**Dialogic® PowerMedia™ Extended Media Server (PowerMedia XMS)**

**XMS 2.2 RESTful demo for C#**

**Introduction**

This article contains a simple demonstration of how to control a Dialogic® PowerMedia™ XMS 2.2 server using the RESTful API and a simple C# library including how to automatically generate a simple API from the XML schema definition that is provided with PowerMedia XMS and how to send and process HTTP and web service requests and responses. It also contains a sample library and working sample application that implements a call flow using the RESTful API. The simple C# library "MCXMSLib" which contains some functions and objects that can be used to communicate with PowerMedia XMS 2.2 using the xmsrest.xsd (XML Schema Definition). Additionally, this article describes a simple console based application that uses the library and implements a sample call flow which will answer inbound calls, play a video and then hangup. This sample code can be used as a base to add additional functionality.

**Details**

As per the RESTful API user interface guide (<http://www.dialogic.com/webhelp/XMS/2.2/XMS_RESTfulAPIUser.pd>f) page 12:

*"PowerMedia XMS uses an XML schema definition (also referred to herein as "XSD"). The XSD formally describes the structure, content, and semantics of the XML payload for the PowerMedia XMS RESTful API call and media commands.*

*An XSD may be used to generate client-side code, allowing contents of XML documents to be treated as objects. The generated code usually enforces type-checking, thus supporting client-side validation of the XML payload before it is sent to the PowerMedia XMS."*

### ****Details****

The project, "MCSimpleXMSTest" is a Microsoft Visual Studio 2010 based solution containing two projects, "MCXMSLib" (the XMS library) and "MCSimpleXMSTest" (the sample application). Both of these projects have a reference to the external DLL "Syzygy.Common.dll" which is contained in the "External" directory and contains various useful C# classes including a logging object that is used in these projects.

Microsoft Visual Studio contains the "xsd.exe" tool for exactly this purpose. This can be run from the "Visual Studio Command Prompt". As of the PowerMedia XMS 2.2 release, the XSD is contained in the "xmsrest.xsd" file which is located on the PowerMedia XMS server in the /etc/xms directory. Note that the XSD will change periodically as features are added so it is worth regenerating the C# API when there is a new release or service update.

The "xsd.exe" tool is run as follows:

>xsd xmsrest.xsd /classes  
Microsoft (R) Xml Schemas/DataTypes support utility  
[Microsoft (R) .NET Framework, Version 4.0.30319.1]  
Copyright (C) Microsoft Corporation. All rights reserved.  
Writing file 'xmsrest.cs'.

This will generate the file "xmsrest.cs" which contains the C# code.

From here, three additional functions are required to [1] convert from the generated API to XML, [2] convert from XML to the generated API and [3] to send the request via HTTP.

[1] Convert from the generated API to XML

      public static String RESTapiToXML(object api, Type type)  
      {  
         XmlSerializer serializer     = new XmlSerializer(type);  
         StringBuilder requestContent = new StringBuilder();  
  
         try  
         {  
            using (StringWriter writer = new StringWriter(requestContent))  
            {  
               XmlTextWriter xmlwriter = new XmlTextWriter(writer);  
               xmlwriter.Formatting = Formatting.Indented;  
               serializer.Serialize(xmlwriter, api);  
            }  
         }  
         catch (Exception ex)  
         {  
            return String.Empty;  
         }  
   
         return requestContent.ToString();  
      }

[2] Convert from XML to the generated API

      public static object XMLToRESTapi(String xml, Type type)  
     {  
         XmlSerializer serializer = new XmlSerializer(type);  
  
         using (StringReader reader = new StringReader(xml))  
         {  
            return serializer.Deserialize(reader);  
         }  
      }

[3] Send a request via HTTP

      public static bool SendHttpRequest(out String responseString, String uri, String method, MCantale.XMS.RESTapi.web\_service request)  
      {  
         String requestString = RestHelpers.RESTapiToXML(request, typeof(MCantale.XMS.RESTapi.web\_service));  
         return SendHttpRequest(out responseString, uri, method, requestString, "application/xml");  
      }

...

