`	This condition occurs during an inspection event. The criteria checks whether the inspection type that triggered the event is a final inspection and whether Civic Platform scheduled an electrical inspection.You can use this criteria during an InspectionScheduledBefore event to prevent a final inspection before an electrical inspection.
`feeExists("LICFEE") && balanceDue <= 0`	This condition checks to see if the fee item LICFEE exists on the current record and whether the balance due on that fee item is less than or equal to 0.You can use this condition to ensure that the license includes the required license fee and that the applicant does not owe any fees.Master scripts set the balanceDue variable before Civic Platform evaluates the script controls.
`wfTask == "Supervisor Review" && (wfStatus == "Approved" || wfStatus == "Not Required")`	This criteria uses parenthesis to evaluate the "or" clause before evaluating the "and" clauses. The criteria says, If you update the Supervisor Review task to Approved or Not Required, do the associated action.An alternative way to write this criteria is: wfTask == "Supervisor Review" && matches(wfStatus,"Approved","Not Required"). 📝 **Note:** The matches function works similarly to a SQL IN clause. It is checking to see if the value in the first parameter is equal to any of the following parameters.

Understanding Actions (the Then Clause)

The right side of the script control (to the right of the caret) tells Civic Platform what to do if the criteria evaluates to true. In most cases the action portion calls a master script function to perform an action (Master Script Function List).

To execute multiple actions, you can write your script controls two ways; 1) list each action separated by a semicolon (;) on the same line (Table 6: Multiple Actions on Same Line), or 2) put each action on a different line (Table 7: Multiple Actions on Different Lines).

Table 6: Multiple Actions on Same Line

#	Value Description
01	{Review Required} == "Yes" ^ addFee("REVIEWFEE","FEESCHEDULE","FINAL",1,"Y"); scheduleInspection("Special Review Inspection",1); //any additional action…

When you put multiple actions on different lines, start each new line with a caret (^).

Table 7: Multiple Actions on Different Lines

#	Value Description
01	{Review Required} == "Yes" ^ addFee("REVIEWFEE","FEESCHEDULE","FINAL",1,"Y") ;
02	^ scheduleInspection("Special Review Inspection",1);
03	^ //any additional needs for the action…

Best practice recommends the multiple line approach due to width limitations for Standard Choice item entries.

To maintain consistency, best practice recommends the use of semicolons at the end of each line, even for single action script controls (Table 8: Single Action Script Controls with Semicolons).

Table 8: Single Action Script Controls with Semicolons

Action	Description
activateTask("Plan Review");	Activates the workflow task on the current record specified by the first parameter.
addAppCondition("Permit","Applied","Re-Inspection Fee","Re-Inspection Fee","Hold");	Applies a hold condition to the current record with the specific type, name, and description.
childApp = createChild("Building","Commercial","Plumbing","NA","New Walmart");	Creates a child record for the Building/Commercial/ Plumbing/NA record type instance named New Walmart.
editAppSpecific("Total Value",parseInt({Sq Ft}) * parseInt({Price per Sq Ft}));	Updates the ASI field Total Value to the product of the ASI fields {Sq Ft} and {Price per Sq Ft}.

Specifying Script Controls as Standard Choices

Standard Choice Annotated shows a Standard Choice script control named ApplicationSubmitAfter and Standard Choice Script Controls defines the components on the Standard Choice UI.

Figure 13: Standard Choice Annotated

Table 9: Standard Choice Script Controls

#	Name	Description
A	Name	Standard Choice name. The master script for each event designates the name of the standard choice that is the entry point for script execution. A script control can implement the branch function to refer to other script controls.
B	Description	Text area used to describe the purpose of the script controls that the Standard Choice contains. You can use this area to maintain a script control change log.
C	Status	You can designate a Standard Choice as Enabled or Disabled. When disabled, Civic Platform does not execute the script controls in the Standard Choice and does not return an error if a master script calls the Standard Choice.
D	Type	Specifies the type of Standard Choice. Use EMSE for script controls. The EMSE type designation does not affect any Civic Platform functions.
E	Value	Best practice recommends that you increment script controls by ten (eg. 10, 20, 30) to leave room for inserted script controls in the future. As of version 2.0 of the master script framework does not require sequential script control numbering.
F	Value Desc	Contains the script controls.
G	Debug Options	showMessage – when set to true, this option presents a pop-up window to the user with a custom message about script execution.showDebug – when set to true,1, 2 or 3, this option present a pop-up window that displays debug information including variable values and script control results.
H	Script Controls Example	Lines 20-70 contain script control examples. The master script evaluates script controls in the order the Standard Choice specifies.
I	Active	You can set a script control to Active or Inactive. Select Update to enable a change. Civic Platform skips over script controls set to Inactive.
J	Delete	You can delete a script control. After confirming a deletion, Civic Platform permanently removes the item. You cannot undo a delete operation.

#	Name	Description
K	Update	Use to commit changes. This includes updating the description, status, type, value, value desc, and active flag.
L	Add	Enables the addition of a new Standard Choice.
M	Cancel	Enables you to navigate back to the page from which you came without committing changes.

Some additional standard choice details to be remember:

- Standard Choices do not have an auto-save feature. Update your Standard Choice often to ensure you do not lose your work.

- You cannot delete Standard Choices. Be careful when naming your Standard Choices.

- You cannot lock a Standard Choice. An update someone else makes to a Standard Choice refreshes the Standard Choice with their changes and wipes out any changes you might have made, but not yet committed.

Understanding Script Control Branching

Each individual master script specifies the Standard Choice that provides the script controls for processing an event (Setting the controlString).

Figure 14: Setting the controlString

The master script represented in Setting the controlString is for the event ApplicationSubmitAfter event. The value of the controlString variable name of the Standard Choice. For most master scripts the controlString value matches the event name for which the Standard Choice contains the script controls.

Master script evaluation of script controls begins with the first line in the Standard Choice and ends with the last line in the Standard Choice.

You can branch a script control process from one Standard Choice to another Standard Choice. The branch script control action functions like a sub-routine in traditional programming.

When a master script encounters a branch script control action, the master script stops evaluation of the current standard choice and begins evaluation of the script controls in the branched to Standard Choice. Use the following syntax to specify a branch action:

```
branch("<Standard Choice Name>")
```

where: *<Standard Choice Name>* is the name of the Standard Choice containing the branched to script controls.

In the example branch action (Branch Action Example) the master script branches to the "Calculate Permitting Application Fees" Standard Choice when a workflow approves an application for processing. The master script then evaluates all the script controls in the "Calculate Permitting Application Fees" Standard Choice implement the application fees' business rules.

Table 10: Branch Action Example

#	Value Description
10	wfTask == "Application Acceptance" && wfStatus == "Approve for Processing" ^ branch("Calculate Permitting Application Fees");

After the master script evaluates all script controls in the branched to Standard Choice, the master script returns to the place that contains the originating branch action, evaluates anymore actions that remain on the same line, then moves onto the next line in that Standard Choice (Branching Flow).
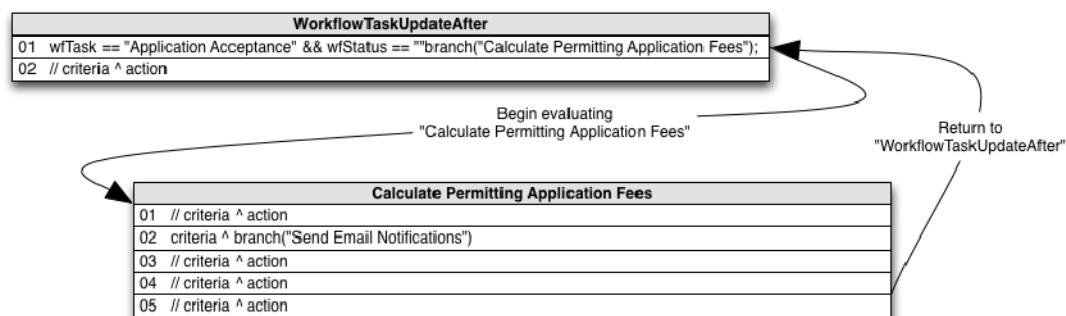
Figure 15: Branching Flow

You can branch to as many levels as required. The same rules that apply single level branching apply to multiple level branching. The master script completes evaluation of all script controls in the lowest level Standard Choice to which you branch first, and completes evaluation of all the script controls in the

highest level Standard Choice, the Standard Choice that served as the entry point for the master script, last (Multiple Level Branching).
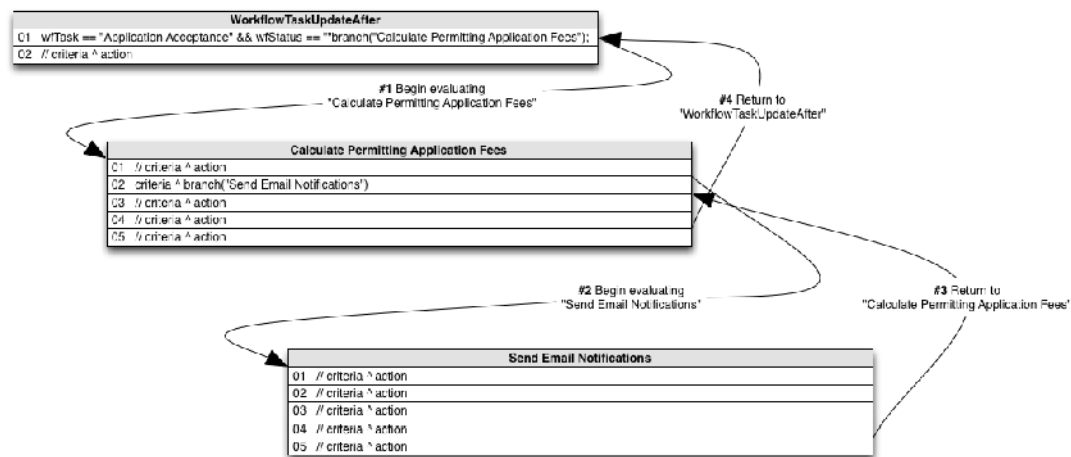
Figure 16: Multiple Level Branching

The flow of script control shown in Multiple Level Branching is as follows:

- Begin script control evaluation with line 01 of "WorkflowTaskUpdateAfter"

- Branch line 01 of "WorkflowTaskUpdateAfter" to "Calculate Permitting Application Fees"

- Continue script control evaluation with line 01 of "Calculate Permitting Application Fees"

- Branch line 02 of "Calculate Permitting Application Fees" to "Send Email Notifications"

- Continue script control evaluation with line 01 of "Send Email Notifications", and continue script control evaluation through line 05

- Return to "Calculate Permitting Application Fees" and continue to evaluate script controls that follow the branch action, on line 02 through line 05

- Return to "WorkflowTaskUpdateAfter" and continue to evaluate script controls that follow the branch action, on line 01 through line 02

- End script control evaluation after evaluating line 02 of "WorkflowTaskUpdateAfter"

Topics:

- Using Branching to Implement a For Loop

- Using Pop-Up Messages

- Using Data Validation

- Using Variable Branching

- Branching to the Same Standard Choice from Different Events

Using Branching to Implement a For Loop

By default, JavaScript uses curly brackets { } to indicate the start and end of a unit of code for conditional statements or loops. In master script syntax, curly brackets indicate retrieval of a value (Understanding

Curly Brackets) not the start and end of a unit of code. As a workaround, use branching to implement body of code functionality and loop functionality.

Incorrect Loop Using Curly Brackets provides an incorrect example of a loop implemented with curly brackets.

Table 11: Incorrect Loop Using Curly Brackets

#	Value Description
01	contactArray.length > 0 ^ for (ca in contactArray) { thisContact = contactArray[ca];
02	^ if (thisContact["email"] != "") email("noreply@accela.com",thisContact["email"],"Permit Update","Your permit has been issued."); }

The master script returns several errors for these script controls due to incorrect use of curly brackets:

• Line 01 opens a curly bracket but does not close the curly bracket on the same line

• Line 02 closes a curly bracket but does not open the curly bracket on the same line

To workaround the syntax issue, you can use a branch action to designate a body of code for a loop (Branch Implementation for Body of Code Loop and Contact Email Loop).

Table 12: Branch Implementation for Body of Code Loop

#	Value Description
01	contactArray.length > 0 ^ for (ca in contactArray) branch("Contact Email Loop");

Table 13: Contact Email Loop

#	Value Description
01	true ^ thisContact = contactArray[ca];
02	^ if (thisContact["email"] != "") email("noreply@accela.com",thisContact["email"],"Permit Update","Your permit has been issued.");

When using the branch action for a body of code loop, best practice prescribes appending the word "loop" to the end of the Standard Choice name.

Using Pop-Up Messages

Master scripts use two variables to specify whether or not to complete the transaction and the message contents to display in a pop-up window. The ScriptReturnCode variable specifies whether or not to complete the transaction.

```
aa.env.setValue("ScriptReturnCode", "<value>");
```

where: <*value*> can be 0 or 1 and:

• 0 – indicates to proceed as normal

• 1 – stop the user action and return to the previous page.

The ScriptReturnMessage variable specifies the content of a message to display in a pop-up window.

```
aa.env.setValue("ScriptReturnMessage", "<myMessage>");
```

where: <*myMessage*> contains the content of the message to display.

You can use the ScriptReturnMessage to:

• notify users of an additional required activity

- notify users of a completed an activity (sent an email and added a condition, for example)

- notify users of useful information (the current location of the application, for example).

> 📝 **Note:**
>
> Civic Platform does not display an empty message.

Pop-Up Message Example shows and example of a pop-up message and the accompanying variables in the master script.
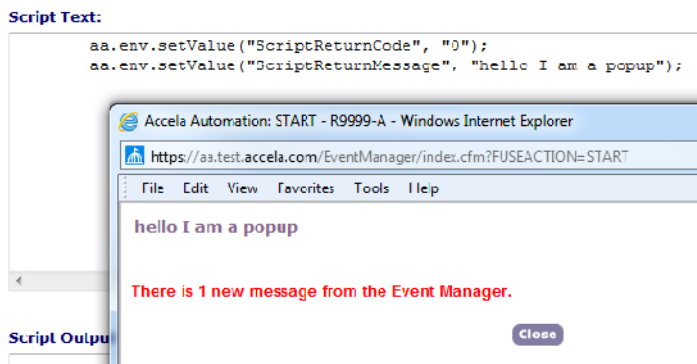
Figure 17: Pop-Up Message Example

You can call the comment function for different script control actions to generate message text specific to evaluation of particular master script functions. Each message returned from the comment function displays on a new line in the pop-up window.

To display a pop-up message after evaluation of the last script control, set the showMessage function to true. If you do not set the showMessage function to true, no message displays, regardless of the number of times you call the comment function.

Script Controls for Displaying Pop-up Messages shows how to call the comment and showMessage functions from a script control.

Table 14: Script Controls for Displaying Pop-up Messages

#	Value Description
10	true ^ showMessage = true;
20	true ^ comment("The service request submission is successful");
30	true ^ comment("Please remind the citizen to sign up on Citizen Access to submit future requests and receive email status updates.");

Message Window shows the resulting pop-up window generated by the script controls in Script Controls for Displaying Pop-up Messages resulting from submission of a service request in Civic Platform.
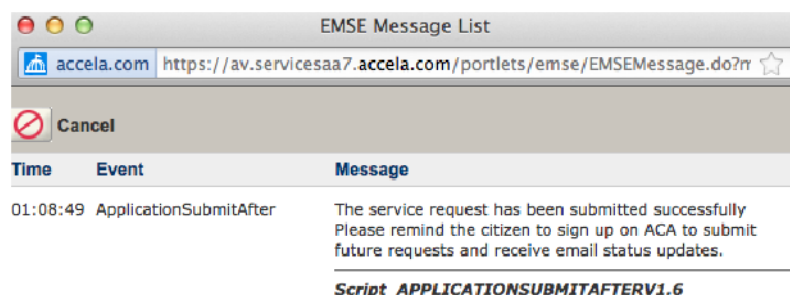
Figure 18: Message Window

📝 **Note:**

If you set the showMessage function to true in an early evaluated script control, but the pop-up message never appears, you can set the showMessage function to false in a later evaluated script control.

You can use HTML tags in the strings submitted to the comment function to add additional formatting (bold, underlined, additional blank lines, for example).

The EMSE_DISABLE_MESSAGES Standard Choice controls display of messages to internal and public users. If you set the entry for either InternalUsers or PublicUsers to "Yes", no pop-up messages display to the user.

Using Data Validation

You can use a 'before' event type to validate submitted data, before saving to the database (Understanding Events), and cancel the transaction if the submitted data does not meet the data validation business rules that your scripts prescribe.

If a data submission attempt fails data validation, provide a message to the user as to why you cancelled the transaction (Using Pop-Up Messages). To stop the transaction, set the cancel variable in the script control to true.

```
cancel = true
```

Script Control for Data Validation provides script control example that cancels a transaction and tells the user why Civic Platform cancelled the transaction. Make sure that the message indicates the reason for cancelling the transaction so the user can correct the situation.

Table 15: Script Control for Data Validation

#	Value Description
01	{Project Name} == "" ^ showMessage = true; comment("You must designate a 'Project Name' in the ASI prior to processing the application"); cancel = true;

Cancelled Transaction Message shows the message displayed to the user.
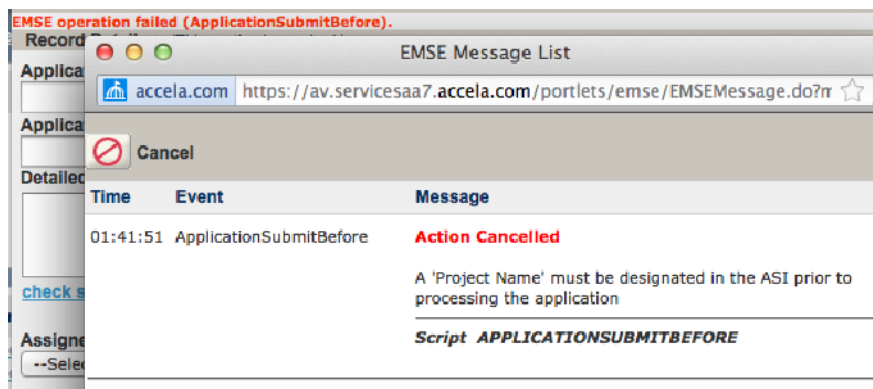
Figure 19: Cancelled Transaction Message

Data validation can be especially helpful for many events, including the following:

- ApplicationSubmitBefore

- WorkflowTaskUpdateBefore

- InspectionScheduleBefore

- InspectionResultSubmitBefore

- PaymentReceiveBefore

- ApplicationStatusUpdateBefore

Using Variable Branching

To enable variable branching for all master scripts, set the enableVariablebranching variable in the "User Configurable Parameters" section of the INCLUDES_ACCELA_GLOBALS script to true.

```
enableVariablebranching = true;
```

📝 **Note:**

When you set variable branching to true, the documentOnly functionality does not work.

ApplicationSubmitAfter – Without Variable Branching shows an example of how the ApplicationSubmitAfter Standard Choice uses branching to organize scripts.

Figure 20: ApplicationSubmitAfter – Without Variable Branching

Without variable branching, you provide a separate script control branch action for each four level record type specification. If you have many unique record types in your implementation that require scripting, this approach involves many lines of script controls.

With variable branching, you use variables to specify the argument of the branch function instead of a literal string value. The master scripts resolve these variables and the branch function calls the appropriate Standard Choice.

Variable branching enables the branch function to accept string variables, in addition to hard coded strings concatenated together, as a single parameter. For example, with variable branching you can write the following:

```
true ^ branch("Assess Fees");
```

like the following:

```
true ^ variable1 = "Assess";
true ^ variable2 = "Fees";
true ^ branch(variable1 + " " + variable2);
```

You can use this principle to represent all possible four level record type specifications (Group/Type/Subtype/Category) with the following six variables:

```
branch(appTypeArray[0] + "/*/*/*");
```

```
branch(appTypeArray[0] + "/" + appTypeArray[1] + "/*/*");
```

```
branch(appTypeArray[0] + "/" + appTypeArray[1] + "/" + appTypeArray[2] + "/
*");
```

```
branch(appTypeArray[0] + "/*/*/" + appTypeArray[3]);
```

```
branch(appTypeArray[0] + "/" + appTypeArray[1] + "/*/" + appTypeArray[3]);
```

```
branch(appTypeString);
```

where the appTypeArray number in square brackets represents the level, of the four-level record type specification, contained in the array. When an event triggers, the master script resolves these variables based on the record type specification for the selected record.

The following provides an example resolution for an instance of the Licenses/Business/Taxi/Application record type:

```
branch(Licenses/*/*/*)
```

```
branch(Licenses/Business/*/*)
```

```
branch(Licenses/Business/Taxi/*)
```

```
branch(Licenses/*/*/Application)
```

```
branch(Licenses/Business/*/Application)
```

```
branch(Licenses/Business/Taxi/Applica
```

```
tion)
```

The branched to Standard Choices contain the script controls for all records in the record type hierarchy level indicated in the branch argument. For example, the script controls in the "Licenses/*/*/*" Standard Choice apply to all license record types, including the (Licenses/Business/Taxi/Application) record type, whereas the script controls in the "Licenses/Business/Taxi/Application" Standard Choice only apply to instances of the Licenses/Business/Taxi/Application record type.

The preceding example branches to the same Standard Choice, regardless of the event trigger. To branch to a different Standard Choice for each event trigger, manually add an event specification into the variable.

```
branch("<my_event>:" + appTypeArray[0] + "/*/*/*");
```

```
branch("<my_event>:" + appTypeArray[0] + "/" + appTypeArray[1] + "/*/*");
```

```
branch("<my_event>:" + appTypeArray[0] + "/" + appTypeArray[1] + "/" +
 appTypeArray[2] + "/*");
```

```
branch("<my_event>:" + appTypeArray[0] + "/*/*/" + appTypeArray[3]);
```

```
branch("<my_event>:" + appTypeArray[0] + "/" + appTypeArray[1] + "/*/" +
 appTypeArray[3]);
```

```
branch("<my_event>:" + appTypeString);
```

where *<my_event>* is the three to five character abbreviation that represents the event (Scriptable Event Abbreviations). For example, you can use the ASA abbreviation to represent the ApplicationSubmitAfter/Before event in the branch variable.

```
branch("ASA:" + appTypeArray[0] + "/*/*/*");
```

```
branch("ASA:" + appTypeArray[0] + "/" + appTypeArray[1] + "/*/*");
```

```
branch("ASA:" + appTypeArray[0] + "/" + appTypeArray[1] + "/" +
 appTypeArray[2] + "/*");
```

```
branch("ASA:" + appTypeArray[0] + "/*/*/" + appTypeArray[3]);
```

```
branch("ASA:" + appTypeArray[0] + "/" + appTypeArray[1] + "/*/" +
 appTypeArray[3]);
```

```
branch("ASA:" + appTypeString);
```

which resolves to the following:

```
branch(ASA:Licenses/*/*/*)
```

```
branch(ASA:Licenses/Business/*/*)
```

```
branch(ASA:Licenses/Business/Taxi/*)
```

```
branch(ASA:Licenses/*/*/Application)
```

```
branch(ASA:Licenses/Business/*/Application)
```

```
branch(ASA:Licenses/Business/Taxi/Application)
```

You must create a Standard Choice with the same name as each possible evaluation outcome of the branch argument variables. Use event acronyms and record type variables, in your branch arguments, to ensure a standard naming convention for your branched to Standard Choices, and to facilitate the organization and reuse of branched to script controls in the Standard Choices for group level record types (Licenses/*/*/*). When you apply this standard naming convention for your Standard Choices, you can use wildcard searches to return an inventory of Standard Choices setup for a specific record type. For example:

• %Licenses/Business/Taxi/% - returns all Standard Choices for taxi business licenses across all events

• %ASA:Licenses/Business/% - returns all Standard Choices for business licenses application submittal

• %/Application/% - returns all standard choices for application record types.

Table 16: Scriptable Event Abbreviations

Event	Abbrev.	Event	Abbrev.
AdditionalInfoUpdateAfter/Before	AIUA / AIUB	InvoiceFeeAfter	IFA
ApplicationConditionAddAfter	ACAA	LicProfLookupSubmitAfter/Before	LPLSA / LPLSB
ApplicationConditionDeleteBefore	ACDB	LicProfUpdateAfter/Before	LPUA / LPUB
ApplicationConditionUpdateAfter/Before	ACUA / ACUB	ParcelAddAfter/BeforeV360ParcelAddAfter	PAA / PAB
ApplicationSpecificInfoUpdateAfter/Before	ASIA / ASIB	ParcelUpdateAfter	PUA
ApplicationStatusUpdateAfter/Before	ASUA / ASUB	PaymentProcessingAfter/Before	PPA / PPB

Event	Abbrev.	Event	Abbrev.
ApplicationSubmitAfter/Before	ASA / ASB	PaymentReceiveAfter/Before	PRA / PRB
ContactAddAfter/Before	CAA / CAB	PaymentReceiveBefore	PRB
ContactEditAfter/Before	CEA / CEB	RenewalInfoUpdateAfter	RIUA
ContactREmoveAfter/Before	CRA / CRB	TimeAccountingAddAfter/Before	TAAA / TAAB
ConvertToRealCapAfter	CRCA	VoidPaymentAfter/Before	VPA / VPB
DocumentUploadAfter/Before	DUA / DUB	WorkflowTaskUpdateAfter/Before	WTUA / WTUB
FeeAssessAfter/Before	FAA / FAB		
InspectionMultipleScheduleAfter/BeforeInspectionScheduleAfter/Before	ISA / ISB		
InspectionResultSubmitAfter/BeforeInspectionResultModifyAfter/BeforeV360InspectionResultSubmitAfter/Before	IRSA / IRSB		

Branching to the Same Standard Choice from Different Events

If you branch to the same Standard Choice from different events:

• Prefix the name of the branched to Standard Choice with the letters "CMN" (common).

• Followed the prefix with the record type scope.

• Append the end of the script control with a short description of its function.

```
wfTask == "Final Review" && wfStatus == "Ready to Issue" ^
 branch("CMN:Permits/*/*/*:INVOICE_ALL_FEES");
```

Naming Inspection Result Events

The following three events, that occur after an inspection result, violate the rule that the entry point Standard Choice (controlString value in the master script) match the event name:

- InspectionResultSubmitAfter (inspection result from AA Classic, GovXML, AMO, AW)

- V360InspectionResultSubmitAfter (inspection is result from AA)

- InspectionResultModifyAfter (inspection updated from AA - FID 8400 disabled)

Best practice prescribes use of the same InspectionResultSubmitAfter Standard Choice for each of these events. Update the master script variable controlString in each event's master script to "InspectionResultSubmitAfter". Use the IRSA acronym to refer to this event in your branch variable (Using Variable Branching).

Exploring an Object

When working with an object while writing scripts you can reference the Javadocs documentation (http://community.accela.com/p/doc_interfaces.aspx) to explore the class it belongs to including its properties and methods. Use the getClass() function to determine the class from which EMSE instantiated an object.

You can use Script Test to create an object and use a for loop to explore the methods and properties available to the object (Show all methods of an object and Show all properties and their values for an object).

Script Text:
```
capId = aa.cap.getCapID("C03-000659").getOutput();
capDetail = aa.cap.getCapDetail(capId).getOutput();

aa.print("capDetail is a " + capDetail .getClass());

for (x in capDetail )
        if (typeof(capDetail[x]) == "function")
                aa.print("  " + x);
```

Submit

Script Output (script debug output will appear in this box when you submit this f
```
capDetail is a class com.accela.aa.emse.dom.CapDetailScriptModel
  hashCode
  getGaFname
  setBuildingCount
  setGaAgencyCode
  notifyAll
  setCreateBy
  getGaAgencyCode
  getGaMname
  getDfndtSignatureFlag
  equals
  getEndAsgnDate
```

Figure 21: Show all methods of an object

Script Text:

```
capId = aa.cap.getCapID("C03-000659").getOutput();
capDetail = aa.cap.getCapDetail(capId).getOutput();

aa.print("capDetail is a " + capDetail .getClass());

for (x in capDetail )
        if (typeof(capDetail[x]) != "function")
                aa.print("  " + x + " = " + capDetail[x]);
```

[Submit]

Script Output (script debug output will appear in this box when you submit this f

```
capDetail is a class com.accela.aa.emse.dom.CapDetailScriptModel
  infractionFlag = null
  referenceType =
  asgnStaff =
  asgnDept = null
  balance = 23.5
  percentComplete = 0
  anonymousFlag = null
  offnWitnessedFlag = null
  ID3 = 00000
  ID2 = 00000
  ID1 = 30HIS
```

Figure 22: Show all properties and their values for an object

Citizen Access Page Flow Scripts

Related Information

Understanding Citizen Access Page Flow Scripts

Using Model Objects

Creating a Page Flow Master Script

Understanding Citizen Access Page Flow Scripts

When a citizen uses Citizen Access to create an application, Citizen Access creates a temporary record in the Civic Platform database and Citizen Access stores application information in a capModel object. Citizen Access stores capModel object data in memory for the duration of a user's session.

The capModel object contains all the details about the application. As the user progresses through the forms on each Citizen Access page, Citizen Access updates the capModel object data in memory. Upon completion and submittal of the application, Citizen Access passes the capModel object data to Civic Platform and Civic Platform creates a new record in the Civic Platform database.

You can use Expression Builder of page flow scripts to apply advanced business rules for Citizen Access applications. If you need to populate data on an Citizen Access form, try to use Expression Builder. If you need to populate data not displayed in Citizen Access, use page flow scripts.

Unlike Civic Platform scripts, page flow scripts associate with events from the Citizen Access Page Flow Admin tool. The user triggers page flow scripts when they navigate through different Citizen Access pages. You can associate a script to the following three page flow events:

- Onload – triggers when the page loads

- Before – triggers when the user clicks the continue button, it can prevent the user from progressing if data validation fails

- After – triggers when the user clicks the continue button, can implement automation in the application process

Civic Platform master scripts interact with record data that Civic Platform stores in the database. The Civic Platform master scripts do not work on Citizen Access page flow data.

