Tivoli. software

IBM

IBM Tivoli Identity Manager 4.6 – Extending Workflows with Java

White Paper

Ori Pomerantz

March 2006

Table of Contents

Introduction

White Paper

Introduction

About this Paper

IBM Tivoli Identity Manager has a very flexible workflow engine. It can request additional information, send e-mail messages, provision additional accounts, and so on. When the default functionality is insufficient, the workflow engine can be extended using Java. This allows workflows to store additional fields for accounts, interface with third-party products, and access all the capabilities of the server.

This white paper presents the use of Java extensions, both directly into the workflow engine and indirectly into the JavaScript interpreter. These extensions will be used to implement a recertification policy that stores the last certification date for each account in a database table. You will also learn how to use a database table to virtually extend account schemas.

Audience

This paper is for implementers and senior system administrators who need to extend workflow functionality and know Java programming. It builds on the knowledge provided in the *Extending IBM Tivoli Identity Manager 4.6* class.

White Paper

1 Using this White Paper

In this white paper you will learn how to use extension elements and JavaScript extensions to implement a recertification policy. A ***recertification policy*** requires that the continuing need for accounts will be certified periodically. To do this, it is necessary to keep track of the last time the need for an account was certified.

To get the most out of this white paper, run IBM Tivoli Identity Manager 4.6 on a test computer and follow along with the examples.

1.1 IBM Tivoli Identity Manager 4.6 Configuration

Install all the components of IBM Tivoli Identity Manager (ITIM) 4.6 on a single computer, including at least one adapter. Use **idsldap** as the owner of IBM Tivoli Directory Server and IBM DB2. Call the database you configure for ITIM **itimdb**, and configure the owner of the database as **enrole** with the password **object00**.

1.2 Database Configuration

IBM Tivoli Identity Manager already has access to a database. For a test implementation like this one, it is best to add tables to that database:

1. Source the database environment:

   ```
   . ~idsldap/sqllib/db2profile
   ```

2. Run the SQL interpreter:

   ```
   db2sql92 -d itimdb -a enrole/object00
   ```

3. Enter these SQL commands to create the tables required for this white paper:

   ```
   CREATE TABLE certDates (
      acctDN varChar(120),
      certTime bigInt
   );
   ```

```
CREATE TABLE acctAttr (
   globalID char(30),
   name varChar(200),
   value varChar(200)
);
```

2 Extension Elements

Extension elements in a workflow call Java methods. Extension elements are atomic, meaning that each element performs its full action, and cannot be customized by the workflow to perform part of it. For a recertification policy, the following operations are required:

- **initialCert** – set the certification date for an account to the present, creating a new row in the database.
- **updateCert** – recertify an account, setting the certification date in an existing row to the present.
- **checkCert** – check if a certification for an account has expired.
- **deleteCert** – delete the certification information, for use when an account is deleted.

2.1 Writing the Extensions

When programming extensions, it is better to test them through the command line before using them in IBM Tivoli Identity Manager. This allows the programmer to identify and solve problems with the extension methods before encountering problems related to placing the extensions to the workflow.

A copy of the extensions without the ITIM specific code will not be included here. You can compile and run the Java program in the next section. If you followed the setup instructions in section 1, the following script will compile and execute a test to verify the extensions work correctly:

```
#! /bin/sh

. ~idsldap/sqllib/db2profile
CLASSPATH=$CLASSPATH:/opt/IBM/itim/lib/itim_api.jar

javac CertExt.java
java CertExt
```


2.2 Preparing the Extensions for ITIM

The following changes are required in a class to make its methods usable as application extensions in IBM Tivoli Identity Manager:

- Have the class implement the interface **com.ibm.itim.workflow.application.WorkflowInterface**.
- Implement the sole method of that interface, **setContext(com.itim.workflow.applications.WorkflowExecutionContext)**. The workflow engine will use this method to provide the class with information about the current workflow.
- If the class has a constructor, make sure that it is public and does not require any parameters. If the constructor is private or protected, the workflow engine will not be able to instantiate it.

In addition, each method that will be called from an extension has to follow these restrictions:

- The method arguments have to be objects, not primitive types.
- The method needs to return a **com.ibm.itim.workflow.model.ActivityResult** object with the results. At a minimum, this object has to specify if the method was successful or not. It can also contain result descriptions, as well as any values returned by the extension.

2.2.1 Simple Examples

This section shows simple code snippets that implement only the basic functionality required to connect to IBM Tivoli Identity Manager.

```
import com.ibm.itim.workflow.application.*;
import com.ibm.itim.workflow.model.*;
```

These lines import the classes that IBM Tivoli Identity Manager requires in application extensions.

```
public class AppExtensionExample
    implements WorkflowApplication
{
```

The class that contains the application extension methods has to implement **WorkflowApplication**. Of course, it also needs to be a public class.

```
// Workflow context, passed from the workflow
// engine
WorkflowExecutionContext ctx;

/**
 * Set the workflow execution context
 *
 * @param ctx context of the current activity
 */
public void setContext(WorkflowExecutionContext ctx)
{
    this.ctx = ctx;
}
```

These lines implement the one function required by **WorkflowApplication**, which enables the workflow engine to set the context for the application extension. This context includes information about the current process, the reason it is running, and so on.

```
public AppExtensionExample()
{
}
```

If there is a constructor, it has to be public, to allow the workflow engine to create new instances of the class to run the extension methods.

```
public ActivityResult success(String param)
{
    // Return with success, no message
    return new ActivityResult();
}
```

This is a simple extension that receives a single parameter, a string. It returns an **ActivityResult** that always reports success, and does not have any output parameters.

```
public ActivityResult failure()
{
    // Return with success, no message

    return new ActivityResult(
       ActivityResult.STATUS_ABORT,
       "Error message.");
}
```

This is another extension, which always reports failure with an error message.


```
public ActivityResult echo(String echoMe)
    {
        // Create the list of values to return:

        // Create a vector of objects with size 1
        java.util.Vector  = new java.util.Vector(1);

        // Add the string to the results vector
        retVal.add(echoMe);


        // Create the Activity Result to return
        ActivityResult res = new ActivityResult();

        // Add the results vector
        res.setDetail(retVal);

        // Return
        return res;
    }
```

This extension shows how to return parameters from an extension:

1. Create a **java.util.Vector** of the required size.
2. Use the **add** method to add the objects for the return parameters to vector.
3. Use the **setDetail** method of the **ActivityResult** object to add the vector to the results.

