IBM® Tivoli Workload Scheduler Integration Workbench V8.6.: How to customize your automation environment by creating a custom Job Type plug-in

Abstract
This document describes how to customize your automation environment by creating a custom Job Type plug-in with very few lines of code using the IBM® Tivoli Workload Scheduler Integration Workbench V8.6.

Intended Audience
Read this guide if you want to create your Job Type plug-in in order to extend, integrate and maximize your IBM® Tivoli Workload Scheduler automation environments.
Introduction

The main purpose of IBM Tivoli Workload Scheduler since the very first versions is to minimize the production time and costs, by instructing a production facility on what to do, when to do it, and what IT resources to use. During the years, Tivoli Workload Scheduler became increasingly used in enterprises to orchestrate the integration of real-time business activities with traditional background IT processing, across various operating systems, platforms and business application environments.

The Dynamic Workload Console provides Tivoli Workload Scheduler with a Web Graphical User Interface that acts as a single point of control for the definition and monitoring of background executions in a distributed network of workstations.

The main advantage of a scheduled orchestration consists in cost saving. Usually, when an enterprise invests in scheduling and automation, after an initial high cost (compared with the unit cost of the single operations), it eventually gets an improvement of efficiency and operation in production, therefore reducing costs. The main disadvantages of older versions of the Tivoli Workload Scheduler were related to a specific technical limitation: in fact, previous versions were not able to easily automate all the desired tasks.

Environments and operations, alongside with the business, are increasingly growing in complexity and difficulty of managing: just think of the difficulties often encountered in the integration of business software products with scheduling software, especially in heterogeneous environments. This is true both for architectural solutions with high complexity and for small homemade integrations. In most cases, to reach a good baseline of automation tasks that can be performed, you need to implement a huge amount of scripts, often difficult to write and only supported on a single operating system. This solution has low reliability, is impossible to reuse and its production cost will probably disencourage the prosecution of the implementation.

In order to cope with these kinds of issues, Tivoli Workload Scheduler version 8.6 introduced the new Job Type plug-in. The main feature of this type of job is that everyone can create a new job type that suits various business needs, from the integration with an advanced reporting suite to a file copy. In this way the powerful set of the features of scheduling is included with the flexibility of Java technology, in a single integrated solution. We will now describe how to create a Job Type Plug-in that meets our automation needs.
Scenario

In our example we will make a simple, but powerful job, that with a few lines of code can work with files on the filesystem, interact with a Java-based existing software solution and finally interact with a second software solution through a command line.

In this way we will demonstrate how the use of this new technology can simplify not only the integration with highly technological products, but also the automation of its daily tasks.

Imagine the following scenario in which the user is responsible for the same daily operations:

• from a particular input directory, select a folder (SELECT)

• using Java code, perhaps written by the user, analyze and aggregate all the information contained in the selected folder. The result of this operation is a series of output files (ANALYZE)

• the output, with appropriate input parameters, is given as an input to a command line that compresses all the data and saves this information in a remote repository (COMPRESS)

Then our Job, from now on called Select Analyze Compress (SAC), aims to automate all the tasks described above, therefore simplifying the daily tasks of our example user.
The Select Analyze Compress Job Type Plug-in

First of all, install *IBM Tivoli Workload Scheduler Integration Workbench version 8.6* on your development machine.

The operation is easy, fast and does not require more than 5 minutes.

Once installed, we can start the Tivoli Workload Scheduler Integration Workbench.
First thing to do is to point the correct Target Platform so that our Job Type Plug-in can be compiled.

Do the following in order to create a new Target Platform definition:

- From the toolbar, select Window → Preferences → Plug-in Development → Target Platform.

- Add a new empty target definition and in Locations panel type your `<Integration_Workbench_Home>/eclipse/features/com.ibm.tws.sdk_8.6.0.v<version>/JavaExt/eclipse` directory
Select the checkbox corresponding to the newly created target definition. Click Apply and OK.

Now your Tivoli Workload Scheduler Integration Workbench is ready.

The next step is to create our Job Type Plug-in. Usually, we start from the interface from where our users will define, run, and manage their jobs. This task involves all the activities related to the user's input panels that will be shown into the Tivoli Dynamic Workload Console.

