A MITEL
PRODUCT
GUIDE

MiContact Center Enterprise

Web Services Integration Script Manager User Guide

Release 9.6
Document Version 1.0

September 2022

Mitel
Powering connections

Notices

The information contained in this document is believed to be accurate in all respects but is not warranted by **Mitel Networks™ Corporation (MITEL®).**

The information is subject to change without notice and should not be construed in any way as a commitment by Mitel or any of its affiliates or subsidiaries. Mitel and its affiliates and subsidiaries assume no responsibility for any errors or omissions in this document. Revisions of this document or new editions of it may be issued to incorporate such changes. No part of this document can be reproduced or transmitted in any form or by any means - electronic or mechanical - for any purpose without written permission from Mitel Networks Corporation.

Trademarks

The trademarks, service marks, logos and graphics (collectively “Trademarks”) appearing on Mitel's Internet sites or in its publications are registered and unregistered trademarks of Mitel Networks Corporation (MNC) or its subsidiaries (collectively "Mitel") or others. Use of the Trademarks is prohibited without the express consent from Mitel. Please contact our legal department at legal@mitel.com for additional information. For a list of the worldwide Mitel Networks Corporation registered trademarks, please refer to the website: http://www.mitel.com/trademarks.

®, ™ Trademark of Mitel Networks Corporation

INTRODUCTION

This document describes how to incorporate web service calls from Script Manager.

A tutorial based on a sample script is also detailed.

SAMPLE SCRIPT FILES

The sample script files used in this document are available on the MiCC Enterprise product DVD. See document *Installing Sample Scripts* for details.

WEB SERVICES IN SCRIPT MANAGER

OVERVIEW

Script Manager can interact with web services by means of the VBScriptExecute block available in the System Object component library.

Although there are many ways to invoke a web service from the VBScriptExecute block, the easiest is the following:

- Create a client .NET class library (DLL) in C# or VB.NET using Visual Studio.
- In this DLL, add the service reference (SOAP or WCF service) or create a client class compatible with the web service technology (REST, XML-RPC, JSON-RPC, XMPP, etc.)
- From this DLL, expose a very simple API to be invoked by the script (just 1 method if possible).
- Implement each method of this API by invoking as many web service methods as necessary, taking into considerations all service-specific features (session management, authentication, security certificate, etc.).
- Create a small test executable (or unit tests suite) to validate the DLL.
- In the script, use a single VBScript block to reference this DLL and invoke its method(s), passing Script Manager variables as input & output arguments.
- On the production server, copy this DLL to the <InstallDir>\ScriptManager\Bin directory.

The “philosophy” is thus to

- do all the complex stuff inside the client DLL and
- use just a single VBScript block to call a method of this DLL and pass SM variables as arguments (→ 2 or 3 lines).

TUTORIAL

The Web Services tutorial demonstrates how to use Script Manager to invoke some **Configuration Web Service** methods.

More specifically, the script will change the skill level of a particular agent. Both the skill and the agent must be identified by their name.

PREREQUISITES

In order to complete this tutorial, you will need:

- The Microsoft .NET Framework 4.5.
- The Microsoft Visual Studio Express 2013 for Windows Desktop or later version.

CREATE A C# DLL USING VISUAL STUDIO

CREATE A C# PROJECT

1. Start Visual Studio.
2. On the menu bar, choose **File**, **New,** and **Project**.
   The **New Project** dialog box opens.
3. Expand **Installed**, expand **Templates**, and click **Visual C#** to display all available Visual C# templates**.**
4. In the center section of the dialog, select **Class Library**.
5. In the **Name** box, specify a name for your project, for instance **ConfigurationWebServiceClient.**
6. Press the **OK** button.
   The new project appears in **Solution Explorer**.
7. In **Solution Explorer**, right-click **ConfigurationWebServiceClient**, click **Add,** and then click **Class**.
   The **Add New Item** dialog box opens.
8. Type **Configuration i**n the **Name** box and then click **Add** to create the class.
   A class named **Configuration** is added to the class library.
9. In **Solution Explorer**, right-click **Class1.cs** and then click **Delete**.
   This deletes the default class that is provided with the class library, because it will not be used in this tutorial.
10. In the **File** menu, click **Save All** to save the project.

CREATE A PROXY CLASS

A proxy is a local object that represents a web service in a form that the client application can use to communicate with the remote service.