For example, the editAppSpecific master script function updates an ASI field on the database record, but does not update the Citizen Access capModel object data stored in memory. The Citizen Access capModel object data overwrites the ASI field on the database record when Citizen Access submits a completed application.

Using Model Objects

The master script functions use get and set functions, with the "cap" variable, to retrieve and update information about the current record. Table 17: Retrieving the capModel Object Value shows how to get information, from the application specific information table, from a script control.

Table 17: Retrieving the capModel Object Value

#	Value Description
01	true ^ asit = cap.getAppSpecificTableGroupModel ();

Table 18: Updating the capModel Object Value shows how to update information, from the application specific information table, from a script control.

Table 18: Updating the capModel Object Value

#	Value Description
01	true ^ cap.setAppSpecificTableGroupModel(asit) ;

Table 19: Updating the capModel Object in Citizen Access shows how to pass capModel updates to Citizen Access at the end of your script.

Table 19: Updating the capModel Object in Citizen Access

#	Value Description
01	true ^ aa.env.setValue("CapModel",cap) ;

Creating a Page Flow Master Script

You must customize each page flow script to the associated page flow.

To create a custom page flow script

1. Make a copy of the universal script.

2. Create a new name for the copy that accurately identities. For example, ACA TN ASI Before, where:

   • ACA – indicates where the script runs

   • TN – is the page flow

   • ASI – is the page flow step

   • Before – is the event type (eg. Onload, After, Before)

3. Open the script in a text editor.

4. Update the controlString variable to match the name (Updating the controlString).

Figure 23: Updating the controlString

5. Install the script (Managing Scripts). Save the script with the same name as the controlString variable set in the previous step.

6. Log in to Citizen Access Admin.

7. Navigate to the proper page flow and page flow step.

8. Associate the newly added script to the proper event (Associating a script to an Citizen Access Page Flow event).
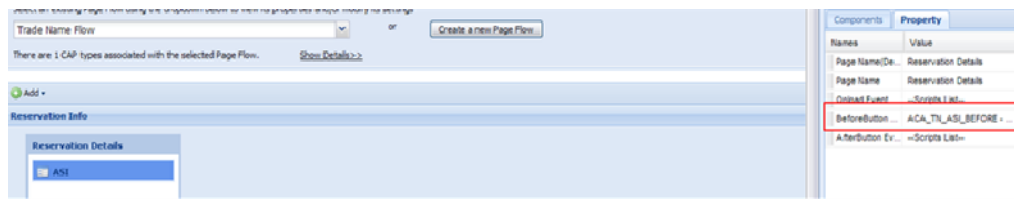
Figure 24: Associating a script to an Citizen Access Page Flow event

9. Create the standard choice entry point with the same name as the controlString variable.

10. Write appropriate script controls that interact with information stored in memory for the capModel object.

Join the conversation in the Accela Community for additional articles and discussions about Citizen Access Page Flows: http://community.accela.com/search/SearchResults.aspx?q=aca+page+flows

Batch Job Scripting

Batch jobs trigger scripts through a scheduled job in contrast to a user-invoked action. For example, you can schedule a nightly batch job, with an associated script, that looks for expired permits or licenses and updates them to an expired application and/or expiration status. At a high level batch scripts contain instructions to query records based on a specified filter, evaluate each returned record and take action for each record according to certain criteria. Civic Platform provides the Batch Job portlet (Batch Jobs Portlet) from where you can use UI controls to set parameters for the associated batch job script.
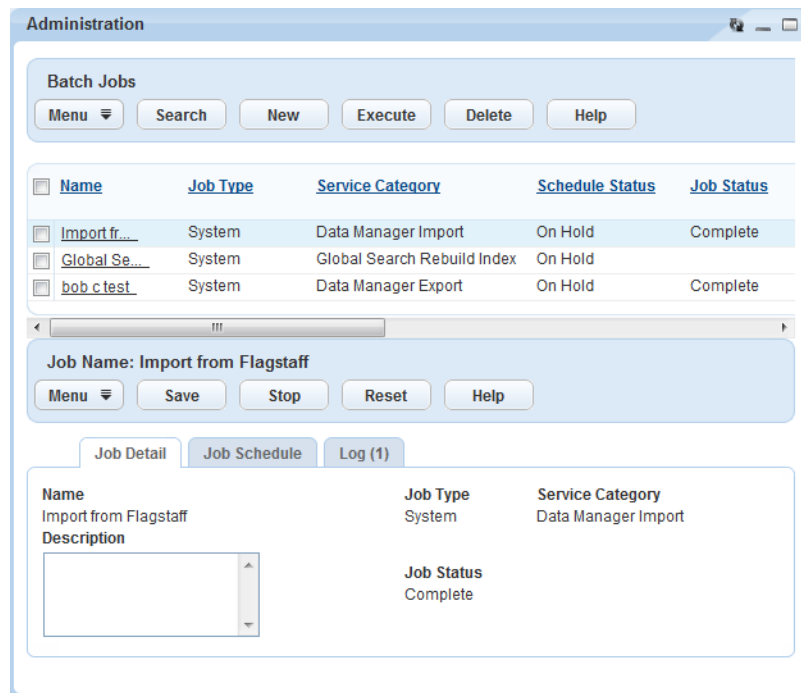
Figure 25: Batch Jobs Portlet

In addition, Civic Platform provides a batch job transaction manager for you to control transactions by scripts. The batch job transaction manager uses the following three methods to begin, commit, and roll back transactions separately.

```
aa.batchJob.beginTransaction(int seconds)
```

```
aa.batchJob.commitTransaction()
```

```
aa.batchJob.rollbackTransaction()
```

Script Testing

Related Information

Understanding the Script Test Tool

Testing an Event and Script Association

Running a

Troubleshooting

Understanding the Script Test Tool

Civic Platform provides the Script Test tool to test EMSE scripts. The Script Text tool simulates script execution by running scripts and displaying the resulting output. However, scripts run in the Script Text tool do not change any values in Civic Platform, Citizen Access, or the Civic Platform database.

You can use the Script Test tool to:

• Develop and test batch scripts.

• Develop and test custom functions.

• Troubleshoot and debug EMSE scripts.

📝 **Note:** The JavaScript framework (introduced in Civic Platform 8.0.1) allows you to use a Script Tester utility to test new or converted scripts without using the legacy Standard Choice script controls . See JavaScript Scripting Framework and Testing Scripts.

To access the Script Test tool

1. From the Classic Admin portlet, click **Admin Tools** > **Events** > **Script Test**.

Civic Platform displays the Script Test interface.

Figure 26: Script Test

Script Test Field Definitions provides information on the fields in the Script Test interface.

Table 20: Script Test Field Definitions

Script Transaction	A drop-down list that provides two options:
	• Always Rollback. The script outputs results to the Script Output window, but does not commit actions in Civic Platform. For example, if the script updates a workflow task for each license that meets certain criteria, the Script Output window indicates an updated workflow task while the status of the record in Civic Platform remains unchanged. The Always Rollback selection does allow scripts to affect autonumbers for Civic Platform objects. For example, if the script assesses and invoices a fee item, the sequence numbers for fees and invoices increments even though Civic Platform did not create the fee as a part of the script test.
	• Commit if SuccessfulThe script commits requested actions in Civic Platform and displays a result message in the Script Output window.
Script Initializer	Contains initialization requirements for testing the script. For example, when testing a batch script, you can set batch script parameter values, like specifying an email address, to set the scope of the batch script to a record type designation. You can also set script initialization values in Script Text.
Script Text	Contains the contents of the script. In general, you should create scripts in a text editor, then copy and paste them into the Script Test section.
Script Output	Contains the returned output upon the completion of the script execution. The script only displays text strings it sends to the aa.print method ("This should be sent to the Script Output window"). If your script contains the logMessage or logDebug function, make sure you send the variable that contains the debug or message output to the aa.print method (Using the aa.print Function).

> **Note:**
> Scripts that run longer than the specified EMSE time-out do not exit as gracefully as they do from a batch job or a set script execution.
> Script errors display in the next encountered error pop-up window as well as the Script Output section.
> The first couple lines of an error message often indicate an undefined variable or a non-existent function.
> The error message typically the line in the script where the error occurred.

Testing an Event and Script Association

In this scenario we install a test script and we associate the test script with an event. When the event triggers, the script displays the "EVENT TEST" pop-up message.

You can associate the test script with any event. When the "EVENT TEST" message appears in a pop-up window, you know the event that triggered the test script.

During your script development, create the association between the test script and the event you want to script first, then replace the test script with your developed script. This way, you know that you associated the developed script with the correct event.

To create the test script

1. From the Classic Admin portlet, click **Admin Tools** > **Events** > **Scripts**.

2. Click Submit to return a list of existing scripts.

3. Click Add.

4. Enter the following two script lines in the Script Content section:

```
aa.env.setValue("ScriptReturnCode","1");
aa.env.setValue("ScriptReturnMessage", "EVENT TEST");
```

5. Enter the name "EVENT_TEST" in the Script Code and Script Title sections.

6. Click Add.

Topics

- Associating the script to an event

- Testing the event

Associating the script to an event

To associate an event to the test script

1. From the Classic Admin portlet, click **Admin Tools** > **Events** > **Events**.

2. Click Submit to return a list of existing events.

3. If required, add a new event.

   a. Click Add.

   b. Select the event name from the Events drop-down list, then click Add.

4. Select an existing event by clicking on the red dot to the left of its name.

5. From the Event Detail screen select the script "EVENT_TEST" from the Script Name drop-down list.

6. Click Save.

Testing the event

Test the event to script association by triggering the event with a test case. For example, to test the "ApplicationSubmitAfter" event, create a new record of any type and click Submit, he "EVENT TEST" message appears in a pop-up window.

To disassociate the script from the event

1. From the Classic Admin portlet, click **Admin Tools** > **Events** > **Events**.

2. Click Submit to return a list of existing events.

3. Select the event by clicking on the red dot to the left of its name.

4. From the Script Name drop-down list, scroll to the top of the list and select the blank entry.

5. Click Save.

To delete the event

1. From the Classic Admin portlet, click **Admin Tools** > **Events** > **Events**.

2. Click Submit to return a list of existing events.

3. Select the event by clicking on the red dot to the left of its name.

4. Click Delete to disable the event.

Figure 27: Script and Event Detail pages

Running a Script Test

Incorrect scripts can permanently alter or erase data on your system. Always test your scripts before you implement them.

Use the Civic Platform script test utility to run your script in a test situation and to view the effects of the scripts or any errors that it generates.

When you run a script in the script test utility, Civic Platform runs the script and changes your system accordingly. You configure runtime parameters for the script test tool that instructs Civic Platform whether to rollback all changes resulting from the script or commit the changes resulting from the script.

Topics

- Using ScriptTester

Using ScriptTester

Civic Platform provides the ScriptTester.js master script file for you to test script controls without triggering an event from the user interface. You can copy and past the content of the ScriptTester.js file into the Script Test tool.

ScriptTester is a file that allows you to copy and paste its contents into Script Test and test script controls without having to trigger an event from the user interface.

To use the Script Test tool

1. Copy and paste the ScriptTester.js contents into the Script Text area of the Script Test tool.

2. Edit the myCapId variable to the tested altId.

3. Edit the myUserId to the tested user.

4. Update controlString to the standard choice entry point.

    📝 **Note:**

    The control string can be a standard choice entry point for the event (eg. ApplicationSubmitAfter) to test an events standard choices or a specific standard choice to test specific functionality

The ScriptTester.js master script inherits the native functionality of the Script Test tool to Always Rollback or Commit if Successful. You can use Always Rollback to repeatedly test a script and not commit the results to the database. You can select Commit if Successful after you troubleshoot your script and want to update the database.

📝 **Note:**

Always Rollback is the default.

Figure 28: ScriptTester in Script Test

Troubleshooting

Topics

- Launching the EMSE Debug Tool

- Understanding the Script Output Window

- Setting the showDebug Script Control

- Using the aa.print Function

- Using Biz Server Logs

Launching the EMSE Debug Tool

Civic Platform provides an EMSE debug tool so that users can debug scripts conveniently. Follow the instructions to launch the EMSE debug tool.

1. Log in to Civic Platform Classic.

2. Navigate to **Admin Tools** > **Events** > **Script Test**.

3. Enter

```
// @Open sesame
```

in the Script Text field.

4. Click the Submit button.
   The Rhino JavaScript Debugger appears on Civic Platform Application Server.
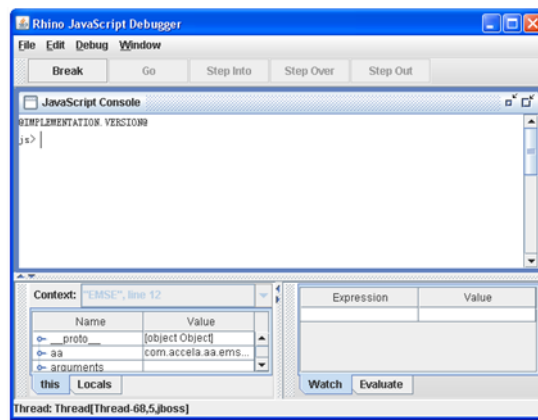
> **Note:**
> The EMSE debug tool only appears on the Civic Platform Application Server, not on any other computers where you can log in to Civic Platform Classic.

5. Debug the scripts that populate the JavaScript Console of the debug tool.

Several kinds of scripts, such as batch job scripts and scripts that EMSE events trigger, can populate the debug tool. For example, when submit an application, you trigger the

```
ApplicationSubmitAfter
```

event and the associated script automatically populates the debug tool.The debugger window looks like the following.
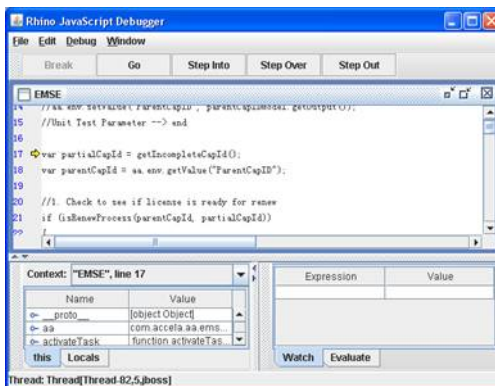
Understanding the Script Output Window

You can use the script output pop-up window to provide additional details about an event and associated script to help locate problems or areas in the script on which you want to focus.

The showDebug variable controls script output according to the following settings:

- 0 or false – no output to the screen or server logs

- 1 or true – generates screen output only

- 2 – outputs to the server log only

- 3 – outputs to the screen and the server log

Figure 29: Script Output Window

The script output window displays the script flow, which includes evaluated criteria and executed actions (yellow section in Script Output with Action).

An action only appears in the script output window if the criterion allows it. To locate an action in the script output window, use your browser to search for the word Action.

Figure 30: Script Output with Action

The script tester writes more and different information to the server.log file than the script output window. If the script output window does not provide enough information to troubleshoot your problem, check the server.log file.

Script Output Error Message shows an error message resulting from a misspelled function, but it does not show where it occurred. Server Logs Error Detail shows the server.log file with the debug output. The server.log file shows the error and where in the standard choice the script stopped executing.

Figure 31: Script Output Error Message

Figure 32: Server Logs Error Detail

📝 **Note:**

In order for the script output window to display showDebug must equal true, 1, 2, or 3 when the master script completes evaluating all the script controls.

If you set the showDebug variable correctly and the script output window does not display, check for the following two situations:

- an error prevents the display

- a script control later in the flow sets the showDebug variable to false.

Setting the showDebug Script Control

You can set the showDebug script control, with true as the criteria (Common showDebug Implementation). In this case, the script output window displays to any user that triggers the associated event, which can confuse users not familiar with the testing process and EMSE.

Table 21: Common showDebug Implementation

#	Value Description
10	true ^ showDebug = 3;
20	// several more lines of script controls

To limit the showDebug function to a specific user, add a criteria to specify that the showDebug script control only applies to a specific user (showDebug for Specified User).

Table 22: showDebug for Specified User

#	Value Description
10	currentUserID == "ADMIN" ^ showDebug = 3; //replace ADMIN with your username
20	// several more lines of script controls

Using the aa.print Function

The aa.print function outputs text to the Script Output window. A script test or a script control can call the aa.print function. When called from a script control, the aa.print function output displays at the bottom of the script output window (aa.print()).

Finished: ApplicationSpecificInfoUpdateAfter, Elapsed Time: 0.047 Seconds

Script **APPLICATIONSPECIFICINFOUPDATEA**

here is aa.print. My capID is 12CAP-00000-0001F

Figure 33: aa.print()

Using Biz Server Logs

You can write your script results to the server.log file on the Civic Platform Biz server.

To write script results to the Biz server

Set the DEBUG Standard Choice with the following settings (DEBUG standard choice):

Standard Choice Item Name	DEBUG
Status:	Enable
Type:	System Switch
Standard Choice Value	ENABLE_DEBUG
Value Desc	YES

Standard Choices Item – Edit

Use this form to set up a Standard Choices Item.
Standard Choices Item Name: DEBUG
Description:
(250 char max)

Status: ⦿ Enable ○ Disable
Type: ⦿ System Switch ○ Share

Standard Choices Value	**Value Desc**
ENABLE_DEBUG	YES

Figure 34: DEBUG standard choice

You can locate the log file on the server running the Civic Platform Biz server in the following location:

```
\\Biz_Server\C$\Accela\av.biz\log\server.log
```

where C$ represents the root drive specification for the Civic Platform installation.

The server.log file represents the log for the current day. Each day at 12:00AM server time the server appends the date to the name of the previous day's log file. The daily history of log files remains on the server until an administrator decides to purge historical log files to free up space on the server hard disk.

If you do not have access to the server, you can request the server administrator deploy a log monitor. See the Accela Community for details of deploying a log monitor: http://community.accela.com/accela_automation/m/aascripts/384.aspx

A log viewing application such as BareTailPro (http://www.baremetalsoft.com/baretailpro/index.php) can enhance the log review process and speed up research and troubleshooting with the biz server logs.

To write specific messages to the biz server log, use the aa.debug function. The aa.debug(string,string) function writes the strings captured in the function call to the server.log file (aa.debug(string,string)).

Standard Choices Value	Value Desc
01	true ^ showDebug = 3; showMessage = true; logDebug("capId has no function");
01aa.print	true ^ aa.print("here is aa.print. My capID is " + capId);
01debug	true ^ aa.debug("here is a debug statement","my capId is " + capId);

```
2012-04-10 14:21:47,936 INFO [STDOUT] -----EMSE Debug Out----- here is a debug statement ; my capId is 12CAP-00000-0001F
```

Figure 35: aa.debug(string,string)

Civic Platform Object Model

This chapter provides a tutorial-like discussion of the Civic Platform object model. The functions that the Civic Platform master scripts provide reference these objects (Master Script Function List).

Related Information

Discussing the Civic Platform Object Model

Understanding Script Return Values

Discussing the Civic Platform Object Model

The Civic Platform object model comprises a hierarchy of objects that organizes access to different parts of Civic Platform. At the root of the tree is the aa object.

Object Model shows that the aa object is at the root, with a few of the objects underneath the aa object listed. Each of the objects listed beneath the aa object is a property of the aa object.
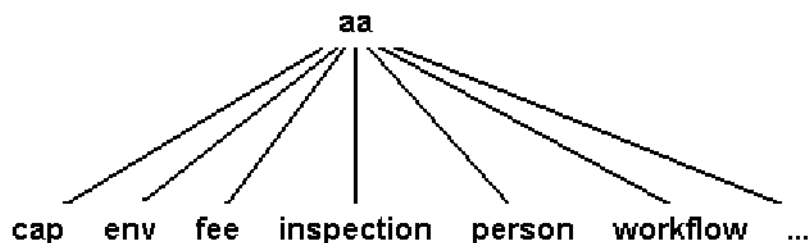
Figure 36: Object Model

From earlier examples, we know that the aa object also has methods.Object Model Root Methods shows two of the methods of the aa object. You can find documentation for all the methods of the aa object in the EMSE Javadocs.
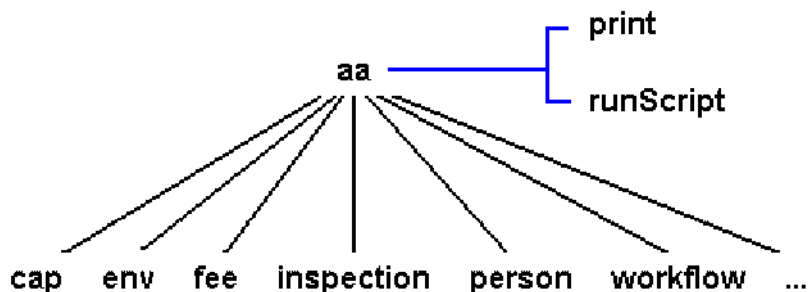
Figure 37: Object Model Root Methods

Notice that each of the objects under the aa object has a name that corresponds to a piece of Civic Platform. Each of these objects beneath aa also provides methods for interacting with (Object Model Object Methods).
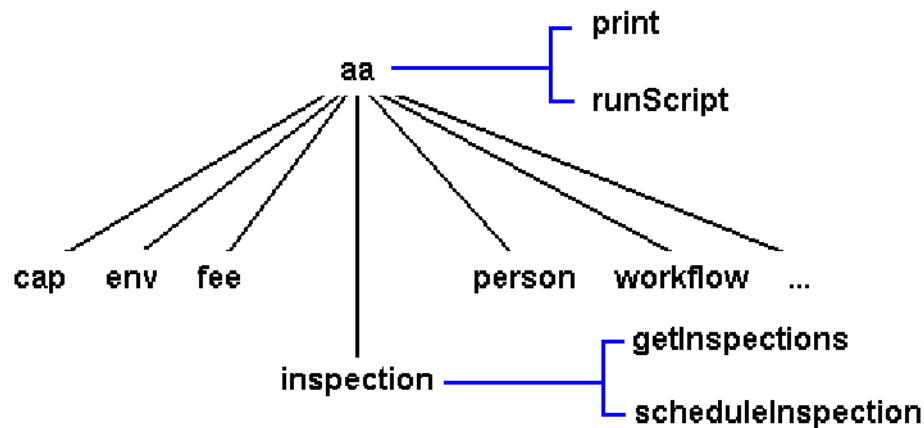
Figure 38: Object Model Object Methods

In this diagram, we can see that the inspection object has some methods that we can use. If we look at the documentation for the getInspections method we find the method definition:

getInspections(CapID capID) returns Result

The method syntax tells us that the name of the method is getInspections and that this method takes one parameter. Two words describe each parameter. The first word tells us the kind of parameter. The second tells us the parameter named. The name helps us to understand how to use the parameter inside the method. The type is "String", "Number", or perhaps the name of an object. In the case of this parameter the type is CapID. We can look at the documentation for the method's parameters and see:

capID – The CapID for the record from which you want to get the array of inspections.

We can also look up the CapID object, and read its description to find out more about it. So now, we know that we need to pass in a CapID object that identifies the record for which we want to get an array of inspections.

If we look at the end of the method definition we see "returns Result". This last part of the method tells us that, when we call this method, we get back a result object. The result object provides an indicator of whether the method succeeded, an error type and error message if the method failed, and the output if the method succeeded.

The getSuccess method returns a Boolean value that is true if the method succeeded and false if it did not. If the getSuccess method returns false you can check the getErrorType and getErrorMessage methods return values to retrieve some information about the error. The getSuccess method returns true you can retrieve the actual output of the method by calling getOutput.

When we look at the documentation for the getInspections method we can see that the getOutput method of the result object returned by the getInspections method returns InspectionItem. The double brackets [ ] tell us that it is an array of InspectionItem objects and not just one InspectionItem object.

Return Value:

Result – Object, see description in this document. The getOutput method of the result object returns InspectionItem[], an array of InspectionItem object. See InspectionItem object description in this document.

At this point, we still have three questions. First, how do we create a CapID object that identifies the record we want? Second, how do we call the method? Third, how do we work with the array of InspectionItem objects returned to us by calling this method?

For the first question, we notice that there is a cap object beneath the aa object. We go to the reference documentation and see that the cap object has this method:

getCapID(String id1, String id2, String id3) returns Result

Return Value:

Result - Object, see description in this document. The getOutput method of the result object returns a CapID object. The getErrorMessage method returns CapNotFound if the method does not find a record that matches the three five digit ids.

This method returns a result object that has the CapID object we need. The method takes three strings that are the three ids for the permit. You can set up your Civic Platform instance to use a custom id, rather than a fifteen digit id, and a method of the cap object allows you to retrieve a CapID object using a custom id. Now we need to know how to call this method. Here is how:

```
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
 myCap = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
```

We can see from this line that we access the cap object as a property of the aa object and then call the getCapID method of the cap object. We pass the three strings, that identify the record we want, to the method . The method returns the result object, which we use to see if the getCapID method succeeded. The retrieved CapID object, by calling the getOutput method of the result object, identifies our record and we assign that object as the value of the myCap variable.

Now we have the value we need as the parameter to pass in to the getInspections method. We can see, from the example of calling a method of the cap object, how to call the getInspections method of the inspection object. We know that we get back an array object from the getInspections method call, so now we know how to write a two line script that retrieves an array of inspections for a particular record. Here is our script so far:

```
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
 myCap = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);

if(myResult.getSuccess()) {
 myInspections = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
```

This script does not display any output yet. At this point, we should note two things. First, use a record id that exists in the Civic Platform instance for which you are writing scripts. Second, use a record that has scheduled, resulted, or cancelled inspections.

What happens if you choose a record that does not exist? If the method does not find a record that matches the id, the result object's getSuccess method returns false, and you need to check the error type and error message.

What happens if the record does not include any scheduled, resulted, or cancelled inspections? You get back a zero length array that means there are no inspections for the record you selected. We come back to these possibilities a bit later, but for now let us assume that we have the right record id, and that the record includes inspections.

Now we need to do something with the array of InspectionItem objects we got back from calling the getInspections. What can we do? Well, let us start by trying to print out some information about the inspections scheduled. We want to print out a few important pieces of information about each inspection.

We go to each element in the array and call some methods on the object stored at that position. This example reminds us of the example use of a 'while' loop. Here is the example again for your review:

```
myArray = new Array();
myArray[0] = "Oranges";
myArray[1] = "Bagels";
myArray[2] = "Spinach";
i=0;
while(i < myArray.length) {
 aa.print(myArray[i]);
 i = i + 1;
}
```

This example approximates what we need. We have an array and we want to loop over its elements. So lets try adding a rough version of this to our script:

```
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
 myCap = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);

if(myResult.getSuccess()) {
 myInspections = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
i=0;
while(i < myInspections.length) {
 // At this point we need to get the inspection and do something with it to
 print.
 i = i + 1;
}
```

Now we have added five more lines to our script that execute a while loop one time for each element in the array. If we have three items in our array, the loop counter has the values 0, 1, and 2. When the loop counter reaches three, the loop stops repeating. Inside the loop, we have a comment as a placeholder for a print function.

Inside the loop, we retrieve the InspectionItem object from the array, that is at the position specified by the loop counter, and we use that object to print out the information. Add the line to retrieve the object:

```
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
 myCap = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);
if(myResult.getSuccess()) {
 myInspections = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
i=0;
while(i < myInspections.length) {
 theItem = myInspections[i];
 // At this point we need to use the object to print some stuff.
```

```
   i = i + 1;
 }
```

After adding a line to the beginning of the loop we now have a variable that contains the InspectionItem object at the current position in the array. Now we just use that object to print out the inspection id number, type, and status. Here is the script:

```
 myResult = aa.cap.getCapID("01BLD", "00000", "00027");
 if(myResult.getSuccess()) {
  myCap = myResult.getOutput();
 } else {
  aa.print(myResult.getErrorMessage());
  aa.abortScript();
 }
 myResult = aa.inspection.getInspections(myCap);
 if(myResult.getSuccess()) {
  myInspections = myResult.getOutput();
 } else {
  aa.print(myResult.getErrorMessage());
  aa.abortScript();
 }
 i=0;
 while(i < myInspections.length) {
  theItem = myInspections[i];
  aa.print(theItem.getIdNumber());
  aa.print(theItem.getInspectionType());
  aa.print(theItem.getInspectionStatus() + "\n");
  i = i + 1;
 }
```

If you run this script with the right record id you receive an output that, depending on the inspections for the record and their status, looks something like this:

```
4238
Trenches
Scheduled
4257
Reinforcing
Approved
4293
Foundation Wall
Approved
```

Now we have a useful script that retrieves some important information about a record. In our script we used three methods of the InspectionItem object: getIdNumber, getInspectionType, and getInspectionStatus. These methods do not take any arguments because their purpose is only to return information about the inspection to your script.

The tenth line of the script shows that we added the special character "\n" at the end of the value that the getInspection status method returned and passed the resulting string to the print method. This special character added an extra blank line in between each inspection's printed values.

Up until this point, we have always used the print method to produce output from our script that we can see, but many of the scripts that you write for Civic Platform do not produce output in this way. You may want to modify a record's workflow or automatically assess a fee when you schedule a new inspection. In other words, the output of your script may modify some data in Civic Platform.

As an example, we are going to check for a problem with the statuses of the inspections of our record, and if a problem exists, we are going to create a smart notice to let staff know. Let us suppose that we do not want to approve a Foundation Wall inspection before there is an approved Trenches inspection. If this scenario happens we want to create a smart notice that informs staff that the record with the record id we were checking has this problem.

So how do we approach this scenario? Well, first we need to know if there is an approved Foundation Wall inspection. If there is, then we need to know if there is an approved Trenches inspection. We already have a script that loops over all the inspections for a record. We can start from there, but instead of printing out information about the inspection, we want to see if the inspection is an approved Foundation Wall inspection. Let us add this check to the script:

```
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
 myCap = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);
if(myResult.getSuccess()) {
 myInspections = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
i=0;
while(i < myInspections.length) {
 theItem = myInspections[i];
 if(theItem.getInspectionType() == "Foundation Wall" &&
 theItem.getInspectionStatus() == "Approved") {
 //Check to see if there is an approved Trenches inspection.
 } i = i + 1;
}
```

When this script executes let us suppose that it finds an inspection that is a Foundation Wall inspection with a status of Approved. When this scenario happens we need to check to see if there is an approved Trenches inspection.

