

Contents

Urban Code Deploy Siebel 8.x Plugin

OVERVIEW and DESIGN

Dennis Stavroyiannopoulos

Senior SOFTWARE ARCHITECT IBM Greece

2015

[Versioning 2](#_Toc433278531)

[Overview 3](#_Toc433278532)

[Siebel Java Data Bean Overview 3](#_Toc433278533)

[Business Object 3](#_Toc433278534)

[Business Component 3](#_Toc433278535)

[Business Service 3](#_Toc433278536)

[How these objects are used 3](#_Toc433278537)

[Getting and changing data remotely on a Siebel Server 3](#_Toc433278538)

[Invoking a Siebel Service 3](#_Toc433278539)

[How the UCD Siebel Plugin is constructed 4](#_Toc433278540)

[UCD Deploy Siebel Step Code structure 4](#_Toc433278541)

[Naming convention 4](#_Toc433278542)

[Creating a new Step. 4](#_Toc433278543)

[Business Object/Component Single Step 5](#_Toc433278544)

[Business Service Step 8](#_Toc433278545)

[Shell Steps 10](#_Toc433278546)

[Steps that are provided for v3 10](#_Toc433278547)

[How you should adapt the UCD Siebel Plugin 12](#_Toc433278548)

[How to use the Siebel jar files 12](#_Toc433278549)

[How to build the Siebel plugin 12](#_Toc433278550)

[How the UCD Siebel Plugin Is used in UCD 13](#_Toc433278551)

[Logging and Error Reporting 15](#_Toc433278552)

# Versioning

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comments |
| 0.7 | 01/07/2015 | Dennis Stavroyiannopoulos | Putting design notes in formal document |
| 0.8 | 10/07/2015 | Dennis Stavroyiannopoulos | Adding screenshots |
| 0.9 | 09/10/2015 | Dennis Stavroyiannopoulos | Correcting some errors and tidying up. |
| 0.91 | 09/10/2015 | Dennis Stavroyiannopoulos | Added extra instructions and cleaned up. |
| 0.92 | 21/10/2015 | Dennis Stavroyiannopoulos | Final Design and break down of steps |
| 0.93 | 21/10/2015 | Dennis Stavroyiannopoulos | Some corrections. |
| 1.0 | 22/10/2015 | Dennis Stavroyiannopoulos | RC1 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Overview

The UCD Siebel Plugin utilizes the Siebel Java Data Bean (JDB) API in order to connect to a specific Siebel Server infrastructure (Siebel Server) and remotely deploy and configure Siebel Servers.

## Siebel Java Data Bean Overview

In order to understand the way the UCD Siebel Plugin works, it is worth describing internal Siebel Architecture. For our purposes Siebel Server basically operates with three major objects: Business Object, Business Component and Business Service. There are other components in the Siebel Server (e.g. Applets), which do not come into play for remote deployments).

### Business Object

This is effectively a collection of Business Components. It is used to group Business Components in order to make accessing them easier.

### Business Component

A Business Component is the main data handling object in a Siebel Server. It is mapped to a database structure and usually an instance of a business component holds data corresponding to a row of a database.

### Business Service

A Siebel Business Service is a piece of code that is executed on the Server when the Business Service is called remotely. It is a generic way of executing code on the Siebel Server.

## How these objects are used

In order to understand how the plugin is constructed let’s explore how the above objects are used.

### Getting and changing data remotely on a Siebel Server

In order to access data on a Siebel Server we have to first find the Business Object on the Siebel Server and then select the specific Siebel Component that we want to access. The easiest way to see how these are structured is to select Help->About View… from the Siebel Web UI menu. The resulting popup will describe the Business Object and Business Components that are used to create a view. Once we discover the Object and Component that are used, the best tool to investigate the structure is Siebel Tools. In Siebel Tools, we can open the Business Object and Component and see the fields that are defined in each one.

Using the information we get from the Siebel JDB API methodology we can get a reference to an object in the Siebel Server and change the data that is contained in it.

### Invoking a Siebel Service

Again the best tool for investigating the available Siebel Services and the methods that can be invoked, is Siebel Tools. Once we know the name of the service that we want to use, we again use the Siebel JDB API to access the service and invoke the methods that it provides with the correct arguments.