2.2.2 Complete Example – Extensions for the Recertification Policy

This Java code implements the extensions that a recertification policy requires.

```
import java.util.Date;
import java.sql.*;
import com.ibm.itim.workflow.application.*;
import com.ibm.itim.workflow.model.*;
```

```
/**
 * CertExt implement IBM Tivoli Identity Manager
 * workflow extensions for account certification.
 * </P>
 * This code is provided as part of the IBM Tivoli Identity
 * Manager 4.6 - Extending Workflows with Java white paper,
 * and is intended for education use only.
 */
public class CertExt
    implements WorkflowApplication
{
    // Connection to the database
    Connection dbConn;

    // Workflow context, passed from the workflow
    // engine
    WorkflowExecutionContext ctx;

    /**
     * Set the workflow execution context
     *
     * @param ctx context of the current activity
     */
    public void setContext(WorkflowExecutionContext ctx)
    {
        this.ctx = ctx;
    }

    // A small main to check the functionality. This
    // method creates a certification row, updates it,
    // checks if the certification was in the last day,
    // and then deletes it.
    public static void main(String[] args)
        throws java.io.IOException
    {
        CertExt me = new CertExt();
        final String acctDN =
            "erglobalid=XXXXXXXXXXXXXXXXXX,"
            + "ou=0,ou=accounts,"
            + "erglobalid=0000000000000000000,"
            + "ou=xyz,o=xyz";

        me.initialCert(acctDN);
        System.out.println("Check in the database that " +
                           "there is a cert date for " +
                           "XXXXX.");
        getEnter();

        me.updateCert(acctDN);
        System.out.println("Check in the database that " +
                           "the cert date for XXXX " +
                           "has been updated.");
        getEnter();
```



```
        ActivityResult res = me.checkCert(acctDN);
        System.out.println("Is the cert valid? " +
                           res.getDetail().get(0));

        me.deleteCert(acctDN);
        System.out.println("Check in the database that " +
                           "the row for XXXX has been " +
                           "deleted.");
    }


    /**
     * Get from stdin the line until an
     * Enter and discard it. This is used by main to wait
     * for user confirmation before it continues.
     */
    static void getEnter()
        throws java.io.IOException
    {
        System.out.println("Press Enter when you are " +
                           "ready to proceed.");
        do
            System.in.read();
        while (System.in.available() > 0);
    }


    /**
     * Connect to the database.
     */
    public CertExt()
    {
        try {
            // Open a connection to the database
            dbConn = connectDB();
        }
        catch (SQLException ex) {
            System.out.println("SQL Exception: " + ex);
        }
    }
```

```
/**
 * Close the database connection.
 */
protected void finalize()
    throws Throwable
{
    try {
        // Close the database connection
        dbConn.close();
    }
    catch (SQLException ex) {
        System.out.println("SQL Exception: " + ex);
    }
}

/**
 * Connect to the database.
 * </P>
 * This method is hard-wired to connect to the itimdb
 * database as enrole, with the password object00.
 * Modify it for your own installation's values.
 *
 * @return a connection to the database
 */
Connection connectDB()
    throws SQLException
{
    try {
        // Load the JDBC driver for DB2
        Class.forName("COM.ibm.db2."+
                      "jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException e) {
        System.out.println("Can't load JDBC driver.");
        System.out.println("Exception: " + e);
        System.out.println("Source the DB2 environment"
                           + " if you haven't yet.");
    }

    // Connect to the database
    return DriverManager.
        getConnection("jdbc:db2:" +
                      // Database name, as specified
                      // in the local catalog on the
                      // ITIM Server
                      "itimdb",
                      // User for the database
                      "enrole",
                      // Password for the database
                      "object00"
                      );
}
```



```
/**
 * Create the initial certification for an
 * account. Set the certification time to the present.
 *
 * @param acctDN the DN for the account entity in the
 * directory
 *
 * @return The result of the activity
 */
public ActivityResult initialCert(String acctDN)
{
    // The current time, in milliseconds
    long now = new Date().getTime();

    // Write information to the database.
    try {
        // Create a PreparedStatement with
        // the SQL command.
        PreparedStatement pstmt =
            dbConn.prepareStatement(
               "INSERT INTO CertDates " +
               "(acctDN, certTime) " +
               "VALUES ('" + acctDN + "', "
                               + now + ");");

        // Update the database
        pstmt.executeUpdate();

        // Close the statement
        pstmt.close();

        // Return with success, no message
        return new ActivityResult();
    }
    catch (SQLException ex) {
        // Write the failure
        System.out.println("SQL Exception: " + ex);

        // Return with failure
        return new ActivityResult(
          ActivityResult.STATUS_ABORT,
          "SQL Exception: " + ex);
    }
}
```

```
/**
 * Update the certification for an account.
 * Set the certification time to the present.
 *
 * @param acctDN the DN for the account entity in
 * the directory
 *
 * @return The result of the activity
 */
public ActivityResult updateCert(String acctDN)
{
    // The current time, in milliseconds
    long now = new Date().getTime();


    // Write information to the database.
    try {
        // Create a PreparedStatement with
        // the SQL command.
        PreparedStatement pstmt =
            dbConn.prepareStatement(
                "UPDATE certDates " +
                    "SET certTime=" + now + " " +
                "WHERE acctDN='" + acctDN + "';");

        // Update the database
        pstmt.executeUpdate();

        // Close the statement
        pstmt.close();

        // Return with success, no message
        return new ActivityResult();
    }
    catch (SQLException ex) {
        // Write the failure
        System.out.println("SQL Exception: " + ex);

        // Return with failure
        return new ActivityResult(
          ActivityResult.STATUS_ABORT,
          "SQL Exception: " + ex);
    }
}
```



```
/**
 * Create a Vector with the boolean value that
 * checkCert needs to return.
 *
 * @param value the boolean value to return
 *
 * @return A vector that has value in the first
 * location and nothing else.
 */
java.util.Vector returnString(String value)
{
    java.util.Vector retVal =
        new java.util.Vector(1);

    retVal.add(value);

    return retVal;
}


/**
 * Check if the account certification is recent
 * enough to still be valid (in the last 45 days).
 *
 * @param acctDN the DN for the account entity in the
 * directory
 *
 * @return The result of the activity. The
 * ActivityResult also contains list of returned
 * parameters. In this case, it will have one returned
 * parameter - a string that specifies if the
 * certification is valid or not. The first character
 * of the string is Y for valid certifications, N for
 * invalid ones.
 */
public ActivityResult checkCert(String acctDN)
{
    // The current time, in milliseconds
    long now = new Date().getTime();

    // Read information from the database.
    // This could cause an SQL exception.
    try {
        // Create a PreparedStatement with
        // the SQL command.
        PreparedStatement pstmt =
            dbConn.prepareStatement(
               "SELECT certTime FROM certDates " +
               "WHERE acctDN='" + acctDN + "';");

        // Read from database
        ResultSet rs = pstmt.executeQuery();
        rs.next();
```

```
                // The date from the database
                long certDate = rs.getLong(1);

                // Close the statement and result set
                rs.close();
                pstmt.close();

                String result;
                // Notice the l following the 45. This
                // converts the multiplication to a long
                // value - otherwise it wraps around and
                // becomes negative.
                if (now > (certDate + 45l*24*3600*1000)) {
                    result = "No, the certification is " +
                        (now - certDate) / (24*3600*1000) +
                        " days old.";
                } else {
                    result = "Yes, the certification " +
                        "is just " +
                        (now - certDate) / (24*3600*1000) +
                        " days old.";
                }
                java.util.Vector retVal = returnString(result);
                ActivityResult res = new ActivityResult();
                res.setDetail(retVal);
                return res;
            }
            catch (SQLException ex) {
                System.out.println("SQL Exception: " + ex);

                // Create the list of returned parameters
                java.util.Vector retVal =
                    returnString("No, due to SQL: " + ex);

                // Return the results
                return new ActivityResult(
                           ActivityResult.STATUS_ABORT,
                    "SQL Exception: " + ex, "",
                    retVal);
            }
        }
```



```
    /**
     * Delete the certification for an account.
     *
     * @param acctDN the DN for the account entity in
     * the directory
     *
     * @return The result of the activity
     */
    public ActivityResult deleteCert(String acctDN)
    {
        // Delete information in the database.
        try {
            // Create a PreparedStatement with
            // the SQL command.
            PreparedStatement pstmt =
                dbConn.prepareStatement(
                    "DELETE FROM certDates " +
                    "WHERE acctDN='" + acctDN + "';");

            // Update the database
            pstmt.executeUpdate();

            // Close the statement
            pstmt.close();

            // Return with success, no message
            return new ActivityResult();
        }
        catch (SQLException ex) {
            // Write the failure
            System.out.println("SQL Exception: " + ex);

            // Return with failure
            return new ActivityResult(
              ActivityResult.STATUS_ABORT,
              "SQL Exception: " + ex);
        }
    }
}
```

