Plugin API Example Scripts

Overview

Setting a default attribute value Whenever a new element is created, certain attributes will be prefilled with a default values	**Enforcing upper limits for numbers** Whenever a element is created or updated, hard limits for numeric attribute values are enforced	**Validate status and date interval values (and send emails)** Checks if status and productive period are compatible. Otherwise an email is sent.
Distributing costs Whenever an IT Service element is updated, the value of the costs attribute is distributed to all related business units	**Realising element seals** Seals for elements help with verifying entered data for an element. They can have statuses like 'Valid', 'Invalid' and 'Outdated.	
Changing related elements Shows how to access and update a related element	**Traversing the element hierarchy** Shows how to access and update parent and children elements	**Connecting and disconnecting business mappings** Shows how to create and remove relations and the famous Business Mappings
List of all attributes and relations Write all attribute and relation names to the log file	**Information about an attribute** Writes information about a certain attribute to the log file, e.g. type, mandatory or multiple values	**Information about an user** Write information about a user (name, email) to the log file
Working with enums and multiple values Shows how to generate the array of values of an enumeration attribute like it was before the edit	**Execute an LDAP query** Shows how to execute an LDAP query	**Execute an HTTP request** Shows how to execute an HTTP request

Setting a default attribute value

In this script we set a default value for a newly created Information System. By changing the attribute to be set and the building block type this script can be adapted to set different default values.

The new element already exists in the model that the plugins obtains, hence its properties can be checked in the model.

```
'use strict';

// The function setDefaultValue is called each time a change event
// for an information system is received by the plugins
api.subscribeFor('InformationSystem', setDefaultValue);


function setDefaultValue(buildingblocktype, event) {

    // More than one information system can be changed and
    // the following code should be applied for each of them
    for (var index = 0; index < event.length; index++) {

        //We take one of the changes
        var change = event[index];

        // Default values should only be set when the element is newly created
        // This is shown by the changeType
        if (change.changeType == 'INSERT') {

             // The plugin works with an updated model,
             // hence the new building block is already contained in the model
             // and has all values set that where set during the creation
            var changedObject = api.datamodel.findByTypeAndId('InformationSystem', change.id);

            // If no value is set, we add the default value.
            // In case a value was set when creating the element no default value is needed
            if (!changedObject.getValue("Status values")){
                changedObject.setValue("Status values", "Current");
            }
        }
    }
}
```

Enforcing upper limits for numbers

Th following script enforces an upper limit of 80 for the attribute Costs of Projects. This is done by subscribing to the changes of Projects and setting the value to 80 each time the Costs are set to more than 80.

```
'use strict';

// We subscribe to the changes of projects
api.subscribeFor('Project', enforceUpperLimit);

// This function is called with a the currently used BBT and the changes that were made for this BBT
function enforceUpperLimit(buildingblocktype, event) {

         // We loop over all changes for this building block type and save the current one in the variable
"change"
         for (var index = 0; index < event.length; index++) {
         var change = event[index];

                // We check if the element is created or updated, for deleted elements no values have to be
adapted
        if (change.changeType == 'INSERT' || change.changeType == 'UPDATE') {

                        // We get the BBT for which this change is
            var changedObject = api.datamodel.findByTypeAndId(buildingblocktype, change.id);

                        // We loop over all changes to check if the costs value was changed
            for (var i = 0; i < change.buildingBlockChanges.length; i++) {
                      if (change.buildingBlockChanges[i].persistentName == 'Costs') {

                                        // If Costs are higher than 80: set to 80
                                         if (changedObject.getValue("Costs") > 80){
                                changedObject.setValue("Costs", 80);
                            }
                                 }
            }
        }
    }
}
```

Validate status and date interval values (and send e-mails)