For the full reference of the Siebel JDB API please look at <https://docs.oracle.com/cd/E14004_01/books/OIRef/OIRefTOC.html>

# How the UCD Siebel Plugin is constructed

The UCD Siebel Plugin implements a utility library of methods that hides the complexity of the Siebel JDB API. It is designed in such a way in order to facilitate the building of new UCD steps (more on this later). However, the Siebel JDB API expects the client to know the data structure of the Business Components, otherwise the data validation that happens internally will reject the request. So in order to be able to remotely change data or invoke services, we need to implement the Siebel Business Component data structure layer. These are implemented in the UCD Step code itself.

## UCD Deploy Siebel Step Code structure

The UCD Siebel Plugin Steps are designed to follow the specific Siebel Server implementation that is deployed in each customer. Some standard Siebel screens are implemented out of the box, but there is the possibility of adding new Steps for specific Siebel customizations. As mentioned, the UCD Siebel steps follow the way that Siebel Views are structured in a Siebel Server. For each view, a step is implemented. This step is used to add/update one row of data and potentially, if it is needed any associated objects that need to be populated (e.g. parameter Business Components).

The step takes the necessary argument for the business component and parameters.

There are also steps that will need to be run under a shell ***(PLEASE NOTE, for the time being the plugin is ONLY tested for Siebel on AIX – I expect to be little problems for Linux, but there is no provisioning for Siebel on Windows).***

### Naming convention

The steps are named in a specific way based on the names of the Views. They are named like <Applet View>\_<Major View>. For example, for List Of Values (LOV) we need to navigate to Site Map->Data –> List Of Values. Consequently, the step is called Data\_LOV\_single.

There is no specific reason for this naming convention, other than making the use of these steps easier when designing processes, since I expect that the plugin could expand to many different steps and this is the way that I found easier to remember. If you make your own steps, you can name them anything you want.

The shell steps are labelled in a way to just describe their functionality.

Additionally, the steps are labelled with a specific word in the plugin.xml configuration to pick up a specific icon in the UCD designer. So the JDB steps are labelled “Configure”, while the shell steps are labelled “Shell”. Again feel free to prefix anything you like when you make your own steps.

### Creating a new Step.

In order to create a new step, 1 file needs to be created, the groovy script file that will run the step. I expect people to copy an existing step and adapt it accordingly and this is the main reason for providing so many different out-of-the-box steps. **Please keep in mind that these steps were developed and tested against a specific Siebel deployment and if your deployment has customized Siebel Business Components, you might have to adapt the existing steps.**

In the eclipse project, there are two files to help you create a new step, in addition to copying an existing step. These are

#### NONWORKING\_EXAMPLE\_WITH\_COMMENTS\_OBJECT.groovy and NONWORKING\_EXAMPLE\_WITH\_COMMENTS\_SERVICE.groovy

This is a non-working script with extensive comments on the function of each groovy step.

#### testScript.groovy

This is a simple groovy file, where you can hard code the details of the Siebel Connection Broker server and test your assumptions about Siebel Objects and Siebel Components.

### Business Object/Component Single Step

Here’s an example of the step script for the class presented above. Effectively, we set up the properties we need to do the Action and then execute the action. There are many more examples in the “scripts” directory of the plugin project, covering most, if not all, of the different Siebel Objects combinations.

More details can be seen in the *NONWORKING\_EXAMPLE\_WITH\_COMMENTS\_OBJECT.groovy* file.

/\*

\* This is a little list in which we add all the Siebel components we need to clean

\* in an error condition.

\*/

**def** siebObjList = []

//Create a new Siebel Java Data Bean Wrapper.

SiebelJavaDataBeanWrapper mySiebJDB = **new** SiebelJavaDataBeanWrapper();

/\*

\* Here we connect to the Siebel Connection Broker Server. We connect to the

\* EAIObjMgr\_enu Object Manager, which seems to be the most well behaved for Siebel.

\* We pick up all the options from the Step options.

\*/

**if**(!mySiebJDB.connect(props['SiebelCBServer'], props['SiebelCBPort'], props['SiebelEnt'], "EAIObjMgr\_enu", props['SiebelUser'], props['SiebelPass']))

{

println "${myName}: Connection to Siebel Server failed - check log upwards";

System.exit(-1);

}

/\*

\* Let's create a utilitity object. We pass it the reference to the

\* Siebel Java Data Bean wrapper, since some things are used internally

\* in the utility class.