2.3 Registering the Extensions with ITIM

You have to register the extensions with ITIM so they can be called from workflows. To do this, add the new extensions to an XML file, place the class file in the ITIM class path, and then restart the application server.

2.3.1 The XML File

All workflow extensions are registered in **$ITIM_HOME/data/workflowextensions.xml**. For every extension, this file contains the extension's name, the name of the class and method that implement it, and the input and output parameters along with their types.

The input parameters are sent as arguments to the method. The output parameters are retrieved from the details of the **com.ibm.itim.workflow.model.ActivityResult** object the method returns.

Note: Each workflow node that calls the extension has a copy of the information from **workflowextensions.xml**. When you change the extension parameters in this file, edit the nodes that implement it. Choose the extension again, and then fill the parameters.

Here are the tags to add to **workflowextensions.xml** for the extensions in section 2.2.2.

```
<ACTIVITY ACTIVITYID="initialCert" LIMIT="0">
    <IMPLEMENTATION_TYPE>
        <APPLICATION CLASS_NAME="CertExt"
             METHOD_NAME="initialCert"/>
    </IMPLEMENTATION_TYPE>
    <PARAMETERS>
        <IN_PARAMETERS  PARAM_ID="acctDN"   TYPE="String"/>
    </PARAMETERS>
</ACTIVITY>

<ACTIVITY ACTIVITYID="updateCert" LIMIT="0">
    <IMPLEMENTATION_TYPE>
        <APPLICATION CLASS_NAME="CertExt"
             METHOD_NAME="updateCert"/>
    </IMPLEMENTATION_TYPE>
    <PARAMETERS>
        <IN_PARAMETERS  PARAM_ID="acctDN"   TYPE="String"/>
    </PARAMETERS>
</ACTIVITY>
```



```
<ACTIVITY ACTIVITYID="deleteCert" LIMIT="0">
    <IMPLEMENTATION_TYPE>
        <APPLICATION CLASS_NAME="CertExt"
             METHOD_NAME="deleteCert"/>
    </IMPLEMENTATION_TYPE>
    <PARAMETERS>
        <IN_PARAMETERS  PARAM_ID="acctDN"   TYPE="String"/>
    </PARAMETERS>
</ACTIVITY>

<ACTIVITY ACTIVITYID="checkCert" LIMIT="0">
    <IMPLEMENTATION_TYPE>
        <APPLICATION CLASS_NAME="CertExt"
             METHOD_NAME="checkCert"/>
    </IMPLEMENTATION_TYPE>
    <PARAMETERS>
        <IN_PARAMETERS  PARAM_ID="acctDN"   TYPE="String"/>
        <OUT_PARAMETERS PARAM_ID="result"   TYPE="String"/>
    </PARAMETERS>
</ACTIVITY>
```

2.3.2 The Application Class Path

ITIM runs under IBM WebSphere Application Server. To add a new class to ITIM's class path:

1. Put the new class in a Java archive (JAR) file.
2. Copy the jar file to **$WAS_HOME/installedApps/<machine-name>/enRole.ear**.
3. Add the name of the jar file to the manifest, which is located at **$WAS_HOME/installedApps/<machine-name>/enRole.ear/app_web.war/META-INF/MANIFEST.MF**.

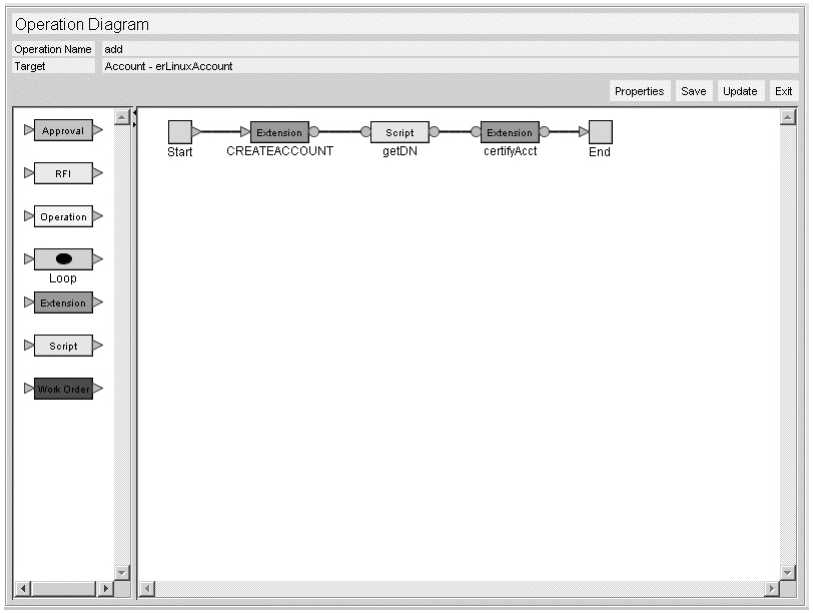
2.4 Using the Extensions

Once the extensions are written, compiled, and registered with IBM Tivoli Identity Manager, the final step is to create the workflows and lifecycle rules that use them.

