1 Best Practices for using own VBScript code in ECR

There are different approaches for including own VBScript code in SwyxWare's extended call routing. This document will give some background information how ECR scripting is structured, how the graphical script editor generates scripts and how SwyxServer loads them. That information helps understanding the best practices hints which follow.

2 Background information

2.1 SwyxWare ECR script structure

Cal routing scripts are common VBScript files generated by Swyx and shipped with SwyxWare or generated by call routing manager (CRM) and graphical script editor (GSE). ECR scripts offer all features of VBScript. SwyxServer additionally recognizes two additional statements, includes and signatures. Both constructs are inside vbscript comments. This ensures that from the vbscript engine's point of view ECR scripts are normal vbscripts.

2.1.1 Script Includes

SwyxServer recognizes an include statement with works similar as the "#include" in C/C++.

```
'#include "<filename>"
```

<filename> is the name of the file to include, e.g. `ruleMyECRRule.vbs`. In SwyxWare v6.0x and older, relative and absolute paths are supported, but not recommended. In SwyxWare v6.10 and Hosted SwyxWare all path components are ignored. Find more details about the script loading process below

2.1.2 Script signature

To be able to distinguish CRM from ECR scripts all script files have a digital signature at the end of the file in form of several VBScript comment lines

```
'' SIG '' Begin Signature
'' SIG '' XQFPPLPIQDJDOBFPTNNNVNVWZOOOSMSOFMZKAOUHFWJBWMKKGPBUKUKEPOTA
'' SIG '' LNRLPASYSOPCPSRKOFZQPLPGPKWKYYGPYEKAKVQPYVYZBIZAKLFOJEIFLMIK
'' SIG '' PDMOGSJUENFARNWNMGQTJPPBLEMIIHZBIUKMBZLPZDMKCOGVTKSMNWZNCCDO
'' SIG '' YDPPQMNPBWNVELIGVOWSWIOAJWPCLKTNNSXPOAKPTCCPMKCSGLKEAPYRROPM
'' SIG '' VUKYYKNZOMRKOTUOOOJPUFOFKBJPHQCLLNKEKYNGXIPRLIKMXLAMQUKYBHXL
'' SIG '' HNBADUBXGAUERKLVIOJGNMMKJZTLJNKLPQOJOOMLWLQVMNPIOXPNYCSBWWGY
'' SIG '' SIAUBDIQUGTOKZMYUPWMVCLTKJIJGGPJCOLOZKQLPAULVIFGFHTLLLHRDYXM
'' SIG '' NGFLPONOXBFFPNQNCBBDNOWOQQMPCHCRBJSOYWNWLPFMIWXAEEMVUYNCQMQP
'' SIG '' MXNRHSLHXLFJONTTAALZJXLZHZAEPUXFXPUDWNOGALUSLMSYJZVPOYVKFXLP
'' SIG '' UPZYIKNONKANPYMGDXMNBMAOZPJTCYTOUVUOBJOAZNWNKLPNISKKIEDYLHFP
'' SIG '' LIMROVYBUHARONDMXKVJNFAKRXGZEHGBTOVJZFSRNONIDLENOOKGJPMKNERX
'' SIG '' MHKZXIMLZTIVKPNDAZDESPXIUMTVSCRLBOZRVEPPLYZMNUPRKICAHMDMFNNL
'' SIG '' MVCPMZAMKFOEVQUFNKPLNHYZJJKHXJNRUFOKCLHMPHNYOHBQPYKECKJYMGJQ
'' SIG '' GOACMQAKZHXRFPEXIBMPOPYLSWMPMITSPKPOLNREUKIHLNJXHYXYRCAAAPCG
'' SIG '' SQKMLPTKPZSHAHHZCLMWPOLIMKYGTIFXSRGAKNNPQYNKPJMVCMXGLSUOPLMV
'' SIG '' ONOGPILHRATEUKIOTKPIMABTXNKGINHOLQLHUZZUPTBMIMNMWDVJNYLVWSKW
'' SIG '' WQFHSLVMIHFFBSKKNQAVVKSPRPSPRGLXERKBGTMMUNUTVZOGGPASMRNBLMMG
'' SIG '' WXRD
'' SIG '' End Signature
```

Scripts generated by CRM have a different signature than script generated by GSE.