\*/

UCDSiebelUtility myUCDStepObject=**new** UCDSiebelUtility(mySiebJDB);

/\*

\* These are the objects that we are going to use later. We define them

\* here in order to be able to use them in the closures we define later

\*/

SiebelBusObject m\_BusObj = **null**;

SiebelBusComp m\_BusComp = **null**;

SiebelJavaDataBeanWrapper m\_dataBean=myUCDStepObject.getSiebelJDB()

/\*

\* This is the getObjects() closure that get the object and component(s)

\* we want. It's defined in every script, since it is different in every

\* script. No matter how many components or objects we want to use, it is

\* better to defined them here. We also add them to the list in case we have

\* to release them in an error exit from the script.

\*/

**def** getObjects = {

//BUSINESS OBJECT NAME

String busObjName = **new** String("Admin Signal VOD Definition");

m\_BusObj=m\_dataBean.getBusObject(busObjName);

**if**(m\_BusObj==**null**)

{

**return** **false**;

}

//BUSINESS COMPONENT NAME - child of BUSINESS OBJECT

String busCompName = **new** String("Signal VOD BusComp");

m\_BusComp=m\_dataBean.getBusComp(busCompName, m\_BusObj);

**if**(m\_BusComp==**null**)

{

**return** **false**;

}

siebObjList.add(m\_BusObj)

siebObjList.add(m\_BusComp)

**return** **true**

}

/\*

\* This is a little closure that we will use if we exit with an error.

\*/

**def** exitGracefullyWithFail = {

siebObjList.each {**if**(it){it.release()}}

mySiebJDB.disconnect()

System.exit(-1)

}

/\*

\* Let's call getObjects() and if it succeeds, let's continue.

\*/

**if**(getObjects())

{

System.out.println(myName+": SetProperties");

//BE VERY CAREFUL WITH THE CASE OF THE SIEBEL FIELDS, they are case sensitive.

/\*

\* Here we set the properties of the object we want to set. The names of

\* the properties in the dProps hash is the same as the field names in the

\* Business Component. Here we translate the step properties to

\* Business Component Properties.

\*/

Properties dProps=**new** Properties();

dProps['VOD Name']=props['SignalName'];

dProps['Description']=props['SignalDescription'];

/\*

\* If we only want to search on different fieldset, then we also

\* make a dPropsSearch hash that only holds the fields that we want

\* to perform a query on in the Siebel Component.

\*/

Properties dPropsSearch=**new** Properties();

dPropsSearch['VOD Name']=props['SignalName'];

/\*

\* This is a little flag to see if this is an update.

\* If it is an update and we don't find anything in the

\* Business Component, then we fail. If we don't set this field,

\* if the entry is not found it is created, otherwise it is updated.

\*/

**boolean** isUpdate =props['Update'].equals("true")

/\*

\* Here we create a Siebel Property set that we are going to use when

\* we create the Business Object row.

\*/

SiebelPropertySet mainSiebelProps=myUCDStepObject.createSiebelPropertySet(dProps)

/\*

\* If we have a dPropsSearch, then we create a search string on those,

\* otherwise we create a search string on all fields that we have defined

\* in dProps.

\*/

String mySearchString=**null**

**if**(dPropsSearch)

{

mySearchString=myUCDStepObject.createSearchExpr(dPropsSearch)

}

**else**

{

mySearchString=myUCDStepObject.createSearchExpr(dProps)

}

/\*

\* Let's actually set the fields and write the record.

\*/

**if**(!myUCDStepObject.setObjectFields(m\_BusComp, mainSiebelProps, mySearchString, dProps, isUpdate))

{

System.out.println(myName+": doAction failed, check log upwards.")

//If something fails, call the exitGracefullyWithFail closure.

exitGracefullyWithFail.call()

}

/\*

\* Everything went well, let's release the object. I am not using the list

\* on purpose here, to make sure that we know what we are releasing.

\*/

m\_BusComp.release();

m\_BusObj.release();

}

**else**

{

//getObjects() failed, call the exitGracefullyWithFail closure.

exitGracefullyWithFail.call()

}

//We are finished, disconnect from the Siebel server.

mySiebJDB.disconnect();

## Business Service Step

The Business Service Step is slightly different than the previous step because it does not directly manipulate data. Instead it invokes a method (effectively calls a function) on the Siebel server, which could be anything (eScript, C++, something else) with the correct arguments and receives the results of the method. Effectively, the method implements business and data logic and manipulates data in the background. The only thing we need to pay attention to here is that the arguments we provide are the correct ones. An example of a service step would be:

/\*

\* This is a little list in which we add all the Siebel components we need to clean

\* in an error condition.