2.4.1 Initial Certification

New accounts need to have a certification row in the database. To create it when they are provisioned, modify the **add** operation. Add a relevant data variable for the account's distinguished name (DN), a script to fill it with the appropriate value, and then an extension to run **initialCert**.

Figure 1



2.4.1.1 Account's Distinguished Name

The extension requires the DN of the account. However, the **account** parameter that the **add** workflow receives does not have a DN yet. The **CREATEACCOUNT** extension assigns a DN when it adds the account to the directory, but it does not update the parameter in the workflow. To get the DN, the script has to read the account entity from the server.

IBM Tivoli Identity Manager does not support LDAP searches for accounts. Therefore, to read the account, it is necessary to read all of the accounts owned by the account's owner, and then identify the correct one. The correct account is the one whose service and user ID attributes are identical to the ones of the **account** parameter.

The following script implements this:

```
var search = new AccountSearch();
var accounts = search.searchByOwner(owner.get().dn);
var serviceDN = service.get().dn;
var uid = account.get().getProperty("eruid")[0];

for (var i=0; i<accounts.length; i++)
   if ((accounts[i].getProperty("erservice")[0] == servDN)
           &&
      (accounts[i].getProperty("eruid")[0] == uid))
           acctDN.set(accounts[i]).dn;
```

2.4.2 Recertification

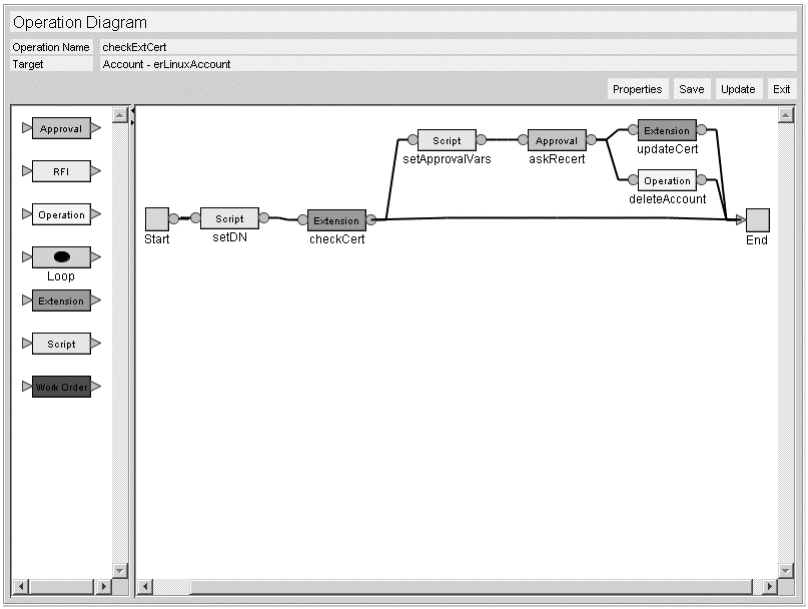
This workflow implements the recertification process. It needs to be called periodically from a lifecycle rule. Because the certification date is not an LDAP attribute, the lifecycle rule cannot identify which accounts need to be recertified. Instead, it calls the workflow on all accounts, or all accounts of a particular service.

The workflow then uses the **checkCert** extensions to identify which accounts need to be recertified. If an account's certification is no longer valid, the workflow uses an approval element to ask for recertification. If the account is then recertified, it uses the **updateCert** extension to change the certification date; otherwise it uses the **delete** operation to remove it.

This workflow uses the following relevant data variables:

Variable	Type	Set By	Used By	Meaning
Entity	Account	The lifecycle rule	setDN, deleteAccount	The account that the workflow processes, and if necessary recertifies or deletes
acctDN	String	setDN	checkCert, updateCert	The DN of the account
retVal	String	checkCert	The transitions out of checkCert	The result of **checkCert**, a string that starts with **Y** if the account's certification is still valid, **N** otherwise
owner	Person	setApprovalVars	askRecert	The owner of the account
service	Service	setApprovalVars	askRecert	The service of the account

Figure 2



The workflow starts with the **setDN** script element. This element sets a variable with the DN of the account. Because it is an existing account, the DN is available, and the script only has to use it:

```
acctDN.set(Entity.get().dn);
```

The extension element that checks the certification, **checkCert**, uses **acctDN** as input, and **retVal** as output.

If the account's certification is still valid, the workflow is done. The transition between **checkCert** and **End** uses the following custom condition to check the certification:

```
res = retVal.get();
return (res.substring(0,1) == "Y");
```

If the account's certification is no longer valid, the workflow needs to run **setApprovalVars**. The transition between **checkCert** and **setApprovalVars** uses this custom condition to check that the certification is no longer valid:

```
res = retVal.get();
return (res.substring(0,1) == "N");
```

Approval elements that request approval for an account, such as this one, require three relevant data variables: the owner, the account, and the service. The account is already available in **Entity**. The script **setApprovalVars** sets the other two variables:

```
var acct = Entity.get();
owner.set(new Person(acct.getProperty("owner")[0]));
service.set(new Service(acct.getProperty("erservice")[0]));
```

The **setApprovalVars** script is followed by the approval element, **askRecert**. The transitions out of the approval element use the predefined conditions: the one to **updateCert** uses the **Approved** condition, and the other, to **deleteAccount**, uses the **Rejected** condition.

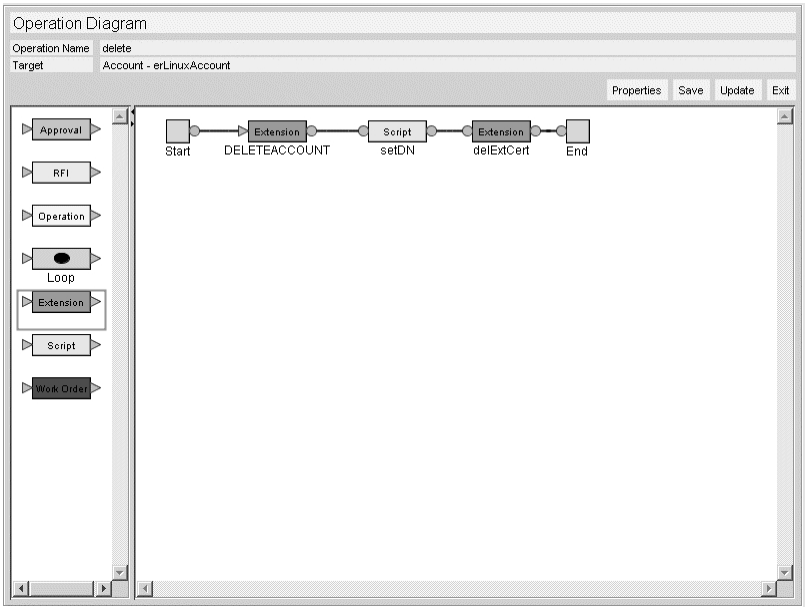
The **updateCert** extension node calls the **updateCert** extension to update the certification time, if the account is recertified.