The check for a Trenches inspection requires that we use a second loop inside our main loop, but we can simplify things by using a function. Let us add a function that checks to see if there is a Trenches inspection that is Approved and a condition that uses our new function to do the checking:

```
Function checkForApprovedTrenchesInspection(inspectionItemArray) {
 j = 0;
 while(j < myInspections.length) {
 currentInspection = myInspections[j];
 if(currentInspection.getInspectionType() == "Trenches" &&
 currentInspection.getInspectionStatus() == "Approved") {
 return true;
 }
 j = j + 1;
 }
 return false;
}
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
 myCap = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);
if(myResult.getSuccess()) {
 myInspections = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
```

```
  i=0;
while(i < myInspections.length) {
 theItem = myInspections[i];
 if(theItem.getInspectionType() == "Foundation Wall" &&
 theItem.getInspectionStatus() == "Approved") {
if(checkForApprovedTrenchesInspection(myInspections) == false) {
 // If we reach this line we have confirmed that the record has a problem.
 }
 }
 i = i + 1;
}
```

Our function looks very similar to the previously written part of our script. We have placed the function at the top of the script, although you could also put it at the bottom as a matter of preference. We have tried to give our function a meaningful name that tells us what it does. We could also put a comment before the function to explain to other people reading our script what the function does.

The function accepts one parameter, an inspection array. We named the parameter inspectionItemArray to remind ourselves of what type of value we need to pass in when calling the function. The function loops over the array passed in and checks to see if each of the inspections meets the condition that it is a Trenches inspection with Approved status. As soon as the function finds an inspection it returns the value true.

When a return statement appears in the middle of a function like this, the function stops what it is doing and immediately returns the specified value; it does not wait for the loop to finish. If the loop goes all the way through the inspections for the record and does not find a matching inspection, the loop exits and the next command after the loop executes. The command after the loop is "return false", so if the function gets through the loop without finding a matching inspection, the function returns false.

The condition we added to the middle of the previously written loop uses the new function to check for an Approved Trenches inspection. If the function does not find an inspection, we know that the record has inspection statuses inconsistent with how we want to run our agency and we need to do something.

Let us suppose that the inspection matches the conditions we have set up so far. In this case, we need to replace the comment line in our script with a command to insert a smart notice with the information that we need. The smartNotice object has this method:

addNotice(String id1, String id2, String id3, String activityType, String activityComment) **returns** null

So when we confirm that the record has the problem, we need to call this method to create the new smart notice. After adding this method call to the script here is what we get:

```
function checkForApprovedTrenchesInspection(inspectionItemArray) {
 j = 0;
 while(j < myInspections.length) {
 currentInspection = myInspections[j];
 if(currentInspection.getInspectionType() == "Trenches" &&
 currentInspection.getInspectionStatus() == "Approved") {
 return true;
 }
 j = j + 1;
 }
 return false;
}
myResult = aa.cap.getCapID("01BLD", "00000", "00027");
if(myResult.getSuccess()) {
 myCap = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);
if(myResult.getSuccess()) {
 myInspections = myResult.getOutput();
} else {
```

```
  aa.print(myResult.getErrorMessage());
  aa.abortScript();
}
i=0;
while(i < myInspections.length) {
 theItem = myInspections[i];
 if(theItem.getInspectionType() == "Foundation Wall" &&
 theItem.getInspectionStatus() == "Approved") {
if(checkForApprovedTrenchesInspection(myInspections) == false) {
 aa.smartNotice.addNotice("01BLD", "00000", "00027", "Inspection Problem",
 "Application 01BLD 00000 00027 has an approved Foundation Wall inspection"
 +
 "but no approved Trenches inspection.");
 }
 }
 i = i + 1;
}
```

So now, we have a script that checks a record to see if it meets a certain criteria, and takes action by inserting a new smart notice if the record does meet the criteria.

The list of Civic Platform events includes an event called InspectionResultSubmitAfter that is a good place to run our script and check the application. However, as our current script only checks the record 01BLD-00000-00027. We need to modify our script to that it uses the input parameters from the event to dynamically determine which application to check.

The documentation for the InspectionResultSubmitAfter event shows three parameters that can tell us which record to check:

```
IN: PermitId1
IN: PermitId2
IN: PermitId3
```

We need to use the getValue method of the env object to retrieve parameters passed in to our script from an event. We add the following three lines at the top of our script to retrieve the permit id values:

```
myId1 = aa.env.getValue("PermitId1");
myId2 = aa.env.getValue("PermitId2");
myId3 = aa.env.getValue("PermitId3");
```

After adding these lines at the beginning of the script, we use the three values we retrieved in place of the unchanging strings we passed as parameters to getCapID. We also use these values to dynamically create the message for the smart notice. Here is the script:

```
function checkForApprovedTrenchesInspection(inspectionItemArray) {
 j = 0;
 while(j < myInspections.length) {
 currentInspection = myInspections[j];
 if(currentInspection.getInspectionType() == "Trenches" &&
 currentInspection.getInspectionStatus() == "Approved") {
 return true;
 }
 j = j + 1;
 }
 return false;
}
myId1 = aa.env.getValue("PermitId1");
myId2 = aa.env.getValue("PermitId2");
myId3 = aa.env.getValue("PermitId3");
myResult = aa.cap.getCapID(myId1, myId2, myId3);
if(myResult.getSuccess()) {
 myCap = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
```

```
  aa.abortScript();
 }
myResult = aa.inspection.getInspections(myCap);
if(myResult.getSuccess()) {
 myInspections = myResult.getOutput();
} else {
 aa.print(myResult.getErrorMessage());
 aa.abortScript();
}
i=0;
while(i < myInspections.length) {
 theItem = myInspections[i];
 if(theItem.getInspectionType() == "Foundation Wall" &&
 theItem.getInspectionStatus() == "Approved") {
if(checkForApprovedTrenchesInspection(myInspections) == false) {
 aa.smartNotice.addNotice(myId1, myId2, myId3, "Inspection Problem",
 "Application " + myId1 + " " + myId2 + " " + myId3 +
 " has an approved Foundation Wall inspection" +
 " but no approved Trenches inspection.");
 }
 }
 i = i + 1;
}
```

You can now associate the script with the InspectionResultSubmitAfter event. However, there is one more thing to do.

Currently, we use aa.print to send messages to the user when something goes wrong. While aa.print works with the Script Test page, use the environment object to send messages back to the user when you attach the script to an event. Here is the final script with the aa.print statements, replaced with the appropriate statements for informing the user with a message:

```
function checkForApprovedTrenchesInspection(inspectionItemArray) {
 j = 0;
 while(j < myInspections.length) {
 currentInspection = myInspections[j];
 if(currentInspection.getInspectionType() == "Trenches" &&
 currentInspection.getInspectionStatus() == "Approved") {
 return true;
 }
 j = j + 1;
 }
 return false;
}
myId1 = aa.env.getValue("PermitId1");
myId2 = aa.env.getValue("PermitId2");
myId3 = aa.env.getValue("PermitId3");
myResult = aa.cap.getCapID(myId1, myId2, myId3);
if(myResult.getSuccess()) {
 myCap = myResult.getOutput();
} else {
 aa.env.setValue("ScriptReturnMessage", myResult.getErrorMessage());
 aa.abortScript();
}
myResult = aa.inspection.getInspections(myCap);
if(myResult.getSuccess()) {
 myInspections = myResult.getOutput();
} else {
 aa.env.setValue(""ScriptReturnMessage", myResult.getErrorMessage());
 aa.abortScript();
}
i=0;
while(i < myInspections.length) {
 theItem = myInspections[i];
```

```
  if(theItem.getInspectionType() == "Foundation Wall" &&
  theItem.getInspectionStatus() == "Approved") {
 if(checkForApprovedTrenchesInspection(myInspections) == false) {
  aa.smartNotice.addNotice(myId1, myId2, myId3, "Inspection Problem",
  "Application " + myId1 + " " + myId2 + " " + myId3 +
  " has an approved Foundation Wall inspection" +
  " but no approved Trenches inspection.");
  }
  }
  i = i + 1;
 }
```

Before you deploy a script like this to an environment where real users use Civic Platform, test the script thoroughly to make sure that it works as expected. Use a test environment where any mistakes do not affect your production data.

To test a script, go in to the Event Manager pages and associate the script with the event, then try to exercise different parts of the script. For example, try doing several different kinds of inspection results that do not insert a smart notice. Then try some inspection results that do insert a smart notice. With proper testing you can be much more sure that the script works, before you deploy it in your production environment.

Understanding Script Return Values

When execution of a script completes, the script sends the values stored in the scripts environment object back to Civic Platform. Some events have documented special values in the environment, which you can set to send information to the Civic Platform interface. The event documentation provides the names for these values and the expected input from your script.

In addition to the environment return values related specifically to the event there are some return values that are always available for you to set. You do not need to provide these parameters a value unless you want something specific to happen.

Note that there are both before and after events in Civic Platform. An event with a name that ends in "Before" means the event takes place before the user action updates the database. An event with a name that ends in "After" means the event takes place after the user action updates the database.

Because script return values can stop Civic Platform from continuing a user action, if you associate a script with an after event, the user action completes before the script has a chance to stop it. If you want to be able to cancel a user action, use a before event.

Topics:

- ScriptReturnCode

- ScriptReturnMessage

- ScriptReturnRedirection

ScriptReturnCode

This return value is a numeric value (ScriptReturnCode Values) which allows the script to choose one of several actions to occur after the script finishes.

Table 23: ScriptReturnCode Values

Value	Action
0	Proceed as normal.
1	Request Civic Platform to stop the user action and return to the previous page.
2	Request Civic Platform to stop the user action and return to the main menu.
3	Request Civic Platform to stop the user action and proceed to the page designated by the ScriptReturnRedirection value.
4	Request Civic Platform to stop the user action and log user out.

ScriptReturnMessage

When this return parameter has a value, it displays to the user as a message when the Civic Platform user request, of which the script is a part, completes. You can use this function to send informative messages, or explanations of why an error occurred.

You can use HTML characters to format the message. The text of the returned message displays to the user in a popup window.

When you set the ScriptReturnCode parameter to something other than zero, you can set a value for ScriptReturnMessage to explain to the user why they are redirected from the normal flow of the Civic Platform interface.

ScriptReturnRedirection

When you set the ScriptReturnCode parameter to three, and you set a value for ScriptReturnCode, the script sets the URL of the browser to the value of the ScriptReturnRedirection parameter.

Master Script Function List

This section provides reference information for the master script functions.

Conventions

- Unless otherwise stated, all function names and parameter values are case sensitive.

- Enter function parameters in the order listed.

> 📝 **Note:**
>
> The name of a function parameter is for descriptive purposes only. The name of the function parameters in this chapter can differ from the name of the corresponding function parameter in the UniversalMasterScript file.

- For the string data type, enclose the parameter value in double-quotes.

- Subscripts 1 and n in parameter names (e.g., wfTask1 … wfTaskn) indicate that you can add between one and any number of such parameters, each in double-quotes and separated by commas.

- This reference shows Boolean values as true or false.

- This reference does not document internal functions in the master script files.

CapIDModel Type

The master script functions use the CapIDModel type for the capID parameter. Civic Platform Object Model provides additional details on the capID parameter. The EMSE Javadocs contain details on the CapIDModel class. The com.accela.aa.aamain.cap package in the EMSE Javadocs contains the CapIDModel class and defines the constructor for this class as follows.

Parameter	Type
serviceProviderCode	string
ID1	string
ID2	string
ID3	string
customID	string
trackingID	long

activateTask

Makes workflow task wfstr active and not completed, so that users can edit wfstr.

Version

1.3

Parameters

Parameter	Type	Description
wfstr	string	Name of task to activate.
wfRelationSeqId (optional)	number: long	Relation sequence ID of workflow process to which wfstr belongs.

Notes

If workflow uses sub-processes that contains duplicate workflow task names, use parameter wfRelationSeqId to specify the process or subprocess whose wfstr you want to activate. You can find the value of wfRelationSeqId by querying workflow tables (e.g. GPROCESS.RELATION_SEQ_ID)

addAddressCondition

Adds a condition to the specified reference address. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

Version

2.0

Parameters

Parameter	Type	Description
addNum	long	Reference address number or null.
cType	string	Type of condition (from admin->condition->condition type).
cStatus	string	Status (from admin->condition->condition status).
cDesc	string	Description of the condition.
cComment	string	Condition comment.
cImpact	string	Must be Lock, Hold, Notice, Required, or "".

Notes

If addNum is null, the function adds the condition to all reference addresses associated with the current record.

addAddressStdCondition

Adds a standard condition to the specified reference address. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

Version

2.0

Parameters

Parameter	Type	Description
addNum	long	Reference address number or null.
cType	string	Type of the standard condition.
cDesc	string	Description of the standard condition.
cStatus (optional)	string	Condition status.

Notes

If addNum is null, the function adds the condition to all reference addresses associated with the current record.

addAllFees

Adds all fees within a fee schedule to the record. Optionally flags the fees for automatic invoicing by the script.

Version

1.3

Parameters

Parameter	Type	Description
fsched	string	Fee schedule to be added.
fperiod	string	Fee period to be used.
fqty	integer	Quantity to be entered.
finvoice	string	Flag for invoicing ("Y" or "N").

addAppCondition

Adds the condition to the record. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

Version

2.0

Parameters

Parameter	Type	Description
cType	string	Type of condition (from admin->condition->condition type).
cStatus	string	Status (from admin->condition->condition status).
cDesc	string	Description of the condition.
cComment	string	Condition comment.
cImpact	string	Must be Lock, Hold, Notice, Required, or "".

addASITable

Populates the ASI table with values.

Version

1.6

Parameters

Parameter	Type	Description
tableName	string	Name of the ASI table to add to the record.
tableValueArray	array of associative arrays	Values to populate the table.
capID (optional)	CapIDModel	Record to add table to.

Notes

tableValueArray is an array of arrays. Each array object within tableValueArray must contain an associative index for each column in the target table.

Example

```
masterArray = new Array();
elementArray = new Array();
elementArray["Table Column 1"] = "Row 1, column 1 Value";
elementArray["Table Column 2"] = "Row 1, column 2 Value";
masterArray.push(elementArray);
addASITable("table name",masterArray);
```

This example populates the 2-column table with one row.

addASITable4ACAPageFlow

Used by page flow scripts to add rows to an ASIT table. You can use this function to dynamically populate an ASIT based on data from earlier pages.

Version

2.0

Parameters

Parameter	Type	Description
DestinationTable GroupModel	appSpecificTable GroupModel	ASIT object from the current record in Citizen Access.
tableName	string	Destination table name.
tableValueArray	associative array	Array of ASI table values to add.

Example

The following example adds a row to the TBL-DOCREQ table.

```
var cap = aa.env.getValue("CapModel");
var conditionTable = new Array();
var c = new Array();
c["Document Type"] = new asiTableValObj("Document Type","Document","Y");
c["Name"] = new asiTableValObj("Name","Dangerous / Vicious Dog
 Waiver","Y");
conditionTable.push(c);
asit = cap.getAppSpecificTableGroupModel();
new_asit = addASITable4ACAPageFlow(asit,"TBL-DOCREQ", conditionTable);
```

addContactStdCondition

Adds a standard condition to the specified reference contact. If contactSeqNum is null, the function adds the condition to all reference contacts associated with the current record. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

Version

2.0

Parameters

Parameter	Type	Description
contactSeqNum	long	Reference contact sequence number or null.
cType	string	Type of the standard condition.
cDesc	string	Description of the standard condition.
cStatus (optional)	string	Condition status.

addCustomFee

Adds a custom fee feecode to the record, from the fee schedule feesched with fee period feeperiod.

Version

1.5

Parameters

Parameter	Type	Description
feecode	string	Fee code to be added.
feesched	string	Fee schedule of the fee to be added.
feeDescr	string	A description of the custom fee item.
feePeriod	string	Fee period to be used.
feeAm	double	Fee quantity.
feeACC	string	Fee account code 1.
capID (optional)	CapIDModel	Record to add fee to.

Returns

Returns the Fee Sequence number of the fee added.

The fee period feeperiod must be a valid fee period for feecode in feesched, or this function throws an error.

See also

addAllFees

addFee

Adds a single fee fcode to the record, from the fee schedule fsched with fee period fperiod and quantity of fqty.

Version

1.3

Parameters

Parameter	Type	Description
fcode	string	Fee code to add.
fsched	string	Fee schedule of the fee to add.
fperiod	string	Fee period to use.
fqty	integer	Quantity to enter.
finvoice	string	Flag for invoicing ("Y" or "N").
capID (optional)	CapIDModel	Record to add fee to.

Returns

The fee period fperiod must be a valid fee period for fcode in fsched, or this function throws an error.

Notes

If *finvoice* is Y, the function invoices the fee. If finvoice is N, the function assesses the fee but does not invoice the fee.

If you use the capID optional parameter, the function updates record capID. If you do not use the capID parameter, the function updates the current record.

getApplication( ), getParent( ), createChild() functions each returns a record ID object that you can use in the capID parameter.

See Also

addAllFees

addFeeWithExtraData

Identical to the addFee function, but also allows you to populate the comment and user defined fields.

Version

1.6

Parameters

Parameter	Type	Description
fcode	string	Fee code to be added.
fsched	string	Fee schedule of the fee to be added.
fperiod	string	Fee period to be used.
fqty	integer	Quantity to be entered.
finvoice	string	Flag for invoicing ("Y" or "N").
feeCap	CapIDModel	Record ID object.
feeComment	string	Comment field on the fee item.
UDF1	string	Value for user defined field on fee item.
UDF2	string	Value for user defined field on fee item.

addLicenseCondition

Adds the condition (cType, cStatus, cDesc, cComment, cImpact) to the reference record for each licensed professional on the record. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

Version

2.0

Parameters

Parameter	Type	Description
cType	string	Condition type.
cStatus	string	Condition status.
cDesc	string	Condition (30 characters maximum).
cComment	string	Condition comment (free text).
cImpact	string	Condition severity: Lock, Hold, Notice, Required, or "".
stateLicNum (optional)	string	State license number.

Notes

If you use the stateLicNum parameter, the function adds the condition to the licensed professional reference record whose State License Number is stateLicNum. This licensed professional may not be on the current record.

addLicenseStdCondition

Adds a standard condition to the specified reference licensed professional. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

Version

2.0

Parameters

Parameter	Type	Description
licSeqNum	long	Reference license sequence number or null.
cType	string	Type of the standard condition.
cDesc	string	Description of the standard condition.
cStatus (optional)	string	Condition status.

Notes

If licSeqNum is null, the function adds the condition to all reference licensed professionals associated with the current record.

addLookup

Adds a lookup entry to an existing standard choices item. Adds a new value called stdValue with the value description of stdDesc to standard choices item name stdChoice.

Version

1.3

Parameters

Parameter	Type	Description
stdChoice	string	Standard choices item name.
stdValue	string	Standard choices value.
stdDesc	string	Standard choices value description.

Notes

If the standard choices item stdChoice already has a value entry called stdValue, the function does not add or update stdValue. This function does not create the standard choices item stdChoice if it does not exist.

addParcelAndOwnerFromRefAddress

Copies the associated parcel and owner from a reference address to the specified record. If you do not specify a record, the function uses the current record as the target.

Version

1.6

Parameters

Parameter	Type	Description
refAddress	long	Reference address number to copy data from.
capID (optional)	CapIDModel	Target record for parcel and owner.

addParcelCondition

Adds a condition to the reference parcel whose number is parcelNum. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

Version

2.0

Parameters

Parameter	Type	Description
parcelNum	string	Parcel number to add the condition to. If null, the function adds the condition to all parcels on the record.
cType	string	Condition type.
cStatus	string	Condition status.
cDesc	string	Condition name.
cComment	string	Condition comment.
cImpact	string	Condition severity.

Notes

The condition's Type, Condition (description), Status, Severity and Comment corresponds to cType, cDesc, cStatus, cImpact, and cComment, respectively. The condition's Apply and Effective dates equal the current date. The condition's Applied By and Action By staff names equal the current user's name.

If you use null for the parcelNum parameter, the function adds the condition to all parcels on the current record.

addParcelDistrict

Adds a district to the parcel on a record.

Version

1.6

Parameters

Parameter	Type	Description
parcelNum	string	Parcel number that district adds to.
districtValue	string	Value of district entry to add.

Notes

Does not edit reference parcel data.

If parcelNum is null, the function adds the district to all parcels on the current record.

addParent

Adds the current record as a hierarchal child to the parent record parentAppNum.
Version
1.3
Parameters

Parameter	Type	Description
parentAppNum	string	App number (B1_ALT_ID) of the record to be parent of the current record.

addrAddCondition

Adds a condition (pType, pStatus, pDesc, pComment, pImpact) to the address on the record whose address number is pAddrNum.

Version

1.4

Parameters

Parameter	Type	Description
pAddrNum	number	Address number. Use null for all addresses on record.
pType	string	Condition type.
pStatus	string	Condition status.
pDesc	string	Condition name.
pComment	string	Condition comment.
pImpact	string	Condition severity.
pAllowDup	string	Determines whether to add duplicate condition to address.

Returns

True if the function adds the condition, false otherwise.

Notes

If pAddrNum is null, adds the condition to all the addresses on the record. If pAllowDup is N, the function does not add a condition to the address if the same condition is already on the address. If pAllowDup is Y, the function adds the condition to the address even if this action duplicates the condition on the address.

The function adds the condition to the reference Address record. The function adds the condition only if you use the Search button on the record's Address screen or use the Get Associated Object button on the record's parcel screen to add the address to the record. If you enter the address manually, the function does not add the condition.

The pAddrNum value comes from B3ADDRES.

L1_ADDRESS_NBR, not B3ADDRES.B1_ADDRESS_NBR.

addReferenceContactByName

Adds a reference contact to the current record, based on the name of the contact. The function only adds the first matching contact.

Version

1.6

Parameters

Parameter	Type	Description
vFirst	string	First name of reference contact.
vMiddle	string	Middle name of reference contact.
vLast	string	Last name of reference contact.

addressExistsOnCap

Returns true if there is at least one address on the record.

Version

1.6

Parameters

Parameter	Type	Description
capID (optional)	CapIDModel	Record ID to check.

addStdCondition

Retrieves all standard conditions named cDesc whose type is cType and adds them to the record. If a standard condition is associated with an ASI group (condition template), the method adds the condition with the template fields and tables. You can call the method to add duplicate conditions to a record.

Version

2.0

Parameters

Parameter	Type	Description
cType	string	Condition type.
cDesc	string	Condition name.
capID (optional)	CapIDModel	Record to add condition to.

Notes

The function assigns the following values to the condition:

- Status = Applied

- Applied By = current user

- Action By = current user

- Apply Date = current date

- Effective Date = current date

- Expiration Date = blank

You can only use the function with Civic Platform 6.4 and later.

addTask

Dynamically adds a task.
Version
2.0
Parameters

Parameter	Type	Description
sourceTaskName	string	Name of the task to replicate.
newTaskName	string	Name of the new task.
insertTaskType	char	Type of task to add (P for parallel or N for next).
recordId (optional)	CapIdModel	Record to which to add the task.

Notes

The function uses the source task for all task information such as assignment and statuses. If insertTaskType equals N, the function adds the task to the end of the workflow in series.

addTimeAccountingRecord

Adds a time accounting entry that associates with a record.

Version

2.0

Parameters

Parameter	Type	Description
taskUser	string	User ID of the Civic Platform user.
taGroup	string	Group of the time accounting entry.
taType	string	Type of the time accounting entry.
dateLogged	string	Date of the time accounting entry.
hoursSpent	string	Number of hours for the entry.
itemCap	CapIdModel	Record to associate to the entry.
billableBool	boolean	True to set the billable flag, otherwise false.

Example

```
capID = aa.cap.getCapID("11CAP-00000-0000D").getOutput()
```

```
addTimeAccountingRecord("BSMITH","Actual","Inspection",
     "07/28/2011","1.1",capID,true);
```

addTimeAccountingRecordToWorkflow

Adds a time accounting entry associated with a workflow task on a record.

Version

2.0

Parameters

Parameter	Type	Description
taskUser	string	User ID of the Civic Platform user.
taGroup	string	Group of the time accounting entry.
taType	string	Type of the time accounting entry.
dateLogged	string	Date of the time accounting entry.
hoursSpent	string	Number of hours for the entry.
itemCap	CapIDModel	Record to associate to the entry.
taskName	string	Name of the task to associate with the entry.
processName	string	Name of the workflow process that contains the task.
billableBool	boolean	True to set the billable flag, otherwise false.

Example

```
capID = aa.cap.getCapID("11CAP-00000-0000D").getOutput()
```

```
addTimeAccountingRecordToWorkflow("BSMITH","Actual","Inspection",
    "07/28/2011","1.1",capID,"Inspection","BLD_MAIN",true);
```

addToASITable

Adds one row of values (tableValues) to the application specific info (ASI) table called tableName.

Version

1.4

Parameters

Parameter	Type	Description
tableName	string	The application specific information table name.
tableValues	array of strings	Values for a single table row, as an associative array of strings.
capID (optional)	CapIDModel	Record ID object for record.

Notes

The tableValues parameter must be an associative array of string values, where each element name is a column name in the ASI table tableName, and the element stores the column value. If you use the capID parameter, the function adds tableValues to tableName in the record whose record ID object is capID.

The parameter tableValues does not have to contain all the columns in the ASI table tableName. The ASI table tableName must already exist on the record.

allTasksComplete

Version

1.3

Parameters

Parameter	Type	Description
stask	string	Process name of workflow to check.
igTask1 … igTaskn (optional)	string	Names of tasks to ignore. Enter one or more task name parameters. Case sensitive.

Returns

Returns true if all tasks (excluding tasks in optional igTask1… igTaskn list) in workflow process / subprocess stask are complete. Returns false if any task is incomplete.

Notes

stask is R1_PROCESS_CODE in the GPROCESS and SPROCESS tables.

Examples

To determine if all tasks in workflow BLDG are completed:

```
allTasksComplete("BLDG")
```

To determine if all tasks in workflow BLDG are complete, except for the Optional Review task and Closure task:

```
allTasksComplete("BLDG","Optional Review", "Closure")
```

appHasCondition

Version

1.4

Parameters

Parameter	Type	Description
pType	string	Condition type.
pStatus	string	Condition status.
pDesc	string	Condition name.
pImpact	string	Condition severity.

Returns

Returns true if the record has a record condition whose type is pType, name is pDesc, status is pStatus, and severity is pImpact; otherwise, returns false.

Notes

Use null in place of any parameter if you do not want to filter by that item. For example, to check if the record has any condition at all, use appHasCondition(null, null, null, null).

applyPayments

On the current record (capID) this function takes any unapplied payments and distributes them to any invoiced fee items.

Version

2.0

Parameters

None

Notes

The function loops through all fee items and applies the payments until all funds are applied, or no more unpaid fee items remain.

appMatch

Version

1.3

Parameters

Parameter	Type	Description
ats	string	Four level record type. Must contain 3 slash (/) characters. Case sensitive. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.
capID (optional)	CapIDModel	Record to check.

Returns

Returns true if ats matches the current record's record type, false if it does not.

Notes

Compares the current record type to ats. You can use the asterisk (*) as a wildcard to match all entries for a given level. For example: appMatch("Building/*/Sign/*/*") evaluates to True for record type Building/Commercial/Sign/Billboard as well as Building/Residential/Sign/Garage Sale.

ats must contain 3 slash characters (/). Do not add spaces immediate before or after the slash (/).

appNameIsUnique

Version

1.4

Parameters

Parameter	Type	Description
gaGroup	string	Record group (the 1st level of record type).
gaType	string	Record type (the 2nd level of record type).
gaName	string	Record name to test.

Returns

Returns true if none of the other records, whose app type begins with gaGroup / gaType, used the record name gaName. Returns false if gaName is not unique.

assignCap

Assigns the staff whose user ID is assignId to the current record. Also assigns the user's department.

Version

1.5

Parameters

Parameter	Type	Description
assignId	string	User ID of the user to whom to assign the record.
capID (optional)	CapIDModel	Record ID to which to assign the user.

Notes

If you use the optional parameter capID, the function assigns the staff and department to the record capID instead.

assignInspection

Assigns the inspector whose user ID is iName to the inspection whose sequence number is iNumber.

Version

1.4

Parameters

Parameter	Type	Description
iNumber	number	Inspection sequence number.
iName	string	Inspector's user ID.
capID (optional)	CapIDModel	Record ID to which to assign the inspector.

Notes

The inspection must already be scheduled on the record.

assignTask

Assigns the staff whose user ID is username to workflow task wfstr.

Version

1.3

Parameters

Parameter	Type	Description
wfstr	string	Workflow task to which to assign a user.
username	string	User ID of the user to whom to assign the task. Case sensitive.
wfProcess (optional)	string	Process name of workflow for wfstr. Case sensitive.

Notes

The function does not create a workflow history for the record.

If record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to activate.

wfProcess is R1_PROCESS_CODE in the GPROCESS and SPROCESS tables. username and wfProcess are normally in uppercase.

autoAssignInspection

Uses the automatic inspection assignment function to assign the specified inspection.

Version

1.6

Parameters

Parameter	Type	Description
iNumber	long	Sequence number for the inspection to assign.

branch

Executes the standard choice script control whose name is iNumber as a sub-control.

Version

1.3

Parameters

Parameter	Type	Description
stdChoice	string	Standard choices item namestring. Case sensitive.

Notes

The script stdChoice must contain only valid criteria/action pairs sequentially numbered.

Example

```
branch("Inspection:Update Expiration")
```

branchTask

Updates the workflow task wfstr as follows

- Status = wfstat

- Status Date = current date

- Status Comment = wfcomment

- Action By = current user

Version

1.3

Parameters

Parameter	Type	Description
wfstr	string	Workflow task name.
wfstat	string	Status.
wfcomment	string	Comment.
wfnote	string	Note to add to the workflow task.
wfProcess (optional)	string	ID (R1_PROCESS_CODE) for the process that the task belongs to. Required for multi-level workflows.