1. In **Solution Explorer**, right-click the **References** node underneath the **ConfigurationWebServiceClient** project, and click **Add Service Reference**.
   The **Add Service Reference** dialog box is displayed.

2. In the **Address** box, enter the URI of the **Configuration Web Service**, for instance http://MiCC Enterprise-server:8524/Configuration.
3. Press the **Go** button.
   If the **Configuration Web Service** is accessible at the specified URI, the Services and Operations lists are populated.
4. Enter a name in the **Namespace** box, for instance **ConfigurationServiceReference**.
   The proxy and its associated data structures will be created as new classes in this namespace.
5. Press the **OK** button.
6. After this step

- the **ConfigurationServiceReference** appears under Service References in the project, and
- the **app.config** file is updated to include elements that can be used to instantiate the proxy.

The following is a view of the configuration file (app.config):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <bindings>
            <basicHttpBinding>
                <binding name="BasicHttpBinding_IConfiguration">
                    <security mode="TransportCredentialOnly">
                        <transport clientCredentialType="Basic" />
                    </security>
                </binding>
            </basicHttpBinding>
        </bindings>
        <client>
            <endpoint address="http://MiCC Enterprise-server:8524/Configuration/soap"
                binding="basicHttpBinding"
                bindingConfiguration="BasicHttpBinding_IConfiguration"
                contract="ConfigurationServiceReference.IConfiguration"
                name="BasicHttpBinding_IConfiguration" />
        </client>
    </system.serviceModel>
</configuration >
```

It shows the following endpoint information under the <client> element:

The endpoint that the proxy uses to access the service that is located at the following address: **http://MiCC Enterprise-server:8524/Configuration/soap**.

The endpoint element specifies that the **ConfigurationServiceReference.IConfiguration** service contract is used for communication between the proxy and the service while the configured binding is the system-provided **basicHttpBinding**.

INSTANTIATING THE PROXY

After defining the proxy class, client code can be added to instantiate the proxy.

Instantiating the proxy consists of specifying the endpoint that the proxy uses to access the service. An endpoint has an address, a binding and a contract, and each of these must be specified in the process of instantiating the proxy.

This can be done by defining endpoints in code or using the configuration file (app.config) that was updated or created together with the proxy class.

Instantiating the proxy using the application configuration file

The endpoint configuration name **BasicHttpBinding_IConfiguration** will be used as argument of the proxy constructor so that proxy instance properties such as security mode and transport are correctly set.

In addition, the remote address of the service may also be provided as argument of the proxy constructor.

```
// Create a proxy with given client endpoint configuration.
proxy = new ConfigurationClient("BasicHttpBinding_IConfiguration");
```

Note that the configuration file will not be used by the class library project. You will need to add the settings in the generated configuration file to the .config file for the executable that will call the class library. See section 0.

Instantiating the proxy using configuration information specified in code

Instead of obtaining the binding definition from a .config file, it is also possible to set them in code as in the following excerpt.

```
// Specify the binding to be used for the proxy.
BasicHttpBinding binding = new BasicHttpBinding();
binding.Security.Mode = BasicHttpSecurityMode.TransportCredentialOnly;
binding.Security.Transport.ClientCredentialType = HttpClientCredentialType.Basic;


// Specify the address to be used for the proxy.
EndpointAddress endpointAddress = new
                            EndpointAddress("http://localhost:8524/Configuration/soap");


// Create a proxy using the endpoint address and binding.
proxy = new ConfigurationClient(binding, endpointAddress);
```

It is usually the best practice to specify the binding and address information declaratively in configuration rather than imperatively in code. Keeping the binding and addressing information out of the code allows them to change without having to recompile or redeploy the application.

AUTHENTICATION

To determine what client credential type is required, examine the **<bindings>** section of the configuration file. Within the section are binding elements that specify the security requirements. Specifically, examine the **<security>** element of each binding. That element includes the mode attribute, which you can set to one of three possible values (**Message**, **Transport**, or **TransportWithMessageCredential**). The value of the attribute determines the mode, and the mode determines which of the child elements is significant.

The **<security>** element can contain either a **<transport>** or **<message>** element, or both. The significant element is the one that matches the security mode.

For example, the following code specifies that the security mode is "**TransportCredentialOnly**", and the client credential type for the <**transport**> element is "**Basic**".