2.1.3 GSE rules and actions

Graphical script editor rules and actions are stored in two files, a binary .rse (rule)/.ase (action) and a .vbs file. The filename always is the rule/action name prefixed by “rule” or “action”, e.g.

ruleMyFirstGSERule.vbs

actionRedirection.vbs

The binary .rse/.ase files are used by GSE to store additional information like the graphical layout. SwyxServer only reads the .vbs files.

GSE creates one VBScript function per GSE rule or action. The function is stored inside a vbs file named as described above. Example: A GSE rule MyFirstGSERule is a function called ruleMyFirstGSERule and is store inside a file named ruleMyFirstGSERule.vbs. The function signature always looks like this

```
Function ruleMyFirstGSERule(ByRef rInputParams)
```

Find more details about the parameter in the next section below. The return value of a GSE rule is a Boolean which is true when the rule has been left via the **Rule Executed** block and false otherwise.

Each GSE block like **Play Announcement** or **Connect To** is implemented as separate vbscript function located inside a special vbscript file called template_v4.3.vbs[1]. Some more complicated functions like the standard remote inquiry is implemented as separate GSE action. GSE automatically writes an appropriate include statement when creating a rule vbs file with such a block. Using separate GSE actions has the additional advantage that the functionality can be changed by using GSE. If the remote inquiry shipped with SwyxWare does not fit, an administrator can just use GSE to change it as needed.

2.1.4 Rule/Action parameter

Each action or rule can have one or more parameters defined via the **File – Script properties** function in GSE. When using a rule in CRM the rule’s description is shown with the parameters as hyperlinks so that the user can define the actual values. SwyxWare uses a custom vbscript parameter class to provide parameters to GSE rule functions. We could have just generated specific rule functions, e.g.

```
Function ruleMyFirstGSERule(szParam1, param2, param3)
```

where the three parameters are the ones defined in the rule’s script properties. But this approach has a drawback. When this rule is used inside a callrouting script, the user’s main callrouting.vbs would have a statement like this:

```
bSkipRemainingRules = ruleMyFirstGSERule(“value1”,2,3)
```

[1] The name is misleading. The file has nothing to do with templates, but nobody seemed to question the naming and therefore it got released with SwyxWare v1.0. However, after having this name in a shipped version it stuck to remain backward compatible. The version number suffix was meant to distinguish different versions of the file, but we found that this was not optimal. It prevented older call routing scripts from using new functionality or bug fixes. Since SwyxWare v4.3 the name remained unchanged and the file kept compatible.

Now let's assume that the user changes his GSE rule to have one more parameter. Now the function signature would be

```
Function ruleMyFirstGSERule(szParam1, param2, param3, szParam4)
```

This would break the user's call routing, because callrouting.vbs calls ruleMyFirstGSERule with three parameters. VBScript does not support optional parameters. GSE would have to recreate the user's callrouting.vbs. But GSE cannot easily do this, because generating call routing.vbs is functionality of the CRM module. We could have done this by spending some additional effort. But that would not really help if the GSE rule is not used by one single user, but by all users, because the administrator has provided it globally. Such a change would require recreating all user's callrouting.vbs. But it gets worse. Consider a GSE action which is used in arbitrary GSE rules by many users, e.g. actionStandardRemoteInquiry. Adding parameter would break all GSE rules which use that action. Scanning all user's rules and action an recreate them to match the new definition is simply not practical.

SwyxWare always uses the same function signature for all rules and actions:

```
Function ruleMyFirstGSERule(ByRef rInputParams)
```

rInputParams is a GSEParamList object which is defined as VBScript class inside template_v4.3.vbs. It's a container of name/value pairs. A GSE rule or action extracts the parameter like this:

dim param1

param1 = rInputParams.GetParamValueByName("param1", "defaultvalue")

If the rInputParams object has no parameter with name "param1" the defaultvalue "defaultvalue" is used.

A GSE rule or action calls such a rule/action function like this:

```
Dim rInParams
Set rInParams = new GSEParamList
rInParams.Count = 2
rInParams.AddParam "Param1", "1234567890"
rInParams.AddParam "Param2", 42

actionMyAction rInParams
```

If actionMyAction is later changed to have more parameter, this will still work as long as the action author considers backward compatibility. Adding or removing parameters is ok. Changing of the parameter meaning probably fails.