Notes

The function closes the task wfstr and the workflow proceeds to the branch task.

If record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to edit.

capHasExpiredLicProf

Version

1.4

Parameters

Parameter	Type	Description
pDateType	string	Expiration date to check. Options (use one): EXPIRE, INSURANCE, BUSINESS.
pLicType (optional)	string	License type.
pCapId (optional)	CapIDModel	Record ID object of record. If null, the function applies to the current record.

Returns

Returns true if any licensed professional on the record has expired; otherwise, returns false.

Notes

Checks for expiration by retrieving the licensed professional reference record having the same license # and checking the expiration date specified by pDateType. If the expiration date is on or before the current date, the script returns true. Skips disabled licensed professionals.

Use parameter pLicType to check a specific license type. Use parameter pCapId to check licensed professionals on a record other than the current record.

dateType	Expiration Date Field Checked
EXPIRE	License Expiration Date
INSURANCE	Insurance Expiration Date
BUSINESS	Business License Expiration Date

capIdsFilterByFileDate

Searches though the records in pCapIdArray and returns only records whose file date is between pStartDate and pEndDate, as an array of capId (CapIDModel) objects

Version

1.4

Parameters

Parameter	Type	Description
pCapIdArray	array of CapIDModel objects	Array of record ID (CapIDModel) objects to filter.
pStartDate	string	Start date of the file date range, in MM/DD/YYYY format.
pEndDate	string	End date of the file date range, in MM/DD/YYYY format.

Notes

To find the number of records returned, store the return value to a variable and use the length property to find the number of records in the array.

Example

```
capArray = capIdsFilterByFileDate(myCapArray, "01/01/2006", "12/31/2006");
 capCount = capArray.length;
```

capIdsGetByAddr

Returns records that have the same property address as the current record, as an array of capId (CapIDModel) objects.

Version

1.4

Parameters

None

Returns

If the current record has no property address, returns false.

Notes

The function matches addresses based on these fields:

- House Nbr Start

- Street Direction

- Street Name

- Street Suffix

- Zip

You can use this function with all events except ApplicationSubmitBefore. The records returned include the current record. If the current record has more than one property address, the function uses the first address to match.

To find the number of records returned, store the return value to a variable and use the length property to find the number of records in the array.

Example

```
capArray = capIdsGetByAddr(); logDebug("Number of CAPs: " +
  capArray.length);
```

capIdsGetByParcel

Returns records that have the same parcel as the current record, as an array of capId (CapIDModel) objects.

Version

1.4

Parameters

Parameter	Type	Description
pParcelNum (optional)	string	Parcel number to search for. If null or omitted, the function uses the first parcel number on the current record.

Returns

If the current record has no parcel, returns false.

Notes

The records returned include the current record.

To find the number of records returned, store the return value to a variable and use the length property to find the number of records in the array.

Example

```
capArray = capIdsGetByParcel(); logDebug("Number of CAPs: " +
  capArray.length);
```

capSet

Version

2.0

Parameters

Parameter	Type	Description
desiredSetId	string	The ID of the set to create or operate on by the capSet object.

Notes

capSet is a helper object that assists in managing Civic Platform Sets of records. If the desiredSetId already exists as a Set, it loads automatically. If the desiredSetId does not exist, function creates it as an empty set.

Methods

refresh()	The capSet object reloads and all properties refresh.
add(capId)	Adds the supplied capId to the set.
remove(capId)	Removes the supplied capId from the set.
update()	The header information about the set updates to the current values. This header information includes the set name and set comment.

Properties

id	The Id of the set.
name	The name of the set.
comment	The set comment.
size	The number of records in the set.
empty	True if the set has no members.
members	An array or CapIDModel objects representing the membership of the set.

checkCapForLicensedProfessionalType

Returns true if a licensed professional of the type exists on the current record.

Version

1.6

Parameters

Parameter	Type	Description
licProfType	string	Licensed professional type to check for.

checkInspectionResult

Version

1.3

Parameters

Parameter	Type	Description
insp2Check	string	Inspection to check. Case sensitive.
insp2Result	string	Inspection result (or status) to look for. Case sensitive.

Returns

Returns true if the inspection insp2Check has the result of insp2Result, or false if it does not.

Notes

You can use Scheduled as the value for the insp2Resultin parameter to check if inspection insp2Check is scheduled (not yet resulted).

childGetByCapType

Searches through all child records and returns the record ID object for the first child record whose record type matches pCapType.

Version

1.4

Parameters

Parameter	Type	Description
pCapType	string	Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.
pParentCapId (optional)	CapIDModel	Record ID object for parent record. Use null if you use the skipChildCapId parameter.
skipChildCapId (optional)	CapIDModel	Record ID object of child record to skip.

Notes

If you use the pParentCapId parameter, the function searches child records of the record whose record ID object is pParentCapId. If you use the skipChildCapId parameter, the function skips over any child record whose record ID object is skipChildCapId.

To find the sibling of the current record, use the function getParent() as the parentCapId parameter and capId as the skipChildCapId parameter.

Example

```
siblingCapId = childGetByCapType("*/*/*/*", getParent(), capId)
```

See also

getChildren

closeCap

Version

2.0

Parameters

Parameter	Type	Description
userId	string	ID of user who closes the record.
capId (optional)	CapIDModel	Record to perform action on.

Notes

Sets the **Closed Date** value to the current date and the **Close by Staff** field to the ID of the user who closes the record.

closeSubWorkflow

A function that is useful when working with sub-processes.

Version

1.6

Parameters

Parameter	Type	Description
thisProcessID	long value	ID of the process to check.
wfStat	string	Status to use when closing the parent task.
capId (optional)	CapIDModel	Record to perform action on.

Notes

Checks all the tasks in the subprocess for completeness. If all tasks are complete, the function closes the parent task with the specified status.

Example

```
closeSubWorkflow(wfProcessID,"Completed");
```

closeTask

Updates the workflow task wfstr as follows:

- Status = wfstat

- Status Date = current date

- Status Comment = wfcomment

- Action By = current user

Version

1.3

Parameters

Parameter	Type	Description
wfstr	string	Workflow task name.
wfstat	string	Status to update.
wfcomment	string	Comment to add.
wfnote	string	Note to add to the workflow task.
wfProcess (optional)	string	ID (R1_PROCESS_CODE) for the process that the task belongs to. Required for multi-level workflows.

Notes

Closes the task wfstr and promotes the workflow to the next task, even if wfstat loops or branches. If workflow needs to loop or branch, use loopTask or branchTask functions.

If record's workflow contains duplicate wfstr tasks, use wfProcess parameter to specify the process or subprocess whose wfstr to edit.

This old name for this function is closeWorkflow2.

comment

You can use this function to display messages to the user, as well as variables to aid in debugging issues.

Version

1.3

Parameters

Parameter	Type	Description
cstr	string	Comment to display.

Notes

Use logMessage and logDebug functions instead.

Adds the message cstr to the message/debug window when the script executes. If you enable debugging (i.e., showDebug = true), the comment shows in the debug messages. If you enable messages (i.e., showMessage = true), the comment shows in the messages. If you do not enable debugging or messages, the comment does not display.

Use this function instead of directly assigning value to message variable in script control.

Example

```
true ^ comment("calcValue is " + calcValue)
true ^ comment("The building fees have been added automatically")
```

comparePeopleGeneric

This function passes as a parameter to the createRefContactsFromCapContactsAndLink function.

Version

1.6

Parameters

Parameter	Type	Description
peop	peopleModel	The peopleModel object containing the criteria.

Returns

Takes a single peopleModel as a parameter, and returns the sequence number of the first G6Contact result. Returns null if there are no matches

Notes

To use attributes, you must implement Salesforce case 09ACC-05048.

completeCAP

Version

1.5

Parameters

Parameter	Type	Description
userId	string	ID of user that completes the record.
capId (optional)	CapIDModel	Record with which to perform the action.

Notes

Assigns the staff whose user ID is userId to the Completed by Staff field on a record. Also sets the Completed by Date value to the current date.

If you use the capId optional parameter, the function updates record capId. If you do not use the capId parameter, the function updates the current record.

contactAddFromUser

Version

1.6

Parameters

Parameter	Type	Description
pUserId	string	User ID used as criteria to search for contact.

Notes

Searches for a reference contact that matches the supplied userID, based on first, middle, and last names. If the function finds a matching contact, the function adds the record contact to the current record.

contactSetPrimary

Sets the supplied contact to be the primary contact on the current record

Version

1.6

Parameters

Parameter	Type	Description
pContactNbr	long	Sequence number of the contact to make primary.

contactSetRelation

Sets the relationship code on the supplied contact, on the current record.

Version

1.6

Parameters

Parameter	Type	Description
pContactNbr	long	Sequence number of the contact.
pRelation	string	Set to this relationship code.

convertDate

Converts a scriptDateTime date to a javascript date.

Version

1.6

Parameters

Parameter	Type	Description
thisDate	scriptDateTime	The date to convert.

convertStringToPhone

Converts the string to phone codes (A=1, D=3, etc), useful with the setIVR function.

Version

1.6

Parameters

Parameter	Type	Description
theString	string	String containing information to convert.

copyAddresses

Copies all property addresses from record pFromCapId to record pToCapId. If record pToCapId has a primary address, any primary address in pFromCapId becomes non-primary when copied over.

Version

1.4

Parameters

Parameter	Type	Description
pFromCapId	CapIDModel	ID of record from which to copy.
pToCapId	CapIDModel	ID of record to which to copy. If null, the function uses the current record.

Notes

getApplication( ), getParent( ), createChild(), createCap() functions each returns a record ID object.

copyAppSpecific

Copies all app spec info values from current record to the record whose record ID object is newCap. If the target record does not have the same app specific info field, the does not copy the value.

Version

1.3

Parameters

Parameter	Type	Description
newCap	CapIDModel	ID of record from which to copy.
ignoreArr (optional)	string array	Array of ASI labels not to ignore and not copy.

copyASIFields

Copies all ASI fields from the sourceCapId record to the targetCapId record with the exception of the ASI subgroups listed in ignore1 . . . ignoren

Version

1.5

Parameters

Parameter	Type	Description
sourceCapId	CapIDModel	ID of record from which to copy.
targetCapId	CapIDModel	ID of record to which to copy.
ignore1 to ignoren (optional)	string	ASI subgroups to ignore during the copy.

Notes

This function moves the ASI fields themselves, not the values. You can add an ASI group to a record that did not previously include the ASI group. This function does not copy the form portlet designer settings, which can cause problems.

copyASITables

Copies ASI Tables from one Record to another. This function depends on the addASITable function.

Version

2.0

Parameters

Parameter	Type	Description
pFromCapId	CapIDModel	ID of record from which to copy.
pToCapId	CapIDModel	ID of record to which to copy.
ignoreArr (optional)	string array	Array of table names to ignore and not copy.

copyCalcVal

Copies the calculated job value from the current record to the record whose record ID object is pToCapId.

Version

1.4

Parameters

Parameter	Type	Description
fromcap	CapIDModel	ID of record from which to copy.
newcap	CapIDModel	ID of record to which to copy.

copyConditions

Copies all conditions from record capId to the current record (if you do not specify *toCapId*) or the specified record.

Version

1.3

Parameters

Parameter	Type	Description
fromCapId	CapIDModel	ID of record from which to copy.
toCapId (optional)	CapIDModel	ID of record to which to copy.

Example

```
true ^ subdivapp = getApplication(lookup("SubdivisionXref",{SubDiv})) ;
copyConditions(subdivapp)
```

copyConditionsFromParcel

Copies conditions from the reference parcel parcelIdString and adds them as conditions to the current record (not to parcels on the current record).

Version

1.4

Parameters

Parameter	Type	Description
parcelIdString	string	Parcel number of source parcel.

copyContacts

Copies all contacts from record pFromCapId to record pToCapId.

Version

1.3

Parameters

Parameter	Type	Description
pFromCapId	CapIDModel	ID of record from which to copy.
pToCapId	CapIDModel	ID of record to which to copy. If null, the function uses the current record.

Notes

If target record has a primary contact and the source record also has a primary contact, the target record ends up with 2 primary contacts.

getApplication( ), getParent( ), createChild(), createCap() functions each return a Cap ID object.

copyContactsByType

Copies only contacts of the specified type from record pFromCapId to record pToCapId.

Version

2.0

Parameters

Parameter	Type	Description
pFromCapId	CapIDModel	ID of record from which to copy.
pToCapId	CapIDModel	ID of record to which to copy. If null, the function uses the current record.
pContactType	string	Contact type to copy.

Notes

If target record has a primary contact and the source record also has a primary contact, the target record ends up with 2 primary contacts.

getApplication( ), getParent( ), createChild(), createCap() functions each return a Cap ID object.

copyFees

Copies all fees from record sourceCapId to record targetCapId. Excludes voided or credited fees.

Version

1.5

Parameters

Parameter	Type	Description
sourceCapId	CapIDModel	ID of record from which to copy fees.
targetCapId	CapIDModel	ID of record to which to copy.

copyLicensedProf

Copies all licensed professionals from *sCapId* to record *tCapId.*

Version

1.6

Parameters

Parameter	Type	Description
sCapId	CapIDModel	ID of record from which to copy licensed professionals.
tCapId	CapIDModel	ID of record to which to copy.

copyOwner

Copies a contacts from sCapID to tCapID.

Version

1.6

Parameters

Parameter	Type	Description
sCapID	CapIDModel	ID of record from which to copy.
tCapID	CapIDModel	ID of record to which to copy.

copyOwnersByParcel

Copies reference owners from all attached parcels to the current record.

Version

2.0

Parameters

None

copyParcelGisObjects

Copies parcel GIS objects to the record.

Version

1.3

Parameters

None

copyParcels

Copies all parcels, and parcel attributes, from record pFromCapId to record pToCapId.

Version

1.4

Parameters

Parameter	Type	Description
pFromCapId	CapIDModel	ID of record from which to copy.
pToCapId	CapIDModel	ID of record to which to copy. If null, the function uses the current record.

Notes

capId is the record ID object for the current record.

getApplication( ), getParent( ), createChild(), createCap() functions each return a record ID object.

copySchedInspections

Copies all scheduled inspections from record pFromCapId to record pToCapId.

Version

1.4

Parameters

Parameter	Type	Description
pFromCapId	CapIDModel	ID of record from which to copy.
pToCapId	CapIDModel	ID of record to which to copy. If null, the function uses the current record.

Notes

Includes inspections that have a pending-type result, but copies status over as Scheduled. You do not need to copy the inspection type to the target record. The function can copy duplicate inspections to the target record.

capId is the record ID object for the current record.

getApplication( ), getParent( ), createChild(), createCap() functions each return a record ID object.

countActiveTasks

Returns the number of active tasks in the workflow whose process name is processName.

Version

1.4

Parameters

Parameter	Type	Description
processName	string	Process name of workflow.

countIdenticalInspections

Returns the number of inspections that have the same inspection description and status (or result) as the inspection in the current event.

Version

1.4

Parameters

None

Notes

Use this function only with the following events:

- InspectionResultSubmitAfter

- InspectionScheduleAfter

- InspectionScheduleBefore

createAddresses

Adds an address to the record.

Version

2.0

Parameters

Parameter	Type	Description
targetCapID	CapIDModel	Record ID object.
addressModel	AddressModel	Address.

createCap

Creates a record of type pCapType with the record name of pAppName.

Version

1.4

Parameters

Parameter	Type	Description
pCapType	string	Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes.
pAppName	string	Record name.

Returns

Returns the new record's record ID object.

createCapComment

Creates a record comment for the specified record

Version

1.6

Parameters

Parameter	Type	Description
vComment	string	Comment to add.
capId (optional)	CapIDModel	Record for which to create a comment.

createChild

Creates a record of type grp/typ/stype/cat with the record name, and links it as a child to the current record's hierarchy.

Version

1.3

Parameters

Parameter	Type	Description
grp	string	App Group. Top classification of the record.
typ	string	App Type. Second classification of the record.
stype	string	App SubType: 3rd Classification of the record.
cat	string	App Category: 4th Classification of the record.
desc	string	Record name.
capId (optional)	CapIDModel	Record to be the parent of new record.

Returns

The new child record's ID.

Notes

The function copies the following data from the current record to the new child record.

- parcels

- contacts

- property addresses

createParent

Creates a record of type grp/typ/stype/cat with the record name, and links it as a parent to the current record's hierarchy.

Version

2.0

Parameters

Parameter	Type	Description
grp	string	App Group. Top classification of the record.
typ	string	App Type. Second classification of the record.
stype	string	App SubType: 3rd Classification of the record.
cat	string	App Category: 4th Classification of the record.
desc	string	Record name.

Returns

The new parent record's record ID object, to be used in other functions.

Notes

The following data are copied from the current record to the new parent record.

• parcels

• contacts

• property addresses

createPendingInspection

Creates a pending inspection of the specified group and type on the specified record.

Version

2.0

Parameters

Parameter	Type	Description
iGroup	string	Inspection group of the inspection to create.
iType	string	Inspection type of the inspection to create.
capId (optional)	CapIDModel	Record on which to create the inspection.

Notes

Uses the current record (capId global variable) if no capId parameter supplied.

createPendingInspFromReqd

Creates a pending inspection for all inspections that are configured as required in the inspection group associated to the record type.

Version

2.0

Parameters

Parameter	Type	Description
capId (optional)	CapIDModel	Record on which to create the inspection.

createPublicUserFromContact

Creates a public user account (Citizen Access) with information based on the contact.

Version

1.6

Parameters

Parameter	Type	Description
contactType (optional)	string	The public user is based on this contact type, default is Applicant.

Notes

Useful for automatically creating an online account for applicants that apply in the office.

- Creates the public user record

- Assigns to current agency

- Activates for the current agency

- Issues a password reset to their email address

- Sends activation email

createRefContactsFromCapContactsAndLink

This function can be used as the basis for maintaining a contact-centric database within Civic Platform.

Version

1.6

Parameters

Parameter	Type	Description
pCapId	CapIDModel	Record to work with.
contactTypeArray	array	The contact types to process, or null for all. This parameter is ignored if the REF_CONTACT_ENFORCE_TYPE_FLAG_WITH_EMSE standard choice is configured. See description for more detail.
ignoreAttributeArray	array	An array of attributes to ignore when creating a REF contact, or null.
replaceCapContact	boolean	Not implemented.
overwriteRefContact	boolean	If true, refreshes the linked ref contact with record contact data.
refContactExists	function	Function used to determine if the reference contact exists.

Example

```
iArr = new Array();
iArr.push("Partner Percent")
createRefContactsFromCapContactsAndLink(capId,null,iArr,false,true,comparePeopleGeneric)
```

In this example, when this code is executed, the function loops through all contacts on the current record. If the contact was hand-entered (not selected and validated from reference contacts) the reference contacts searches for a match using the comparePeopleGeneric function. If a match is found, the record contact links to the reference contact. Also, the reference contact refreshes with data from the cap contact. All attributes refresh except for the "Partner Percent" field.

Version 2.0 Update: This function now checks for the presence of a standard choice "REF_CONTACT_CREATION_RULES". See screenshot below for configuration.

This setting determines whether to create the reference contact, as well as the contact type with which to create the reference contact. If this setting is configured, the function ignores the contactTypeArray parameter. The "Default" in this standard choice determines the default action of all contact types. Other types can be configured separately. Each contact type can be set to "I" (create ref as individual), "O" (create ref as organization), "F" (follow the indiv/org flag on the cap contact), "D" (Do not create a ref contact), or "U" (create ref using the transactional contact type").

createRefLicProf

Creates a new reference Licensed Professional from the Contact on the current record whose contact type is pContactType.

Version

1.4

Parameters

Parameter	Type	Description
rlpId	string	State license number.
rlpType	string	License type.
pContactType (optional)	string	Contact type.

Notes

The Licensed Professional has the state license # of rlpId and license type of rlpType. If a reference Licensed Professional with state license # rlpId already exists, it updates with data from the Contact.

Contact's State field must be populated for the Licensed Prof to be created.

The function does not copy the Contact's middle name and address line 3 to the Licensed Prof.

If available, the following app specific info fields copy to the Licensed Prof (field labels must match exactly):

- Insurance Co

- Insurance Amount

- Insurance Exp Date

- Policy #

- Business License #

- Business License Exp Date

createRefLicProfFromLicProf

Retrieves the first licensed professional on the record and creates a reference licensed professional record. If a reference record already exists for this licensed professional, updates the reference licensed record with the licensed professional's data from the record.

Version

1.4

Parameters

None

dateAdd

Returns date that results from adding amt days to td, as a string in "MM/DD/YYYY" format.

Version

1.3

Parameters

Parameter	Type	Description
td	string	Starting date, in format "MM/DD/YYYY" (or any string that converts to JS date). If null is used, td is the current date.
amt	integer	Number of days to add to td. Use negative number (e.g. –20) to subtract days from td.
workDays (optional)	string	'Y' if amt workdays should be added to td. Omit if amt calendar days should be added to td.

Notes

Does not work if date is wfDate. Returns NaN/NaN/NaN.

dateAddMonths

Returns date that results from adding pMonths months to pDate, as a string in "MM/DD/YYYY" format.

Version

1.4

Parameters

Parameter	Type	Description
pDate	string	Starting date, in format "MM/DD/YYYY" (or any string that converts to JS date). If null is used, td is the current date.
pMonths	integer	Number of months to add to pDate. Use negative number (e.g. -12) to subtract months from td.

Notes

If pDate is the last day of the month, the returned date is the last day of the month. If pDate is not the last day of the month, the new date has the same day of month, unless such a day doesn't exist in the new month (e.g. if baseDate is 1/30/2007 and the returned month is February), in which case the new date is the last day of the month.

Does not work if baseDate is wfDate. Returns NaN/NaN/NaN.

dateFormatted

Returns formatted date in YYYY-MM-DD or MM/DD/YYYY format (default).

Version

1.3

Parameters

Parameter	Type	Description
pMonth	string	Month of new date, as 2-digit month.
pDay	string	Day of new date, as 2-digit day.
pYear	string	Year of new date as 4-digit year.
pFormat	string	Format to produce string in.

dateNextOccur

Returns the next occurrence of pMonth and day after pDate. If oddEven is "odd", gets the next occurrence of pMonth and day after pDate in an odd year (for example, year is an odd number). If oddEven is "even", gets the next occurrence of pMonth and day after pDate in an even year.

Version

1.3

Parameters

Parameter	Type	Description
pMonth	string	Month of new date, as 2-digit month.
pDay	string	Day of new date, as 2-digit day.
pDate	string	Date from which new date is determined. In format MM/DD/YYYY or YYYY-MM-YY as used by wfDate variable.
oddEven (optional)	string	Specifies if the new date should be in an odd or even year. Enter "odd" or "even".

Notes

The pDate parameter can be a date string in MM/DD/YYYY format, or an event-specific variable (e.g. wfDate) whose date format is YYYY-MM-DD.

deactivateTask

Deactivates the task, similar to setting Active? = N in the workflow supervisor portlet

Version

1.6

Parameters

Parameter	Type	Description
wfstr	string	Workflow task to be deactivated.
wfProcess (optional)	string	Process name of workflow task wfstr.

deleteTask

Permanently removes the named task from the workflow.

Version

1.6

Parameters

Parameter	Type	Description
targetCapId	CapIDModel	Record to affect.
deleteTaskName	string	Name of task to delete.

editAppName

Updates record name to newName.

Version

1.5

Parameters

Parameter	Type	Description
newName	string	New record name.
capId (optional)	CapIDModel	Record ID object for record.

Returns

Returns true if successful or false if update fails.

editAppSpecific

Updates the value of the app specific info field itemName with the value itemValue. Also updates the internal list of values, so that future criteria/action pairs do not see the correct value. If no capId is supplied, then the current record is used.

Version

1.3

Parameters

Parameter	Type	Description
itemName	string	App Specific Info field to edit.
itemValue	string	Value that the app spec info field itemName should be changed to.
capId (optional)	CapIDModel	Record ID object for record whose app spec info field itemName is to be changed to itemValue.

editBuildingCount

Edits the building count on the record detail.

Version

1.6

Parameters

Parameter	Type	Description
numBuild	string	New number of buildings.
capId (optional)	CapIDModel	The capID to affect.

editCapContactAttribute

Changes the value of a record contact attribute.

Version

2.0

Parameters

Parameter	Type	Description
contactSeq	long	Sequence number of the record contact to edit.
pAttributeName	string	Label of the attribute to edit.
pNewAttributeValue	string	New value of the attribute.
itemCapId (optional)	CapIDModel	Record on which the record contact belongs.

Notes

The attribute name must be in ALL CAPS.

Example

```
editCapContactAttribute(60549773,"HAIR COLOR","Yellow",thisCapId);
```

editChannelReported

Changes the channel reported value to value passed to function.

Version

2.0

Parameters

Parameter	Type	Description
channel	string	Value to change channel reported to.
capId (optional)	CapIDModel	Record to change value on.

Example

```
editChannelReported("PHONE",capId);
```

editContactType

Updates Contact Type for all contacts on a record to newtype when the existing Contact Type is equal to the existingType.

Version

1.5

Parameters

Parameter	Type	Description
existingType	string	Existing contact type.
newType	string	New contact type.
capId (optional)	CapIDModel	Record ID object.

Notes

getApplication( ), getParent( ), createChild( ) functions each returns a record ID object that can be used in the capId parameter

editHouseCount

Updates the record's house count field to numHouse.

Version

1.5

Parameters

Parameter	Type	Description
numHouse	string	New house count.
capId (optional)	CapIDModel	Record ID object for record.

Returns

Returns true if successful or false if update fails.

editInspectionRequiredFlag

Sets the inspection milestone flag 'Inspection Required" to Y or N.

Version

1.6

Parameters

Parameter	Type	Description
inspType	string	Inspection type to edit.
reqFlag	boolean	If true, sets the required flag to "Y", otherwise "N".
capId (optional)	CapIDModel	Target record ID.

editLookup

Attempts to find existing standard choices value called stdValue in the standard choices item called stdChoice. If found, updates the existing Value Description for stdValue. If stdValue is not found, adds the new value stdValue with the Value Desc of stdDesc.

Version

1.5

Parameters

Parameter	Type	Description
stdChoice	string	Name of standard choice.
stdValue	string	Name of standard choice value.
stdDesc	string	New standard choice description.

editPriority

Updates the record's Priority field to priority.

Version

1.5

Parameters

Parameter	Type	Description
priority	string	New priority.
capId (optional)	CapIDModel	Record ID object for record.

Returns

Returns true if successful or false if update fails.

editRefLicProfAttribute

Updates the attribute (template data) on a reference licensed professional record.

Version

1.6

Parameters

Parameter	Type	Description
pLicNum	string	License number of reference LP.
pAttributeName	string	Label of the attribute to update.
pNewAttributeValue	string	New attribute value.

editReportedChannel

Updates the record's Reported Channel field to reportedChannel.

Version

1.5

Parameters

Parameter	Type	Description
reportedChannel	string	New reported channel value.
capId (optional)	CapIDModel	Record ID object for record.

Returns

Returns true if successful or false if update fails.

editScheduledDate

Edits the schedule date in record detail on the selected record.

Version

1.6

Parameters

Parameter	Type	Description
scheduledDate	string	New schedule date value.
[capId] (optional)	CapIDModel	Record ID to modify.

editTaskComment

Adds the status comment wfcomment to workflow task wfstr. If wfstr has an existing comment, the comment is replaced by wfcomment. wfstr does not have to be active. Status date is not updated. No workflow history record is created.

Version

1.3

Parameters

Parameter	Type	Description
wfstr	string	Workflow task whose comment should be updated.
wfcomment	string	Comment to be given to wfstr.
wfProcess (optional)	string	Process name of workflow task wfstr.

Notes

If record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr should be edited.

editTaskDueDate

Sets the due date of the workflow task wfstr to wfdate. If wfstr is "*", sets due dates on all workflow tasks on the record. No workflow history record is created.

Version

1.3

Parameters

Parameter	Type	Description
wfstr	string	Workflow task.
wfdate	string	Due date to be given to wfstr.
wfProcess (optional)	string	Process name of workflow task wfstr.

Notes

If record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr should be edited.

editTaskSpecific

Updates the value of the task specific info field itemName for workflow task wfName to the value itemValue. Also updates the internal list of values, so that future criteria/action pairs see the correct value. If capId is supplied, updates the specified task specific info field on the record whose record ID object is capId.

Version

1.3

Parameters

Parameter	Type	Description
wfName	string	Workflow task.
itemName	string	Task Specific Info field to edit .
itemValue	string	Value that the task spec info field itemName should be changed to.
capId (optional)	CapIDModel	Record ID object for record whose task spec info field itemName is to be changed to itemValue.

email

Sends an email to the email address pToEmail from the email address pFromEmail. The email's subject line is pSubject and its content is pText.

Version

1.4

Parameters

Parameter	Type	Description
pToEmail	string	Email address of recipient.
pFromEmail	string	Email address of sender.
pSubject	string	Text that appears in subject line of email.
pText	string	Text that appears in body of email.

emailContact

Sends an email to the contact on the current record whose Contact Type is contactType. Uses the email address in the contact screen. Default contact is "Applicant".

Version

1.3

Parameters

Parameter	Type	Description
mSubj	string	Text that appears in subject line of email.
mText	string	Text that appears in body of email.
contactType (optional)	string	Contact Type that email is sent to. Default is "Applicant".

Example

```
inspResult.equals("Passed") ^ emailContact("Inspection Results", "Your
  inspection " + inspType + " has passed.", "Contractor")
```

endBranch

Immediately stops execution of the branch (standard choice) that is currently executing. Script controls continue executing from the calling standard choice, if any.

Version

1.6

Parameters

None

Example

```
01 true ^ endBranch()
02 true ^ comment("this will not execute")
```

executeASITable

Executes an ASI table as if it were script commands. No capability for else or continuation statements. Assumes that there are at least three columns named "Enabled", "Criteria", and "Action". Replaces token in the controls.