```
// Specify the binding to be used for the proxy.
BasicHttpBinding binding = new BasicHttpBinding();
binding.Security.Mode = BasicHttpSecurityMode.TransportCredentialOnly;
binding.Security.Transport.ClientCredentialType = HttpClientCredentialType.Basic;
```

To be allowed to invoke **Configuration Web Service** methods, the client application must provide valid MiCC Enterprise credentials.

To ensure that each method invocation is authenticated by the **Configuration Web Service**, the **ClientCredentials** property of the proxy instance must be set.

```
// Specify the client credential value on the proxy in code
proxy.ClientCredentials.UserName.UserName = userName;
proxy.ClientCredentials.UserName.Password = password;
```

CALLING OPERATIONS

Once you have a proxy object created and configured, call operations in the same way that you would if the object were local. Close the proxy once the operation invocations are completed.

```
// Making calls.
User user = proxy.GetUserByLogonId(tenantId, logonId);
Skill skill = proxy.GetSkillByName(tenantId, skillName);
SkillLevel skillLevel = new SkillLevel();
skillLevel.SkillId = skill.Id;
skillLevel.Level = level;
proxy.AddSkillLevelToUser(tenantId, user.Id, skillLevel);

//Closing the client gracefully closes the connection and cleans up resources.
proxy.Close();
```

The **using** directive (**Imports** in Visual Basic) at the beginning of the file enables you to use the unqualified class names to reference the Configuration Web Service methods.

```
using ConfigurationWebServiceClient.ConfigurationServiceReference;
```

EXCEPTION HANDLING

Exceptions can occur in a client application when opening or closing the proxy, or when using the proxy to call operations. It is recommended at a minimum that applications expect to handle possible **System.TimeoutException** and **System.ServiceModel.CommunicationException** exceptions in addition to any **System.ServiceModel.FaultException** objects that may be triggered by operations.

The recommended way to handle these exceptions is to surround the code that invokes proxy methods with a try block and append a catch block for each potential exception.

```
try
{
    // Client code using proxy


}
catch (TimeoutException timeProblem)
{
    Console.WriteLine("The service operation timed out. " + timeProblem.Message);
    proxy.Abort();
    return 1;
}
catch (FaultException unknownFault)
{
    Console.WriteLine("An unknown exception was received. " + unknownFault.Message);
    proxy.Abort();
    return 2;
}
catch (CommunicationException commProblem)
{
    Console.WriteLine("There was a communication problem. " +
                                          commProblem.Message + commProblem.StackTrace);
    proxy.Abort();
    return 3;
}
catch (Exception ex)
{
    Console.WriteLine("An unknown exception was received. " + ex.Message);
    proxy.Abort();
    return 4;
}
```

COMPILING THE CODE

```
using System;
using System.ServiceModel;
using ConfigurationWebServiceClient.ConfigurationServiceReference;

namespace ConfigurationWebServiceClient
{
    public class Configuration
    {
        ConfigurationClient proxy;

        public Configuration(string userName, string password)
        {
            // Specify the binding to be used for the proxy.
            BasicHttpBinding binding = new BasicHttpBinding();
            binding.Security.Mode = BasicHttpSecurityMode.TransportCredentialOnly;
            binding.Security.Transport.ClientCredentialType =
                                                    HttpClientCredentialType.Basic;

            // Specify the address to be used for the proxy.
            EndpointAddress endpointAddress = new
                            EndpointAddress("http://localhost:8524/Configuration/soap");

            // Create a proxy using the endpoint address and binding.
            proxy = new ConfigurationClient(binding, endpointAddress);

            // Create a proxy with given client endpoint configuration.
            //proxy = new ConfigurationClient("BasicHttpBinding_IConfiguration");

            // Specify the client credential value on the proxy in code
            proxy.ClientCredentials.UserName.UserName = userName;
            proxy.ClientCredentials.UserName.Password = password;
        }

        public int AddSkillLevelToUser(int tenantId, string logonId, string skillName,
                                                                        int level)
        {
            try
            {
                // Making calls.
                User user = proxy.GetUserByLogonId(tenantId, logonId);
                Skill skill = proxy.GetSkillByName(tenantId, skillName);
                SkillLevel skillLevel = new SkillLevel();
                skillLevel.SkillId = skill.Id;
                skillLevel.Level = level;
                proxy.AddSkillLevelToUser(tenantId, user.Id, skillLevel);
```