Schedule or directly execute a check: If the status of an information system is 'Current', the date interval 'Productive period' must contain the current day. For each violation the accountable person for the element will be notified by email.

```
'use strict';

// The function validate is registered so it can be executed directly
// or by the scheduler
api.registerExecution(validate);

function validate() {
        // We compare the end of the timespan with the current date,
        // as this is the same for all instances we can create one variable
        var currentDate = new Date().getTime();

        // Returns all InformationSystems in the model
        var allIS = api.datamodel.findByType('InformationSystem');

        // We check all Informationsystems, one after the other
        for (var i=0; i < allIS.length; i++) {

            // We load the is and the attribute values necessary for the validation
            var is = allIS[i];
            var productivePeriod = is.getValue("Productive period");
            var status = is.getValue("Status values");

            // Ensure that all values are set, otherwise we cannot validate
            // and calls would lead to errors
            if (productivePeriod && productivePeriod.to && status) {

                // The validation part, checks if an invalid state is there
                if(status=="Current" && productivePeriod.to < currentDate){

                    // We want to send email to the accountable people,
                    // So we load the necessary data
                    var accountability = is.getValues("Accountability");

                    // More than one person could be accountable, we send the email to all of them
                    for (var index=0; index<accountability.length; index++){
                        var acc = accountability[index];

                        // In the attribute not only the loginName but also first and second name are saved
                        // We need to obtain the loginname itself
                        var loginName = acc.substring(acc.indexOf("(")+1, acc.length-1);

                        // Search for the user to get the email address
                        var user = api.user.get(loginName);

                        // Send an email only if the user has an email address
                        if(user.getEmail()){
                            var subject = "Invalid IS: "+ is.getValue("Name");
                            api.sendEmail(user.getEmail(), subject, "Please fix it");
                        }
                    }
                }
            } //if
        } //for
}
```

Distributing Costs

Each time the attribute 'IT Costs' of the building block type IT Service is changed, this difference of the value is transferred to the connected business units (a fraction depending on the number of related Business Mappings).

```
'use strict';

// This script is called each time an ItService changed
api.subscribeFor('ItService', onBuildingBlockChange);

function onBuildingBlockChange(buildingblocktype, event) {
```

```
// We loop over all changes to find the relevant ones
for (var index = 0; index < event.length; index++) {

    // Take the next change
    var change = event[index];

    // This function is not called for deleted building blocks
    if (change.changeType != 'DELETE') {
        // The changed ItService and its connected BMs are taken from the model
        var changedObject = api.datamodel.findByTypeAndId('ItService', change.id);
        var bm = changedObject.getRelatedObjects('businessMappings');

        // We initialize the variable for the cost differences
        var costsDiff;

        // We search for the change where the IT Costs changed
        // and calculate the difference of these Costs
        for (var i = 0; i < change.buildingBlockChanges.length; i++) {
          if (change.buildingBlockChanges[i].persistentName == 'IT Costs') {
            var added =  change.buildingBlockChanges[i].added;
            var removed =  change.buildingBlockChanges[i].removed;
            //api.printLog("change " + i + " added: " + added);
            //api.printLog("change " + i + " removed: " + removed);
            costsDiff = added - removed;
          }
        }

        // Log Statments help to understand what the script does
        // Especially when developing a script
        api.printLog("costsDiff: " + costsDiff);
        api.printLog("changedObject: " + changedObject.getId());
        api.printLog("Business Mappings: " + bm.length);

        // We only have to continue it the costs changed,
        // Hence if costDiff is initialized (not undefined) and not 0
        if (costsDiff) {
            // In order to distribute related BMs are needed,
            // otherwise nothing can be distributed
            if (bm) {
                for (var j = 0; j < bm.length; j++) {
                    // Within the BM we are interested in the connected business Unit
                    var bu = bm[j].getRelatedObject('businessUnit');

                    api.printLog("bu #" + j + ":  " + bu.getId());
                    // We only continue if there is a related BU,
                    // otherwise there is nothing to obtain the costs
                    if (bu) {
                        // Load the costs of the related BU
                        var buCost = bu.getValue('IT Costs');
                        api.printLog("buCosts: " + buCost);

                        // We distribute the costs to the BUs
                        // by giving it the fraction of the Costs that applies
                        // when dividing equally for all BMs
                        if (!buCost) {

                            // If the BU does not yet have costs,
                            // the costs are set to the distributed value
                            var valueToSet = costsDiff / bm.length;
                            api.printLog("setting to " + valueToSet);
                            bu.setValue("IT Costs", valueToSet);

                        } else {

                            // If the BU has costs these costs are adapted
                            // corresponding to the distributed value
                            var valueToSet = buCost + (costsDiff / bm.length);
                            api.printLog("setting to " + valueToSet);
                            bu.setValue("IT Costs", valueToSet);

                        }
```

```
                        }
                    }
                }
            }
        }
    }
}
```

Realising element seals

This example shows how the functionality of seals can be realised as a plugin script.