Version

1.5

Parameters

Parameter	Type	Description
tableArray	array	Application specific info table array.

exists

Searches the array eArray for the value eVal. Returns true if the value is found in the array.

Version

1.6

Parameters

Parameter	Type	Description
eVal	string	The search value.
eArray	array of strings	Potential matches.

Example

```
Values = new Array("Apple","Pear","Banana");
X = exists("Apple",Values);
X is true.
```

externalLP_CA

Validates a license with the California State License Board and refreshes LP information with results.

Version

1.6

Parameters

Parameter	Type	Description
licNum	string	Valid CA license number. Non-alpha, max 8 characters. If null, the function uses the LPs on the supplied record ID.
rlpType	string	License professional type to use when validating and creating new LPs.
doPopulateRef	boolean	If true, creates/refreshes a reference LP of this number/type.
doPopulateTrx	boolean	If true, copies create/refreshed reference LPs to the supplied Cap ID. doPopulateRef must be true for this to work.
itemCap	CapIDModel	If supplied, licenses on the record are validated. Is also refreshed if doPopulateRef and doPopulateTrx are true.

Notes

See the "CSLB Interface using the externalLP_CA function - v3_0.pdf" document for detailed information.

Example

appsubmitbefore (validates the LP entered, if any, and cancels the event if the LP is inactive, cancelled, expired, etc.)

```
cslbMessage = externalLP_CA(CAELienseNumber,false,false,CAELienseType,null);
```

appsubmitafter (update all CONTRACTOR LPs on the record and REFERENCE with data from CSLB. Link the record LPs to REFERENCE. Pop up a message if any are inactive...)

```
cslbMessage = externalLP_CA(null,true,true,"CONTRACTOR",capId)
```

feeAmount

Returns the total amount of the all fees on the record whose fee code is feestr. If optional fStatus1 … fStatusn parameter(s) are supplied, also checks that feestr has one of the statuses in fStatus1 … fStatusn.

Version

1.5

Parameters

Parameter	Type	Description
feestr	string	Fee code.
fStatus1 … fStatusn (optional)	string	List of fee statuses to check for. Enter one or more statuses.

Notes

A fee has one of the following statuses: NEW, INVOICED, VOIDED, CREDITED.

feeAmountExcept

Returns the total amount of the all fees on the record. Ignores fees that are supplied as additional parameters.

Parameters

Parameter	Type	Description
checkCapId	CapIDModel	Record ID to search.
feeCodeToIgnore1… feeCodeToIgnoren (optional)	string	One or more fee codes to ignore.

feeBalance

Returns the total balance due for all fees on the record whose fee code is feestr. If parameter feeSchedule is used, retrieves those fees whose schedule is feeSchedule.

Version

1.4

Parameters

Parameter	Type	Description
feestr	string	Fee code.
feeSchedule (optional)	string	Fee schedule.

feeCopyByDateRange

On the current record, searches for fees in the given date and status criteria, then copies the fees onto the current record.

Version

1.6

Parameters

Parameter	Type	Description
pStartDate	string	Starting search date for fee items.
pEndDate	string	Ending search date for fee items.
feeStatus (optional)	string	Search for fee items of this status.
feeStatus (optional)	string	Search for fee items of this status.

feeExists

Version

1.3

Parameters

Parameter	Type	Description
feestr	string	Fee code of fee to check for.
fStatus1 … fStatusn (optional)	string	List of fee statuses to check for. Enter one or more statuses.

Returns

Returns true if a fee whose fee code is feestr has been added to the record.

Notes

If optional fStatus1 … fStatusn parameter(s) are supplied, also checks that feestr has one of the statuses in fStatus1 … fStatusn.

A fee has one of the following statuses: NEW, INVOICED, VOIDED, CREDITED.

Example

To determine if fee "FEE001" has been added and not invoiced:

```
feeExists("FEE001","NEW")
```

feeGetTotByDateRange

Version

1.3

Parameters

Parameter	Type	Description
pStartDate	string	Start of date range, in format MM/DD/YYYY.
pEndDate	string	End of date range, in format MM/DD/YYYY.
fStatus1 … fStatusn (optional)	string	List of fee statuses to check for. Enter one or more statuses.

Returns

Returns total amount of fees that were assessed during the date range pStartDate to pEndDate.

Notes

If optional fStatus1 … fStatusn parameter(s) are supplied, the fee must have one of the statuses in fStatus1 … fStatusn.

A fee has one of the following statuses: NEW, INVOICED, VOIDED, CREDITED.

Fees are retrieved by their initial assess date, not invoiced date.

feeQty

Version
1.6

Parameters

Parameter	Type	Description
feestr	string	Fee item to search.

Returns
On the current record, returns the quantity field of the given fee item.

getAddressConditions

Searches for address conditions by the following parameters. Additionally *pType*, *pStatus*, *pDesc*, and *pImpact* can be passed as null values for wildcard searches.

Version

2.0

Parameters

Parameter	Type	Description
pType	string	Condition type.
pStatus	string	Condition status.
pDesc	string	Condition description.
pImpact	string	Condition impact code.
capId (optional)	CapIDModel	Record to search.

Notes

This function can only work well when the Civic Platform site supports Arabic.

getAppIdByASI

Version

1.4

Parameters

Parameter	Type	Description
ASIName	string	App specific info field name to search for.
ASIValue	string	App specific info field value to search for. Record ID object for record whose app spec info field ASIValue is to be changed to ASIValue.
ats	string	Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.

Returns

Returns the record number (cap ID string) of the first record whose record type matches ats and whose application specific info field ASIValue has the value of ASIValue.

getAppIdByName

Version

1.3

Parameters

Parameter	Type	Description
gaGroup	string	Record group.
gaType	string	Record type.
gaName	string	Record name.

Returns

Returns the cap ID string of the first record whose record type begins with gaGroup / gaType and whose record name is gaName.

Notes

The parameter gaType is the 2nd value in the 4 level record type.

getApplication

Version
1.3

Parameters

Parameter	Type	Description
applicationNumber	string	Application # (B1_ALT_ID).

Returns

Returns the record ID object for record applicationNumber that can be used by other functions.

getAppSpecific

Version
1.3

Parameters

Parameter	Type	Description
itemName	string	Application Specific Info field to get.
capId (optional)	CapIDModel	Record ID object for record.

Returns
Returns the value of the application spec info field itemName. If you provide *capId*, returns the value of itemName on the record whose record ID object is capId.

getCapByAddress

Version

1.4

Parameters

Parameter	Type	Description
ats	string	Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.

Returns

Returns the first record having the same address as the current record and whose record type matches ats, as a record ID object. If the search does not return any records, the function does not return any value.

Notes

The function matches addresses by Street # (start), Street Name, Street Direction, Street Suffix, and Zip. The function can return the current record.

getCAPConditions

Searches for record conditions by the following parameters. Additionally you can pass *pType*, *pStatus*, *pDesc*, and *pImpact* as null values for wildcard searches.

Version

2.0

Parameters

Parameter	Type	Description
pType	string	Condition type.
pStatus	string	Condition status.
pDesc	string	Condition description.
pImpact	string	Condition impact code.
capId (optional)	CapIDModel	Record to search.

Notes

This function can only work well when the Civic Platform site supports Arabic.

getCapId

Gets the ID of the record associated with the event.

getCapsWithConditionsRelatedByRefContact

Searches for records that share the same reference contact and same record condition, and returns the result as an array of CapIDModels.

Version

2.0

Parameters

Parameter	Type	Description
itemCap	string	The capIDModel of record.
capType	string	Application type.
pType	string	Condition type, leave null for wildcard search.
pStatus	string	Condition status.
pDesc	string	Condition description.
pImpact	string	Condition impact code.

getChildren

If you use the skipChildCapId parameter, the function excludes any child record whose record ID object is skipChildCapId.

Version

1.4

Parameters

Parameter	Type	Description
pCapType	string	Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.
pParentCapId (optional)	CapIDModel	Record ID object for parent record. Use null if skipChildCapId parameter is used.
skipChildCapId (optional)	CapIDModel	Record ID object of child record to exclude.

Returns

Returns all child records whose record type matches pCapType, as an array of record ID objects. If the pParentCapId parameter is used, returns child records of the record whose record ID object is pParentCapId.

Notes

If the skipChildCapId parameter is used, the function excludes any child record whose record ID object is skipChildCapId.

See also

childGetByCapType

getChildTasks

Version

1.6

Parameters

Parameter	Type	Description
taskName	string	Name of criteria parent task.
capId (optional)	CapIDModel	Record to search.

Returns

Returns an array of taskScriptModel objects, which represent the child tasks (sub process) of the criteria task.

getConditions

Searches for cap conditions, address conditions, contact conditions, parcel conditions, and licensed professional conditions by the following parameters. Additionally pType, pStatus, pDesc, and pImpact can be passed as null values for wildcard searches.

Version

2.0

Parameters

Parameter	Type	Description
pType	string	Condition type.
pStatus	string	Condition status.
pDesc	string	Condition description.
pImpact	string	Condition impact code.
capId (optional)	CapIDModel	Record to search.

Notes

This function can only work well when the Civic Platform site supports Arabic.

getContactArray

Retrieves field values and customizes attribute values for all contacts and returns them as an array of associative arrays. Each element in the outer array contains an associative array of values for one contact. Each element in each inner associative array is a different field.

Version

2.0

Parameters

Parameter	Type	Description
capIdFrom (optional)	CapIDModel	Record ID object for source application.

Notes

The following fields are retrieved:

Contact Field	Element Name
First Name	firstName
Middle Name	middleName
Last Name	lastName
Business Name	businessName
Phone 1	phone1
Phone 2	phone2
Contact Type	contactType
Relationship	relation
Sequence Number	contactSeqNumber
Reference Contact ID	refSeqNumber
E-mail	email
Address Line 1	addressLine1
Address Line 2	addressLine2
City	city
State	state
Zip Code	zip
Fax	fax
Notes	notes
Country/Region	country
Full Name	fullName

All custom attributes are also added to the associative array, where the element name is the attribute name (in upper-case). Note that the attribute name may not be the same as the attribute label.

If the parameter capIdFrom is used, function retrieves contacts from the record whose record ID object is capIdFrom.

getContactConditions

Searches for contact conditions by the following parameters. Additionally *pType*, *pStatus*, *pDesc*, and *pImpact* can be passed as null values for wildcard searches.

Version

2.0

Parameters

Parameter	Type	Description
pType	string	Condition type.
pStatus	string	Condition status.
pDesc	string	Condition description.
pImpact	string	Condition impact code.
capId (optional)	CapIDModel	Record to search.

Notes

This function can only work well when the Civic Platform site supports Arabic.

getCSLBInfo

Selects the first licensed professional on the record and retrieves its data from the California State License Board (CSLB). If doWarning is true, shows a warning message if the license has expired. If doPop is true, updates the record's licensed professional with data from CSLB.

Version

1.4

Parameters

Parameter	Type	Description
doPop	boolean	Use true if the record's license professional must be updated with data from the California State License Board (CSLB); otherwise, use false.
doWarning	boolean	Use true if warning message should appear if license has expired; otherwise, use false.

Returns

Returns false if the record has no licensed professional, if the license cannot be found at CSLB, or if any error is encountered.

Notes

The following fields are updated:

- Business Name

- Phone Number

- Address Line 1

- Issued Date

- Address Line 2

- Expiration Date

- City

- State

- Zip

getDepartmentName

Version

1.4

Parameters

Parameter	Type	Description
username	string	User's ID.

Returns

Returns the department of the user whose ID is username.

getGISBufferInfo

Version

1.4

Parameters

Parameter	Type	Description
svc	string	GIS service name.
layer	string	GIS layer on which the function creates buffer zones around the input GIS object.
numDistance	integer	The distance (in feet) around the to-be-buffered GIS object in which buffer zones are created on the specified layer. A positive distance means creating buffers outside the GIS object while a negative distance means creating buffers inside the GIS object. When the buffer distance is negative, GIS checks whether the to-be-buffered GIS object is a point, line, or polygon. If it is a point or line, GIS changes the negative buffer distance to 0.01 to avoid the script error.
attribute1… attributen (optional)	strings	Additional attributes of the GIS layer to retrieve.

Returns

Returns an array of associative arrays. Each element in the outer array is a GIS object (from the indicated layer) within the buffer from the record's GIS object. Each element in the inner associative array is a requested attribute.

Example

```
x = getGISBufferInfo("NewtonCounty","Parcels","50","NAME1","TOTACRES");
x[0]["TOTACRES"] = 0.46
x[0]["NAME1"] = "JENNINGS DEMETRIA C"
x[1]["TOTACRES"] = 0.46
x[1]["NAME1"] = "SIMMS ROCK & VALARIE"
x[2]["TOTACRES"] = 0.46
x[3]["NAME1"] = "PAUL NEVILLE & MARGARET"
```

getGISInfo

Use with all events (and master scripts) except ApplicationSubmitBefore.

Version

1.4

Parameters

Parameter	Type	Description
svc	string	GIS service name.
layer	string	GIS layer.
attributename	string	Name of attribute to retrieve.

Returns

Returns the attribute value for attributename in the GIS layer for the last GIS object on the record.

getGISInfoArray

Version

1.6

Parameters

Parameter	Type	Description
svc	string	GIS service name.
layer	string	GIS layer.
attributename	string	Name of attribute to retrieve.

Returns

Similar to getGISInfo, except it returns an array of values for the given attribute, instead of the first value found.

getGuideSheetObjects

Version

2.0

Parameters

Parameter	Type	Description
inspId	long	Sequence number of the inspection that contains the guidesheet objects to retrieve.
capId (optional)	CapIDModel	Record Id to search.

Returns

Returns an array of guideSheetObject objects that represent the guidesheet data on the inspection.

Notes

See the guideSheetObject for more information

getInspector

Version

1.3

Parameters

Parameter	Type	Description
insp2Check	inspDesc	Inspection description.

Returns

Returns the user ID of the inspector assigned to inspection insp2Check whether scheduled or completed.

Notes

If more than one insp2Check is on the record, the first inspection found is selected, which may or may not be the insp2Check with the earliest inspection date.

getLastInspector

Version

1.4

Parameters

Parameter	Type	Description
insp2Check	string	Inspection description.

Returns

Returns the user ID of the last inspector to result the inspection insp2Check.

getLastScheduledInspector

Version

1.6

Parameters

Parameter	Type	Description
Insp2Check	string	Inspection description.

Returns

Returns the user ID of the last inspector to be schedule on the inspection insp2check

getLicenseConditions

Searches for licensed professional conditions by the following parameters. Additionally *pType*, *pStatus*, *pDesc*, and *pImpact* can be passed as null values for wildcard searches.

Version

2.0

Parameters

Parameter	Type	Description
pType	string	Condition type.
pStatus	string	Condition status.
pDesc	string	Condition description.
pImpact	string	Condition impact code.
capId (optional)	CapIDModel	Record to search.

Notes

This function can only work well when the Civic Platform site supports Arabic.

getLicenseProfessional

Version

1.6

Parameters

Parameter	Type	Description
itemcapId	CapIDModel	Record ID to use.

Returns

Returns an array of LicensedProfessional objects that represent all LPs on the specified record.

getParcelConditions

Searches for parcel conditions by the following parameters. Additionally *pType*, *pStatus*, *pDesc*, and *pImpact* can be passed as null values for wildcard searches.

Version

2.0

Parameters

Parameter	Type	Description
pType	string	Condition type.
pStatus	string	Condition status.
pDesc	string	Condition description.
pImpact	string	Condition impact code.
capId (optional)	CapIDModel	Record to search.

Notes

This function can only work well when the Civic Platform site supports Arabic.

getParent

Version

1.3

Parameters

None

Returns

Returns the record ID object for the first parent of the current record.

getParents

Version

1.5

Parameters

Parameter	Type	Description
itemCap (optional)	string	Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.

Returns

Returns all parents on the current record in a record ID object array. If itemCap parameter is passed, only returns parent records whose record type matches the itemCap parameter string pattern.

getRefLicenseProf

Version

1.6

Parameters

Parameter	Type	Description
refstlic	string	State license number to search for.

Returns

Returns a reference licensed professional object for the LP that matches the state license number value

getRelatedCapsByAddress

Version

1.4

Parameters

Parameter	Type	Description
ats	string	Four level application type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.

Returns

Returns all records having the same address as the current record and whose record type matches ats, as an array of record ID objects. If the function does not find any related records, the function does not return any value.

Notes

The function matches addresses by Street # (start), Street Name, Street Direction, and Street Suffix. The function does not include the current record in the returned array. Retrieve records do not have to be a parent or child of the current record.

getRelatedCapsByParcel

Version

1.4

Parameters

Parameter	Type	Description
ats	string	Four level record type. Must contain 3 slash (/) characters. Do not add spaces before or after slashes. You can use the asterisk (*) as a wildcard to match all entries for a given level.

Returns

Returns all records having the same parcel as the current record and whose record type matches ats, as an array of record ID objects. The function does not include the current record in the returned array. If the function does not find any related records, the function does not return any value.

Notes

Records retrieved do not have to be a parent or child of the current record.

getReportedChannel

Version

1.5

Parameters

Parameter	Type	Description
capId (optional)	CapIDModel	Record ID object for application.

Returns

Returns the value of the Reported Channel field as a string. If null, the function returns an empty string.

getScheduledInspId

Version
1.6

Parameters

Parameter	Type	Description
insp2Check	string	Inspection description.

Returns

Returns the internal sequence number for the inspection record that matches the description. Only returns values for scheduled inspections, not resulted inspections.

Notes

You can use the returned sequence number with other functions, such as autoAssignInspection.

getShortNotes

Version

1.5

Parameters

Parameter	Type	Description
capId (optional)	CapIDModel	Record ID object for record.

Returns

Returns the value of the Short Notes field as a string. If null, the function returns an empty string.

getTaskDueDate

Version

1.6

Parameters

Parameter	Type	Description
wfstr	string	Workflow task name.
wfProcess (optional)	string	Workflow process name.

Returns

Returns the due date of the requested workflow task on the current record.

Notes

If a record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to check.

wfProcess is R1_PROCESS_CODE in the GPROCESS and SPROCESS tables. wfProcess is normally in uppercase.

getTaskStatusForEmail

This function retrieves all completed tasks on workflow stask and returns their task name, status, and comments (if any) in the following format:

- Task Name: {task name}

- Task Status: {task status}

- Task Comments: {status comments}

The function repeats the previous block for each completed task.

Version

1.3

Parameters

Parameter	Type	Description
stask	string	Process name of workflow.

hasPrimaryAddressInCap

Checks whether a record has a primary address.

Version

2.0

Parameters

Parameter	Type	Description
capID	CapIDModel	Record ID object.

insertSubProcess

Dynamically adds a workflow process as a subprocess to an existing task.

Version

2.0

Parameters

Parameter	Type	Description
taskName	string	Name of the task that is the parent for the sub-process.
process	string	Name of the reference workflow process that the function adds a subprocess.
completeReqd	boolean	True if you must complete the subprocess before you promote the parent task. Otherwise, false.
itemCap (optional)	CapIDModel	Optional target capId.

Example

```
insertSubProcess("Reviews","PLAN_REVIEW_VER1",true);
```

inspCancelAll

Cancels all scheduled and incomplete inspections on the current record.

Version

1.4

Parameters

None

Returns

Returns true if at least one inspection is cancelled; otherwise, returns false.

invoiceFee

Invoices all assessed fees with fee code of fcode and fee period of fperiod.

Version

1.5

Parameters

Parameter	Type	Description
fcode	string	Fee code of the fee to invoice.
fperiod	string	Fee period of the fee to invoice.

Returns

Returns true if the function finds the assessed. Otherwise, returns false.

isScheduled

Version

1.3

Parameters

Parameter	Type	Description
inspType	string	Inspection description.

Returns

Returns true for scheduled or resulted inspections inspType for the current record.

Notes

To identify a scheduled, but not yet resulted inspection, use the checkInspectionResult function and use Scheduled for the insp2Result parameter.

isTaskActive

Version

1.3

Parameters

Parameter	Type	Description
wfstr	string	Workflow task name.
wfProcess (optional)	string	Workflow process name.

Returns

Returns true if workflow task wfstr is active, or false if it is not.

If used with the WorkflowTaskUpdateAfter event, this function returns true if wfstr becomes active as a result of the WorkflowTaskUpdateAfter event. The function returns false if wfstr becomes inactive as a result of the WorkflowTaskUpdateAfter event.

Notes

If record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to check.

wfProcess is R1_PROCESS_CODE in the GPROCESS and SPROCESS tables. wfProcess is normally in uppercase.

isTaskComplete

Version

1.3

Parameters

Parameter	Type	Description
wfstr	string	Workflow task name.
wfProcess (optional)	string	Workflow process name.

Returns

Returns true for a completed workflow task *wfstr*. Otherwise, returns false.

If used with the WorkflowTaskUpdateAfter event, this function returns true if wfstr becomes completed as a result of the WorkflowTaskUpdateAfter event.

Notes

If record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to check.

wfProcess is R1_PROCESS_CODE in the GPROCESS and SPROCESS tables. wfProcess is normally in uppercase.

isTaskStatus

Version

1.3

Parameters

Parameter	Type	Description
wfstr	string	Workflow task name.
wfstat	string	Workflow status.
wfProcess (optional)	string	Workflow process name.

Returns

Returns true if workflow task wfstr has the current status of wfstat, or false if it does not. Returns false if the function does not fine wfstr.

Notes

If record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to check.

wfProcess is R1_PROCESS_CODE in the GPROCESS and SPROCESS tables. wfProcess is normally in uppercase.

jsDateToASIDate

Converts the JavaScript Date object to a string, with a zero pad date format, that you can use in ASI, TSI, and ASI Table date fields.

Version

1.5

Parameters

Parameter	Type	Description
dateValue	JavaScript date	JavaScript date object.

jsDateToMMDDYYYY

Converts the JavaScript Date object pJavaScriptDate to a string in the format MM/DD/YYYY.

Version

1.4

Parameters

Parameter	Type	Description
pJavaScriptDate	JavaScript date	JavaScript date object.

Returns

Returns the date as a string in the format MM/DD/YYYY.

Notes

Use this function to display a JavaScript date in the format MM/DD/YYYY. Do not use the result of this function directly to compare against another date.

licEditExpInfo

Changes the record's expiration status to pExpStatus and expiration date to pExpDate.

Version

1.4

Parameters

Parameter	Type	Description
pExpStatus	string	Expiration status. Use null if you only edit expiration date.
pExpDate	string	Expiration date. Use null if you only edit expiration status.

Notes

If pExpStatus is null, expiration status does not change. If pExpDate is null, expiration date does not change. Use this function with license records only, that is the record type begins with Licenses.

pExpDate can be in YYYY-MM-DD or MM/DD/YYYY format.

Script throws an error if record does not have Renewal Info.

loadAddressAttributes

Populates thisArr as a associate array of address attributes and address values based on the address associated with the record.

Version

1.6

Parameters

Parameter	Type	Description
thisArr	array	Target array of address attributes.
capId (optional)	CapIDModel	Record ID to search.

loadAppSpecific[4ACA]

Retrieves all application specific info fields and adds them to the associative array thisArr.

Version

1.4

Parameters

Parameter	Type	Description
thisArr	array	Associative array.
capId (optional)	CapIDModel	ID for record from where to copy all app spec info fields.

Notes

The element name is the application specific info field name and the element value is the field value. If the user configurable variable useAppSpecificGroupName on the master script equals true, the function appends group name to the beginning of the field name with a period, that is CONSTRUCTION_INFO.Construction Type. The function does not retrieve application specific information table data.

If the function uses the capId parameter, the function retrieves application info fields from the record whose record ID object is capId.

The loadAppSepecific4ACA performs the same function as loadAppSpecific but it specially works with Citizen Access pageflow scripts.

loadASITable

Version

1.6

Parameters

Parameter	Type	Description
tname	string	Name of ASI table to load.
capId (optional)	CapIDModel	Record ID from which to load the table.

Returns

Returns an array of associate arrays that contain objects representing the contents of the ASI table for the selected record.

Notes

The underlying object is an "asiTableValObj" that contains three properties:

- fieldValue = value of the table

- columnName = name of the column for this value

- readOnly = Y for a read only field, N if not.

Example

```
myTable = loadASITable("EXAMPLE TABLE")
firstRow = myTable[0];
columnA = firstRow["Column A"]
columnB = firstRow["Column B"]
comment("value of column a is : " + columnA.fieldValue)
comment("column a read only property is : " + columnA.readOnly)
```

The fieldValue property of the asiTableValObj object is the default property, so the following also works:

```
comment("value of column a is : " + columnA);
```

loadASITables[4ACA][Before]

Similar to the loadASITable function, except the function creates global variables for each ASI table on the requested record.

Version

1.6

Parameters

Parameter	Type	Description
capId (optional)	CapIDModel	Record ID from which to load the table.

Notes

You can edit the names of the tables remove whitespace and leading digits, so that they become appropriate JavaScript variables.

Example

```
loadASITables();
if (typeof(PROPERNAMES) == "object")
comment("number of rows in the 'PROPER NAMES' table : " +
 PROPERNAMES.length)
```

Variables are not created for tables that do not have any data, so you must first use the JavaScript typeof operator to check for the presence of the table variable, as shown in the previous example.

By default, all master scripts execute loadASITables.

The loadASITables4ACA performs the same function as loadASITables but it specially works with Citizen Access pageflow scripts.

The loadASITablesBefore is an alternate version of this function that works specifically with the ApplicationSubmitBefore event.

loadFees

Version

1.5

Parameters

Parameter	Type	Description
capId (optional)	CapIDModel	Record ID object of record from which to load fees.

Returns

Retrieves all assessed fees for the record capId and returns them as an array of associative arrays.

Notes

Each element in the outer array contains an associative array of values for one fee. Each element in each inner associative array is a different field. The function retrieves the following fields:

Fee Field	Element Name
Sequence Num	sequence
Fee Code	code
Description	description
Unit	unit
Amount	amount
Amount Paid	amountPaid
Applied Date	applyDate
Effective Date	effectDate
Status	status
Received Date	redDate
Fee Period	period
Display Order	display
Account Code 1	accCodeL1
Account Code 2	accCodeL2
Account Code 3	accCodeL3
Fee Formula	formula
Sub Group	subGroup
Calculation Flag	calcFlag

loadGuideSheetItems

Version

1.6

Parameters

Parameter	Type	Description
inspId	long	Inspection sequence number to load.
capId (optional)	CapIDModel	Record to search.

Returns

Returns an associative array of guidesheet items from the indicated inspection.

Example

```
gsArray = loadGuideSheetItems(234323);
comment(gsArray["Privacy Violation"])
```

Displays the value of the Privacy Violation guidesheet item.

loadParcelAttributes

Retrieves all parcel fields (including custom attributes) and adds them to the associative array thisArr.

Version

1.4

Parameters

Parameter	Type	Description
thisArr	array	Associative array.
capId (optional)	CapIDModel	Record ID object for the record from where to copy parcel attributes.

Notes

The element name is the field name (prefixed with "ParcelAttribute.") and the element value is the field value. The function includes the following standard parcel fields:

ParcelAttribute.Block	ParcelAttribute.LegalDesc
ParcelAttribute.Book	ParcelAttribute.Lot
ParcelAttribute.CensusTract	ParcelAttribute.MapNo
ParcelAttribute.CouncilDistrict	ParcelAttribute.MapRef
ParcelAttribute.ExemptValue	ParcelAttribute.ParcelStatus
ParcelAttribute.ImprovedValue	ParcelAttribute.SupervisorDistrict
ParcelAttribute.InspectionDistrict	ParcelAttribute.Tract
ParcelAttribute.LandValue	ParcelAttribute.PlanArea

If the record has multiple parcels, the function only retrieves fields for the last parcel. If the function uses the capId parameter, the function retrieves parcel fields from the record whose record ID object is capId.

loadTasks

Version
1.3

Parameters

Parameter	Type	Description
ltcapidstr	string	Application # (B1_ALT_ID).

Returns
Returns an array of workflow task objects for the record ltcapidstr.

loadTaskSpecific

Retrieves all task specific info fields and adds them to the associative array thisArr.

Version

1.4

Parameters

Parameter	Type	Description
thisArr	array	Associative array.
capId (optional)	CapIDModel	Record ID object for the record from where to copy all task spec info fields.

Notes

The element name is the task specific info field name and the element value is the field value. If the user configurable variable useTaskSpecificGroupName on the master script equals true, the function prepends the workflow process code and workflow task name to the field name, for example, BLDGPROCESS.Application Submittal.Date Received.

If the function uses the capId parameter, the function retrieves task specific info fields from the record whose record ID object is capId.

logDebug

Displays debug information, depending on the showDebug global variable setting.

Version

1.6

Parameters

Parameter	Type	Description
dstr	string	Value to display on the debug window.
debugLevel (optional)		Debug content destination.

Notes

debugLevel overrides this setting for this message only.

```
debugLevel = false    // no output
debugLevel = 1         //  screen output
debugLevel = 2         //  output to biz server log
debugLevel = 3         //  output to screen and biz log
```

lookup

Looks up valueName in standard choices item stdChoice, and returns its value description. Essentially uses standard choices as a lookup table.

Version

1.3

Parameters

Parameter	Type	Description
stdChoice	string	Standard choices item name.
stdValue	string	Standard choices value.

Returns

Returns the Value Desc corresponding to the standard choices value stdValue in the standard choices item stdChoice. If the function does not find stdValue, returns undefined.

lookupDateRange

Matches dateValue against a series of dates in the standard choices called stdChoiceEntry.

Version

1.4

Parameters

Parameter	Type	Description
stdChoiceEntry	string	Item Name of standard choices used as lookup table.
dateValue	string	Date that determines which row to return. Use string in format MM/DD/YYYY, e.g. "07/21/2000".
valueIndex (optional)	integer	Determines the value to return. Defaults to 1, the first value.

Returns

If dateValue falls after date 1 but before or on date 2, returns the value following the caret (^) on date 1's right. If the function uses the valueIndex parameter, returns the value immediately after the valueIndex'th caret (^), following the matching date.