      public static bool SendHttpRequest(out String responseString, String uri, String method = "GET", String requestContent = "", String contentType = "application/xml")  
      {  
         responseString = String.Empty;  
         HttpWebRequest request  = (HttpWebRequest)WebRequest.Create(uri);  
         request.Accept          = null;  
         request.ContentType     = null;  
         request.Expect          = null;  
         request.KeepAlive       = true;  
         request.Method          = method;  
         request.ProtocolVersion = HttpVersion.Version11;  
  
         try  
         {  
            if (!String.IsNullOrWhiteSpace(requestContent))  
            {  
               request.ContentType   = contentType;  
               request.ContentLength = requestContent.Length;  
  
               using (StreamWriter stream = new StreamWriter(request.GetRequestStream()))  
               {  
                  stream.Write(requestContent);  
                  stream.Close();  
               }  
            }  
  
            LoggingSingleton.Instance.Message(LogType.Library, LogLevel.Debug2,  
                                              "XMLHelper::SendHttpRequest : Sent {0} {1}", method, uri);  
  
            if (!String.IsNullOrWhiteSpace(requestContent))  
            {  
               LoggingSingleton.Instance.Message(LogType.Library, LogLevel.Debug2, "XMLHelper::SendHttpRequest : Sent web service request :\n{0}", requestContent);  
            }  
  
            using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())  
            {  
               using (StreamReader stream = new StreamReader(response.GetResponseStream()))  
               {  
                  String result = stream.ReadToEnd();  
                  responseString = result;  
  
                  /// Parse the response...  
                  ///  
                  LoggingSingleton.Instance.Message(LogType.Library, LogLevel.Debug2,   
                                                    "XMLHelper::SendHttpRequest : Received {0:D} {1}",  
                                                    response.StatusCode, response.StatusDescription);  
               }  
            }  
         }  
         catch (WebException ex)  
         {  
            if (ex.Status == WebExceptionStatus.ProtocolError)  
            {  
               HttpWebResponse response = ex.Response as HttpWebResponse;  
  
               if (response != null)  
               {  
                  LoggingSingleton.Instance.Message(LogType.Library, LogLevel.Warning,  
                                                    "XMLHelper::SendHttpRequest : Received HTTP failure response {0} {1}",  
                                                    (int)response.StatusCode, response.StatusDescription);  
               }  
            }  
            else  
            {  
               LoggingSingleton.Instance.Message(LogType.Library, LogLevel.Warning,  
                                                 "XMLHelper::SendHttpRequest : {0}\n{1}",  
                                                 ex.Message.ToString(), ex.StackTrace.ToString());  
            }  
  
            return false;  
         }  
         catch (Exception ex)  
         {  
            LoggingSingleton.Instance.Message(LogType.Library, LogLevel.Except,  
                                              "XMLHelper::SendHttpRequest : {0}\n{1}",  
                                              ex.Message.ToString(), ex.StackTrace.ToString());  
  
            return false;  
         }  
  
         return true;  
      }

Putting the above together, a web service request would be sent with a command to a PowerMedia XMS server to, for example, answer an incoming call:

      public bool AnswerCall(string CallURI)  
      {  
         RESTapi.web\_service ws = new RESTapi.web\_service()  
         {  
            Item = new RESTapi.call()  
            {  
               answer                    = RESTapi.boolean\_type.yes,  
               answerSpecified           = true,  
               async\_completion          = RESTapi.boolean\_type.yes,  
               async\_completionSpecified = true,  
               media                     = RESTapi.media\_type.audiovideo,  
               mediaSpecified            = true,   
            }  
         };  
   
         String responseString = String.Empty;  
   
         if (RestHelpers.SendHttpRequest(out responseString, CallURI, "PUT", ws))  
         {  
            LoggingSingleton.Instance.Message(LogType.Library, LogLevel.Debug1, "Call::AnswerCall : AnswerCall OK");  
            LoggingSingleton.Instance.Message(LogType.Library, LogLevel.Debug1, responseString);  
         }  
         else  
         {  
            LoggingSingleton.Instance.Message(LogType.Library, LogLevel.Error, "Call::AnswerCall : AnswerCall failed!");  
            Hangup(ResourceID);  
            return false;  
         }   
         return true;   
      }

#### MCXMSLib

This PowerMedia XMS library contains four main components:

[1] xmsrest.xsd / xmsrest.cs

This is the XML Schema Definition (XSD) and the C# classes generated from the XSD respectively. The generated C# definition file was modified to make sure everything is in the "MCantale.XMS.RESTapi" namespace but aside from that is left as generated by the "xsd.exe" tool.

[2] EventDispatcher.cs

The event dispatcher. This will set up an event handler with the PowerMedia XMS at startup (optionally removing any existing event handlers that may be left behind / orphaned) and process any inbound HTTP responses or requests from PowerMedia XMS.