2.1.5 GSE Action Output Parameter

GSE actions are able to return values to the calling GSE rule. This is done by updating the parameter inside the action like this:

```
rInputParams.UpdateParamValue "Param1", szParam1
rInputParams.UpdateParamValue "Param2", Param2
```

This updates the Param1 and Param2 value inside the rInputParam object by assigning them the values of variable szParam1 and Param2. rInputParam is a ByRef parameter to the action function, so the updated values are visible in the calling GSE rule.

The calling GSE rule uses the updated parameter values when the action has been called with a variable. You do this by setting the actual parameter value in a **Run GSE Action** block to something like

=MyGSERuleVariable

MyGSERuleVariable has been set before with a **Set Variable** block. Inside the GSE rule script your action is now called like this:

```
Dim rInParams
Set rInParams = new GSEParamList
rInParams.Count = 2
rInParams.AddParam "Param1", MyGSERuleVariable
rInParams.AddParam "Param2", "42"

actionNewAction rInParams

MyGSERuleVariable = actionNewAction.GetParamValueByName("Param1", p1)
```

2.1.6 GSE action return values

The **End Action** block of a GSE action defines the return value. It's 0 per default. You can either specify a constant or use a variable. The return value is stored in a special field inside the GSEParamList object. The calling GSE rule uses that value to determine the **Run GSE Action** block exit to use.

2.1.7 Callrouting.vbs

The starting point for call routing scripts is a function called main() inside a script file called callrouting.vbs. When a user changes his/her call routing, CRM generates this file. CRM rules are implemented as separate VBScript functions in that file, too. The main() function calls all enabled CRM rule functions and all enabled GSE rule functions. When a user adds or enables a GSE rule, CRM creates adds the appropriate include statement to callrouting.vbs and modifies the main() function to call the rule.

Main() also calls a particular function at the beginning to check for special call types, like ad-hoc conference establishments or remote inquiry. If such a special call is detected all CRM/GSE rules are skipped.

2.1.8 Preprocessing

Per default the main() function in callrouting.vbs calls a GSE rule named "Preprocessing", i.e. calls a vbscript function rulePreprocessing.vbs. The default implementation of the rule does nothing. A customer can add own functionality here for tasks to be executes for all users' calls.

2.1.9 Own script code

SwyxWare offers several methods to include own script code inside call routing scripts.

- **Start Block**
  Each GSE rule or action has a start block. The properties of this block allow defining own script code. Everything entered there is written verbatim into the rule/action script file. It's the global scope, i.e. outside the rule/action function. Therefore it is executed as soon as the script is loaded, even before the main() function in callrouting.vbs is called. Or in other words, it is execution is not related to the execution of the rule/action. When you enable a rule or use a GSE action in an enabled rule, start block code of that rule/action is called immediately after the script has loaded.
  If you have **Start** block code in a GSE action and this action is used more than once, e.g. from different rules, your code is executed only once, because the GSE action script file is included only once. See 2.2 below.
- **Insert Script Code Block**
  VBScript code defined in this block is also written verbatim into the script inside the rule/action VBScript function. It is executed when the block is executed.
- **Own VBScript file**
  You can place code in a separate vbscript file and include that by manually adding an "include" statement in the Start block or an insert code block.
  - Include own file via **Start** block
    During script loading SwyxServer replaces the include statement with the contents of the included files. As a consequence, global code in that file is executed as directly entered start block code. Script functions or subs in your own file are executed when they're called
  - Include own file via **Insert Script Code** block
    The "include" statement entered in this block is written verbatim inside the rule/action vbscript function and is replaced during script loading with the contents of the included file. Example

    ```
    Function ruleMyRule(…)
    …
    '#include "MyOwnCode.vbs"
    …
    End Function
    ```

    After loading the include statement is replaced with the contents of MyOwnCode.vbs. As a consequence you cannot use functions or subs in code included there, because nested functions are not supported.

  Note: Own script files need to be signed, too. You can use a GSE rule script, add your code in the start block and refer to that rule script in the include statement. A better alternative is to use a VBScript editor of your choice and sign the file manually. See http://www.swyx.com/partnernet/ssdb.html?kbid=kb2635 for a separate tool to do this.