Notes

Set up the standard choices lookup table as follows:

- Value column = Four digit incremental index. Must be left zero padded to four digits. Entire table must be consecutive.

- Value Desc column = at least two values separated with the caret (^) symbol. Returns the first value as the effective date (MM/DD/YYYY format). Returns the remaining values by the function.

Examples

Standard Choices Item Name: test date lookup
Description:
(250 char max)

Status: ⦿ Enable ◯ Disable

Standard Choices Value	Value Desc
0001	1/1/2000^111 1^12222
0002	1/1/2001^22222^23333
0003	1/1/2002^33333^34444
0004	1/1/2006^44444^45555

lookupDateRange("test date lookup","5/5/2002") returns 33333

lookupDateRange("test date lookup","1/5/2000",2) returns 12222

lookupDateRange("test date lookup","1/1/2010") returns 44444

lookupDateRange("test date lookup","1/1/1999") returns undefined since there is no entry effective for that date.

lookupDateRange("test date lookup","1/5/2000",3) returns undefined since there are not 3 values.

Sample script controls:

```
01  appMatch("Building/Residential/SFD/*") ^lookupIndex=1
02  appMatch("Building/Residential/Duplex/*") ^ lookupIndex = 2
03  true ^ addFee("FEECODE","FEESCHED","FEEPERIOD", lookupDateRange("test
 date lookup", filedate, lookupIndex), "Y")
```

lookupFeesByValuation

Looks up the Value Desc for the stdChoiceValue Value in the standard choices called stdChoiceEntry.

Version

1.4

Parameters

Parameter	Type	Description
stdChoiceEntry	string	Item name of standard choices used as lookup table.
stdChoiceValue	string	Standard choices value.
capval	number	Number value (e.g. valuation) to compare.
valueIndex (optional)	integer	Determines which value to return. Defaults to 1, the first value.

Notes

Compares capval against the series of numbers in the Value Desc. If valueIndex is null or 1, uses the value following the 1st pipe (|) on the matching number's right to calculate the base fee. If valueIndex is 2, usrs the value following the 2nd pipe (|) on the matching number's right to calculate an add on fee.

Set up the standard choices lookup table as follows:

- Value column = Lookup value.

- Value Desc column = one or more 3-number series, where

  - 1st number = number to compare compareValue against

  - 2nd number = base fee

  - 3rd number = used to calculate add-on fee

Use a pipe(|) to separate each number. Use a caret(^) to separate each 3-number series.

Example


```
06  true ^ theBase = lookupFeesByValuation("PlanCheck2007","A-1-
Group2",5600)
07  true ^ theAddOn = lookupFeesByValuation("PlanCheck2007","A-1-
Group2",5600,2)
08  true ^ newTotal = newTotal +(parseFloat(theBase) +parseFloat(theAddOn))
```

5,800 is between 2,000 and 10,000 so the function returns 1,177.62

There are 36 additionl 100 Sq Ft over the base of 2000 so the function returns 36 * 6.1227 = 220.42

Total Fee = 6,6706.04
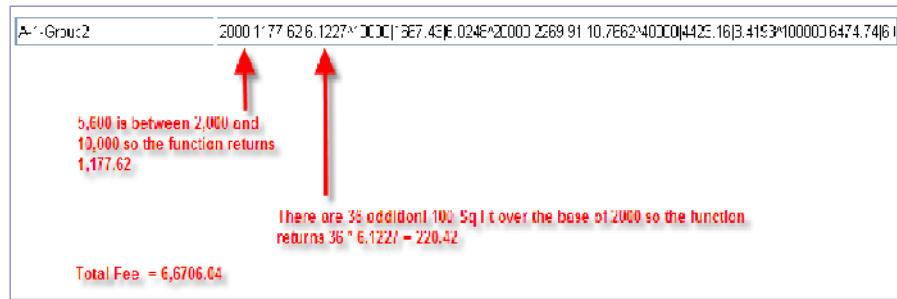
lookupFeesByValuationSlidingScale

Similar to the lookupFeesByValuation function, but introduces another element in the standard choice tables which serves as a divisor for the capval.

Version

1.6

Parameters

Parameter	Type	Description
stdChoiceEntry	string	Item name of standard choices used as lookup table.
stdChoiceValue	string	Standard choices value.
capval	number	Number value (e.g. valuation) to compare.
valueIndex (optional)	integer	Determines which value to return. Defaults to 1, the first value.

Notes

Set up the standard choices lookup table as follows:

- Value column = Lookup value.

- Value Desc column = one or more 3-number series, where

    - 1st number = number to compare compareValue against

    - 2nd number = divisor (e.g., 100, 1000, etc.)

    - 3rd number = base fee

    - 4th number = used to calculate add-on fee

Use a pipe(|) to separate each number. Use a caret(^) to separate each 4-number series.

loopTask

Version

1.3

Parameters

Parameter	Type	Description
wfstr	string	Workflow task name.
wfstat	string	Status to assign.
wfcomment	string	Comment to add.
wfnote	string	Note to add to the workflow task.
wfProcess (optional)	string	ID (R1_PROCESS_CODE) for the process that the task belongs to. Required for multi-level workflows.

Notes

Updates the workflow task wfstr as follows:

- Status = wfstat

- Status Date = current date

- Status Comment = wfcomment

- Action By = current user

Closes task wfstr and promotes workflow to the loop task.

If record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to check.

Master Script Function List

This section provides reference information for the master script functions.

Conventions

- Unless otherwise stated, all function names and parameter values are case sensitive.

- Enter function parameters in the order listed.

> 📝 **Note:**
>
> The name of a function parameter is for descriptive purposes only. The name of the function parameters in this chapter can differ from the name of the corresponding function parameter in the UniversalMasterScript file.

- For the string data type, enclose the parameter value in double-quotes.

- Subscripts 1 and n in parameter names (e.g., wfTask1 … wfTaskn) indicate that you can add between one and any number of such parameters, each in double-quotes and separated by commas.

- This reference shows Boolean values as true or false.

- This reference does not document internal functions in the master script files.

CapIDModel Type

The master script functions use the CapIDModel type for the capID parameter. Civic Platform Object Model provides additional details on the capID parameter. The EMSE Javadocs contain details on the CapIDModel class. The com.accela.aa.aamain.cap package in the EMSE Javadocs contains the CapIDModel class and defines the constructor for this class as follows.

Parameter	Type
serviceProviderCode	string
ID1	string
ID2	string
ID3	string
customID	string
trackingID	long

matches

Version

1.3

Parameters

Parameter	Type	Description
eVal	string	String to match.
argList	strings	The m1 [, … mn] list. List of values to test for a match. Enter any number of values, each enclosed in double quotes and separated by comma.

Returns

Returns true if the function finds value in the m1 [, … mn] list. Function looks for an exact, case-sensitive match. Returns false if the function finds nothing in the m1 [, … mn] list that matches value.

nextWorkDay

Version

1.4

Parameters

Parameter	Type	Description
td (optional)	string	Date, in format "MM/DD/YYYY" (or any string that converts to a JavaScript date).

Returns

Returns the first agency work day following the current date, by checking the Agency Workday calendar defined for the agency. If the function uses the td parameter, returns the first agency work day following td. The date returned is a string in the format MM/DD/YYYY.

Notes

You can only use this function with Civic Platform 6.3.2 and later.

openUrlInNewWindow

Opens a new browser window and shows the web page whose URL is myurl.

Version

1.4

Parameters

Parameter	Type	Description
myurl	string	URL of web page to open.

Notes

Either user-configurable variable showDebug or showMessage must be true for this function to work.

parcelConditionExists

Version

1.4

Parameters

Parameter	Type	Description
condtype	string	Condition type.

Returns

Returns true if any parcel has a condition of type condtype; otherwise, returns false.

parcelExistsOnCap

Version

1.6

Parameters

Parameter	Type	Description
capId (optional)	CapIDModel	Record ID to check.

Returns

Returns true if a parcel exists on the record

paymentByTrustAccount

This function uses the trust account associated with a record to pay for a specific fee item.

Version

2.0

Parameters

Parameter	Type	Description
fSeqNbr	long	Sequence number of the fee item to be paid.
itemCap (optional)	CapIDModel	Optional target record ID.

Notes

The logic behind the function is:

- Retrieves the primary trust account on the record.

- Initiates payment from this trust account for the amount of the fee.

- If payment successful, applies payment to the fee.

- Generates a receipt for the payment.

- Returns false if any of the previous fails. Otherwise returns true.

- You can only pay invoiced fees.

Example

```
feeSeq = addFee("C","F","P",20,"Y");
paymentByTrustAccount(feeSeq);
```

paymentGetNotAppliedTot

Gets the total amount of unapplied payments on the current record (capId), as a float number.

Version

2.0

Parameters

None

proximity

Version

1.3

Parameters

Parameter	Type	Description
svc	string	GIS service name.
layer	string	GIS layer, that is, the object that the function is testing proximity to.
numDistance	integer	Distance of parcel, associated with the current record, to the object that you identify with the layer parameter.
unit (optional)	string	Unit for numDistance measurement. Optional. Default is feet.

Returns

Returns true if the parcel on the current record is within numDistance feet (or other unit specified) of the object in layer; otherwise, returns nothing.

proximityToAttribute

Version

1.4

Parameters

Parameter	Type	Description
svc	string	GIS service name.
layer	string	GIS layer, i.e., object that function is testing proximity to.
numDistance	integer	Distance of parcel, associated with the current record, to the object that you identify with the layer parameter.
distanceType	string	Unit for distance measurement.
attributeName	string	Attribute name.
attributeValue	string	Attribute value.

Returns

Returns true if the record has a GIS object in numDistance proximity that contains an attribute called attributeName with the value attributeValue.

Example

```
proximityToAttribute("flagstaff","Parcels","50", "feet","BOOK","107") ^
 DoStuff...
```

refLicProfGetAttribute

Version

1.4

Parameters

Parameter	Type	Description
pLicNum	string	State license number.
pAttributeName	string	Custom attribute name.

Returns

Returns the value of the custom attribute named pAttributeName for the reference Licensed Professional whose license # is pLicNum.

Notes

Note that pAttributeName is not necessarily the same as the attribute label. You can find the attribute name in the attribute's configuration screen.

If the function does not find a reference Licensed Professional with license # of pLicNum, the function returns NO LICENSE FOUND. If the function does not find the attribute pAttributeName, the function returns ATTRIBUTE NOT FOUND.

refLicProfGetDate

Version

1.4

Parameters

Parameter	Type	Description
pLicNum	string	State license number.
pDateType	string	Date field to retrieve. Options (use one): EXPIRE, ISSUE, RENEW, INSURANCE, BUSINESS.

Returns

Returns the date specified by pDateType for the reference Licensed Professional whose license # is pLicNum. The date returned is a JavaScript Date object.

Notes

The table below shows the date returned for each pDateType parameter value.

dateType	Date Field Value Returned
EXPIRE	License Expiration Date
ISSUE	License Issue Date
RENEW	License Last Renewal Date
INSURANCE	Insurance Expiration Date
BUSINESS	Business License Expiration Date

If the function does not find a reference Licensed Professional with license # of pLicNum, the function returns NO LICENSE FOUND. If the function does not find a date, the function returns NO DATE FOUND. If pLicNum is empty, the function returns INVALID PARAMETER. The function skips disabled reference Licensed Professional.

To format a JavaScript Date as a MM/DD/YYYY string, use function jsDateToMMDDYYYY.

removeAllFees

Removes all un-invoiced fees on the record

Version

1.6

Parameters

Parameter	Type	Description
itemCap	CapIDModel	The capIDModel of record.

removeASITable

Removes all entries for ASI Table Name

Version

1.5

Parameters

Parameter	Type	Description
tableName	string	Table name to remove.
capId (optional)	CapIDModel	Record ID object for record.

removeCapCondition

Deletes the condition whose type is cType and name is cDesc from the current record. If you use the optional parameter capId, the function deletes the condition from the record capId.

Version

1.5

Parameters

Parameter	Type	Description
cType	string	Condition type.
cDesc	string	Condition name.
capId (optional)	CapIDModel	Record ID object.

removeFee

Deletes all assessed fees with the fee code of fcode and fee period of fperiod. The function does not delete invoiced fees.

Version

1.4

Parameters

Parameter	Type	Description
fcode	string	Fee code of the fee to delete.
fperiod	string	Fee period of the fee to be delete.

removeParcelCondition

Removes the condition whose name is cDesc and type is cType from the reference parcel whose number is parcelNum. If you set the parameter parcelNum to null, the function removes any condition, whose name is cDesc and type is cType, from all parcels on the record.

Version

1.4

Parameters

Parameter	Type	Description
parcelNum	string	Parcel number from which to remove the condition.
cType	string	Condition type.
cDesc	string	Condition name.

removeTask

Dynamically edits the workflow on the indicated record by removing the task.

Version

2.0

Parameters

Parameter	Type	Description
targetCapId	CapIDModel	Record Id to edit.
removeTaskName	string	Name of the task to remove.
wfProcess (optional)	string	Workflow process name.

replaceMessageTokens

Used for formatting emails, this function parses through the string, replacing tokens with variable values.

Version

1.6

Parameters

Parameter	Type	Description
m	string	String to do the token replacement.

Notes

The function replaces values inside pipes (e.g. |capIdString|) by their script values.

The function replaces values inside curly brackets (e.g. {ASIVal}) by ASI values.

Example

```
EmailContent = "Thank you for submitting |capIDString| on |fileDate|. The
  balance due is |balanceDue|. The ASI field is {ASI Field}"
```

```
EmailSend = replaceMessageTokens(EmailContent);
```

This function can access any variable that the script uses.

resultInspection

This function posts a result for a scheduled inspection. If no scheduled inspection exists (of that type for the record) then the function does nothing.

Version

1.6

Parameters

Parameter	Type	Description
inspType	string	Inspection type to result.
inspStatus	string	Resulting status.
resultDate	string	Posted date of the result.
resultComment	string	Comment to add to the result.
capId (optional)	CapIDModel	Record ID to result.

scheduleInspectDate

Schedules the inspection iType for the date DateToSched. If you supply inspectorID, the function assigns the scheduled inspection to the inspector whose Civic Platform user ID is inspectorID.

Version

1.5

Parameters

Parameter	Type	Description
iType	string	Inspection type.
DateToSched	string	Scheduled date of inspection.
inspectorID (optional)	string	User ID of inspector.
inspTime (optional)	string	Inspection time in HH12:MIAM format or AMPM (e.g. "12:00PM" or "PM").
inspComm (optional)	string	Inspection comment.

Note

To specify the optional inspection time without passing in inspection use scheduleInspectDate("Desc","01/01/2001",null, "AM").

To specify the option inspection comment without the other option parameters you can use scheduleInspectDate("Desc","01/01/2001",null,null, "My Comment").

scheduleInspection

Schedules the inspection iType for DaysAhead days after current date. If you supply inspectorID, the function assigns the scheduled inspection to the inspector whose Civic Platform user ID is inspectorID.

Version

1.5

Parameters

Parameter	Type	Description
iType	string	Inspection type.
DaysAhead	number	Number of days in the future to schedule the inspection for.
inspectorID (optional)	string	User ID of inspector.
inspTime (optional)	string	Inspection time in HH12:MIAM format or AMPM (e.g. "12:00PM" or "PM").
inspComm (optional)	string	Inspection comment.

Notes

To specify the optional inspection time without passing in inspection use scheduleInspectDate("Desc",5,null,"AM").

To specify the option inspection comment without the other option parameters you can use scheduleInspectDate("Desc",5,null,null,"My Comment");

searchProject

Searches the entire hierarchy on the current record for related records that match the criteria.

Version

1.6

Parameters

Parameter	Type	Description
pProjType	app type string	Record type marking highest point to search. Ex. Building/Project/NA/NA.
pSearchType	app type string	Record type to search for. Ex. Building/Permit/NA/NA.

Returns

Returns CapID array of all unique matching SearchTypes

setIVR

Sets the record tracking number for IVR

Version

1.6

Parameters

Parameter	Type	Description
ivrnum	long	New IVR tracking number.

setTask

Helper function to edit the active and complete flags on a task.

Version

2.0

Parameters

Parameter	Type	Description
wfstr	string	Name of the task to edit.
isOpen	string	Edits the "active" flag on the task. If "Y" activate the task, if "N" close the task.
isComplete	string	Edits the "complete" flag on the task. If "Y" set the task to complete. If "N" set the task to incomplete.
processName (optional)	string	Optional process name that the target task resides in.

Example

To set a task to inactive/complete:

```
setTask("Peer Review","N","Y");
```

stripNN

Strips all non-numeric characters from the string. Only numerals and the period character remain.

Version

1.6

Parameters

Parameter	Type	Description
fullStr	string	String to strip.

taskCloseAllExcept

Closes all tasks on the record except for tasks in the list wfTask1… wfTaskn. If you only supply the parameters pStatus and pComment, the function closes all tasks on the record.

Version

1.4

Parameters

Parameter	Type	Description
pStatus	string	Status to assign to tasks.
pComment	string	Status comment to add to tasks.
wfTask1 … wfTaskn (optional)	string	Names of tasks to exclude. Enter one or more tasks separated by commas, each in double-quotes.

Notes

Before the function closes each task, the function updates the task as follows:

- Status = pStatus

- Status Date = current date

- Status Comment = pComment

- Action By = current user

taskStatus

Version

1.3

Parameters

Parameter	Type	Description
wfstr	string	Workflow task name.
wfProcess (optional)	string	ID (R1_PROCESS_CODE) for the process that the task belongs to.
capId (optional)	CapIDModel	Record ID object for record to use.

Returns

Returns the status of the workflow task wfstr.

Notes

If record's workflow contains duplicatescheduleins wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to check.

If you use the parameter capId, the function retrieves data from the record capId.

taskStatusDate

Version

1.5

Parameters

Parameter	Type	Description
wfstr	string	Workflow task name.
wfProcess (optional)	string	ID (R1_PROCESS_CODE) for the process that the task belongs to.
capId (optional)	CapIDModel	Record ID object for record to use.

Returns

Returns the current status date of the workflow task wfstr.

Notes

If record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to use.

If you use the parameter capId, the function retrieves data from the record capId.

transferFunds

Version

1.3

Parameters

Parameter	Type	Description
parentAppNum	string	Record number to transfer funds to.
dollarAmount	number: double	Amount to transfer.

Notes

If the current record has sufficient funds (i.e. non-applied amount), transfers dollarAmount from the current record to the record parentAppNum. The function records the transaction as a Fund Transfer transaction on both records. If current record does not have sufficient funds, no fund transfer takes place.

updateAddresses

Updates the address in a record.

Version

2.0

Parameters

Parameter	Type	Description
targetCapID	CapIDModel	Record ID object.
addressModel	AddressModel	Address.

updateAppStatus

Updates record status of record to stat and adds cmt to the status update history.

Version

1.3

Parameters

Parameter	Type	Description
stat	string	Status to update the record to.
cmt	string	Comment to add to status update history.
capId (optional)	CapIDModel	Record ID object.

Notes

If you use the capId optional parameter, the function updates record capId. If you do not use the capId parameter, the function updates current record.

The getApplication( ), getParent( ), createChild(), createCap() functions each return a record ID object that you can use in the capId parameter.

updateFee

Version

1.5

Parameters

Parameter	Type	Description
fcode	string	Fee code of the fee to be updated/added.
fsched	string	Fee schedule of the fee to be updated/added.
fperiod	string	Fee period of the fee to be updated/added.
fqty	integer	Quantity to be updated/added.
finvoice	string	Flag for invoicing ("Y" or "N").
pDuplicate (optional)	string	Allow duplicate invoiced fee ("Y" or "N").
pFeeSeq (optional)	integer	Attempts to update a specific fee item.

Returns

For an updated fee, the function returns null. For an added fee, the function returns the fee sequence number.

Notes

If a fee whose fee code is fcode and fee period is fperiod has been assessed and not invoiced, updates the quantity on the fee to fqty. If invoice is Y, then invoices the fee. If there is more than one assessed fee with fcode and fperiod, updates the first fee found. If the fee is not found, adds the fee.

If this fee already exists and is invoiced, adds another instance of the same fee, unless *pDuplicate* is N. The duplicate fee has an adjusted quantity, which is fqty less quantity on previous fee.

If you use the *pFeeSeq* parameter, the function attempts to find the specified fee. If the function does not find the specified fee sequence number, the function adds a new fee based on the pDuplicate fee flag.

Warning: If adjusted quantity can be negative, do not use this function to add a fee. Civic Platform's cashier feature does not handle negative fees well. Set *pDuplicate* parameter to N.

updateRefParcelToCap

Refreshes parcel data on the specified record. The function refreshes parcel data on the record with reference parcel values.

Version

1.6

Parameters

Parameter	Type	Description
capId (optional)	CapIDModel	Record ID to process.

updateShortNotes

Updates the short notes on the specific capId detail record

Version

1.6

Parameters

Parameter	Type	Description
newSN	string	New short notes value.
capId (optional)	CapIDModel	Record ID to update.

updateTask

Updates the workflow task wfstr as follows:

- Status = wfstat

- Status Date = current date

- Status Comment = wfComment

- Action By = current user

Version

1.3

Parameters

Parameter	Type	Description
wfstr	string	Name of workflow task to update.
wfstat	string	Status to update task to.
wfComment	string	Comment to update status comment to.
wfnote	string	Note to update task note to.
wfProcess (optional)	string	Workflow process that wfstr belongs to.
capId (optional)	CapIDModel	Record ID object.

Notes

The workflow does not promote to the next task. To promote the workflow to the next task, use the closeTask, branchTask or loopTask function.

If record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to check.

If you use the capId parameter, the function updates the record capId. If you use the capId parameter, you must use the wfProcess parameter by entering a process string or entering the word null.

updateTaskAssignedDate

Updated the assigned date of the workflow task wfstr. The function does not create a workflow history record.

Version

1.6

Parameters

Parameter	Type	Description
wfstr	string	Workflow task to edit.
wfAssignDate	string	New assignment date.
wfProcess (optional)	string	Process name of workflow for wfstr. Case sensitive.

Notes

If record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to activate.

updateTaskDepartment

Updated the assigned department for the workflow task wfstr. The function does not create a workflow history record.

Version

1.6

Parameters

Parameter	Type	Description
wfstr	string	Workflow task to edit.
wfDepartment	string representing department	New department code.
wfProcess (optional)	string	Process name of workflow for wfstr. Case sensitive.

Notes

If record's workflow contains duplicate wfstr tasks, use parameter wfProcess to specify the process or subprocess whose wfstr to activate.

Assigned department must be a string with 7 values separated by slashes, such as "ADDEV/DPE/ONLINE/ LICENSE/NA/NA/NA"

updateWorkDesc

Updates the work description on the specific capId detail record.

Version

1.6

Parameters

Parameter	Type	Description
newWorkDes	string	New work description value.
capId (optional)	CapIDModel	Record ID to update.

validateGisObjects

Version

1.3

Parameters

None

Returns

Returns true if all GIS objects on the current record validate in GIS, or false if any GIS object on the current record does not validate in GIS.

workDescGet

Version

1.4

Parameters

Parameter	Type	Description
pCapId	CapIDModel	Record ID object for record.

Returns

Returns work description for the record whose record ID object is pCapId.

Notes

The getApplication(), getParent(), createChild(), createCap() functions each return a record ID object.

zeroPad

Version
1.6

Parameters

Parameter	Type	Description
num	string	Number to zero pad.
count	integer	Number of digits required.

Returns
A zero-padded string of the supplied number that is count digits long.

Example

```
zeroPad("5",4) = "0005"
```

Master Script Object List

Related Information

Fee

genericTemplateObject

guideSheetObject

licenseProfObject

licenseObject

Task

Fee

Defines the a fee object for use by fee functions, loadFees for example.

Parameters

sequence	code	description
unit	amount	amountPaid
applyDate	effectDate	expireDate
status	recDate	period
display	accCodeL1	accCodeL2
accCodeL3	formula	udes
UDF1	UDF2	UDF3
UDF4	subGroup	calcFlag
calcProc	auditDate	auditID
auditStatus		

genericTemplateObject

You can use this object to interact with the Application Specific Information and Application Specific Information Tables stored as generic template information on licensed professionals and conditions

Version

2.0

Constructors

Loads the genericTemplate objects and makes object data accessible through the read only parameters.

Parameter	Type	Description
gtmp	genericTemplateModel	Generic template model from which to read information from.

Example

```
var cond = aa.capCondition.getCapCondition(capId,445392).getOutput();
var tmpObj = genericTemplateObject(cond.getTemplateModel());
```

Parameters

Parameter	Description
ASI	An associative array comprised of data from all the ASI fields that the generic template contains as an associative array. The constructor sets the associated hasASI flag to true if the function finds valid ASI fields during creation. The object stores the associative array as ASI[label name] format.
ASIT	An associative array comprised of data from all the ASIT fields that the generic template contains as an associative array. The constructor sets the associated hasASIT flag to true if the function finds valid ASI fields during creation. The object stores this table in the ASIT[tableName][row][column] format.
hasASI	Boolean flag set to indicate if object has valid ASI loaded.true = valid ASI found false = no ASI found
hasTables	Boolean flag set to indicate if object has valid ASIT loaded.true = valid ASIT found false = no ASIT found

Example

```
If(tmpObj.hasASI)
   var tmpObj = tmpObj.ASI["My ASI Field"];
//List all ASI
If(tmpObj.hasASI)
    For(a in tmpObj.ASI)
       logDebug(a + " : " + tmpObj.ASI[a]);
```

Example

```
//List all ASI Table values
If(tmpObj.hasTables)
    for(table in tmpObj.hasASIT)
       for(row in tmpObj.hasASIT[table])
          for(col in tmpObj.hasASIT[table][row])
             logDebug(table + " : " + row + " : " + col + " : " +
 tmpObj.hasASIT[table][row][col];
```

guideSheetObject

A helper object which represents the data that the guidesheet contains. You can retrieve these objects for a given inspection. Each guide item is represented as a separate object.

You can use this object with Inspection Guidesheets to simplify the interaction with the various guidesheet items and to expose the Applications Specific Information and Application Specific Information Tables for use.

Version

2.0

Constructors

Loads the guideSheetObject for the provided guideSheet and guideSheetItem.

Constructor	Type	Description
gguidesheetModel	gguideSheetModel	Guidesheet object to retrieve.
gguidesheetItemModel	gguidesheetItemModel	Guidesheet item to retrieve.

Example

```
var guideObj = guideSheetObject(guideSheet, guideItem);
```

Parameters

Name	Description
gsType	Guidesheet type.
gsSequence	Guidesheet system sequence number.
gsDescrption	Guidesheet description.
gsIdentifier	Guidesheet identifier.
item	Guidesheet item model object.
text	Guidesheet item text identifier.
status	Guidesheet item status value.
comment	Guidesheet item comments.
score	Guidesheet item score value.
info	Guidesheet item application specific information values. Use the loadInfo method to load.
infoTables	Guidesheet Item application specific information table values. Use the loadInfoTables method to load.
validTables	Boolean value that determines if valid tables exist in the guideSheetObject. True if infoTables has data (item has ASIT).
validInfo	Boolean value that determines if valid application specific information exist in the guideSheetObject. True if info has data (item has ASI)

Methods

Name	Parameters	Description
loadInfo	None	This method populates the info parameter with the Application Specific Information contained in the guidesheet item model.
infoTables	None	This method populates the infoTables parameter with the Application Specific Information Table data that the guidesheet item model contains.

licenseProfObject

You can use this object to interact with the reference licensed professional entities in Civic Platform and to provide many methods to streamline the most common interactions.

Version

2.0

Constructors

Populates licenseProfObject with the license number and license type.

Constructor	Type	Description
licnumber	string	License number to retrieve. This number is the RSTATE_LIC value.
lictype	string	License type to retrieve.

Example

```
var myLic = licenseProfObject("1234","Business");
```

Parameters

Parameter	Description
attribs	An associate array populated with all the valid licensed professional attributes. When valid attributes exist the validAttrs flag sets to true indicating values are available. Use the getAttribute and setAttribute methods to access the licensed professional attribute instead of directly accessing the attribs parameter.
infoTables	This parameter exposes the people info tables multiple dimension array of the following format: infoTables[tableName][row][column]. To access the value of this field you must use the getValue() for the column and to set the value you must use the setValue(val). To add or delete rows please review the methods section for addTableRow(), removeTable(), and removeTableRow().
refLicModel	This parameter loads on object creation and provides direct access to the licensed professional model.
valid	Boolean flag set to indicate if object has a valid reference license professional loaded.
	true = valid professional found
	false = no professional found
validAttrs	Boolean flag set to indicate if object has valid reference licensed professional attributes loaded.
	true = valid professional found
	false = no professional found
validTables	Boolean flag set to indicate if object has valid people info tables loaded.
	true = valid professional found
	false = no professional found

Example (attribs)

```
Ex.
if(myLic.validAttrs)
   var myValu = myLic.attribs["Is Valid Business?"];

//List attributes
if(myLic.validAttrs)
    for(attrib in myLic.attribs)
        logDebug(attrib + " : " + myLic.attribs[attrib]);
```

Example (infoTables)

```
//get value
myLic.infoTables["Codes"][0]["Type"].getValue();

//set value
myLic.infoTables["Codes"][0]["Type"].setValue("Type III");

//list all values
If(myLic.validTables)
    for(table in myLic.infoTables)
       for(row in myLic.infoTables[table])
           for(col in myLic.infoTables[table][row])
               logDebug(table + " : " + row + " : " + col + " : " +
 myLic.infoTables[table][row][col].getValue();
```

Example (refLicModel)

```
myLic.refLicModel.getLicenseType();
```

Example (valid)

```
var myLic = licenseProfObject("1234","Business");
if(myLic.valid)
  //do actions
```

Methods

- addTableFromASIT

  This method copies ASI Tables to reference licensed professional people info tables. This method attempts to add all rows from the ASI Table array to the people info table array for all matching columns.

  - Parameters

Parameter	Type	Description
tableName	string	Name of people info table.
ASITArray	ASIT Array	ASI table array that master script loads.