My personal approach for this operation is the following. Create a new empty Java project and call it UI
Inside this new project we can create a new folder, called panels, that will be the container of our ui panels.

The result of this operation is the following

![Image of Package Explorer showing UI, src, JRE System Library, and panels folders]

After that we need to switch to the AUIML Toolkit perspective. The Abstract User Interface Markup Language (AUIML) is the technology that allows the quick creation of our ui panels.
Inside the panel directory, we can now create a new AUIML File called, for example, panel.auiml. The result is
Select the `panel.auiml` that you created earlier. From the Properties panel, select the Resource field. From the drop-down list, select the value property resource bundle and click Save. The result is the following.

At this point you can use AUIML VisualBuilder tool to create our panels. The AUIML VisualBuilder tool provides What You See Is What You Get (WYSIWYG) editing. For example, you can add the following GUI objects: Buttons, check boxes, combo boxes, edit boxes, list boxes, and so on.

At this point we are ready to create our custom panel, with the following features:

- a field for the input directory with an associated button which allows to make an interactive selection (*SELECT*).
- a field for the output directory (*ANALYZE*).
- a combobox for select the compression type and a field for the remote repository hostname (*COMPRESS*).
So, we can start with the creation of the SAC panel:

Inside of it we can add all the necessary fields, captions and buttons.
For every single field and button, we have to define a caption and a name. In particular, the name will be used when we define the Java code.

Now that our panel is ready we can create the project by going through the following steps, that consist in creating a project with our Job Type Plug-in files and all that is needed to deploy them. Do the following:

- From the toolbar select File → New → Project.... The New Project window opens.
- In the list of wizards, double-click first on IBM Tivoli Workload Scheduler and then on TWS Job Type Plug-in Project.
- The TWS Job Type Plug-in Project wizard opens.
• Enter a name for the project.
• Enter a package name for the plug-in.
• Enter the name of the job type as it will be shown on the Dynamic Workload Console panel.
• Enter the panel.auiml name and enter the names of the panel.properties files.
• And finally click Finish

The result looks like this:
The most important class of our project is the `SACAction.java`. In fact, this Java class will contain the code required to make the Job operational. In particular, the code of the `doAction` method will be executed when the Job is in running state.

At this point we can work on the code in order to make the three main steps operational: **SELECT**, **ANALYZE** and **COMPRESS**.
Select

The goal of this step is to make sure that, when we click on the SelectInput Directory button, a popup with show all the available directories, starting from a path defined inside a properties file present on the agent, and allow the user to make a selection.

So, first of all, we need to connect our button with our code using the procedure to define a response action.

The definition of a response action requires:

- The AUIML definition of the button that triggers the command (this operation has already been done during the panel creation)

- The implementation of a canRunOnConnector method in the SACAction.java that specifies if the action can run in the connector or it must run in the agent requesting the command. In our case, the action runs on the agent only

- The implementation of a runExecutorCommand method in the SACAction.java that runs the command. The command value of the button has been defined in the Class name field associated to it AUIML definition. In our case the command is selectInputDir

The code that we are going to add to the SACAction.java class is the following

```java
@Override
public boolean canRunOnConnector(String command) {
    //if command equals selectInputDir it must run on the agent
    if(command.equals("selectInputDir")) {
        return false;
    } else {
        return true;
    }
}

@Override
public ExecutorCommandResponseType runExecutorCommand(
    Map<String, String> parameters, String command, Locale loc)
    throws JobExecutorException {
    ExecutorCommandResponseType response = RestFactory.eINSTANCE.createExecutorCommandResponseType();

    //Create the response type
    ExecutorCommandResponseType response = RestFactory.eINSTANCE.createExecutorCommandResponseType();

    //Create the picklist action
    ExecutorAction showPickListAction = RestFactory.eINSTANCE.createExecutorAction();

    //Create the picklist object
    ShowPickList showPickList = RestFactory.eINSTANCE.createShowPickList();
    showPickList.setTitle("Select Input Directory");

    //connect the picklist selection with a specific field
    showPickList.setFieldName("inputDirectory");

    //get the picklist items
    List pickListItems = showPickList.getItem();
```
//Get the actions list
List<ExecutorAction> actions = response.getActions();

try {
  //if command is selectInputDir
  if(command.equals("selectInputDir")){
    //retrieve the config dir
    String configDir = System.getProperty("ConfigDir");