Therefore a new enum attribute called "Seal" has to be created with the values "Valid", "Invalid" and "Outdated" and a date attribute called "SealSetDate". Both attributes should be assigned to their own attribute group/attribute groups. Then one can set the permissions for this group so only authorised users can change the seal and the date - or the date can only be changed by the Supervisor to make it invisible for other users.

The different functions are within separate files to make it easier to read and maintain them, but they could also be placed within one file.

Each of the files has its own functionality, the first one sets the seal to "Invalid" when an Information System is changed and the seal is not updated. The second one can be directly executed to invalidate seals that are not changed within the last year. The third one sets the "SealSetDate" each time the seal is set to valid. This is necessary to have a date to check if a building block seal is outdated.

Setting these values is possible within the scripts for each change as the Plugin API works independently from the permissions of the user.

```
'use strict';

// The seal is only used for Information Systems hence only those have to be considered
api.subscribeFor('InformationSystem', onBuildingBlockChange);

// Each time an information System is changed we want to check if the seal breaks
function onBuildingBlockChange(buildingblocktype, event) {

    // Loop over all changes to find the relevant ones
    for (var index = 0; index < event.length; index++) {

        // Each change is checked seperately
        var change = event[index];

        // The seal can only be broken when an element is updated
        if (change.changeType == 'UPDATE') {

            // We search for the changed Information System
            var changedObject = api.datamodel.findByTypeAndId('InformationSystem', change.id);

            // Check if there is a valid seal, otherwise nothing to do
            if (changedObject.getValue("Seal") == "Valid") {

                // We change the value if the seal itself was not changed
                // Hence we initialize the boolean with true
                var changeSeal = true;

                // Check if the property seal was changed
                // in this case the seal is not changed again
                // otherwise it would become impossible to set the seals to valid
                var buildingBlockChanges = change.buildingBlockChanges;
                for (var i = 0; i < buildingBlockChanges.length; i++) {
                    if (buildingBlockChanges[i].persistentName == "Seal") {
                        changeSeal = false;
                    }
                }

                // If the seal becomes invalid because of the change it is invalidated here
                if (changeSeal) {
                    changedObject.setValue("Seal", "Invalid");
                }
            }
        }
    }
}
```

```
'use strict';

// The plugin is registered as Direct Execution Script
api.registerExecution(checkSealsIfOutdated);


function checkSealsIfOutdated() {

    // We want to check all Information Systems
    var allIS = api.datamodel.findByType('InformationSystem');

    // We check each Information System seperately
    for (var i=0; i < allIS.length; i++) {

        // We get the Information System and its attribute value about when the seal was set
        var is = allIS[i];
        var sealDate = is.getValue("SealSetDate");

        // Check if the valid seal was set more than one year ago
        if (is.getValue("Seal") == "Valid" && sealDate && new Date() - sealDate >365*24*60*60000) {

            // In this case the value of the Seal is changed to outdated
            is.setValue("Seal", "Outdated");

        }
    }
}
```

```
'use strict';

// The seal is only used for Information Systems hence only those have to be considered
api.subscribeFor('InformationSystem', setDateOfSeal);


function setDateOfSeal(buildingblocktype, event) {

    // Loop over all changes to find the relevant ones
    for (var index = 0; index < event.length; index++) {

        //Each change is checked seperately
        var change = event[index];

        // When an element is deleted it does not exist anymore,
        // and its values cannot be changed so this function is only called in the other cases
        if (change.changeType == 'UPDATE' ||change.changeType == 'INSERT') {

            // Getting all the changes for different attributes
            var buildingBlockChanges = change.buildingBlockChanges;

            // More than one value might have changed so we loop over all of them
            // to find if there is a relevant change
            for (var i = 0; i < buildingBlockChanges.length; i++) {

                // We look for a change where the seal was set to valid
                if (buildingBlockChanges[i].persistentName == "Seal" && buildingBlockChanges[i].added=="Valid")
{

                    // Obtaining the changed information System
                    var changedObject = api.datamodel.findByTypeAndId('InformationSystem', change.id);

                    // Setting the SealSetDate to the current date,
                    // as this is the time when it was set to valid
                    var currentDate = api.createDate(new Date().getTime());
                    changedObject.setValue('SealSetDate', currentDate);
                }
            }
        }
    }
}
```