  - Return

    If ASI Table loads successfully into the people info tables, the method returns true. If the load fails the method returns false.

  - Example

    ```
    myLic.addTableFromASIT("myTable", CERTIFICATIONS);
    ```

- addTableRow

  Add a new row to the people info table utilizing an associative string array.

  - Parameters

Parameter	Type	Description
tableName	string	Name of people info table.
valueArray	string array	Associative string array where the index name is the column name to load.

  - Example

    ```
    var newRow = new Array();
    ```

```
newRow["Column1"] = "A";
newRow["Column2"] = "B";
myLic.addTableRow("myTable",newRow);
myLic.updateRecord();
```

- copyToRecord

  Copies the current reference licensed professional to the specified record id.

  - Parameters

Parameter	Type	Description
capId	CapIDModel	Record to copy the licensed professional to.
replace	boolean	Flag if existing LP should be replace if found.

  - Example

    ```
    myLic.copyToRecord(capId,true);
    ```

- disable

  Disables the licensed professional

  - Parameters
    None

- enable

  Enables the licensed professional

  - Parameters
    None

- getAssociatedRecords

  Retrieves all records associated to the reference licensed professional in an array.

  - Parameters
    None

  - Example

    ```
    var capArray = myLic.getAssociatedRecords();
    ```

- getAttribute

  Get method for getting a licensed professional attribute value.

  - Parameters

Parameter	Type	Description
attributeName	string	Reference license professional attribute name.

  - Notes

    Method handles error checking. Use this method instead of directly accessing the parameter.

  - Example

    ```
    var val = myLic.getAttribute("myValue");
    ```

- getMaxRowByTable

Gets the max row number for a people info table.

- Parameters

Parameter	Type	Description
tableName	string	People info table name to get the max row from.

- Return
Returns -1 if no rows exist.

- refreshTables

Refreshes the people info table arrays in the object with the data found in database.

- Parameters
None

- removeTable

Removes all rows from a people info table.

- Parameters

Parameter	Type	Description
tableName	string	People info table name to remove.

- removeTableRow

Removes provided row index from provided table.

- Parameters

Parameter	Type	Description
tableName	string	People info table name to remove row from.
rowIndex	long	Row index to remove.

- Return
If method removes the row, returns true. Otherwise, returns false.

- setAttribute

Sets a reference license professional attribute to the provided value and performs error checking.

- Parameters

Parameter	Type	Description
attributeName	string	Reference license professional attribute name.
attributeValue	string	Reference license professional attribute value to set.

- Return
If method sets value, returns true. Otherwise, returns false.

- Example

```
If( myLic.setAttribute("myValue","newValue") )
    logDebug("Value Updated");
```

- setDisplayInACA4Table

Sets the flag to display the reference people info table in Citizen Access.

- Parameters

Parameter	Type	Description
tableName	string	Name of the people info table.
visibleFlag	string	Valid flag values are Y to display the table in Citizen Access or N to hide the table from Citizen Access.

- setTableEnabledFlag

Sets the enabled flag displayed on the people info tables to yes or no for the provided table row.

- Parameters

Parameter	Type	Description
tableName	string	People info table name to remove row from.
rowIndex	long	Row index to remove.
isEnabled	boolean	Enabled flag.

- Return

Returns true if update is successful.

- Example

```
myLic.setTableEnabledFlag("myTable",0,false);
```

- updateFromAddress

This method updates the reference professional with the address information from the provided record.

- Parameters

Parameter	Type	Description
capId	CapIDModel	Record to get the address information from.

- Return

If update is successful the method returns true, otherwise the method returns false.

- Notes

The method first attempts to use the primary address. If no primary address exists the method selects the first address available on the Record.

If the method finds an address the method then attempts to copy the Address Line 1, Address Line 2, City, State, and Zip to the reference licensed professional. In the event an Address Line 1 is not available it attempts to create the line one by concatenating the house number, street direction, street name, street suffix, unit type, and unit number.

- updateFromRecordContactByType

This method attempts to update the contact information on a reference licensed professional from a record contact.

- Parameters

Parameter	Type	Description
capId	CapIDModel	Record to get the contact information from.
contactType	string	Contact type to search, use "" for primary.
updateAddress	boolean	Set to true to update address information.
updatePhoneEmail	boolean	Set to true to update phone information and email information.

- Return

  If the update is successful the method returns true. If the update fails it returns false.

- Notes

  To attempt to use the primary contact use an empty string ("") from the contact type. If you provide a contact type and there are multiple with the same contact type, the method uses the first occurrence of the contact type in the event .

  When found the method updates the first, middle, last, and business name on the reference licensed professional with the first, middle, last, and business name of the contact record.

  If the updateAddress flag is true then the method attempts to copy the address line 1, address line 2, address line 3, city, state, and zip from the contact record to the associate fields of the reference licensed professional.

  If the updatePhoneEmail flag is true then the method also attempts to copy the phone1, phone2, phone3, email, and fax to the associate fields on the reference licensed professional record.

- updateFromRecordLicensedProf

  This method attempts to update the reference licensed professional utilizing a transactional licensed professional.

  - Parameters

Parameter	Type	Description
capId	CapIDModel	Record to get the license professional information from.

  - Return

    If the update is successful the method returns true. If the update fails it returns false.

  - Notes

    This method searches the provided record for a transactional license professional of the same number and the same type. If the method finds a match, the method attempts to copy all licensed professional information from the transactional record to the reference record.

- updateRecord

  This method commits all changes made to the reference licensed professional object to the database.

  - Parameters

    None

  - Return

    If the update is successful the method returns true. If the update fails it returns false.

  - Notes

    If you do not invoke this method, you lose all updates made to the licensed professional prior to the last update.

  - Example

    ```
    myLic.updateRecord();
    ```

licenseObject

This function creates a helper object that you can use to view and modify license information and expiration information.

Version

1.6

Constructors

Parameter	Type	Description
licNumber	string	State license number of the reference licensed professional to link to the license object.
capId (optional)	CapIDModel	Record ID to use for the license object. Identifies the record from which to load renewal information.

Notes

This constructor populates the licenseObject for the license number specified and the currently loaded capId. If licNumber has a value, the helper object attempts to replicate changes to a reference license professional, as well as the record.

Parameters

Parameter	Description
refProf	The referenced licensed professional.
b1Exp	Contains the b1 record (renewal status on record).
b1ExpDate	Returns the license expiration date in mm/dd/yyyy format (read only).
b1ExpCode	Returns the expiration code.
b1Status	Returns the license renewal status (read only).
refExpDate	Returns the license professional expiration date in mm/dd/yyyy format (read only).
licNum	The license number.

Example

```
var licObj = licenseObject("1234");
```

Methods

- getCode

  Gets the expiration status of the record.

  - Parameters
    None

  - Return
    Returns the expiration code configured for the license.

  - Example

    ```
    var licObj = licenseObject("1234");
    var code = licObj.getCode();
    ```

- getStatus

  Gets the expiration status of the record.

  - Parameters

None

- Return

  Returns the expiration status

- Example

```
var licObj = licenseObject("1234");
var status = licObj.getStatus();
```

- setExpiration

  Sets the expiration date on the license record and associate reference license professional to the provided value.

  - Parameters

Parameter	Type	Description
expDate	string	Expiration date in string format.

  - Example

```
licObj.setExpiration("01/01/2020");
```

- setIssued

  Sets the issued date on the license record and associate reference license professional to the provided value.

  - Parameters

Parameter	Type	Description
issDate	string	Issued date in string format.

  - Example

```
licObj.setIssued("01/01/2000");
```

- setLastRenewal

  Sets the renewed date on the license record and associate reference license professional to the provided value.

  - Parameters

Parameter	Type	Description
renewDate	string	Renewed date in string format.

  - Example

```
licObj.setLastRenewal("01/01/2000");
```

Task

Defines the task object for use by task functions, loadTasks for example.

Parameters

status	comment	note
statusdate	process	processID
step	active	

Example Expression Script

This following example illustrates how to define an expression that uses a state agency web service to validate a licensed professional.

In the example, you manually enter an EMSE script in the script mode window of Expression Builder for a selected Professional Execute Fields. The EMSE script verifies the license type and license number on a new application. The expression also updates the licensed professional license to the most current information, for example, status, expiration date, and address.

> 📝 **Note:**
>
> You must build or deploy a Web Service Stub to interface with the external Web Services. For more information about building a Web service stub for your agency, see the *Creating an External Web Service Stub (08ACC-04275) – Accela Automation Technical Bulletin.pdf*.

To verify the licensed professional

1. Navigate to the Building portlet.

2. Select a permit.

3. Click the **Professionals** tab.

   Civic Platform displays the Professionals tab in the detail portlet

4. Click the **New** button to enter the licensed professional information.

5. Select License Type from the drop-down list, or enter a License #.

   If the license number and license type validate successfully, Civic Platform creates the new record. If the license number and license type are not valid, Civic Platform displays the EMSE error message on the Professionals tab.

To create the EMSE script to validate licensed professionals

1. Create a New expression and navigate to the Expression Name field.

   Civic Platform displays the New Expression fields where you define the criteria.
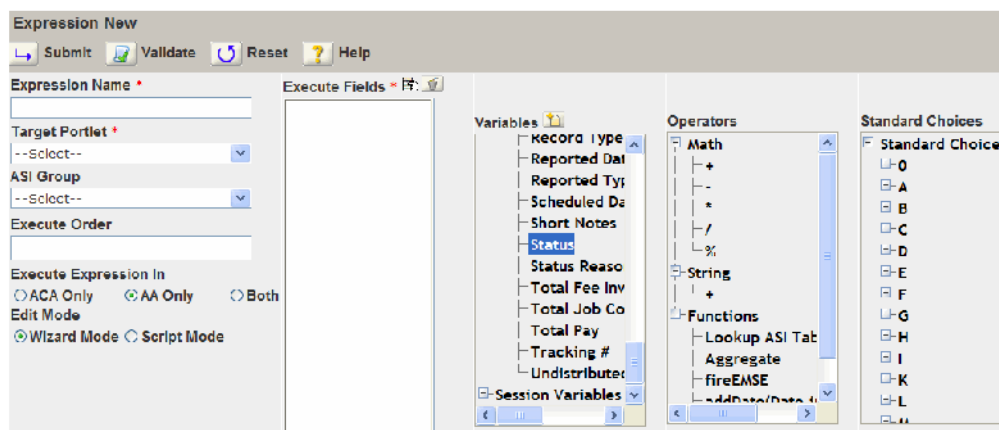
2. Enter an **Expression Name**. This scenario uses

```
Licensed Professional Validation
```

3. Select Record Detail from the Target Portlet drop-down list.

This step specifies that the expression takes effect in the Record detail portlet or application for the selected record type. This scenario uses the record type

```
Building/Building Permit/Commercial/All Categories
```

In general, you do not need to perform this step.

4. Select

```
Script Mode
```

in the **Edit Mode** section.

Civic Platform re-populates the page to display the Script fields.


5. Select

```
Professional
```

from the **Target Portlet** drop-down list.

This step specifies that the expression takes effect in the Professional portlet for the selected record type.

6. In the **ASI Group** field, select the group that contains the record type and the fields for which you want to create an expression.
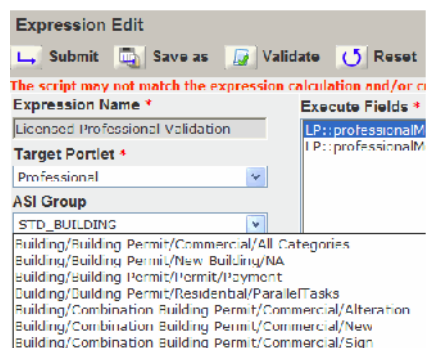
7. Use the Variables section to specify the fields affected by the expression.

   Civic Platform displays ASI, Professional, Record Detail, and Session Variables in the Variables field.
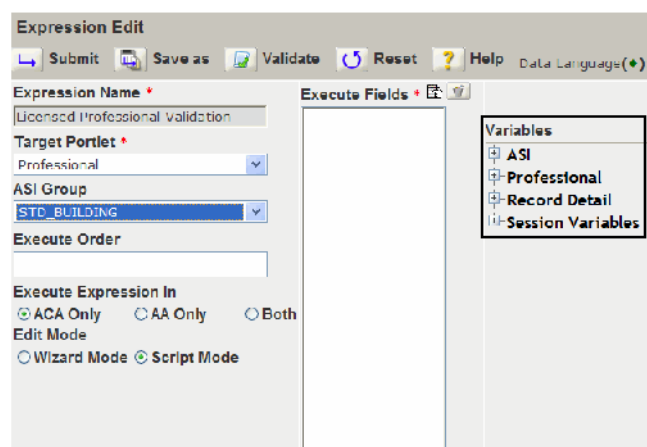
8. Click the **Execute Fields** list picker.

   A pop-up window displays the Execute Fields list.
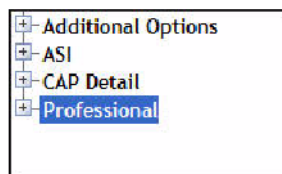
9. Expand **Professional** and click the **License #** and **License Type** options.

   Civic Platform loads the License # and License Type options in the Execute Fields list.
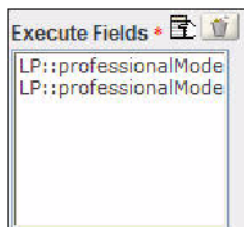
10. In the Script field, enter the EMSE script. The script for this scenario is:

```
/*-----------------------------------------------------------------/
| Program : LicProfAddBefore.js
| Event   : LicProfAddBefore
|
/-----------------------------------------------------------------*/
var LicProfModel = aa.env.getValue("LicProfModel");
var licenseType = LicProfModel.licenseType;
var licenseNbr = LicProfModel.licenseNbr;
var licenseValidateReturnCode = "0";
var licenseValidateReturnMessage = "Follow Licenses are invalid:";
if (!validateLicense(licenseType, licenseNbr))
{
licenseValidateReturnCode = "-1";
licenseValidateReturnMessage += "<br>";
licenseValidateReturnMessage += " * License type: " + licenseType;
licenseValidateReturnMessage += " , License number: " + licenseNbr;
}
// check whether something wrong
if (licenseValidateReturnCode != "0")
{
aa.env.setValue("ScriptReturnCode", licenseValidateReturnCode);
aa.env.setValue("ScriptReturnMessage", licenseValidateReturnMessage);

}
// check whether the licenseType and licenseNbr is valid.
function validateLicense(licenseType, licenseNbr)
{
var accelawsUrl = 'https://www4.cbs.state.or.us/exs/bcd/accela/ws/
accelaws.cfc?method=lic_valid&returnformat=json';
var client = aa.httpClient;

// set url parameters
var params = client.initPostParameters();
params.put('p_lic_type', licenseType);
params.put('p_lic_num', licenseNbr);

// do validate via web service
var scripResult = client.post(accelawsUrl, params);

// check the return value
if (scripResult.getSuccess())
{
var resultString = String(scripResult.getOutput());

//Convert to jsonObject
var result = eval("("+resultString+")");
var valid = String(result["VALID"]);
if (valid.toUpperCase() == "TRUE")
{
return true;
}
}
else
{
aa.print("ERROR: Failed to validate license: " +
scripResult.getErrorMessage());
return false;
}
return false;
}
```

11. Click the **Validate** button to check the EMSE script for errors.

12. Click the **Submit** button.

Example External Document Review Script

The following sample script illustrates the use of the ExternalDocReviewCompleted event. This event is fired by the Check-In Document Review Construct API that is called by a 3rd party document review application when it checks in a reviewed document. As shown in the following sample code snippet, an agency can use this event to script any of the following use cases:

- Automatically update the status of the original document.

- Activate the next workflow status

- Send a notification email to the citizen when the reviewed file is checked in

```
var servProvCode = aa.env.getValue("ServiceProviderCode");
var capType = aa.env.getValue("RecordType").toString();
var checkInDocuments = aa.env.getValue("CheckedinDocuments");
var altID = aa.env.getValue("AlternateID");
var capStatus = aa.env.getValue("CapStatus");

var capId = getCapId();    // CapId object
var year = aa.date.getCurrentDate().getYear();
var month =aa.date.getCurrentDate().getMonth();
var day = aa.date.getCurrentDate().getDayOfMonth();
var currentDate = month+"/"+day+"/"+year;
var sysDate = aa.date.getCurrentDate();
var currentDate =
 dateFormatted(sysDate.getMonth(),sysDate.getDayOfMonth(),sysDate.getYear(),"");
var error = "";
var message = "";
var br = "<br>";

var checkedinFileName='';
var docSeqNbr = '';
var firstName ='';
var middleName ='';
var lastName = '';
var planReview="Plan Review";
var needCorrection="Need Corrections";
var wfObj = aa.workflow.getTasks(capId).getOutput();
var stepnumber;
var processID;
var workflowTaskStatus = capStatus;

aa.print("RecordType      =:" + capType);
aa.print("ServiceProviderCode  =:"+ servProvCode);
aa.print("Alt ID  =:"+ altID);
aa.print("CapStatus  =:"+ capStatus);
aa.print("PreviousDocID  =:"+ aa.env.getValue("PreviousDocID"));

/*------------------------------------------------------------------/
| Update the original document status by previous document id.
/------------------------------------------------------------------*/
var previousDocID  = aa.env.getValue("PreviousDocID").toString();
var result = aa.document.getDocumentByPK(previousDocID);
if(result.getSuccess())
{
   var orgDocModel  = result.getOutput();
   if(orgDocModel!= null)
```

```
      {
      orgDocModel.setDocStatus("");
      aa.document.updateDocument(orgDocModel);


      }
}

/*----------------------------------------------------------------------/
| Get the checked-in document name.
/----------------------------------------------------------------------*/
   if(checkInDocuments != null && checkInDocuments.length > 0)
   {
   for(var i = 0; i < checkInDocuments.length; i++)
   {
      var documentModel = checkInDocuments[i];
         docSeqNbr = documentModel.getDocumentNo();
         checkedinFileName = documentModel.getFileName();
   }
   }

/*----------------------------------------------------------------------/
| Update the current workflow task Plan Review to Need Corrections.
/----------------------------------------------------------------------*/
for (i in wfObj)
{
   fTask = wfObj[i];
   //complete the "Plan Review" task
   if (planReview.equals(fTask.getTaskDescription()))
   {
   aa.print("Complete the task: Plan Review");
   stepnumber = fTask.getStepNumber();
   processID = fTask.getProcessID();
   aa.workflow.adjustTask(capId, stepnumber, processID, "N", "Y", null,
 null);
   }
   if (needCorrection.equals(fTask.getTaskDescription()))
   {
   aa.print("Active the task: Need Corrections");
   stepnumber = fTask.getStepNumber();
   processID = fTask.getProcessID();
   aa.workflow.adjustTask(capId, stepnumber, processID, "Y", "N", null,
 null);
   }
}


/
*-------------------------------------------------------------------------------
| Send email to applicant.
/-------------------------------------------------------------------------------
*/
var capContactResult = aa.people.getCapContactByCapID(capId);
if (capContactResult.getSuccess())
{
   var email = '';
   var primaryFlag ='';
   var Contacts = capContactResult.getOutput();
   for (var contactIdx in Contacts)
   {
   primaryFlag = Contacts[contactIdx].getCapContactModel().getPrimaryFlag();
   email = Contacts[contactIdx].getCapContactModel().getEmail();
   firstName = Contacts[contactIdx].getCapContactModel().getFirstName();
   middleName = Contacts[contactIdx].getCapContactModel().getMiddleName();
   lastName = Contacts[contactIdx].getCapContactModel().getLastName();
```

```
    if ('Y'== primaryFlag)
    {
       if(email !='')
       {
       break;
       }
    }
    }
}
var host = "https://10.50.70.35:5443";
var documentLink = host + "/portlets/document/documentReviewCommentsList.do?
mode=list&entityType=CAP&from=CAP&docSeqNbr="+ docSeqNbr;

var mailFrom = "shell.wang@achievo.com";
var mailCC = "Cindy.q@beyondsoft.com";

var subject ="Plan Review Result for your Application " + altID;
var applicant = buildFullName(firstName, middleName,lastName);
sendNotificationEmail(mailFrom, email,mailCC, subject,documentLink,
 applicant,capType, altID,workflowTaskStatus,checkedinFileName);

function buildFullName(firstName,middleName,lastName)
{
   var fullName = "";
   if(firstName && firstName != null)
   {
   fullName +=firstName;
   }
   if(middleName && middleName != null)
   {
   fullName += " "+ middleName
   }
   if(lastName && lastName != null)
   {
   fullName += " "+ lastName
   }
   return fullName;
}

function sendNotificationEmail(from, to,cc, subject,documentLink,
 applicant,capType, altID,workflowTaskStatus,checkedinFileName)
{

   var templateName = "EXTERNALDOCREVIEWCOMPLETED";
   var emailParameters = aa.util.newHashtable();
   addParameter(emailParameters, "$$applicant$$",applicant);
   addParameter(emailParameters, "$$AltID$$",altID);
   addParameter(emailParameters, "$$CapType$$",capType);
   addParameter(emailParameters, "$$workflowTaskStatus$
$",workflowTaskStatus);
   addParameter(emailParameters, "$$reviewLink$$",documentLink);
   addParameter(emailParameters, "$$subject$$",subject);

   sendNotification(to, templateName, emailParameters, null);
}

function getCapId()  {

    var s_id1 = aa.env.getValue("PermitId1");
    var s_id2 = aa.env.getValue("PermitId2");
    var s_id3 = aa.env.getValue("PermitId3");

    var s_capResult = aa.cap.getCapID(s_id1, s_id2, s_id3);
    if(s_capResult.getSuccess())
```

```
      return s_capResult.getOutput();
    else
    {
      logMessage("**ERROR: Failed to get capId: " +
 s_capResult.getErrorMessage());
      return null;
    }
  }

function dateFormatted(pMonth,pDay,pYear,pFormat)
{
   var mth = "";
   var day = "";
   var ret = "";
   if (pMonth > 9)
   mth = pMonth.toString();
   else
   mth = "0"+pMonth.toString();

   if (pDay > 9)
   day = pDay.toString();
   else
   day = "0"+pDay.toString();

   if (pFormat=="YYYY-MM-DD")
   ret = pYear.toString()+"-"+mth+"-"+day;
   else
   ret = ""+mth+"/"+day+"/"+pYear.toString();

   return ret;
}

function addParameter(pamaremeters, key, value)
{
   if(key != null)
   {
   if(value == null)
   {
      value = "";
   }
   pamaremeters.put(key, value);
   }
}

function sendNotification(userEmailTo,templateName,params,reportFile)
{
   var result = null;
   result = aa.document.sendEmailAndSaveAsDocument(mailFrom, userEmailTo,
 mailCC, templateName, params, getCapScriptModel(), reportFile);
   if(result.getSuccess())
   {
   aa.log("Send email successfully!");
   return true;
   }
   else
   {
   aa.log("Fail to send mail.");
   return false;
   }
}

function getCapScriptModel()
{
   var s_id1 = aa.env.getValue("PermitId1");
```

```
   var s_id2 = aa.env.getValue("PermitId2");
   var s_id3 = aa.env.getValue("PermitId3");
   return aa.cap.createCapIDScriptModel(s_id1, s_id2, s_id3);
}

function addParameter(pamaremeters, key, value)
{
   if(key != null)
   {
   if(value == null)
   {
      value = "";
   }
   pamaremeters.put(key, value);
   }
}

function logMessage(str)
{
   message += str + br;
}

if (error && error.length > 0)
{
   aa.env.setValue("ScriptReturnCode", "1");
   aa.env.setValue("ScriptReturnMessage", error);
}
else
{
   aa.env.setValue("ScriptReturnCode", "0");
   aa.env.setValue("ScriptReturnMessage", message);
}
```

JavaScript Primer

The following sections introduce the basic concepts that you need to write scripts and understand scripts that others write. Accela uses JavaScript as the basis for the Civic Platform scripting engine. Accela has extended pure JavaScript to include features that allow you to interact directly with Civic Platform in your scripts.

Related Information

Understanding Scripts

Using Variables

Using Expressions

Controlling What Happens Next

Using Functions

Using Objects, Properties, and Methods

Understanding Scripts

To help you understand scripts, this section uses an example: a complete script that responds to a specific event. This section also includes information on writing scripts from scratch and a simple tool that can help writing scripts easier.

Topics:

- Our First Example

- Writing And Testing Our First Script

- Using Jext To Make Writing Scripts Easier

Our First Example

This example is a complete script that responds to an InspectionScheduleAfter event by inserting a new smart notice with information about the scheduled inspection. The example is several lines long, and contains comments at lines 1, 6, 9, 16, 19, and 26 that briefly explain what is happening in each section in the script.

Note that each line that begins with a comment starts with a double slash. The double slash tells Civic Platform to ignore that line. It is good practice to add comments to your scripts.

```
1    //Get the permit id.

2    permitId1 = aa.env.getValue("PermitId1");

3    permitId2 = aa.env.getValue("PermitId2");

4    permitId3 = aa.env.getValue("PermitId3");

6    //Prepare the smart notice label.

7    noticeLabel = "Inspection schedule";

9    //Get some information about the scheduled inspection.

10   numberOfInspections = aa.env.getValue("NumberOfInspections");

11   inspectionType = aa.env.getValue("InspectionType");

12   inspectionScheduleMode = aa.env.getValue("InspectionScheduleMode");

13   inspectionDate = aa.env.getValue("InspectionDate");

14   inspectionTime = aa.env.getValue("InspectionTime");

16   //Prepare label for smart notice.

17   noticeLabel = "Inspection Scheduled!";

19   //Prepare the text of the new smart notice.

20   noticeText = numberOfInspections + " Inspection(s) " +

21   inspectionType + " " +

22   inspectionScheduleMode + "d on " +

23   inspectionDate + " " +

24   inspectionTime + ".";

26   //Create the new smart notice using the information gathered.

27   aa.smartNotice.addNotice(permitId1, permitId2,

28   permitId3, noticeLabel, noticeText);
```

Another important feature of our first example is that every line that is not a comment line seems to end with a semicolon (;). We can see that lines 2, 3, 4, 7, 10, 11, 12, 13, 14, 17, 24, and 28 end with a semi-colon. The semi-colon tells Civic Platform that it has reached the end of a command, and should execute it. If we look at line 27 we see that it does not end with a semi-colon, but ends with a comma. The comma means that the command continues on to the next line. We see that line 28 ends with a semi-colon. The semi-colon means that there is one command that begins on line 27 and ends on line 28. If we look at lines 20, 21, 22, 23, and 24 we can see that these lines comprise one big command split across five lines to make it easier to read.

If you forget to end your commands with a semi-colon, Civic Platform is forgiving and the script may run correctly, but it is always good practice to end your commands with a semi-colon when necessary. Some kinds of commands do not have to end in a semi-colon. We investigate what kinds of commands end with a semi-colon and what kinds do not in later sections of this document.

Writing And Testing Our First Script

While you are learning to write scripts, it is useful to be able to test simple scripts and see the results immediately without having to attach your script to an event. To do this we use the Script Test page. For information on testing scripts, see Script Testing. When we cover a new sample script in this document you can copy the script and paste it into the Script Text field on the Script Test page. After you have pasted the text of the script into the form, you can click the Submit button to run the script and view the result.

You can also type this script by hand. Typing the script can be a very helpful learning aid when you start with script writing. However, if you make a mistake in typing you receive a message telling you that there is a problem with your script. When you get a message telling you about the problem, check your script to make sure that it matches the example. Another good tip for learning to write script is to try to modify the sample script to see what happens.

Here is our first sample script for testing:

```
aa.print('Hello World.');
```

If you have read the earlier sections of this document, you may recognize our sample script. The output of this script is

```
Hello World.
```

Let us look at exactly what is happening in our sample script. The script has one line that ends with a semi-colon just like the lines in the first example. The line begins with the two letters 'aa'. These two letters stand for Civic Platform. This line begins with 'aa' because we are going to tell Civic Platform to do something for us. A dot follows the 'aa'. The dot connects the word 'aa' to the word 'print', which means that the word 'print' is a method of the 'aa' object.

An object is a group of associated actions or functions. The previous sample script calls the object aa. The aa object can retrieve data from your database and then use the data to perform tasks. A simple example of the tasks that the aa object is capable of performing is the print task, but there are many tasks that the aa object can perform.

Objects can retrieve information, change stored information, and do many other things for you. Writing and implementing scripts is how we get the aa object (or any other object in JavaScript) to do work with raw data. When we choose a script to initialize in Civic Platform, we are essentially giving a command to our machine. We call the commands we give methods. We learn more about objects and methods in the section Using Objects, Properties, and Methods.

We now know that this line is asking Civic Platform to print something for us. After the word 'print' there is a left parenthesis. After the words 'Hello World.' there is a right parenthesis. When you are writing scripts, a you must follow a method name by a pair of parentheses. Sometimes there are things in between the parentheses, called parameters. Parameters tell a method how to do its job. When we look at the characters between the parentheses we see a single quote followed by the words Hello World, followed by a period, followed by another single quote. Strings are words, numbers, or punctuation marks that appear between single or double quotes.

We know that this script is telling Civic Platform to print the string 'Hello World.', which is exactly what appears in the Script Output box when you use the Script Test page to run this script. For practice, try to change the string passed to the print method of the aa object and see what happens. You can also try to add a second line after the first one with prints out something different, and see what happens then.