    //retrieve the sac.properties file
    File props = new File(configDir + File.separatorChar + "sac.properties");
    Properties properties = new Properties();
    //read properties file
    FileInputStream in = new FileInputStream(props);;
    properties.load(in);

    //read the inputDir path property
    String inputDirPath = (String) properties.get("inputDir");

    //scan the input dir to find all
    //the directory inside of it
    File inputDir = new File(inputDirPath);

    //retrieve all the files inside the inputDir
    File[] inputDirChildren = inputDir.listFiles();

    for(File current : inputDirChildren){
      //if the current file is a directory add his name
      //in the picklist items list
      if(current.isDirectory())
        pickListItems.add(current.getName());
    }

    //connect the picklist action with the picklist object
    showPickListAction.setShowPickList(showPickList);

    //add the picklist action to the action list
    actions.add(showPickListAction);
  }
} catch (Exception e) {
  //If something goes wrong create a ShowPopup action
  //and add it to the actions list in order to shown an error
  //message
  ExecutorAction popupAction =
    RestFactory.eINSTANCE.createExecutorAction();
  ShowPopUp popup = RestFactory.eINSTANCE.createShowPopUp();
  popup.setTitle("Error Message");
  popup.setMessage(e.getMessage());
  popup.setHeight("300px");
  popup.setWidth("200px");
  popupAction.setShowPopup(popup);
actions.add(popupAction);
}

return response;
}
**Analyze**

In this step, the selected input directory should be analyzed with some particular custom code written by the user. The result of this operation will generate an output directory. In our example, this code is inside a particular jar called `analyzer.jar`.

The main class of this jar is the *Analyze.java*

```java
package com.yourcompany.analyze;

/**
 * @author Marco Ganci
 */
public class Analyzer {

    /**
     * Constructor
     */
    public Analyzer() {
    }

    /**
     * @param inputDirPath
     * @param outputDirPath
     */
    public void analyze(String inputDirPath, String outputDirPath) {
        //analyze all the file inside the inputDir and produce a new output directory
        ...
    }
}
```

We need to integrate the jar file inside our Job Type Plug-in and its functionality inside the `doAction` method. Therefore, the `analyze.jar` must be added to the Runtime Classpath, modifying the `META-INF/MANIFEST.MF` file. Do the following:

- Copy the `analyzer.jar` inside the `<project>/lib` directory

- Double click the `META-INF/MANIFEST.MF` file. The Overview panel is displayed. Click Runtime. The Runtime panel is displayed. In the Classpath section of the Runtime panel, click New. The New Library window is displayed. Enter the value lib/analyzer.jar.

- Save the changes to the `META-INF/MANIFEST.MF` file.
At this point we can use the code inside the analyzer.jar into our SACAction.java class.
After this operation, the doAction implementation will look like the following:

```java
@Override
public void doAction(SACParametersType SACParameters) throws Exception{
    //retrieve input parameters from panel
    SAC panel = SACParameters.getSAC();

    //select group
    Select selectGroup = panel.getSelect();

    //input directory
    String inputDirPath = selectGroup.getInputDirectory().getText();

    //analyze group
    Analyze analyzeGroup = panel.getAnalyze();

    //output dir
    String outputDirPath = analyzeGroup.getOutDirectory().getText();

    //analyze the input dir and produce an output into the output dir
    Analyzer analyzer = new Analyzer();
    analyzer.analyze(inputDirPath, outputDirPath);
}
```
**Compress**

In this last step we take the compression type value selected by the user, the remote repository workstation and, using a command line already present in the agent machine, compress the output directory and save it remotely.

So we need to modify the doAction method. First of all we need to retrieve the input parameters, the compression type and the remote repository. After that we can launch the command line. In the last line of code we will verify if the operation has been completed successfully or not. In case of errors the job will automatically fail.