\*/

**def** siebObjList = []

//Create a new Siebel Java Data Bean Wrapper.

SiebelJavaDataBeanWrapper mySiebJDB = **new** SiebelJavaDataBeanWrapper()

/\*

\* Here we connect to the Siebel Connection Broker Server. We connect to the

\* EAIObjMgr\_enu Object Manager, which seems to be the most well behaved for Siebel.

\* We pick up all the options from the Step options.

\*/

**if**(!mySiebJDB.connect(props['SiebelCBServer'], props['SiebelCBPort'], props['SiebelEnt'], "EAIObjMgr\_enu", props['SiebelUser'], props['SiebelPass']))

{

println "${myName}: Connection to Siebel Server failed - check log upwards"

System.exit(-1)

}

/\*

\* Let's create a utilitity object. We pass it the reference to the

\* Siebel Java Data Bean wrapper, since some things are used internally

\* in the utility class.

\*/

UCDSiebelUtility myUCDStepObject=**new** UCDSiebelUtility(mySiebJDB)

/\*

\* These are the objects that we are going to use later. We define them

\* here in order to be able to use them in the closures we define later

\*/

SiebelService m\_BusServ = **null**

//This is the SiebelPropertySet that is the input to the service.

SiebelPropertySet m\_FieldSetIN = **null**

//This is the SiebelPropertySet that is the output to the service.

SiebelPropertySet m\_FieldSetOUT = **null**

//This is the string that we are going to use to call a method on the service.

String m\_sMethod = **null**

//Let's get a reference to the SiebelJavaDataBean.

SiebelJavaDataBeanWrapper m\_dataBean=myUCDStepObject.getSiebelJDB()

/\*

\* This is the getObjects() closure that get the object and component(s)

\* we want. It's defined in every script, since it is different in every

\* script. No matter how many components or objects we want to use, it is

\* better to defined them here. We also add them to the list in case we have

\* to release them in an error exit from the script.

\*/

**def** getObjects = {

//BUSINESS OBJECT NAME

String busServiceName = **new** String("Workflow Administration")

m\_BusServ=m\_dataBean.getBusService(busServiceName)

**if**(m\_BusServ==**null**)

{

**return** **false**;

}

//Add the object to a small list to know what to clean up later

siebObjList.add(m\_BusServ)

**return** **true**

}

/\*

\* This is a little closure that we will use if we exit with an error.

\*/

**def** exitGracefullyWithFail = {

siebObjList.each {**if**(it){it.release()}}

mySiebJDB.disconnect()

System.exit(-1)

}

//In this case we are going to use the Import method.

m\_sMethod="Import"

/\*

\* Let's call getObjects() and if it succeeds, let's continue.

\*/

**if**(getObjects())

{

System.out.println(myName+": SetProperties");

//BE VERY CAREFUL WITH THE CASE OF THE SIEBEL FIELDS, they are case sensitive.

/\*

\* Here we set a couple of propeties that we are going to use later.

\* Here is where we map the \*METHOD ARGUMENTS\* to our properties.

\*/

Properties dProps=**new** Properties()

dProps['FileName']=props['SM\_XMLPath']

dProps['FileType']="XML"

/\*

\* Let's instatiate the property sets (NOT SET VALUES) that we are going to use.

\* The IN set is initialized with the values from dProps and the OUT

\* is initialized empty (will be set by the Service).

\*/

m\_FieldSetIN=myUCDStepObject.createSiebelPropertySet(dProps)

//Let's set the values for the property set.

myUCDStepObject.setSiebelPropertySet(m\_FieldSetIN, dProps)

//Nothing to set here, this is supposed to be empty.

m\_FieldSetOUT=myUCDStepObject.createSiebelPropertySet((Properties)**null**)

**if**(m\_FieldSetIN==**null**)

{

//Something failed with the instatiation, fail gracefully.

exitGracefullyWithFail()

}

//Lets call the service.

String lOutputPropName=**new** String()

/\*

\* Let's invoke the method with the arguments we have.