Changing related elements

We show how to change related building blocks by adding entries to the accountability attribute.

If an Information System is changed, but not deleted, "bob" becomes accountable for all related Technical Components. If a Technical Component is changed, we add all entries of its accountability to the related Infrastructure Elements.

This example also shows that changes by the Plugin API do not trigger the Plugin API again. Search for a technical component that is related to both an information system and an infrastructure element. If you change the information system, the technical component changes, but the infrastructure element does not.

```
'use strict';

// This code snipped contains two functions, which react to changes of different buildingblock types
api.subscribeFor('InformationSystem', onInformationSystemChange);
api.subscribeFor('TechnicalComponent', onTechnicalComponentsChange);

// When an Information System is changed, this function is called
function onInformationSystemChange(buildingblocktype, event) {

    // We loop over all changes for this BBT and save the current one in a varaible "change"
     for (var index = 0; index < event.length; index++) {
        var change = event[index];

        // We print to the log which Information System was created/updated/deleted
        api.printLog(change.changeType + ' of InformationSystem with ID ' + change.id);

                // We only continue if the element was not deleted
        if (change.changeType != 'DELETE') {
                        // Get the element from the current datamodel
           var changedObject = api.datamodel.findByTypeAndId('InformationSystem', change.id);

            // Get all relation building blocks that connect the element with technical components
            var relatedTechnicalComponentAssociations = changedObject.getRelatedObjects
('technicalComponentAssociations');

                        // Each relation is connected to one technical component, add bob to the
Accountability of this technical components
            for (var i=0; i<relatedTechnicalComponentAssociations.length; i++) {
                var technicalComponent = relatedTechnicalComponentAssociations[i].getRelatedObject
('technicalComponent');
                technicalComponent.addValue('Accountability','bob');
            }
        }
    }
}

// When a Technical Component is changed, this function is called
function onTechnicalComponentsChange(buildingblocktype, event) {
    // Again we loop over all changes, and save the current one within an extra varaible.
   for (var index = 0; index < event.length; index++) {
        var change = event[index];

        // We print information about change to the log
       api.printLog(change.changeType + ' of TechnicalComponent with ID ' + change.id);

        // We only continue if the element was not deleted
        if (change.changeType != 'DELETE') {
           // Again we search for the element and the related building blocks connecting this technical
component with infrastructure elements
           var changedObject = api.datamodel.findByTypeAndId('TechnicalComponent', change.id);
           var relatedInfrastructureElementAssociations = changedObject.getRelatedObjects
('infrastructureElementAssociations');

           // Here we get the accountability values for the changed Object
           var accountabilities = changedObject.getValues('Accountability');

            // We loop over all related infrastructure elements and add all accountability values to the
related element
            for (var i = 0; i < relatedInfrastructureElementAssociations.length; i++) {
                var infrastructureElement = relatedInfrastructureElementAssociations[i].getRelatedObject
('infrastructureElement');
                for (var j = 0; j < accountabilities.length; j++) {
                    infrastructureElement.addValue('Accountability', accountabilities[j]);
                }
            }
        }
    }
}
```

Traversing the element hierarchy

The following scripts shows how to set attribute values for the parent.