```
                //Closing the client gracefully closes the connection and cleans up
                  resources.
                proxy.Close();


                return 0;
            }
            catch (TimeoutException timeProblem)
            {
                Console.WriteLine("The service operation timed out. " +
                                                                    timeProblem.Message);
                proxy.Abort();
                return 1;
            }
            catch (FaultException unknownFault)
            {
                Console.WriteLine("An unknown exception was received. " +
                                                                  unknownFault.Message);
                proxy.Abort();
                return 2;
            }
            catch (CommunicationException commProblem)
            {
                Console.WriteLine("There was a communication problem. " +
                                          commProblem.Message + commProblem.StackTrace);
                proxy.Abort();
                return 3;
            }
            catch (Exception ex)
            {
                Console.WriteLine("An unknown exception was received. " + ex.Message);
                proxy.Abort();
                return 4;
            }
        }
    }
}
```

On the Build menu click Build Solution (or press CTRL+SHIFT+B).

TEST THE DLL

To test the **ConfigurationWebServiceClient** class library, you must have a project that uses it.

Use a VB.Net console application, so that you will be able to reuse your code in the Script Manager VBScriptExecute block.

1. Start Visual Studio.
2. On the menu bar, choose **File**, **New**, and **Project**.
   The **New Project** dialog box opens.
3. Expand **Installed**, expand **Templates**, and click **Visual Basic** to display all available

Visual Basic templates**.**

4. In the center section of the dialog, select **Console Application**.
5. In the **Name** box, specify a name for your project, for instance **TestConfigurationWebServiceClient**.
6. Press the **OK** button**.**
   The new project appears in **Solution Explorer**.
7. In **Solution Explorer**, right-click the **References** node underneath the **TestConfigurationWebServiceClient** project, and click **Add Reference**.
   The **Add Reference** dialog box opens.
8. Select the **Browse** tab, locate the folder where the **ConfigurationWebServiceClient** library resides and select it.
9. Press the **OK** button.
10. After this step the **ConfigurationWebServiceClient** appears under References in the project. You can then use the classes and methods of the library like you would use those of the .NET Framework.
11. If **Module1.vb** isn't open in the **Code Editor**, open the shortcut menu for **Module1.vb** in **Solution Explorer**, and then choose **View Code**.
12. Call the library class methods in your application.
    Replace the contents of **Module1.vb** with the following code :

```
Imports ConfigurationWebServiceClient

Module Module1

    Sub Main()

        Dim adminLogonId As String = "Admin"
        Dim password As String = "admin"
        Dim tenantId As Long = -1
        Dim userLogonId As String = "user1"
        Dim skillName As String = "skill1"
        Dim skillLevel As Long = 5
        Dim result As Long
        Dim conf As Configuration

        'Initialize the class from the class library
        conf = New Configuration(adminLogonId, password)

        'Add SkillLevelToUser
        result = conf.AddSkillLevelToUser(tenantId, userLogonId, skillName, skillLevel)

        Console.WriteLine("The result is {0:d}", result)

        Console.WriteLine("Press [Enter] to close.")
        Console.ReadLine()