\*/

**try**

{

m\_BusServ.invokeMethod(m\_sMethod, m\_FieldSetIN, m\_FieldSetOUT)

}

**catch** (SiebelException e)

{

//Something failed, exit gracefully with fail.

exitGracefullyWithFail()

}

//Release the Business Service object.

m\_BusServ.release()

}

**else**

{

//getObjects() failed, call the exitGracefullyWithFail closure.

exitGracefullyWithFail.call()

}

System.out.println(myName+": disconnect")

mySiebJDB.disconnect()

As you can see in order to call a method, we have to define three arguments. A Siebel Property Set for input, a Siebel Property Set for output and the method that we need to call. All Business Service calls are more or less the same (at least I have not seen anything else).

## Shell Steps

The Siebel shell steps are like any other UCD shell step. They just codify the Siebel command line steps (like starting/stopping servers, generating web scripts etc).

## Steps that are provided for v3

Business Object steps

* Configure Applications\_Views\_create\_single
* Configure Applications\_Responsibilities\_create\_single
* Configure Application\_Responsibility\_View\_associate\_single
* Configure Application\_PredefinedQueries\_single
* Configure Data\_LOV\_LOV\_single
* Configure Enterprises\_ProfileConfiguration\_single
* Configure Group\_Position\_single
* Configure Group\_InternalDivision\_single
* Configure Integration\_SymbolicURLList\_single
* Configure OrderManagement\_Signals\_single
* Configure RuntimeEvents\_Events\_single
* Configure ServerConfiguration\_JobTemplates\_single

Business Service Steps

* Configure BusProc\_WorkfImport\_single
* Configure BusProc\_WorkfDeploy\_single
* Configure ImportSiebelMessage\_single (Data Maps, Web Services and some others – look for SiebelMessage in XML).

Shell steps

* Shell Import Repository File
* Shell Start or Stop Servers
* Shell Copy or Move File
* Shell Generate Browser Scripts

# How you should adapt the UCD Siebel Plugin

From what I have seen, Siebel is a highly customizable product. This is a slight problem for an automated deployment integration, since a lot of things will change from one Siebel implementation to another.

This is exactly the reason for explaining how the code is designed and structured and providing a lot of examples of different kinds of objects and ways of manipulating them.

The idea here is that hopefully, I have provided enough examples of functionality that when you find something that is not covered out of the box, you will easily be able to adapt (read: copy-paste-rename) an existing plugin to suit your needs.

The scripts provided for each step are at most 200 lines of Groovy code, with around 100 lines of actual code that might need to be changed. It should be simple enough for someone of moderate Groovy or Java skills to adapt.

# How to use the Siebel jar files

I cannot include the Siebel jar files as part of the plugin distribution. You will need to put them into the Siebel plugin, under the lib directory. The plugin should not build and will definitely not run unless you put the jars in the right place and add them in the Java Build Path of the project (right click on project, select Properties, select Java Build Path, click on the Libraries tab and click on the Add JARs button).

We need two jar files:

* **Siebel.jar**
* **SiebelJI\_enu.jar (this is Siebel J I)**

You should find these files under your Siebel installation directory. They usually can be found in the ***$SIEBEL\_HOME/classes/*** directory.

**If you don’t want to build the plugin, just pick up the already build plugin, unzip it, place the Siebel jars in the lib directory, zip it up again and deploy the resulting zip.**

# How to build the Siebel plugin

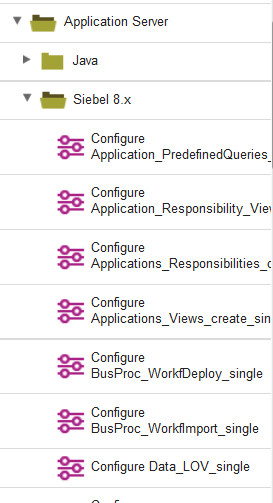
You will need a version of eclipse (I use Luna), and the UCD eclipse plugin. You can find the instructions on how to do that under the following link in the devkit/ [DevKitInstall.pdf](https://hub.jazz.net/project/ucplugin/UrbanCode%20Deploy%20Plug-in%20For%20Eclipse%20Development/overview?cm_mc_uid=79782447039114296083406&cm_mc_sid_50200000=1437513663#https://hub.jazz.net/project/ucplugin/UrbanCode%2520Deploy%2520Plug-in%2520For%2520Eclipse%2520Development/ucplugin%2520%257C%2520UrbanCode%2520Deploy%2520Plug-in%2520For%2520Eclipse%2520Development/_Z54kYGNpEeOEAdgB1rj08Q/_Z7JvsGNpEeOEAdgB1rj08Q/devkit/D) file.