When a new parent is assigned to an Information System, the costs of this Information System is added to the costs of the superordinate Information System. If the superordinate Information System does not have a value "Costs", it obtains the costs of the child.

```
'use strict';

// We subscribe to the changes of information systems
api.subscribeFor('InformationSystem', onBuildingBlockChange);

// This function is called with a the currently used BBT and the changes that were made for this BBT
function onBuildingBlockChange(buildingblocktype, event) {

         // We loop over all changes for this building block type and save th current one in the variable
"change"
         for (var index = 0; index < event.length; index++) {
        var change = event[index];

                // Here we only check changes for newly created and updated elements
                if (change.changeType == 'INSERT' || change.changeType == 'UPDATE') {

            var changedObject = api.datamodel.findByTypeAndId(buildingblocktype, change.id);
            var buildingBlockChanges = change.buildingBlockChanges;

                        // We loop over all changes made for this building block
            for (var i = 0; i < buildingBlockChanges.length; i++) {

                                //We are only interested in changes where the parent changed
                if (buildingBlockChanges[i].persistentName == "parent") {

                                        //We get the parent element
                    var parent = changedObject.getRelatedObject("parent");

                                        //We only continue if a parent is set
                                         if (parent) {

                                                //Get the Costs of the parent and the changed object
                                                 var parentCost = parent.getValue("Costs");
                        var changedObjectCosts = changedObject.getValue("Costs");

                                                //Only continue if the changed element has a cost value assigned
                        if (changedObjectCosts){

                                                        //if the parent has no Costs add the Costs of the
current element
                                                        //otherwise add the costs of the current element to the
costs of the parent
                                                         if (!parentCost) {
                                parent.setValue("Costs", changedObjectCosts);
                            } else {
                                parent.setValue("Costs", parentCost+changedObjectCosts);
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Connecting and disconnecting business mappings and attributable relations

This example shows how to connect two building blocks via an attributable relation and how to disconnect two elements connected via a business mapping when only the ids of the building blocks are known.

Connect

```
'use strict';

// This function is registered on direct execution
api.registerExecution(onDirectExecute);

function onDirectExecute() {
    // Get the information system and the technical component by id
    var is = api.datamodel.findByTypeAndId('InformationSystem', 174);
    var tc = api.datamodel.findByTypeAndId('TechnicalComponent', 311);

    // Create a relation building block
    var relation = api.datamodel.create('Is2TcAssociation');

    // Add both elements to the relation object, this way they are connected to each other
    relation.connect(is, "informationSystem");
    relation.connect(tc, "technicalComponent");
}
```

Disconnect

```
'use strict';

// This function is registered on direct execution
api.registerExecution(onDirectExecute);

function onDirectExecute() {

        // The ids of the two elements to be disconnected
    var bfId = 24;
    var boId = 51;

        // We get all BMs related to the business function so we do not have to check all BMs
    var bf = api.datamodel.findByTypeAndId('BusinessFunction', bfId);
    var allBMs = bf.getRelatedObjects('businessMappings');

        // Print information to the log
    api.printLog('The business function is related to '+ allBMs.length + ' business mappings');

        // Loop over all BMs to find the one to delete
    for (var index=0; index < allBMs.length; index++) {
        if(allBMs[index].getRelatedObject('businessFunction') &&
            allBMs[index].getRelatedObject('businessFunction').getId()== bfId &&
            allBMs[index].getRelatedObject('businessObject') &&
            allBMs[index].getRelatedObject('businessObject').getId()== boId &&
            !allBMs[index].getRelatedObject('informationSystem')&&
            !allBMs[index].getRelatedObject('businessUnit')&&
            !allBMs[index].getRelatedObject('businessProcess')&&
            !allBMs[index].getRelatedObject('product') &&
            !allBMs[index].getRelatedObject('itService')){

                        // When the BM is found its id is printed to the log and the element is removed
                api.printLog(allBMs[index].getId());
                allBMs[index].remove();
        }
    }
}
```

List of all attributes and relations

In this example for each building block type the names of all attributes and relations are printed to the log. It is a script that can be directly executed.