2.1.10 Script file character encoding

SwyxWare v6.02 and older use the standard Windows ANSI character set to store files. This is very similar to the US-ASCII character set. Internally SwyxServer uses the Unicode character set and therefore converts each script file when loading it. This might cause problems when the client who wrote the script and SwyxServer who reads it use different character code pages. Non-ASCII character might change when converted back to Unicode on the server side. As a result the script signature breaks, because it works on the Unicode representation.

Starting with Hosted SwyxWare v1.0 and SwyxWare v6.10 SwyxServer additionally supports script files encoded in UTF-8, UTF-16. Call Routing Manager and GSE write UTF-8, which is considered as the default format.

SwyxServer uses the **Byte Order Mark**, which is Unicode code point U+FEFF to detect the format. The script loader recognizes file as described in the following table:

Script file starts with bytes	Loaded as	Remarks
EF BB BF	UTF-8	default format in v6.10
FE FF	UTF-16 big endian	supported, but not used by Swyx Clients and therefore not thoroughly tested
FF FE	UTF-17 little endian	supported, but not used by Swyx Clients and therefore not thoroughly tested
(Other)	windows character set as in v6.0x	

All four mentioned formats can be read and written by Windows notepad.exe. Modern software development environments like Visual Studio 2005 are doing so, too.

2.2 Script loading

For incoming calls to a SwyxWare user, SwyxServer loads the user's callrouting.vbs. If that file has "include" statements SwyxServer loads the included files, too. It does this recursively. If an "include" has already been loaded, it will not be loaded again. I.e. two GSE rules can include the same file; it will be loaded only once. SwyxServer checks the signature of each file and does not use it when it is invalid. Every loaded script file is stored in an in-memory cache. That cache has a fixed maximum size (configurable via registry). If that maximum has been reached when a new file is added, the oldest ones are removed from the cache until the size is less than max.

The search order for every script file is the same.

Order	SwyxWare <= v6.0x	SwyxWare v6.10
1	\\<swyxserver>\SwyxWare\user\<username>\phoneclient\scripts	user scope
2	\\<swyxserver>\SwyxWare\data\default\<username>\phoneclient\scripts	user default scope
3	(n/a)	Global scope
4	\\<swyxserver>\SwyxWare\data\phoneclient\scripts	server default scope

3 Best Practices

3.1 When to use GSE actions

GSE actions' main purpose is reuse of call routing functionality. The remote inquiry or the auto attendant actions are good examples. They can either be used from Call Routing Manager inside of a CRM sequence of action or from GSE rule or another GSE action via a **Run GSE Action** block. GSE actions have a certain overhead because of the parameter passing scheme explained above. However, when using GSE actions to handle calls, e.g. for an IVR menu, this overhead is negligible. On the other hand, when a GSE action is used very often and the task inside the action is very short, the overhead may be noticeable.

GSE action parameters have full GUI support, i.e. CRM and GSE show the description and parameters you've defined for an action. They're usable without any VBScript knowledge.

3.2 When to use code in GSE Start Block

Only use the start block to define own VBScript functions or subs. Don't place code directly in this block, because it gets executed directly after script loading before the main() function is called. See section 2.1.9 above. The main() function has some mandatory code to detect special calls like ad-hoc conference establishment. There should be no custom script code called before that.

The script editing control in the start block is not very well suited for editing script code. Better implement own functions/subs in separate script files and include them here.

3.3 When to use code in GSE Insert Script Code Block

Insert Script Code GSE block is well suited to call own functions or to execute small amounts of code. You cannot define functions here, because the code is written verbatim into the GSE rule/action function. Do include files here.

3.4 When to use own script files

Use own script files mainly for function libraries, i.e. for VBScript subs and functions you call from GSE actions or rules. It is not recommended to use the GSE block functions implemented in template_vX.X.vbs from your custom script files. Model call control with GSE blocks.

Include own script files only via the GSE Start Block

Own script files need to be signed, too. You can use a GSE rule, add your code in the start block and refer to that rule script in the include statement. A better alternative is to use a VBScript editor of your choice and sign the file manually. See http://www.swyx.com/partnernet/ssdb.html?kbid=kb2635 for a command-line tool to sign SwyxWare script files.