The **deleteAccount** operation calls the **delete** operation workflow if the account is not recertified. It is important to use the workflow, rather than the extension **DELETEACCOUNT**, because the workflow also deletes the certification row from the database table.

2.4.3 Deleting Certifications

When an account is deleted, there in no longer a need for a database row with its certification date. The modifications to the **delete** workflow are similar to the ones to the **add** workflow in section 2.4.1.

Figure 3



The script in **setDN** is the same as in the recertification workflow:

```
acctDN.set(account.get().dn);
```

2.5 Accessing ITIM Objects in Extensions

The extension elements in the recertification policy require the DN as input, which means the workflows that use them have to have a relevant data variable for the DN and a script to set it to the correct value. It would be simpler to pass the account to the extension and let it find the DN.

This section will only show how to change the **deleteCert** extension to do this. The changes in the other extensions are nearly identical.

2.5.1 Changes in the Java

1. Import the package **com.ibm.itim.dataservices.model.domain**, which contains the ITIM objects.
2. Change the parameter type to the appropriate ITIM object. In the case of **deleteCert**, that would be **Account**.
3. Using the javadoc pages, which are installed in **$ITIM_HOME/extensions/api**, modify the method to use the object.

This is the modified **deleteCert** extension:

```
/**
 * Delete the certification for an account.
 *
 * @param acct the Account
 *
 * @return The result of the activity
 */
public ActivityResult deleteCert(Account acct)
{
    // Get the distinguished name
    String acctDN =
        acct.getDistinguishedName().toString();

    // Delete information in the database.
    try {
        // Create a PreparedStatement with
        // the SQL command.
        PreparedStatement pstmt =
            dbConn.prepareStatement(
                "DELETE FROM certDates " +
                "WHERE acctDN='" + acctDN + "';");
        // Update the database
        pstmt.executeUpdate();

        // Close the statement
        pstmt.close();

        // Return with success, no message
        return new ActivityResult();
    }
    catch (SQLException ex) {
        // Write the failure
        System.out.println("SQL Exception: " + ex);

        // Return with failure
        return new ActivityResult(
          ActivityResult.STATUS_ABORT,
          "SQL Exception: " + ex);
    }
}
```


2.5.2 Changes in the Extension Registration

Change parameter type in **workflowextensions.xml**:

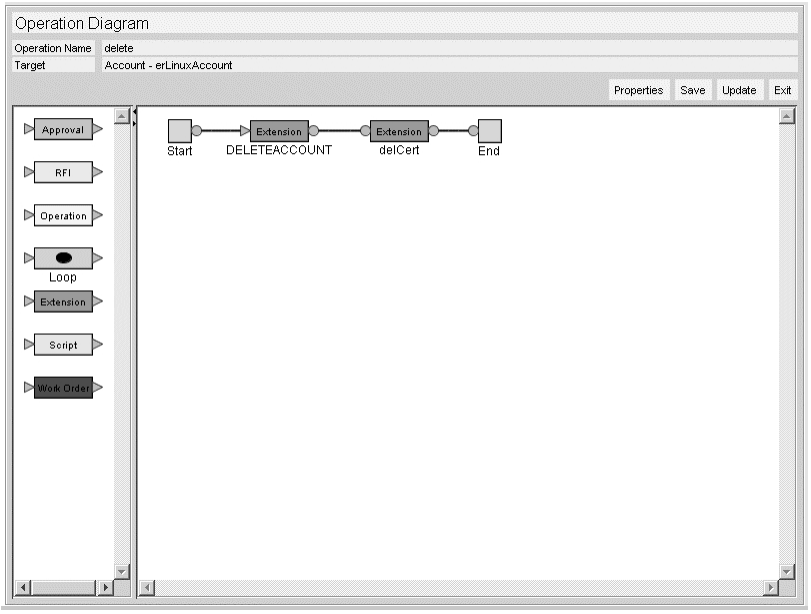
```
<ACTIVITY ACTIVITYID="deleteCert" LIMIT="0">
    <IMPLEMENTATION_TYPE>
        <APPLICATION CLASS_NAME="CertExt"
             METHOD_NAME="deleteCert"/>
    </IMPLEMENTATION_TYPE>
    <PARAMETERS>
        <IN_PARAMETERS PARAM_ID="acct" TYPE="Account"/>
    </PARAMETERS>
</ACTIVITY>
```

Note: Remember to restart ITIM to update the extension after you compile the modified Java, put the jar file in the appropriate location, and modify **workflowextensions.xml**.

2.5.3 Changes in the Workflow

Remove the relevant data variable for the account DN and the script that sets it. In the extension node, put **Entity** as the input parameter.

Figure 4



3 Extending the JavaScript Interpreter

The Java application extensions in the previous section implement the recertification policy correctly. However, they are not very flexible. If the policy changes to require certification every thirty days, or different certification periods for different people, the Java code would have to be changed.

A more flexible solution would be to extend the account schema to allow scripts to set and get arbitrary attributes. That way, the policy can be implemented and maintained in JavaScript. This solution also allows policy changes without restarting ITIM.

3.1 Planning

It is a bad idea to extend account schemas in the directory, because every time the profile is imported the schema is overwritten, and the values in any additional attributes will be lost. A better solution is to extend the schema virtually, using a database table instead of the directory.

To implement this with single value attributes requires three functions to be implemented in Java:

- **setAttr** – set an attribute for an account. If the attribute already exists, overwrite it.
- **getAttr** – get the value of an account attribute.
- **delAcct** – delete all of the attributes of an account. This function will be called when an account is deleted.

To identify the account, the functions will use the account's **erglobalid** attribute. It is a global identifier that is guaranteed to be unique inside an ITIM installation.

3.2 Writing FESI Extensions

ITIM's JavaScript interpreter is an open source package called Free EmcaScript Interpreter (FESI). This interpreter can be extended using Java.

To implement a new function that will be available in JavaScript, follow these steps:

1. Create a new public class that implements the interface **FESI.Extensions.Extension**.

2. In this class, create an inner class that extends **FESI.Data.BuiltinFunctionObject**. This class needs two methods:

   a. A constructor that calls the constructor of the superclass

   b. The function that implements the function for JavaScript, called **callFunction**

3. In the public class, write a function called **initializeExtension** that creates a new object of the inner class and registers it as a property of the global object.

4. Register the public class with FESI. The exact way to do this depends on the program that uses the interpreter. In ITIM, edit the file **$ITIM_HOME/data/fesiextensions.properties**.

Note: The javadocs for the FESI classes are not provided in IBM Tivoli Identity Manager. To get them, download FESI from the Internet and install it.