```
'use strict';

//The function is registered for diret execution as it works independen of a change
api.registerExecution(getNames);

function getNames() {
  //This array contains the names of all standalone building blocks
    var allBBT = ['BusinessDomain', 'BusinessProcess','BusinessUnit','Product','Project','BusinessFunction',
'BusinessObject',  'InformationSystemDomain','InformationSystem', 'InformationFlow',
'ItService','ArchitecturalDomain','TechnicalComponent', 'InfrastructureElement']

        // We loop over all BBTs to get the information for all of them
    for (var index = 0; index < allBBT.length; index++) {
        var buildingBlockType = allBBT[index];

        // Log the current BBT
        api.printLog( buildingBlockType );

        // Print all attributes of the BBT

        var attributedString = '=Attributes= ';
        // Get all attributes for this BBT
        var bbAttributes = api.datamodel.getAllPropertyNamesByType(buildingBlockType);

        // Add them to the String
        for (var attributeIndex = 0; attributeIndex < bbAttributes.length; attributeIndex++) {
            attributedString += ' * ' + bbAttributes[attributeIndex];
        }

        //Log the created string
        api.printLog(attributedString);

        // Print all relations of the BBT

        var relationString = '=Relations= ';

        //Get all relations and add them to the string like for the attributes
        var bbRelationTypes = api.datamodel.getAllRelationshipNamesByType(buildingBlockType);

        for (var relationIndex = 0; relationIndex < bbRelationTypes.length; relationIndex++) {
            relationString += ' * ' + bbRelationTypes[relationIndex];
        }
        //Print the String
        api.printLog(relationString);
    }
}
```

Information about an attribute

The following script shows how to get detailed information about an attribute. It logs the properties of the Accountability attribute at the Project building block type.

getPropertyInfo

```
'use strict';
api.registerExecution(onDirectExecute);

function onDirectExecute() {
    //Get the information about the attribute "Accountability" of the type "Project"
    var propertyInfoObject = api.getPropertyInfo("Project", "Accountability");

    //Create the string to be logged
    var propertyInfo = 'Project - Accountability '
        + '(type: ' + propertyInfoObject.type
        + ', multiple: ' + propertyInfoObject.multiple
        + ', mandatory: ' + propertyInfoObject.mandatory + ')';

    // log the information
    api.printLog(propertyInfo);
}
```

Information about an user

This example script subscribes to changes of Information System and logs simple information about last modification and the user responsible for it.

getUserDetails

```
'use strict';

// We subscribe for changes on Information Systems
api.subscribeFor('InformationSystem', onBuildingBlockChange);

// This function is called for each change on a information system and prints information about the user that
changed it
function onBuildingBlockChange(buildingblocktype, event) {

        // We loop over all changes and save them within a seperate variable
         for (var index = 0; index < event.length; index++) {
        var change = event[index];

                // This function is called only for updated BBs
        if (change.changeType == 'UPDATE') {

                        // We search for the BB
                         var changedObject = api.datamodel.findByTypeAndId('InformationSystem', change.id);

                        // Getting detailed user about the user that changed to object
                    var userString = changedObject.getValue('lastModificationUser');
                  var loginName = userString .substring(userString .indexOf("(")+1, userString .length-1);
            var lastModificationUser = api.user.get(loginName);

                        // We print a string with all necessary information to the log
                         var lastModificationInfo = 'Information system \"' + changedObject.getValue('name') +
'\" was modified by '
                        + lastModificationUser.firstName + ' ' + lastModificationUser.lastName + '. Contact
this person via email for any details: ' + lastModificationUser.getEmail();

                        api.printLog(lastModificationInfo);
        }
    }
}
```

Working with enums and multiple values

In the following example we react on changes of the accountability attribute. Each time the accountability is changed, we print the set of the persons that were accountable for the Information System or Architectural Domain before the change.

By adapting this example it is possible to obtain the set of values before the change for arbitrary multivalue enums.