3.5 Initialization code

Because of the load and include mechanisms described above your own initialization should be done so that you

- Do not initialize something you never use
- Do not initialize more often than necessary

Let's consider a custom object you want to use inside your own GSE rules. If initialization of that objects is costly you should initialize it only once and reuse it while the script is running. Typical example would be a custom object which needs some time to initialize. You should create the object and initialize it only when you're sure that you need it. And you should keep and reuse it while the script is running. To do this the following approach can be used

1. Define the object's variable in the rule start block, e.g.

   ```
   Dim oMyObj
   ```

2. Define an GetMyObject() function in the rule start block, e.g.

   ```
   Function GetMyObject()
     If oMyObj = nothing then
       Set oMyObj = PBXScript.CreateObject("MyApp.MyObject")
       oMyObj.Init
     End if
     Set GetMyObject = oMyObject
   End Function
   ```

3. Call your GetMyObject function whenever you need the object.
4. Use an **Insert Script Code** block at the end of the rule to clean up, i.e. call your object's cleanup method, if any, and set the object variable to nothing. VBScript usually removes all objects automatically when the script is finished, but better be safe than sorry.

This approach works well for one GSE rule or action. But when your solution consists of several rules or actions and you always need to initialize and cleanup some custom objects, consider using Pre- and Postprocessing rules.

3.6 Init/cleanup with Pre and Postprocessing

The Main() function not only calls a GSE rule named PreProcessing before all other rules, but executes a PostProcessing rule at the end, too. You can us that for your own initialization and cleanup code.

1. Use GSE to create a rule named PreProcessing. Use the **start** block of that rule to declare global variables. Use one or more **insert script code** blocks to do the initialization
2. Place the resulting rulePreprocessing.vbs into the global scope (swyxware\data\phoneclient\scripts in v6.0x) when it should be used by every user. Alternatively place the file into the user default scope (swyxware\data\defaults\<username>\phoneclient\scripts in v6.0x) if it is to be used by a specific user only.
3. If you have created the Preprocessing rule with the same user which is later using it, make sure that the Preprocessing rule is not enabled, because it is used automatically and enabling it for a user would execute it twice.

Use the same approach for the Postprocessing rule.

3.7 Start GSE Block and Variable names

Because **Start** GSE block script code is written verbatim into the rule script and all enabled rule scripts of a user are loaded together and form one big scope, you have to make sure that variables you declare in this block are unique. I.e. you cannot have two GSE rules which have a

Dim MyVariable

When both rules are enabled together the resulting script would have two dim statements for the same variable, causing a VBScript error. The user's call routing won't work. Consider using names which are prefixed with the rule name for all variables you define inside a **Start** block.

3.8 Complex functionality

VBScript is not very well suited for complex solutions and lacks support for modern technologies like consuming web services. The best approach is to implement such solutions inside a .NET type, e.g. by using VB.NET or C# and Visual Studio, mark it as COM-visible and register it in the system, e.g. by using regasm.exe, which is part of the .NET Framework. That way it can be instantiated from call routing scripts and you can call methods and use properties. Building COM objects is beyond the scope of this article. Consult the .NET Framework and Visual Studio 2005 documentation for details.

If you have such an object, consider doing the initialization as described in 3.5 above.

It's often not necessary to wrap method calls or property accesses on such objects in GSE actions. **Set Variable** or **Insert Script Code** blocks are sufficient.

3.9 Use UTF-8 encoding

To be able to handle non-ASCII characters SwyxWare v6.10 and Hosted SwyxWare v1.0 support script files encoded in UTF-8 format.

- When you write own script files and include them as described in 2.1.9 above, store your script files in UTF-8 encoding.
- Make sure that your script file editor properly supports writing UTF-8 with the **Byte order Mark** character at the beginning.

3.10 Use trace output

Script files can and should write trace statements into the SwyxServer log file. To generate trace statements from your code use the PPBXScriptOutputTrace method. It takes a string as parameter, e.g.

```
PBXScriptOutputTrace "Function foo, parameters: " & param1 & ", " & param2
```

To see these trace statements in the SwyxServer log file, make sure to set trace module SrvScript to Info3 (decimal 6). You can do this either by using IpPbxTraceFlags.exe from the SwyxWare CD or set the registry REG_DWORD value **SrvScript** at **HKLM\Software\Swyx\IpPbxSrv\CurrentVersion\Tracing** to decimal **6**.

If you have any comment, question or suggestion about this document, please send an email to martin.hueser@swyx.com