The new doAction implementation will be the following:

```java
/*
 * Perform the action
 * Throws an exception on error
 */
@override
public void doAction(SACParametersType SACParameters) throws Exception{
    //Retrieve input parameters from panel
    SAC panel = SACParameters.getSAC();

    //Select group
    Select selectGroup = panel.getSelect();

    //Input directory
    String inputDirPath = selectGroup.getInputDirectory().getText();

    //Analyze group
    Analyze analyzeGroup = panel.getAnalyze();

    //Output dir
    String outputDirPath = analyzeGroup.getOutDirectory().getText();

    //Analyze the input dir and produce an output into the output dir
    Analyzer analyzer = new Analyzer();
    analyzer.analyze(inputDirPath, outputDirPath);

    //Compress
    Compress compress = panel.getCompress();

    //Compression type value, one of zip, tar or rar
    String compressionType = compress.getCompressionType().getText();

    //Remote repository
    String remoteRepository = compress.getRemoteRepository().getText();

    //Create the command for the command line
    String command = "compress -compressionType " + compressionType + " -remote " + remoteRepository + " -dir " + outputDirPath;

    //Run the command line command in this example the compress command line has been already defined in the classpath
    Process commandLineProcess = Runtime.getRuntime().exec(command);

    //Check in the standard output of the
// command if is present the string "success"
BufferedReader stdOutputBuffer =
    new BufferedReader(
        new InputStreamReader(commandLineProcess.getInputStream()));

// put all the stdOut inside this object
StringBuffer stdOut = new StringBuffer();

// read all the stdOut
String line;
while ((line = stdOutputBuffer.readLine()) != null) {
    stdOut.append(line);
}

// verify if the command has been executed with success
// otherwise throw a new exception and the job automatically fails
if (!stdOut.toString().contains("success")) {
    throw new Exception("Operation failed!");
}
Write the Job Log

At this point the base implementation of our Job Type plug-in is completed, but another last thing should be made in order to optimize the job execution and troubleshooting.

Inside the already written code, we easily add the job log associated to the execution of this job. These operations consist in the following action: retrieve the current Job Log file, create an object to write the Job Log, write the Job Log, close the Job Log writer.

The final version of doAction method will be:

```java
/*
 * Perform the action
 * Throws an exception on error
 */
@override
public void doAction(SACParametersType SACParameters) throws Exception{

    //retrieve jobLog file using the getOutputFile api
    File jobLog = new File(getOutputFile());

    //create the job log writer
    BufferedWriter jobLogWriter = new BufferedWriter(new FileWriter(jobLog));

    //retrieve input parameters from panel
    SAC panel = SACParameters.getSAC();

    //select group
    Select selectGroup = panel.getSelect();

    //input directory
    String inputDirPath = selectGroup.getInputDirectory().getText();
    jobLogWriter.append("Input directory is: " + inputDirPath);

    //analyze group
    Analyze analyzeGroup = panel.getAnalyze();

    //output dir
    String outputDirPath = analyzeGroup.getOutDirectory().getText();
    jobLogWriter.append("Output directory is: " + outputDirPath);

    //analyze the input dir and produce an output into the output dir
    Analyzer analyzer = new Analyzer();
    analyzer.analyze(inputDirPath, outputDirPath);
    jobLogWriter.append("Analyze start...");
    jobLogWriter.append("Analyze end");

    //compress
    Compress compress = panel.getCompress();

    //compression type value, one of zip, tar or rar
    String compressionTye = compress.getCompressionType().getText();
    jobLogWriter.append("Compression type is: " + compressionTye);