3.2.1 Simple FESI Extension

This class implements a single function, which returns its first argument.

```
import FESI.Data.*;

/**
 * This code is provided as part of the IBM Tivoli
 * Identity Manager 4.6 - Extending Workflows with Java
 * white paper, and is intended for education use only.
 */
public class SimpleFesiExt
    extends FESI.Extensions.Extension
{
```

Every FESI extension class needs to implement the **FESI.Extensions.Extension** interface, which declares the **initializeExtension** method.

```
    public SimpleFesiExt()
    {
    }
```

If you create a constructor, it has to be public, so FESI will be able to instantiate this class.

```
    /**
     * Implement the echo function.
     */
    class Echo extends BuiltinFunctionObject
    {
```

This is an inner class that implements an extension function. To implement extension functions, extend the class **FESI.Data.BuiltinFunctionObject**.

```
        public Echo(String name,
                FESI.Interpreter.Evaluator evaluator,
                FunctionPrototype fp)
        {
            super(fp, evaluator, name, 1);
        }
```

This is the constructor for the inner class, which calls the constructor for **FESI.Data.BuiltinFunctionObject**.

```
        public ESValue callFunction(ESObject thisObject,
                                    ESValue[] arguments)
            throws FESI.Exceptions.EcmaScriptException
```

The **callFunction** method implements the function that will be accessible from JavaScript. It takes two arguments: a **FESI.Data.ESObject**, which is the object for which the function is called, if it is called as a method, and an array of **FESI.Data.ESValue** objects, which contains the function's arguments.

```
        {
            return new ESString(arguments[0]);
        }
    }
```

This is the function body. It takes the first argument and converts it to a **FESI.Data.ESString**, which is a subclass of **FESI.Data.ESValue**. There are other subclasses for other data types, such as **FESI.Data.ESInteger** and **FESI.Data.ESBoolean**.

This is also the end of the inner class, **Echo**. The following method is part of the public class, **SimpleFesiExt**.

```
    public void initializeExtension(
            FESI.Interpreter.Evaluator evaluator)
        throws FESI.Exceptions.EcmaScriptException
    {
```

This function initializes the extensions. It is called by FESI with the evaluator.

```
        GlobalObject go = evaluator.getGlobalObject();
        FunctionPrototype fp =
            (FunctionPrototype)
            evaluator.getFunctionPrototype();

        go.putHiddenProperty("echo",
           new Echo("echo", evaluator, fp));
     }
}
```

This is the way to register extension functions that are not object methods. Each property has a name, which is a string, and an object, which is its value. In the case of functions, the value is a subclass of **FESI.Data.BuiltinFunctionObject**.