<https://hub.jazz.net/project/ucplugin/UrbanCode%20Deploy%20Plug-in%20For%20Eclipse%20Development/overview?cm_mc_uid=79782447039114296083406&cm_mc_sid_50200000=1437513663>

At the time of writing, the current version is 2.1 and this is what I used to build the plugin.

Once installed, you can build the plugin by opening the build.xml file, right clicking on it and selecting Run As->Ant Build. This will create a UCDSiebelPlugin\_v3.zip file under the releases directory of the project that you can import directly to Urbancode Deploy.

# How the UCD Siebel Plugin Is used in UCD

After the plugin is installed, you can find it in the processes tab under Applications Server/Siebel 8.x UCD plugin. The steps that the plugin implements appear in the tree under the Siebel 8.x UCD plugin

Once you select a step and drag it to the design area the properties of the step pop up. Usually, the connections properties (Server, Port, Enterprise, Username and Password) will be picked up by the component properties (see UCD manual), so they are hidden by default. If you want to test something, just click on the “Show Hidden Properties” checkbox.

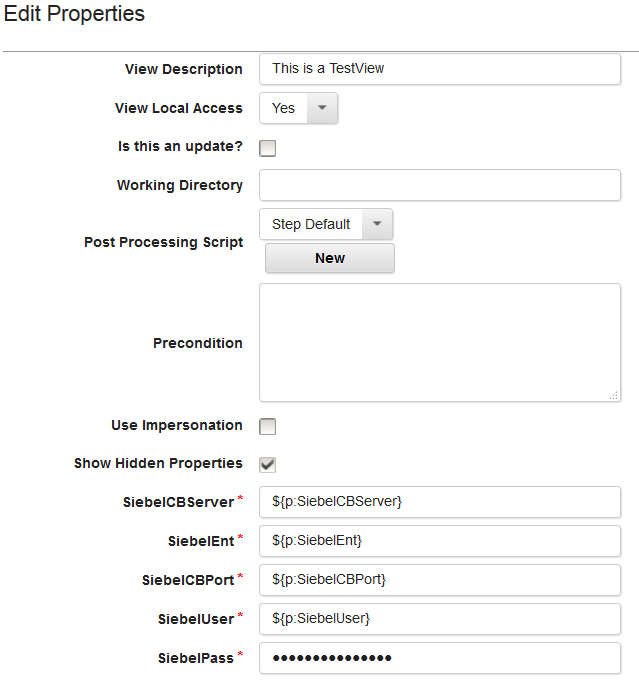
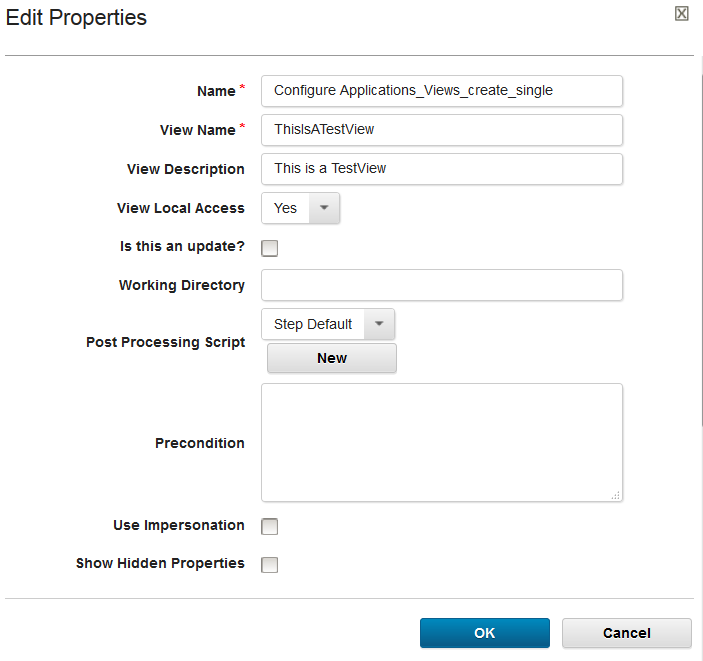


Figure : The View creation Step without and with the hidden properties.