    End Sub
```

```
End Module
```

13. Choose the F5 key to run the application.

WEBSERVICE.MFD

The script presented in Figure 1 demonstrates the inclusion of web service calls during a voice call.

This tutorial emphasizes the use of the **VBScriptExecute** block.

Double-click on WebService.mfd to open the script below.

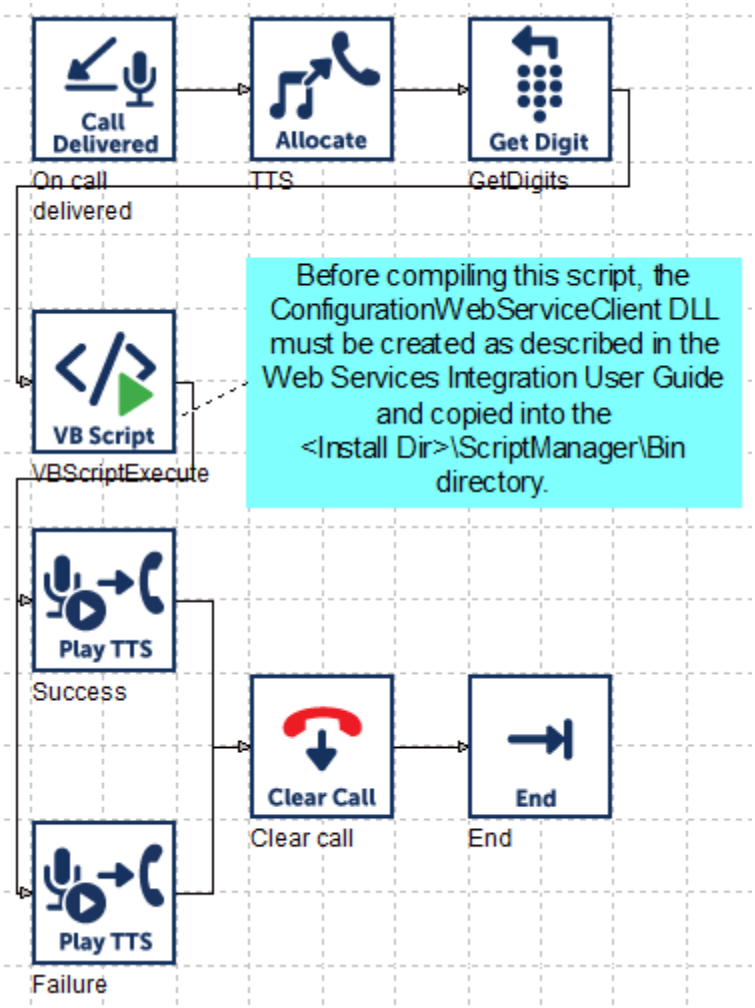

Figure 1: Web Services sample script

This script handles invocation of web services methods. Its successive steps are the following:

- Receive an incoming call.
- Allocate resources and answer call.
- Get digits entered by the caller.

- Call the web service operations.
- Play “success” in case of success, or
  “failure” in case of error.

PREREQUISITES

Before running the script, the following actions must be taken on the production server:

- Copy the **ConfigurationWebServiceClient** DLL to <InstallDir>\ScriptManager\Bin directory.
- If in the **ConfigurationWebServiceClient** project, the proxy has been instantiated using the configuration file, add the settings in the app.config file to the **<InstallDir>\ScriptManager\Bin\flowprocessor.exe.config** file.
  You may need to create this file if it doesn’t already exist.

SESSION VARIABLES

The following session variables are defined.

AdminLogonId

- Type: String.
- Dimension: Zero.
- Initial Value: “Admin” (without quotes).

Password

- Type: String.
- Dimension: Zero.
- Initial Value: “admin” (without quotes).

TenantId

- Type: Long.
- Dimension: Zero.
- Initial Value: -1.

UserLogonId

- Type: String.
- Dimension: Zero.
- Initial Value: “user1” (without quotes).

SkillName

- Type: String.
- Dimension: Zero.
- Initial Value: “skill1” (without quotes).

Digits

- Type: String.

- Dimension: Zero.
- Initial Value:

ONCALLDELIVERED

Figure 2 shows how the Settings tab of the OnCallDelivered block is configured.

Specify the name of an existing BVD in the **Monitored Device List**.

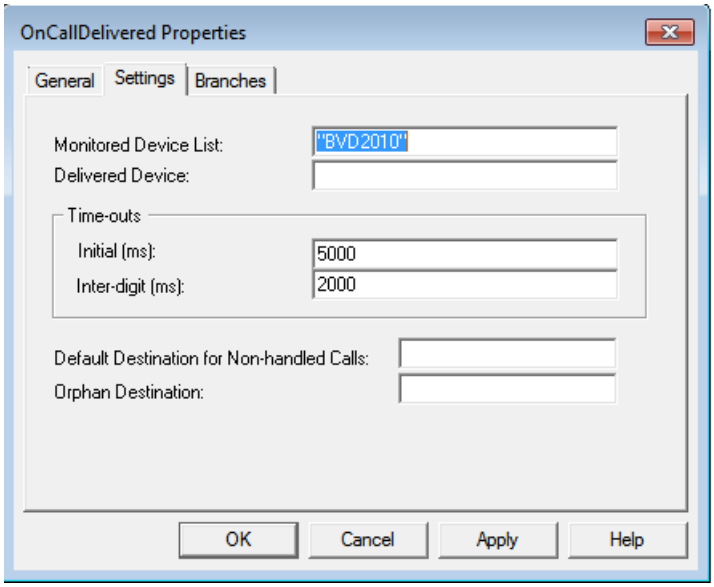

Figure 2: OnCallDelivered Settings

ALLOCATERESOURCE

Figure 3 shows how the Settings tab of the AllocateResource block is configured.

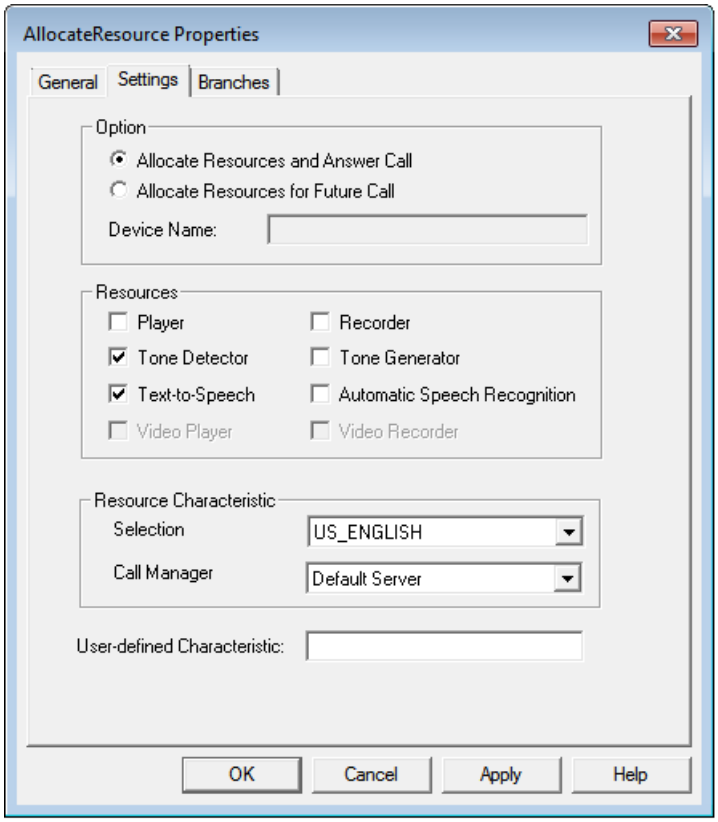

Figure 3: AllocateResource Settings

GETDIGITS

Figure 4 shows how the Settings tab of the GetDigits block is configured.

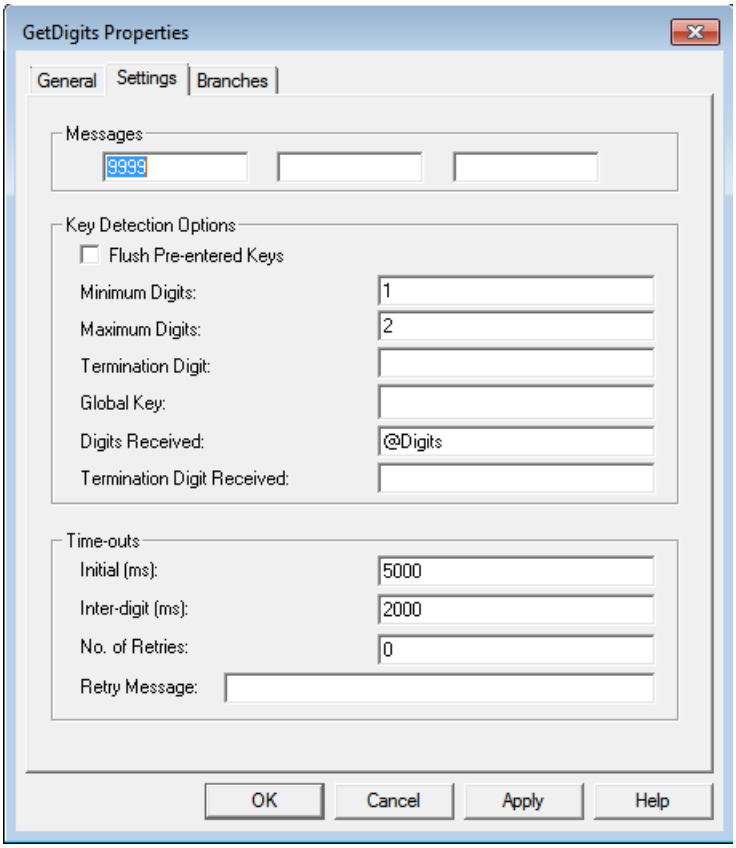

Figure 4: GetDigits Settings

VBSCRIPTEXECUTE

Settings

Figure 5 shows how the Settings tab of the VBScriptExecute block is configured.

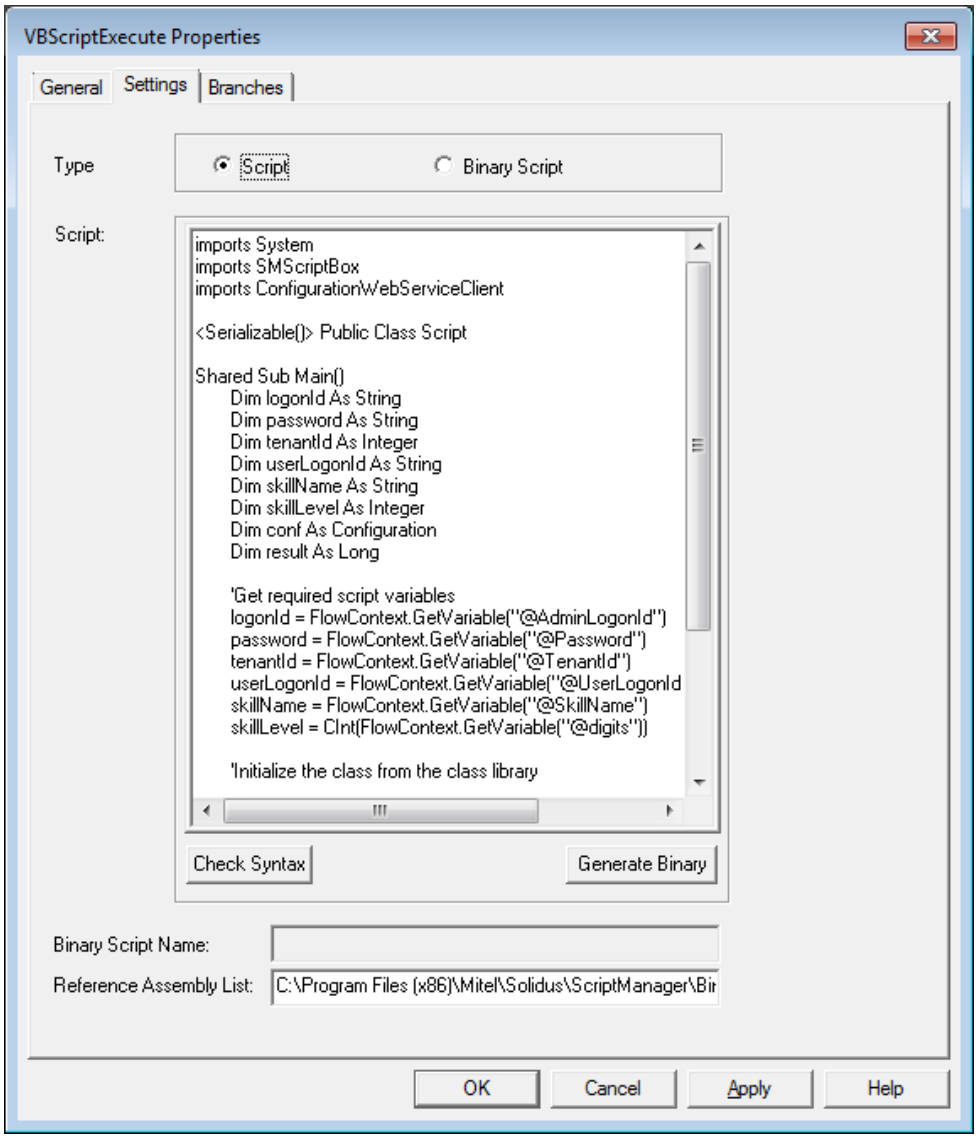

Figure 5: VBScriptExecute Settings

Here is the code inside the Script text field.