3.2.2 FESI Extensions for Extending Account Schemas

This class implements the extensions to extend account schemas.

```
import FESI.Data.*;
import java.sql.*;


/**
 * FesiExt implements extensions to the FESI JavaScript
 * interpreter inside IBM Tivoli Identity Manager to
 * virtually extend the account schema.
 * </P>
 * This code is provided as part of the IBM Tivoli
 * Identity Manager 4.6 - Extending Workflows with Java
 * white paper, and is intended for education use only.
 */
public class FesiExt extends FESI.Extensions.Extension
{
    // Connection to the database
    Connection dbConn;
```

```
/**
 * Connect to the database.
 */
public FesiExt()
{
    try {
        // Load the JDBC driver for DB2
        Class.forName("COM.ibm.db2."+
                      "jdbc.app.DB2Driver");

        // Connect to the database
        dbConn =  DriverManager.
            getConnection("jdbc:db2:" +
                          // Database name, as
                          // specified in the local
                          // catalog on the ITIM Server
                          "itimdb",
                          // User for the database
                          "enrole",
                          // Password for the database
                          "object00"
                          );
    }
    catch (ClassNotFoundException e) {
        System.out.println("Can't load JDBC driver.");
        System.out.println("Exception: " + e);
        System.out.println("Source the DB2 environment"
                           + " if you haven't yet.");
    }
    catch (SQLException ex) {
        System.out.println("SQL Exception: " + ex);
    }
}

/**
 * Close the database connection.
 */
protected void finalize()
    throws Throwable
{
    try {
        dbConn.close();
    }
    catch (SQLException ex) {
        System.out.println("SQL Exception: " + ex);
    }
}


// This is the evaluation interface that uses this
// extension object.
private FESI.Interpreter.Evaluator evaluator = null;
```



```
/**
 * FesiExt$SetAttr implements the setAttr JavaScript
 * function which sets an account attribute.
 */
class SetAttr extends BuiltinFunctionObject
{
    /**
     * The constructor receives the parameters and
     * uses them to call the constructor for the
     * superclass, BuiltinFunctionObject.
     *
     * @param name name of the JavaScript function
     * implemented by this class.
     *
     * @param evaluator evaluator that uses this
     * function.
     *
     * @param fp function prototype used to create
     * new functions.
     */
    public SetAttr(String name,
            FESI.Interpreter.Evaluator evaluator,
            FunctionPrototype fp)
    {
        super(fp, evaluator, name, 1);
    }
```

```
    /**
       * This function is called by the interpreter
       * when a script calls setAttr. It implements
      * that function. Note that setAttr always overwrites
       * previous attribute values.
       *
       * @param thisObject the object for which this
       * method is a method. Will always be the global
       * object, because these extensions do not use the
       * object oriented features of JavaScript.
       *
       * @param arguments The arguments to the function:
       * <OL>
       * <LI> Account erGlobalID
       * <LI> Attribute Name
       * <LI> Attribute Value
       * </OL>
       *
       * @return always ESUndefined.theUndefined, the
       * FESI version of NULL.
       */
      public ESValue callFunction(ESObject thisObject,
                                  ESValue[] arguments)
          throws FESI.Exceptions.EcmaScriptException,
                 SQLException
      {
          // Delete the current value, if any.
          PreparedStatement pstmt =
              dbConn.prepareStatement(
                  "DELETE FROM acctAttr " +
                  "WHERE " +
                  "globalID='" + arguments[0] + "' " +
                  "AND name='" + arguments[1] + "';");
          pstmt.executeUpdate();
          pstmt.close();

          // Create a row with the new value
          pstmt = dbConn.prepareStatement(
                  "INSERT INTO acctAttr " +
                  "(globalID, name, value) " +
                  "VALUES ('" + arguments[0] + "', '" +
                  arguments[1] + "', '" +
                  arguments[2] + "');");
          pstmt.executeUpdate();
          pstmt.close();

          // Undefined return value
          return ESUndefined.theUndefined;
      }
  }
```



```
/**
 * FesiExt$GetAttr implements the getAttr JavaScript
 * function which gets an account attribute.
 */
class GetAttr extends BuiltinFunctionObject
{
    /**
     * The constructor receives the parameters and
     * uses them to call the constructor for the
     * superclass, BuiltinFunctionObject.
     *
     * @param name name of the JavaScript function
     * implemented by this class.
     *
     * @param evaluator evaluator that uses this
     * function.
     *
     * @param fp function prototype used to create
     * new functions.
     */
    public GetAttr(String name,
            FESI.Interpreter.Evaluator evaluator,
            FunctionPrototype fp)
    {
        super(fp, evaluator, name, 1);
    }
```

```
    /**
       * This function is called by the interpreter
       * when a script calls getAttr. It implements
       * that function.
       *
       * @param thisObject the object for which this
       * method is a method. Will always be the global
       * object, because these extensions do not use the
       * object oriented features of JavaScript.
       *
       * @param arguments The arguments to the function:
       * <OL>
       * <LI> Account erGlobalID
       * <LI> Attribute Name
       * </OL>
       *
       * @return value of the attribute, a string
       */
      public ESValue callFunction(ESObject thisObject,
                                  ESValue[] arguments)
          throws FESI.Exceptions.EcmaScriptException,
                 SQLException
      {
          // Read the value from the database
          PreparedStatement pstmt =
              dbConn.prepareStatement(
                  "SELECT value FROM acctAttr " +
                  "WHERE " +
                  "globalID='" + arguments[0] + "' " +
                  "AND name='" + arguments[1] + "';");
          ResultSet rs = pstmt.executeQuery();
          rs.next();
          String value = rs.getString(1);

          // Cleanup
          rs.close();
          pstmt.close();

          // Undefined return value
          return new ESString(value);
      }
  }
```



```
/**
 * FesiExt$DelAcct implements the delAcct JavaScript
 * function which deletes all the attributes associated
 * with an account.
 */
class DelAcct extends BuiltinFunctionObject
{
    /**
     * The constructor receives the parameters and
     * uses them to call the constructor for the
     * superclass, BuiltinFunctionObject.
     *
     * @param name name of the JavaScript function
     * implemented by this class.
     *
     * @param evaluator evaluator that uses this
     * function.
     *
     * @param fp function prototype used to create
     * new functions.
     */
    public DelAcct(String name,
            FESI.Interpreter.Evaluator evaluator,
            FunctionPrototype fp)
    {
        super(fp, evaluator, name, 1);
    }
```

```
    /**
       * This function is called by the interpreter
       * when a script calls delAcct. It implements
       * that function.
       *
       * @param thisObject the object for which this
       * method is a method. Will always be the global
       * object, because these extensions do not use the
       * object oriented features of JavaScript.
       *
       * @param arguments The arguments to the function:
       * <OL>
       * <LI> Account erGlobalID
       * </OL>
       *
       * @return always ESUndefined.theUndefined, the
       * FESI version of NULL.
       */
      public ESValue callFunction(ESObject thisObject,
                                  ESValue[] arguments)
          throws FESI.Exceptions.EcmaScriptException,
                 SQLException
      {
          // Delete the attributes of this account
          PreparedStatement pstmt =
              dbConn.prepareStatement(
                  "DELETE FROM acctAttr " +
                  "WHERE " +
                  "globalID='" + arguments[0] + "';");
          pstmt.executeUpdate();
          pstmt.close();

          // Undefined return value
          return ESUndefined.theUndefined;
      }
  }
```



```
    /**
     * Register the extensions functions as attributes
     * of the global object. This makes them accessible
     * using function calls in JavaScript.
     *
     * @param evaluator evaluator that uses these functions.
     */
    public void initializeExtension(
            FESI.Interpreter.Evaluator evaluator)
        throws FESI.Exceptions.EcmaScriptException
    {
        // Set the class variable. Not used in these
        // extensions, but might be used in a later version.
        this.evaluator = evaluator;

        // Get the global object of the evaluator.
        GlobalObject go = evaluator.getGlobalObject();

        // Get the function prototype used by the evaluator.
        FunctionPrototype fp =
            (FunctionPrototype)
            evaluator.getFunctionPrototype();

        // Register the functions as hidden properties.
        go.putHiddenProperty("setAttr",
           new SetAttr("setAttr", evaluator, fp));
        go.putHiddenProperty("getAttr",
           new GetAttr("getAttr", evaluator, fp));
        go.putHiddenProperty("delAcct",
           new DelAcct("delAcct", evaluator, fp));
     }
}
```

3.3 Registering FESI Extensions with ITIM

The FESI extensions are registered **$ITIM_HOME/data/fesiextensions.properties**. Any property that starts with **fesi.extension.Workflow** is interpreted as a workflow extension. The value of the property is treated as an extension class by FESI.

For example, this line adds the **FesiExt** class to the JavaScript language used by workflows:

```
fesi.extension.Workflow.siteExt1=FesiExt
```

Note: Java properties can only have one value. Therefore, to add multiple extensions, use multiple properties with different names:

```
fesi.extensions.Workflow.siteExt1=FesiExt1
fesi.extensions.Workflow.siteExt2=FesiExt2
```

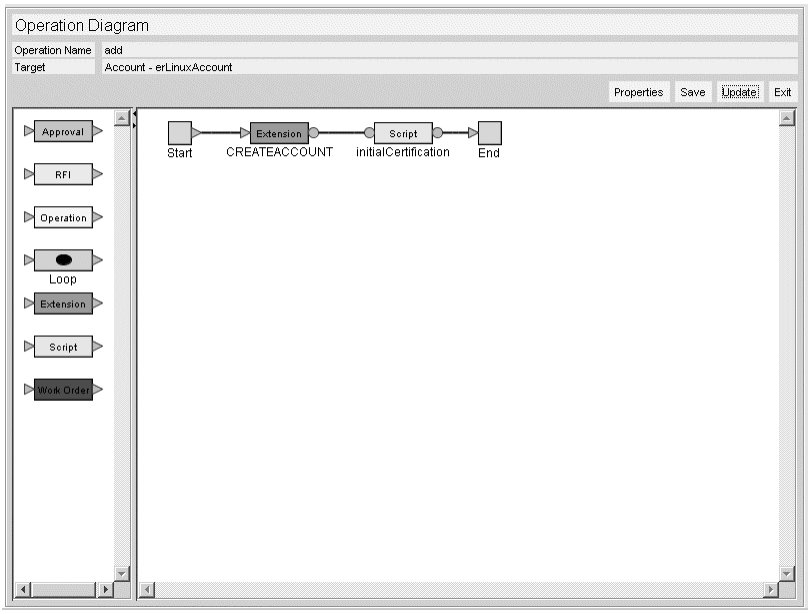
3.4 Using the FESI Extensions

We are now ready to modify the workflows created in section 2.4 to use the FESI extensions.