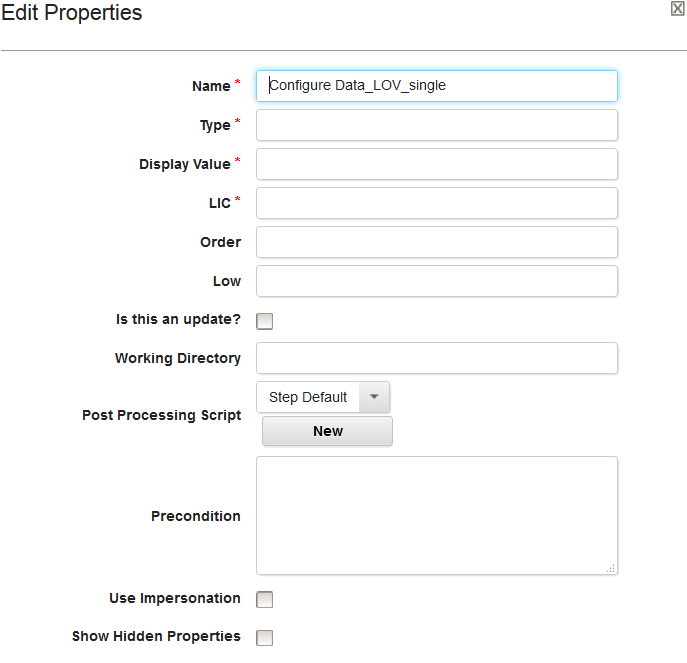


Figure : The Data List Of Values step with the required fields.

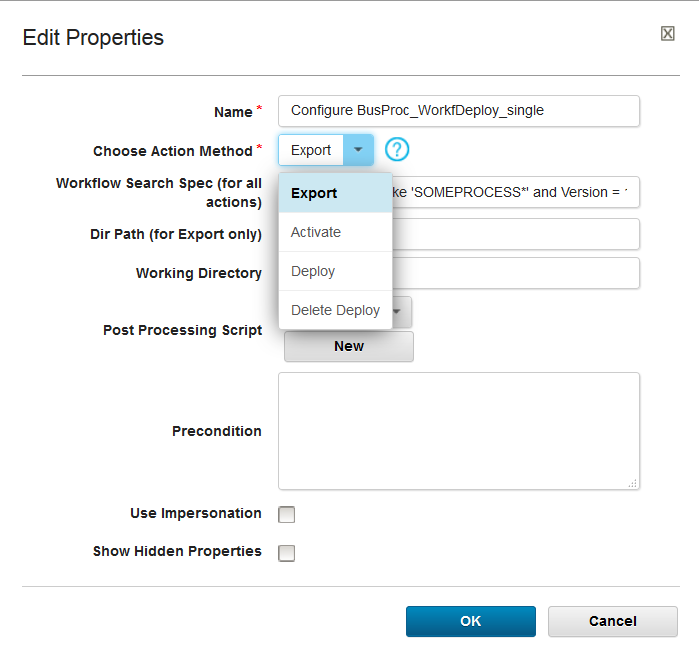
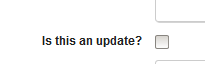


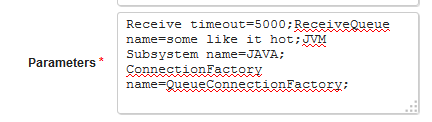
Figure : The Workflow Deploy step with the drop down of the actions that can be done on a workflow (just used deploy for the name).

The step specific properties are the properties that are needed by the step. The required ones are marked by a small red star.

**Finally, there is Update check box for BusObj/Comp steps, which is very important. If this check box is ticked then the step will expect to find the Business Object/Component instance that we are looking for and if it does not it will fail. If this box is not checked then if we don’t find the data we will create it.**



If Parameters are needed then a textbox appears. The parameters should be added as name value pairs separated by a semicolon (<name>=<value>**;**).



# Logging and Error Reporting

The Siebel Plugin will output a lot of information by design, In order to be able to diagnose problems very quickly.

If at any step the plugin fails, it will output the Siebel Error code as well as the complete stack trace of the code in order to see exactly at which point it failed.