Using Jext To Make Writing Scripts Easier

The Jext editor is a freely available text editor with many features that make editing scripts easier. Here is a screenshot of Jext in action:
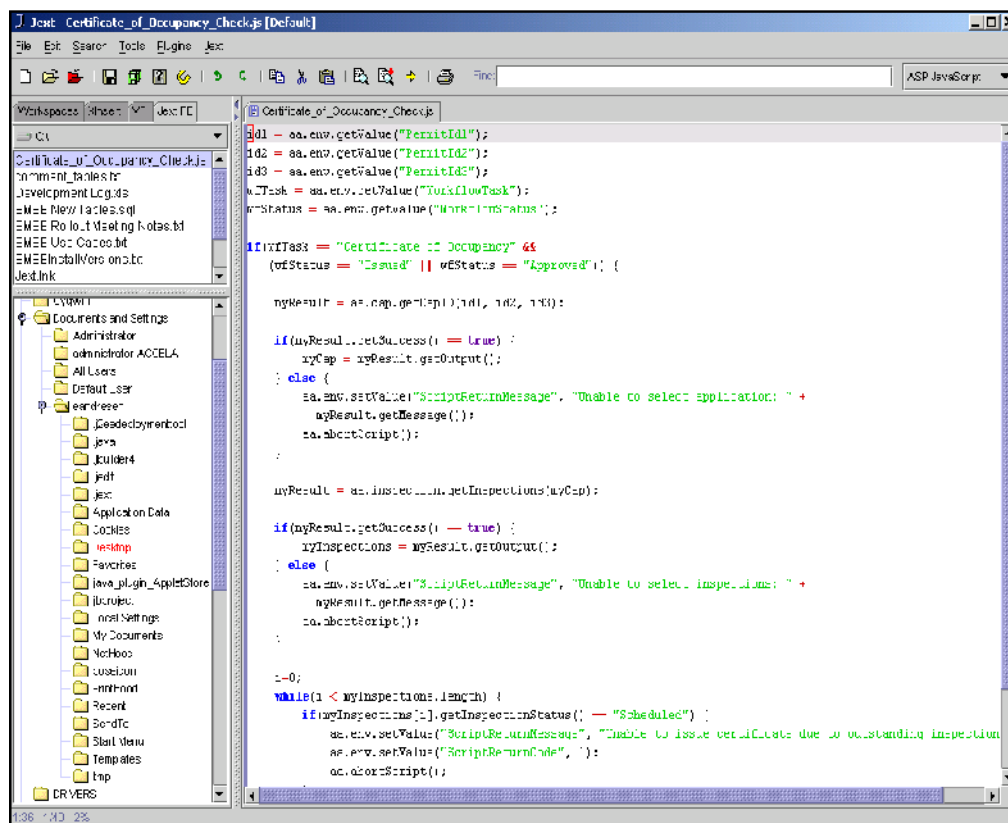
For Jext to recognize your scripts as JavaScript files you must save them with the extension ".js". When opened in Jext, the editor highlights the script text in different colors to make it easier to read. Other useful features include a counter in the bottom left that tells you what line of the script that you are on, and a file explorer in the along the left side to make finding and opening files easier.

Using Variables

A variable is a placeholder in your script that you use to store a value. A variable always has a name that you can use to represent its value. You can use the name of a variable to store something or retrieve it. Variable names, also known as identifiers, must begin with an underscore "_", or letter that you can follow with letters, underscores, and digits. Here are some sample variable names:

```
myVariable?
Number_Of_Inspections?
Test12?
```

In this document, we always begin our variable names with a lower case letter, and capitalize the first letter of each subsequent word in the variable name. We recommend, but do not require, this method of variable naming. Let us look at an example of putting a value into a variable.

```
myVariable = 12;
aa.print(myVariable);
```

This script displays this output:

```
12
```

The first line of this script puts the value 12 into myVariable. The second line uses the print method of the aa object that we investigated earlier, but instead of putting a string of characters in between the left and right parentheses of the print method, we have put the name of our variable. This usage tells the print method to display whatever value myVariable contains. We can also put strings of characters into variables. Here is a script that uses a string as the value of myVariable:

```
myVariable = "Hello World.";
aa.print(myVariable);
```

This script displays this output:

```
Hello World.
```

Another technique that we can use is to assign the value of one variable to the value of a different variable. Here is an example:

```
firstVariable = 101;
secondVariable = firstVariable;
aa.print(secondVariable);
```

This script displays this output:

```
101
```

In this script, we assign the value 101 to firstVariable, and then assign the value of firstVariable to the value of secondVariable. Finally, the script prints out the value of secondVariable. When you use firstVariable on the right side of an equals sign, we call this evaluating a variable. To evaluate a variable is to retrieve its value. We are also evaluating a variable when we pass secondVariable to the print method of the aa object. One might ask, what happens if we try to evaluate the value of a variable to which you did not assign a value. For example,

```
firstVariable = 101;
aa.print(secondVariable);
```

This script displays this output:

```
An error occurred while running your script.
ErrorType: org.mozilla.javascript.EcmaError
Error Detail:
```

```
?undefined: "secondVariable" is not defined. (script; line 1)
```

In this example, we removed the second line of the script. This means there is no line in the script that assigns a value to secondVariable, but on the last line of the script, we try to print out the value of this variable. You receive an error if the script tries to evaluate a variable without an assigned value. When we try to execute this script we see an error message in the output box. The error message tells us that secondVariable is not defined.

There are many potential causes for errors. Civic Platform error messages provide meaningful information to help you solve problems with your scripts. Script writers frequently misspell variable names, so It is a good idea to look carefully at your scripts, check for misspellings, missing semi-colons, and missing parentheses.

Topics:

* Numbers

* Strings

* True and False

* Arrays

* The Special Value "null"

* Objects

Numbers

There are many kinds of numbers you can assign to a variable. We do not provide an exhaustive list here, but we do go over some of the common kinds of numbers that we deal with. Numbers can be positive, negative, zero, integers, decimals, and have exponents and other characteristics. Here are some sample numbers:

```
12
0
-2
1.28
0.94871
-54.09
3.1E12
5E-14
```

The last two sample numbers have an exponent. You probably do not need to use the exponential form of a number, but if your script ever encounters a very large or very small number, then tries to print that value, it may appear in the exponential form. The number after the E is the number of places to the right that you should move the decimal point to get the non-exponential form of the number. If the number after the exponent is negative, it represents the number of places to the left that you need to move the decimal point to get the non-exponential form. If you find that you need to use a kind of number not mentioned here, we encourage you to look up more information on numbers in a JavaScript reference text.

Precision of Numbers

The precision of a number is the number of decimal digits in that number. You can use mathematical expressions (see Mathematical Expressions) or functions (see Using Functions) in your scripts to get various numbers. Because Civic Platform applies the Java class BigDecimal to control the precision of results for mathematical functions, but not for mathematical expressions. If your expected result is a number with decimal part, use mathematical functions instead of expressions to ensure the precision of the result.

With mathematical functions including add, subtract, multiply, divide and round, the default DEF_DIV_SCALE value is 2. You can customize DEF_DIV_SCALE.

For example:

```
aa.print(123.3 / 100); // The precision in the expression is out of
 control.
aa.util.multiply(4.015, 100); // result: 401.5
var DEF_DIV_SCALE=3
aa.util.divide(123.3, 100) //default scale=2, result: 1.23
aa.util.divide(123.3, 100, DEF_DIV_SCALE) // result: 1.233
aa.util.round(12.1542, 1) // result: 12.2
aa.util.round(12.1542, 2) // result: 12.15
```

Strings

Strings comprise a number of characters in between a pair of double or single quotes. Here are some examples:

```
"Hello World."
"This string is surrounded by double quotes."
'This string is surrounded by single quotes.'
'Ok'
'A'
'!'
```

These examples show that a string can be one or many characters long, surrounded by single or double quotes. The following example shows that a string can consist of digits.:

```
"12345"
```

Escape Characters

The next example shows a string with a special escape character inside:

```
"Four score and \n seven years ago."
```

The escape character is the backslash "\" character that you follow with the "n" character. This special character means go to the next line down when printing out this string. This special character calls the new line character. Here is a sample script that prints out this example:

```
aa.print("Four score and \n seven years ago.");
```

This script displays this output:

```
Four score and
 seven years ago.
```

All special characters begin with a backslash. Let us look at the most commonly used special characters:

Table 24: Common Special Characters in Scripting

Special Character	Definition
\n	New Line
\t	Tab
\'	Single quote
\"	Double quote
\\	Backslash

Because we use the single and double quotes to determine the ends of the string, we can only put them into a string by using the special character that represents them. Because we use the backslash to start a special character, we must use a double to put a backslash into a string. Other special characters allow you to insert characters from foreign languages, insert special symbols, and insert other character types. If you find that you need to use these other special characters, consult a standard JavaScript reference book.

True and False

Now we encounter a new kind of variable type that we have not seen before. We call this type of variable Boolean. These variables can only hold either true or false. Here is an example:

```
aTrueVariable = true;
aFalseVariable = false;
```

In this example, we assign the *true* value to one variable and the *false* value to the other variable. Unlike string values, you do not enclose the true and false values in quotes. You can assign these two words to variables as special values. You typically use Boolean variables as parameters for methods of objects or for controlling what happens next in your script. We see some examples of how to use Boolean variables later in this document.

Arrays

An array is a special kind of variable that hold a list of values, and allows you to retrieve and store each of the values separately. Here is an example of creating and using an array:

```
myVar = new Array();
myVar[1] = "Hello";
myVar[2] = "World";
aa.print(myVar[1]);
aa.print(myVar[2]);
```

This script displays this output:

```
Hello
    World
```

The first line of the example tells Civic Platform that myVar is of the special Array type, that is, assign a new empty Array object to myVar. So we can say that myVar contains an array object. The second line of the script puts the string "Hello" in number one position of the array. The third line put the string "World" in the number two position of the array. The fourth line prints out the value stored in the number one position of the array. The fifth line prints out the value stored in the number two position of the array. You can store and retrieve values in any position of the array you like. There is also a position zero, and negatively numbered positions, but most of the time you only use the positively numbered positions.

You can find out how long an array is by using a property of all arrays. Here is an example:

```
myVar = new Array();
myVar[1] = "Hello";
myVar[2] = "World";
aa.print(myVar.length);
```

This script displays this output:

```
3
```

Note that we are printing out something called myVar.length on the last line of the script. Whenever we need to know the length of an array we can always put ".length" after it to get the length. Length is a property of our array. We learn about more arrays and how properties work in the section Objects, Methods, and Properties later in this document. You may ask, if we assigned something to position one and two then why is myVar.length returning three? The answer is that we count position zero in the length of the array. Let us modify this example slightly and see what happens:

```
myVar = new Array();
myVar[1] = "Hello";
myVar[4] = "World";
aa.print(myVar.length);
aa.print(myVar[2]);
```

This script displays this output:

```
5
undefined
```

We changed the third line to put a value in position four rather than in position two. The result is that the total length of the array is now five. There are empty elements in the array at positions 0, 2, and 3. On the last line of the script we tried to evaluate myVar[2] and received a special value called undefined that tells us that we never put anything into the array at that position.

The Special Value "null"

The word null in a script means nothing. Some methods of some objects allow you to pass null in as the value of a parameter, usually to indicate that you do not want to send in any meaningful value for that parameter. We see a little later that Civic Platform may return null to your script when you try to retrieve some information from Civic Platform, usually to indicate that no information is available. We see some of the specific places that use null later in this document.

Objects

A variable can also contain an object. An object is a self-contained module of data and its associated processing. Lets look at an example:

```
myVar = aa;
myVar.print("Hello World.");
```

This script displays this output:

```
Hello World.
```

Here we can see that we assigned the aa object to the myVar variable on the first line of the script. This assignment means that myVar contains the aa object. We then used myVar to execute the print method of the aa object. Here is another example:

```
myVarOne = aa;
myVarTwo = aa;
myVarOne.print("Hello");
```

```
myVarTwo.print("World");
```

This script displays this output:

```
Hello
World
```

Notice that we assign the aa object to both of the variables in this script, and then call the print method on each one. This example shows us that what really happens when you assign an object to a variable is that the variable is only pointing at the object. The variable becomes like a handle to the object that you can use to manipulate it. You can have many variables that all point at the same object.

We look deeper into object in the section Object, Methods, and Properties later in this document. We learn more about variables as we learn about other aspect of writing scripts.

Using Expressions

An expression is a compound value that evaluates to determine a result. We have already encountered one example of an expression called an assignment statement. A simple expression uses an equals sign to assign a value to a variable. Expressions can contain operators that modify or join the values of some variables to come up with a final result. In this section, we look at several different forms of expressions.

Topics:

- Mathematical Expressions

- String Expressions

- Boolean Expressions

- Relational Operators

- Special Operators

- Operator Precedence

Mathematical Expressions

The kinds of expressions that most people are familiar with are arithmetic expressions. Here is an example of an expression that adds two numbers together and assigns the result to a variable:

```
myVar = 2 + 2;
aa.print(myVar);
```

This script displays this output:

```
4
```

In this example the "+" operator joins two numbers. Here is another example:

```
myVar = 1
myVar = myVar + 2;
aa.print(myVar);
firstVar = 7;
secondVar = 5;
myVar = firstVar + secondVar;
aa.print(myVar);
```

This script displays this output:

```
3
12
```

On the second line of this script, we add two to the current value of myVar and put the resulting new value back into and put the resulting new value back into myVar. On the seventh line of this script we add two variables to come up with a result that we place in myVar. There are operators for addition, subtraction, multiplication, division, and many more. We do not cover every arithmetic operator here, but here are six operators arithmetic operators you can use:

Table 25: Mathematical Operators

Symbol	Description	Example
+	Addition	`myVar = 2 + 2;` myVar now contains 4.
–	Subtraction	`myVar = 4 – 2;` myVar now contains 2.
*	Multiplication	`myVar = 2 * 3;` myVar now contains 6.
/	Division. Be careful not to divide by zero or you get an error.	`myVar = 6 / 2;` myVar now contains 3.
%	Modulus. The "remainder" operator. Tells you the left over amount, after division.	`myVar = 5% 3;` myVar now contains 2.

–
 	
(negation)
	Negation. The "unary" operator. Takes whatever value you put immediately to the right of it, and reverses its sign.	`someVar = 3;`
		
		`myVar = -someVar;`
		
		myVar now contains –3.

📝 **Note:**

Civic Platform does not support decimal precision in mathematical expressions. If you want to ensure the precision of your mathematical results, consider using mathematical functions instead. For more information, see Precision of Numbers.

You can also use more than one operator at a time in an expression. For example:

```
firstVar = 7;
secondVar = 5;
myVar = firstVar - 2 + secondVar + 7;
aa.print(myVar);
```

This script displays this output:

```
17
```

When using a single line that contains several operators it is important to remember that, just as in your grade school mathematics classes, some operators have a higher precedence than others do. For example:

```
myVar = 2 + 6 / 3;
```

myVar now contains 4.

The result of the expression in this example was four because the division operator has a higher precedence than the addition operator does. All operators, including the non-arithmetic operators, have a certain level of precedence. When two operators in the same expression have the same level of precedence, Civic Platform evaluates them in left to right order. For example:

```
myVar = 6 * 3 / 3;
```

myVar now contains 6.

You can use parentheses to change the order in which to evaluate an expression:

```
MyVar = (2 + 6) / 4;
```

MyVar now contains 2.

In general, Civic Platform evaluates everything inside a set of parentheses before anything outside the parentheses.

String Expressions

String expressions are quite simple. There is only one operator that works on strings. We use the "+" operator for addition and to concatenate two strings together end to end to form a new string. Here is an example:

```
firstVar = "Hello";
secondVar = "World.";
thirdVar = firstVar + secondVar;
aa.print(thirdVar);
```

This script displays this output:

```
Hello World.
```

You can concatenate more than two strings together:

```
myVar = "Hello" + "to the " + "world.";
aa.print(myVar);
```

This script displays this output:

```
Hello to the world.
```

Boolean Expressions

Boolean expressions always evaluate to either true or false. Boolean Operators shows the most common Boolean operators for boolean expressions.

Table 26: Boolean Operators

Symbol	Description	Example
&&	And	`myVar = true && false` ;myVar now contains false.
||	Or	`myVar = true || 2` ;myVar now contains true.

!	Not	`someVar = true`
		 ;
		 `myVar = !someVar`
		 ;myVar now contains false.

First let us examine the "and" operator "&&". This operator is true when both of its operands are true, and false the rest of the time. The word "operands" refers to the thing that the operator is operating on. So for example, an inspector has an inspection scheduled for today and the inspector has called in sick. The facts that the inspector has an inspection scheduled and that he has called in sick can be operands of the &&

operator. Both are true, so the operation returns a result of true. And Operator Results shows possible results of the "&&" operator.

Table 27: And Operator Results

Example	myVar Contains
`myVar = true && true;`	true
`myVar = true && false;`	false
`myVar = false && true;`	false
`myVar = false && false;`	false

You use the "&&" operator most often when you want to find out if two or more things are true at the same time.

Next we examine the "or" operator "||". You use the "||" operator most often when you want to find out if at least one of two or more things is true. The result is true as long as at least one of the operands is true. You use vertical bar character (also called a pipe), that is on the same key as the backslash character, to type this operator. Press shift backslash to type this character. Or Operator Results shows a set of examples for the "||" operator.

Table 28: Or Operator Results

Example	myVar Contains
`myVar = true || true;`	true
`myVar = true || false;`	true
`myVar = false || true;`	true
`myVar = false || false;`	false

Finally, we examine the "not" operator "!". The "!" is a unary operator that operates on only one operand. Like the unary minus sign, the not operator reverses the state of the value to which it applies (Not Operator Results).

Table 29: Not Operator Results

Example	myVar Contains
`myVar = !true;`	false
`myVar = !false;`	true

You can use multiple Boolean operators in a row, and use parentheses to change the order of precedence of Boolean operators, just like arithmetic operators. However, there is an additional aspect of Boolean operators not shared by other operators called "short-circuit evaluation." Here are two examples of this:

```
myVar = false && ???;
```

myVar contains false no matter what is on the right hand side of the "&&".

```
myVar = true || ???;
```

myVar contains true no matter what is on the right hand side of the "||".

Short-circuit evaluation means that if Civic Platform can determine from the first part of an expression whether the whole expression is going to be true or false it does not bother to evaluate the rest of the expression.

Relational Operators

Relational operators return either true or false. However, unlike Boolean operators they can take different kinds of operands like numbers and strings. The relational operators are ==, !=, <, >, <=, and >=.

The "equals" operator "==" tells us if two values are the same. See Relational Operators.

Table 30: Relational Operators

Example	myVar Contains
`myVar = (true == true);`	true
`myVar = (true == false);`	false
`myVar = (false == true);`	false
`myVar = (false == false);`	true
`myVar = (1 == 2);`	false
`myVar = (2 == 2)`	true
`myVar = ("Hello" == "World");`	false
`myVar = ("Hello" == "Hello");`	true

The "==" operator uses two equals signs to avoid confusion with the assignment operator. You usually use the "==" operator to find out if two things are the same. You can compare any two values using this operator. You can find out if a variable has a special value like null as in this example:

```
myVar = (someVar == null);
```

The "!= operator is the opposite of the "==" operator. See Relational Operators.

Table 31: Relational Operators

Example	myVar Contains
`myVar = (true != true);`	false
`myVar = (true != false);`	true
`myVar = (false != true);`	true
`myVar = (false != false);`	false
`myVar = (1 != 2);`	true
`myVar = (2 != 2);`	false
`myVar = ("Hello" != "World");`	true
`myVar = ("Hello" != "Hello");`	true

The <, >, <=, and >= operators are useful when comparing two numbers. See Relational Operators.

Table 32: Relational Operators

Example	myVar Contains
`myVar = 1 < 2;`	true
`myVar = 2 < 1;`	false
`myVar = 1 > 2;`	false
`myVar = 2 > 1;`	true
`myVar = 1 <= 2;`	true
`myVar = 2 <= 1;`	false
`myVar = 1 <= 1;`	true
`myVar = 1 >= 2;`	false
`myVar = 2 >= 1;`	true
`myVar = 1 >= 1;`	true

Special Operators

We only cover one special operator here. We have already seen this operator when we first investigated creating an array. The "new" operator creates a new object. For example:

```
myVar = new Array();
```

myVar now contains an Array object.

Arrays are really just a special kind of object. We go over more about arrays in a later section. For now, we should take note that the keyword "new" is a special kind of operator that creates a new copy of an object type. In this example the object type was "Array". We learn more about creating objects in the section Objects, Methods, and Properties. There are many special operators that we have not covered. For more information on special operators, consult a JavaScript reference.

Operator Precedence

The operators have the precedence in the order shown.

```
=
||
&&
==  !=
<  >  <=  >=
+  -
*  /  %
!  -  (unary minus)
new
```

Controlling What Happens Next

There are several tools that you can use to indicate the next step in a script.

Topics:

- if … else

- for

- while

- do … while

if … else

You use the conditional when you want to perform a set of commands only if something is true. Here is an example:

```
MyVar = 1;
if(myVar > 0) {
 aa.print("Yes.")
}
```

This script displays this output:

```
Yes.
```

The conditional begins with the word "if" followed by a pair of parentheses. You can place any expression between these parentheses. A pair of braces "{" and "}" follow the parentheses. The braces contain one or more commands. If the expression between the parentheses evaluates to true then the commands between the braces executes. If the expression between the parentheses evaluates to false then the commands between the braces do not execute. The example script does print out the word "Yes" because it is true that myVar, which has the value one, is greater than zero. If you set myVar was to negative one, then nothing prints out.

You use the "else" clause to specify what happens when the condition is false. Here is an example:

```
MyVar = 1;
if(myVar > 2) {
 aa.print("Yes.")
} else {
 aa.print("No");
}
```

This script displays this output:

```
No.
```

In this example, we have changed the conditional to test if myVar is greater than two. Because the condition evaluates to false, the else block executes instead of the main block. We use the word "block" to refer to a group of commands between a matching pair of braces. The first block prints "Yes" if the condition is true. The second block prints "No" if the condition is false. Because the condition is false, this example prints out "No".

You can also create a multi way branch with several blocks. Here is an example:

```
myVar = "Bagels";
if(myVar == "Oranges") {
 aa.print("Fruit.")
} else if(myVar == "Bagels") {
 aa.print("Cereals.");
```

```
} else if(myVar == "Spinach") {
 aa.print("Vegetables.";
} else {
 aa.print("I don't known what food group that is in.";
}
```

This script displays this output:

```
Cereals.
```

This script contains several possible blocks that can execute, depending on the value of myVar. Because myVar has the value "Bagels", the second block executes, and the word "Cereals" prints out. The final else clause is optional.

for

There are several kinds of loops in JavaScript. The for loop allows a script to repeat a set of commands repeatedly until some condition is false. You typically use the for loop when you know how many times you want to repeat the loop. Here is an example:

```
for(i = 1; i < 6; i = i + 1) {
 aa.print("The current value of the loop counter is: " + i);
}
```

This script displays this output:

```
The current value of the loop counter is: 1
The current value of the loop counter is: 2
The current value of the loop counter is: 3
The current value of the loop counter is: 4
The current value of the loop counter is: 5
```

The for loop begins with the word "for" followed by a pair of parentheses that contain three expressions, each separated by semi-colons. After the parentheses are a pair of braces that contain the statements that repeat by the loop. The three expressions in between the parentheses determine how many times the loop repeats. Let us look at these three expressions:

i = 1; i < 6; i = i + 1

The first expression is i=1. This expression set the value of the variable i to one as you might expect. This first expression executes one time, before Civic Platform executes the body of the loop. The body of the loop is the block, surrounded by a pair of braces that comes right after the parenthesis. The second expression is i < 6. This is the condition of the loop, and you only execute the body of the loop if this condition is true. Civic Platform checks the condition just before the body of the loop executes, and the loop continues to repeat until it is false. The third expression, i = i + 1, tells the loop how to update the loop counter each time you reach the end of the body of the loop. When you reach the end of the body of the loop, Civic Platform executes this statement. In this case, the third expression adds one to the counter. From the output of the example, you can see that each time through the loop the counter value updates and the counter value prints out. The loop stops when the value of i reaches six because the second expression is no longer true.

while

The "while" loop is another loop that repeats until its condition is false. You typically use this loop when you do not know how many times the loop executes. Here is an example:

```
myArray = new Array();
myArray[0] = "Oranges";
myArray[1] = "Bagels";
myArray[2] = "Spinach";
i=0;
while(i < myArray.length) {
```

```
 aa.print(myArray[i]);
 i = i + 1;
}
```

This script displays this output:

```
Oranges
Bagels
Spinach
```

In this example, you create an array with three elements and you set the loop counter variable i to zero. The loop begins at the word "while". Next is a pair of parentheses that contain the condition for the loop. While the condition is true, the body of the loop, which comes after the parentheses and is enclosed by a pair of brackets, repeats. We can see two commands inside the body of the loop. The first prints out the value of the array at the position that you indicate by the loop counter. The second line adds one to the loop counter. We need to be careful to remember to always add a line to add to the loop counter to the end of our while loop bodies, because if we do not then the loop never stops, and Civic Platform terminates the script after a time-out period has elapsed.

do … while

This loop also repeats until its condition is false. Use the "do" loop when you want to make sure to execute the body of your loop at least one time even if the condition is false before the loop starts. Here is an example:

```
myArray = new Array();
myArray[0] = "Oranges";
myArray[1] = "Bagels";
myArray[2] = "Spinach";
i=0;
do {
 aa.print(myArray[i]);
 i = i + 1;
} while(i < 0);
```

This script displays this output:

```
Oranges
```

In this example we can see that the body of the do loop executes one time even though the condition i < 0 is false before the loop begins. The example shows that a do loop begins with the word "do" followed by a block, surrounded by braces, for the body of the loop. After the block is the word "while" followed by a pair of parentheses that enclose the condition for the loop. Unlike the other two loops, this last line of the loop, after the parentheses, ends with a semi-colon. Remember to put a command to change the counter in the body of the loop if you are using a counter to control the loop.

Using Functions

A function is a set of commands, with a name, that you can execute by calling that name and passing in any parameters that the function requires. You usually use functions when you have a set of commands that you want to be able to repeat at different places in your script, rather than at one place like with a loop. Let us look at an example:

```
function timesTen(number) {
 result = number * 10;
 return result;
}
myNumber = timesTen(5);
aa.print(myNumber);
```

This script displays this output:

```
50
```

The first four lines of the script create the function. The fifth line stores the function result in a variable and the sixth line uses the variable value as a parameter. We call the four lines that create the function the definition of the function. You can place your function definitions at the beginning or end or your scripts.

Function definitions begin with the word "function" followed by a space, then the name of the function. We can see that the function name is "timesTen". After the function name is a pair of parentheses that enclose the parameter list for the function. We can see that there is one parameter called "number." You can declare as many parameters as you like, but you must separate each one by a comma. Note that parameter names must follow the same rules as variable names.

After the parameter list is a block of one or more commands, enclosed by a pair of braces. The first line in this block for the timesTen function takes the number parameter, multiplies it by ten, and puts the result in the result variable. The second line begins with the keyword "return." This keyword means "send back to whoever called this function the following value." The value following the word "return" on the second line of this script is the variable result. So when one calls the timesTen function, it takes its first parameter, multiples it by ten, and gives as a result the value of the result of that command.

We can see the sixth line of the script calls the timesTen function and the value five passes in as its parameter. The script assigns the result of the timesTen function to the value of the myNumber variable. The last line of the script prints out the value.

Using Objects, Properties, and Methods

An object is a self-contained module of data and its associated processing. We get objects to do work, or retrieve things for us, by calling the methods of the objects. We can also retrieve things from an object using the object's properties. A method is like a function provided to us by an object. When we write script that asks for an object to run a particular method, we say we are calling a method. You can call a method in the following way:

```
objectName.methodName(parameters);
```

Sometimes a method returns a value and that return value is a variable in your script, in which case you call the method this way:

```
myVariable = objectName.methodName(parameters);
```

A property is like a variable that is part of an object. You can always retrieve a property, but you cannot usually change the property. You can retrieve a property value as follows:

```
myVariable = objectName.propertyName;
```

Some objects are available to your script at all times, like the aa object. You retrieve other objects through method calls, or create objects directly by your script like in the examples that use an array. Several predefined objects are available to script writers. Some of these, like Array, Math, and String are part of the JavaScript language. Other predefined objects like aa are additions to JavaScript provided by Accela for interacting with Civic Platform.

Topics:

- The Array Object

- The Math Object

- The String Object

The Array Object

We have already seen one example of how to create an array, but there are other ways to create an array that can be more convenient. An example:

```
myArray = new Array("Oranges", "Bagels", "Spinach");
aa.print(myArray[0]);
aa.print(myArray[1]);
aa.print(myArray[2]);
```

This script displays this output:

```
Oranges
Bagels
Spinach
```

In this example, we initialize an array simultaneously with three elements. This approach provides an easy way to create a small array when you know the contents of that array. The Array object has the property length, and the methods concat, join, pop, push, reverse, shift, slice, splice, sort, and unshift among others.

The Math Object

This object provides access to most if not all of the mathematical functions that you might need when writing scripts. The object defines properties such as E, LN10, LN2, PI and others. Recognize PI as the familiar constant 3.14159. The other properties are also constants. The Math object defines many constants not already mentioned. The Math object also defines these methods: abs, acos, asin, atan,

atan2, ceil, cos, exp, floor, log, max, min, pow, random, round, sin, sqrt, and tan. Let us look at an example of using the math object:

```
piToTheThirdPower = Math.exp(Math.PI, 3);
aa.print(piToTheThirdPower);
```

This script displays this output:

```
23.140692632779267
```

The example calls the "exp" method of the Math object, passes in the PI property of the Math object as the first parameter of the method, and three as the second parameter of the method. The exp method takes its first parameter and raises it to the power of the second parameter. The output is 3.14159 * 3.14159 * 3.14159 = 23.140692632779267. Consult the Civic Platform Script Writer's Object Model Reference documentation or a book on JavaScript for more information on the Math object.

The String Object

When you execute a script with the line:

myVariable = "Hello World."

You are really creating a String object. Let us look at an example:

```
myString = "Hello World";
aa.print(myString.length);
aa.print(myString.toUpperCase());
```

This script displays this output:

```
11
HELLO WORLD
```

We can see from the example that you can use the name of the variable that contains the string to retrieve the length of the string, and call a method of the String object that retrieves an upper case version of a string. The String object has a length property, and the methods slice, split, substr, substring, toLowerCase, and toUpperCase among others. Consult the Civic Platform Script Writer's Object Model Reference or a book on JavaScript for more information on the String object.