```
Imports System
Imports SMScriptBox
Imports ConfigurationWebServiceClient

<Serializable()> Public Class Script
    Shared Sub Main()
        Dim logonId As String
        Dim password As String
        Dim tenantId As Integer
        Dim userLogonId As String
        Dim skillName As String
        Dim skillLevel As Integer
        Dim conf As Configuration
        Dim result As Long

        ' Get required script variables
        logonId = FlowContext.GetVariable("@AdminLogonId")
        password = FlowContext.GetVariable("@Password")
        tenantId = FlowContext.GetVariable("@TenantId")
        userLogonId = FlowContext.GetVariable("@UserLogonId")
        skillName = FlowContext.GetVariable("@SkillName")
        skillLevel = CInt(FlowContext.GetVariable("@Digits"))

        '  Initialize the class from the class library
        conf = New Configuration(logonId, password)

        ' Invoke web service method
        result = conf.AddSkillLevelToUser(tenantId, userLogonId, skillName, skillLevel)

        ' Set the result
        FlowContext.SetResult(result)
    End Sub

End Class
```

To invoke a web service from the VBScriptExecute block, the client DLL need to be referenced as in this example above.

Branches

Figure 6 shows how the Branches tab of the VBScriptExecute block is configured.

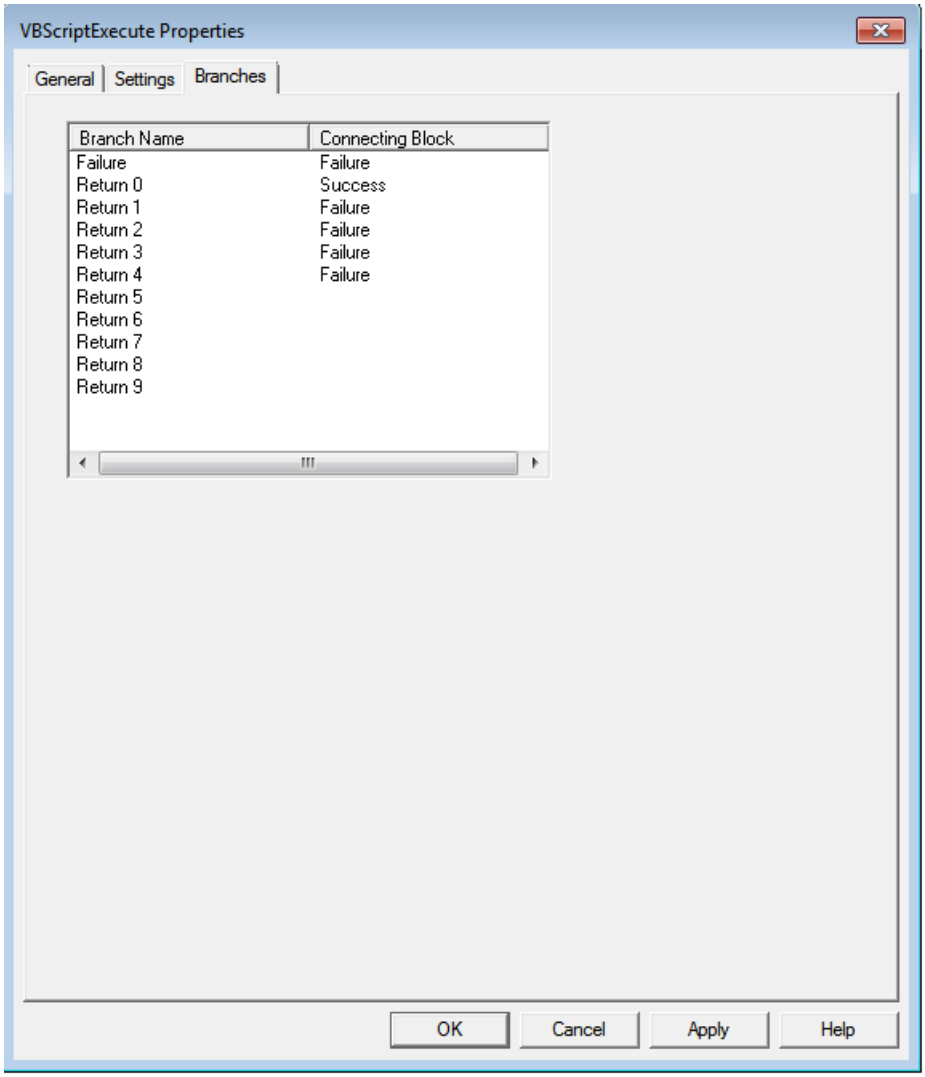

Figure 6: VBScriptExecute Branches

VOICE PROMPT

One voice prompt is used in this script to play a message to the caller.
Make sure the voice prompt is available and is configured in OAS as a play message.

Table 1: Voice Prompt

SCRIPT MANAGER VARIABLE	MESSAGE ID	SAMPLE VOICE PROMPT DESCRIPTION
Msg	9999	“Please enter the skill level”

USAGE

To test this sample script, perform the following steps.

- Call the BVD number associated to the Service Access.
- Specify the desired skill level.
- Depending on the result of web service calls, the system then plays either “Success” , or “Failure”.
- Using MiCC Enterprise Configuration Manager, verify that the skill level of the specified user has been changed.

Mitel
Powering connections

mitel.com