3.4.1 Initial Certification

Use this workflow for the **add** operation:

Figure 5



As with the distinguished name, the **erglobalid** property is not available in the **account** parameter. The **initialCertification** script, therefore, has to read the accounts and identify the correct one before it sets the attribute:

```
var search = new AccountSearch();
var accounts = search.searchByOwner(owner.get().dn);
var servDN = service.get().dn;
var uid = account.get().getProperty("eruid")[0];
var globalID;
```

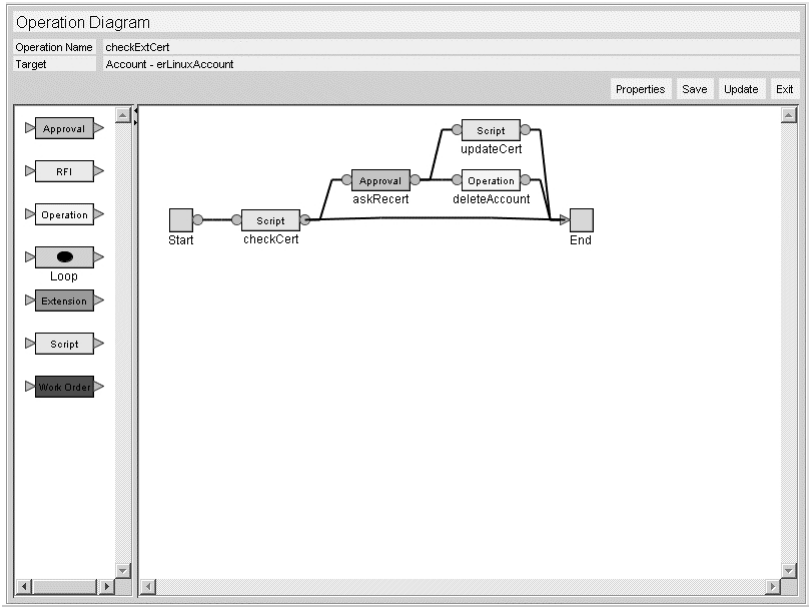


```
for(var i=0; i<accounts.length; i++) {
  if ((accounts[i].getProperty("erservice")[0] == servDN) &&
        (accounts[i].getProperty("eruid")[0] == uid))
        globalID = accounts[i].getProperty("erglobalid")[0];
}

var now = new Date();

// The attribute certDate for this account is
// the number of milliseconds since the beginning
// of the epoch
setAttr(globalID, "certDate", now.valueOf());
```

3.4.2 Recertification

The recertification workflow is similar to the one used with application extensions in section 2.4.2, but it is considerably simpler. In addition to the **Entity** relevant data variable, which is provided by the lifecycle rule, the workflow needs to have **owner**, which is a **Person**, and **service**, which is a **Service**. The approval element requires all three.

Figure 6



The **checkCert** script checks if the certification is still valid. If it is, then it sets the result to **activity.Approved**. If the certification is not valid, **checkCert** sets the result of **activity.REJECTED** and sets the relevant data variables that are required for the approval.

```
// Read the certification date
var acct = Entity.get();
var globalID = acct.getProperty("erglobalid")[0];
var certDate = getAttr(globalID, "certDate");

// Get the current time
var now = new Date();
now = now.valueOf();

// Check if the certification is still valid
if ((now - certDate) > 45*24*3600*1000) {
      // Need to recertify
      activity.setResult(activity.REJECTED);
      owner.set(new Person(acct.getProperty("owner")[0]));
      service.set(
           new Service(acct.getProperty("erservice")[0]));
} else {
      // Account still valid
      activity.setResult(activity.APPROVED);
}
```

The transitions from **checkCert** check the activity results. The transition to **End** has the **Approved** condition, so the workflow will end if the account's certification is still valid. The transition to **askRecert** has the **Rejected** condition, so the workflow will ask for approval for the account if the certification is no longer valid.

Similarly, the transition from **askRecert** to **updateCert**, which updates the certification date if the account is recertified, also has the **Approved** condition. The transition to **deleteAccount** has the **Rejected** condition. Both **askRecert** and **deleteAccount** are identical to the ones used in section 2.4.2.

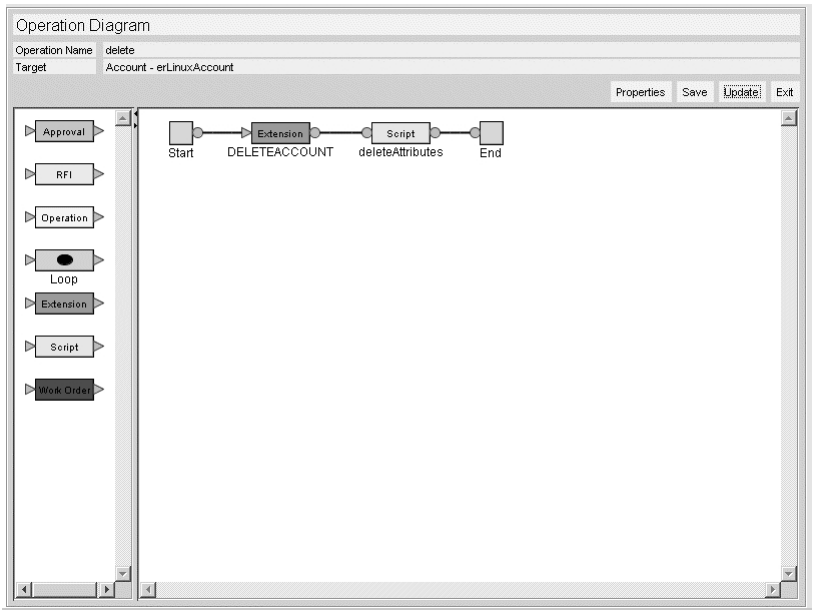
This is the script in **updateCert**, which updates the certification date:

```
var globalID = Entity.get().getProperty("erglobalid")[0];
var now = new Date();

setAttr(globalID, "certDate", now.valueOf());
```


3.4.3 Deleting Certifications

When an account is deleted, its attributes are no longer relevant. To save database storage space and improve database performance, it is best to delete those attributes.

Figure 7



This is the **deleteAttributes** script:

```
delAcct(Entity.get().getProperty("erglobalid")[0]);
```

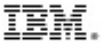

Conclusion

Summary

You should now be able to integrate your own Java modules into IBM Tivoli Identity Manager workflows. This lets you extend workflow functionality to anything the computer can do.

Resources

- David Saucier, *IBM Tivoli Identity Manager, Version 4.5: Defining and Extending Workflows with JavaScript and Application Extensions* Field Guide.
- *IBM Tivoli Identity Manager Extensions*, IBM Publication Number SC32-1683-00. The content of this publication are available on the ITIM Server, at **$ITIM_HOME/extensions/doc** and **$ITIM_HOME/extensions/examples**.



Copyright and trademark information