    //remote repository
    String remoteRepository = compress.getRemoteRepository().getText();
    jobLogWriter.append("Remote repository is: " + remoteRepository);
```
// create the command for the command line
String command = "compress -compressionType " + compressionTye
                + " -remote " + remoteRepository
                + " -dir " + outputDirPath;

// run the command line command in this example the
// compress command line has been already defined in the classpath
jobLogWriter.append("Compression start...");
Process commandLineProcess = Runtime.getRuntime().exec(command);
jobLogWriter.append("Compression end");

// check in the standard output of the
// command if is present the string "success"
BufferedReader stdOutputBuffer = new BufferedReader(
        new InputStreamReader(commandLineProcess.getInputStream()));

// put all the stdOut inside this object
StringBuffer stdOut = new StringBuffer();

// read all the stdOut
String line;
while ((line = stdOutputBuffer.readLine()) != null) {
    stdOut.append(line);
}

// verify if the command has been executed with success
// otherwise throw a new exception and the job automatically fails
if (!stdOut.toString().contains("success")){
    jobLogWriter.append("Operation failed!");
    // close the jobLog writer
    jobLogWriter.close();
    throw new Exception("Operation failed!");
} else{
    jobLogWriter.append("Operation completed with success!");
    // close the jobLog writer
    jobLogWriter.close();
}
Export the Job-Type Plug-in

Now our custom Job is completed and ready to be exported. This step consists in exporting the packaged plug-in files from the Eclipse workbench to a folder on your computer so that you can install them on the agents and make them operational. Do the following to export the plug-in package:

- In the Package Explorer panel, right click on the package name and select Export. In the Export window double-click first on IBM Tivoli Workload Scheduler and then on TWS Job Type Plug-in. A list of defined TWS Job-Type Plug-in projects is displayed.

  - Select the project in the list and enter the destination folder in your computer and click Finish.
The result is the creation of a jar file named `sac_1.0.0.jar` in the destination folder of the computer you selected.
Some more final steps are needed to make the plug-in available to the Dynamic Workload Console and to the agent.
Installing the plug-in on the Master Domain Manager server

To install the plug-in on the Master Domain Manager server, copy the sac_1.0.0.jar file to <TWA_home>/TWS/applicationJobPlugIn on the system where the Master Domain Manager server is installed, and restart the WebSphere Application Server.

This is required to allow the Dynamic Workload Console to discover the jar on the Master Domain Manager.
Installing the plug-in on the agents

You need to install the plug-in on every workstation where you want to run a job of this type (the dynamic agent must be installed on these workstations). The procedure is as follows:

1. Stop the dynamic agent using the `ShutdownLwa` command

2. Copy the plug-in to `<TWA_home>/TWS/JAVAEXT/eclipse/plugins` on the agent

3. Create the sac.properties file under `<TWA_home>/TWS/JAVAEXT/cfg`
4. Inside the sac.properties file, put and set the `inputDir` property, for example

```
inputDir=c:/myinputdir
```

5. Configure the plug-in as follows:

1. Go to: `<TWA_home>/TWS/JAVAEXT/eclipse/configuration`

2. Open the config.ini file in a text editor
3. Add the following value to the first line: `sac@4:start`

4. Start the dynamic agent using the `StartupLwa` command
Test the SAC Job-Type plug-in

Open the Dynamic Workload Console
Open the Workload editor. As you can see, a new Job Definition is available:
Select to create a new SAC job
Select the input directory using the Select Input Directory button
Fill in all the other parameters

After the submission of the Job, using the Monitor Jobs portlet, we can see the execution details of our SAC Job
And finally we can see the JobLog associated with our execution

```
Job Log Details

Job: SACJOB
Workstation (Job): BVTLINUX64
Job Stream: FINAL
Workstation (Job Stream): BVTLINUX64

--------------------------------------------------------------
- JOB : BVTLINUX64\FINAL ([2359 11/22/11] ,[88888888888888888888]) , SACJOB
- USER : tws
tws
- JCLFILE : /space/tws/tws/TWS/SwitchPlan
- Job Number: 28853
- Tue 11/22/11 23:59:55 CET

--------------------------------------------------------------
Input directory is: c:\inputDir\20111120
Output directory is: c:\outputDir
Analyze start...
Analyze end
Compression type is: xip
Remote repository is: remote.myLab.com
Compression start...
Compression end
Operation completed with success!

--------------------------------------------------------------
- Exit Status : 0
- System Time (Seconds) : 0 Elapsed Time (Minutes) : 0
- User Time (Seconds) : 0
- Wed 11/23/11 00:00:11 CET

--------------------------------------------------------------
```
Conclusion

In conclusion, we have discusses how to customize our automation environment by creating a custom Job with very few lines of code. This custom job can perform three different types of operations: select a file, execute Java code and then use a command line. This is possible through the integration of Java and IBM Tivoli Workload Scheduler Integration Workbench. This, combined with the multiple capacities of scheduling of Tivoli Workload Scheduler, allows end users to extend, integrate and maximize their automation environments. Available tasks for automations include not only advanced software suites but also user's daily tasks.

For more information you can see also:

Tivoli Workload Scheduler

IBM Tivoli Workload Automation V8.6 Information Center

IBM Tivoli Workload Automation V8.6 Information Center – Developer's Guide