```
'use strict';

// The script is used for Information Systems and Architectural Domains, we react on changes of them
api.subscribeFor('InformationSystem', onBuildingBlockChange);
api.subscribeFor('ArchitecturalDomain', onBuildingBlockChange);

// Each time an information system changed we check if it obtained a new parent
function onBuildingBlockChange(buildingblocktype, event) {

    // We loop over all changes
    for (var index = 0; index < event.length; index++) {
        var change = event[index];

        // And consider only changes of updated information systems
        if (change.changeType == 'UPDATE') {

              // By using the type that is given as parameter the same script can be used for more than one type
            var changedObject = api.datamodel.findByTypeAndId(buildingblocktype, change.id);

            var buildingBlockChanges = change.buildingBlockChanges;

            for (var i = 0; i < buildingBlockChanges.length; i++) {

                // We check if the attribute accountability was changed, only for those changes we want to do
something
                if (buildingBlockChanges[i].persistentName == "Accountability") {
                    var changesOfAccountability = buildingBlockChanges[i];

                    // We start with the current set
                    var setBeforeChange = changedObject.getValues("Accountability");

                    // Then we add the values that were removed with the change
                    if (changesOfAccountability.removed) {
                        setBeforeChange = setBeforeChange.concat(changesOfAccountability.removed);
                    }

                    // Finally we only keep the values that are not added by the change
                    if (changesOfAccountability.added) {
                        setBeforeChange = setBeforeChange.filter(
                            function (item) {
                                return changesOfAccountability.added.indexOf(item) == -1;
                            }
                        )
                    }

                    // Now we can print the set like it was before the change
                    api.printLog("The set before the changes: " + setBeforeChange);

                }
            }
        }
    }
}
```

Execute an LDAP query

This example script executes LDAP queries and adds users to a responsibility attribute.

Execute LDAP query and populate responsibility attribute

```
'use strict';

api.registerExecution(onDirectExecute);

function onDirectExecute() {
    var ldapResult = [];

    // Executes an LDAP query to obtain information needed to create users
    try {
        ldapResult = api.executeLdapQuery("OU=Engineering", "(&(objectCategory=user)(objectClass=user))",
["sAMAccountName", "givenName", "sn", "mail"]);
    } catch(e) {
        api.printLog("LDAP error: " + e.message);
    }

    // Get the responsibility attribute where the users should be added
    var attr = api.metamodel.getAttribute("responsibilityAttribute");

    // Loop over all the information the LDAP execution returned
    for (var i = 0; i<ldapResult.length; i++) {
        var str = "";

        // Save the information from LDAP in different constants
        var loginName = ldapResult[i].sAMAccountName;
        var firstName = ldapResult[i].givenName;
        var lastName = ldapResult[i].sn;
        var mail = ldapResult[i].mail;

        // Create the user within the attribute
        try {

            // Check if data are null and use an empty string in this case
            api.printLog("Create user: " + loginName);
            firstName = firstName === null ? "" : firstName;
            lastName = lastName === null ? "" : lastName;
            mail = mail === null ? "" : mail;

            // Create the user
            var user = api.user.create(loginName, firstName, lastName);
            user.setEmail(mail);

            // Add the responsibility literal to the attribute
            attr.createResponsibilityLiteral(user.getId(), "c0c883");

        } catch(userEx) {
            api.printLog("User create error: " + userEx.message);
        }
    }
}
```

Execute an HTTP request

This code snippet uses a public API with information about sun rise and sunset to log the sunset-time in Munich of the current day. It shows how to use the HTTP functions of the Plugin API.

```
'use strict';

api.registerExecution(httpSunset);

// Writes the time of the sunset at the current day to the log
function httpSunset(){

    // Setting parameters for the http request
    // This corresponds to GET on "https://api.sunrise-sunset.org/json?lat=48.13743&lng=11.57549"
    api.http.setURL("https://api.sunrise-sunset.org/json");
    api.http.setMethod("GET");
    api.http.addParameter("lat", 48.13743)
    api.http.addParameter("lng", 11.57549)

    // Performing the request, returns a string
    var result = api.http.performRequest();

    // In order to work with the result, it is parsed to a JSON Object
    var resultJson = JSON.parse(result);

    // We log the time of the sunset
    api.printLog(resultJson.results.sunset);
}
```