[3] CallDispatcher.cs

The call dispatcher is called by the event dispatcher to determine which specific call object should receive the incoming message. It maintains a list of all calls that are in progress and is responsible for generating new calls (either in response to a make call request or incoming call request) and for hanging up and deleting them. Again, optionally it can detect and delete any calls that were in progress when the application is started.

[4] CallBase.cs

This is a class which represents a single call. It contains various functions to accept and answer calls and to do things such as play a file. It is an abstract base class and as such, the expectation is that the user of this library would derive the user's own class from this one to implement a specific call flow. By default, CallBase will do nothing apart from to delete the call resource in response to an incoming hangup request.

Finally, "RestSettings.cs" is a static class which contains the PowerMedia XMS server IP address, port and application ID that are used elsewhere in the library. These must be set by the application before the event dispatcher is invoked.

#### MCSimpleXMSTest

The sample application shows how to invoke the EventDispatcher class to start a simple call flow. By default, it will wait for inbound calls (depending on how many channels are licensed on the XMS server) and will answer them in audio+video mode and play a short video. The call will then hangup. The use is free to hangup at any time.

The program itself is contained within the "Program.cs" file, in which there is a class "MyCall" that implements a call flow which derives from the "CallBase" class.

* When incoming call event is detected, the "AcceotCall()" function will be called.
* When the call is ringing "AnswerCall()" which be called.
* Upon the call being connected, "PlayFile()" is called to play a video/audio file.
* Once the play has been completed, "Hangup()" is called to disconnect the call.

Note that the "OnEndPlay()" event handler is still called even if the play was terminated by the call hanging up.  There is a check for the termination reason to make sure that the application doesn't hangup twice.

   class MyCall : CallBase  
   {  
      protected override void OnConnected()  
      {  
         PlayFile("file://verification/video\_clip\_nascar.wav",  
                  "file://verification/video\_clip\_nascar.vid");  
      }  
  
      protected override void OnEndPlay()  
      {  
         foreach (KeyValuePair<string, string> item in this.\_LastEventData)  
         {  
            if (item.Key.ToLower() == "reason" && item.Value.ToLower() == "hangup")  
            {  
               return;  
            }  
         }  
  
         Hangup();  
      }  
  
      protected override void OnIncoming()  
      {  
         AcceptCall();  
      }  
  
      protected override void OnRinging()  
      {  
         if (this.Direction == "inbound")  
         {  
            AnswerCall();  
         }  
      }  
   }

The main application first tells the logging object "LoggingSingleton.Instance" to log to both to a file and to the console.

      LoggingSingleton.Instance.LogToFile   = true;  
      LoggingSingleton.Instance.LogToStdout = true;

It then sets the PowerMedia XMS IP address, port and application ID.   The user will need to modify these values with information that is appropriate for the environment where the sample application will be executed.

      RestSettings.Instance.ServerIP    = "192.168.186.104";  
      RestSettings.Instance.ServerPort = 81;  
      RestSettings.Instance.AppID      = "app";

It then disconnects any existing event handlers. Note that this will actually remove any existing event handlers which use the application ID specified.

      EventDispatcher<MyCall>.DisconnectAllHandlers;

From there, a new call handler is created specifying that it should create call objects with the type "MyCall" and these call objects will then be instantiated:

      EventDispatcher<MyCall> ev = new EventDispatcher<MyCall>() { DeleteAllCallsOnConnect = true; };  
      ev.Start();

Now the application will spin around in a loop waiting for a key press to exit:

      while (!needExit)  
      {  
         ConsoleKeyInfo info = Console.ReadKey(true);  
  
         switch (info.KeyChar)  
         {  
            case 'x':  
            case 'X':  
               needExit = true;  
               break;  
         }  
  
         Thread.Sleep(10);  
      }

Finally the event dispatcher is stopped which will asynchronously delete the event handler and dispose of it:

      ev.Stop();  
      ev.WaitForStop();  
  
      ev.Dispose();  
      ev = null;

The program will create output to the console and also to the file "MCSimpleXMSTest-<DATETIME>.log" where <DATETIME> is a formatted date/time string:

Wed Oct 15 10:43:31.413 2014 | Library     | Debug2  | EventDispatcher::GetEventHandlers : Sent GET http://192.168.186.104:81/default/eventhandlers?appid=app  
Wed Oct 15 10:43:31.441 2014 | Library     | Debug2  | EventDispatcher::GetEventHandlers : Received 200 OK  
Wed Oct 15 10:43:31.442 2014 | Library     | Debug3  | EventDispatcher::GetEventHandlers : <web\_service version="1.0">  
Wed Oct 15 10:43:31.442 2014 | Library     | Debug3  | <eventhandlers\_response size="0">  
Wed Oct 15 10:43:31.442 2014 | Library     | Debug3  | </eventhandlers\_response>  
Wed Oct 15 10:43:31.442 2014 | Library     | Debug3  | </web\_service>  
Wed Oct 15 10:43:31.823 2014 | Library     | Info    | EventDispatcher::GetEventHandlers : Found 0 existing event handlers  
Wed Oct 15 10:43:31.832 2014 | Library     | Debug2  | EventDispatcher::InternalRunThread : Entering InternalRunThread()  
Wed Oct 15 10:43:31.832 2014 | Library     | Info    | EventDispatcher::InternalRunThread : Connecting to event handler...  
Wed Oct 15 10:43:32.199 2014 | Library     | Debug2  | EventDispatcher::ConnectToEventHandler : Sent POST http://192.168.186.104:81/default/eventhandlers?appid=app  
Wed Oct 15 10:43:32.230 2014 | Library     | Debug2  | EventDispatcher::ConnectToEventHandler : Received 201 Created  
Wed Oct 15 10:43:32.246 2014 | Library     | Debug2  | EventDispatcher::ConnectToEventHandler : EventHandler URI is "http://192.168.186.104:81/default/eventhandlers/9d35496f-d3e7-4134-801b-2e4c02fda88b?appid=app", OID is "9d35496f-d3e7-4134-801b-2e4c02fda88b"  
Wed Oct 15 10:43:32.247 2014 | Library     | Debug2  | EventDispatcher::ConnectToEventHandler : EventHandler connected  
Wed Oct 15 10:43:32.255 2014 | Library     | Debug2  | XMLHelper::SendHttpRequest : Sent GET http://192.168.186.104:81/default/calls?appid=app  
Wed Oct 15 10:43:32.257 2014 | Library     | Debug2  | XMLHelper::SendHttpRequest : Received 200 OK  
Wed Oct 15 10:43:32.258 2014 | Library     | Debug1  | CallDispatcher::GetCalls : Received web service request :  
Wed Oct 15 10:43:32.258 2014 | Library     | Debug1  | <web\_service version="1.0">  
Wed Oct 15 10:43:32.258 2014 | Library     | Debug1  | <calls\_response size="0">  
Wed Oct 15 10:43:32.258 2014 | Library     | Debug1  | </calls\_response>  
Wed Oct 15 10:43:32.258 2014 | Library     | Debug1  | </web\_service>  
Wed Oct 15 10:43:32.265 2014 | Library     | Info    | CallDispatcher::GetCalls : Found 0 RESTapi calls  
Wed Oct 15 10:43:32.270 2014 | Library     | Debug2  | EventDispatcher::InternalProcessEventsThread : Entering InternalProcessEventsThread()  
Wed Oct 15 10:43:32.271 2014 | Library     | Debug2  | EventDispatcher::InternalProcessEventsThread : Sent GET http://192.168.186.104:81/default/eventhandlers/9d35496f-d3e7-4134-801b-2e4c02fda88b?appid=app  
Wed Oct 15 10:43:32.273 2014 | Library     | Debug2  | EventDispatcher::InternalProcessEventsThread : Received 200 OK  
Wed Oct 15 10:43:32.273 2014 | Library     | Info    | EventDispatcher::InternalProcessEventsThread : Waiting for events...  
Wed Oct 15 10:43:33.648 2014 | Library     | Debug2  | EventDispatcher::InternalStop : Interrupting InternalRunThread...  
Wed Oct 15 10:43:33.648 2014 | Library     | Debug2  | EventDispatcher::InternalStop : Attempting to join InternalRunThread...  
Wed Oct 15 10:43:33.649 2014 | Library     | Debug2  | EventDispatcher::InternalRunThread : Caught ThreadInterruptedException in InternalRunThread()!  
Wed Oct 15 10:43:33.651 2014 | Library     | Info    | EventDispatcher::InternalRunThread : Disconnecting from event handler...  
Wed Oct 15 10:43:33.654 2014 | Library     | Debug2  | EventDispatcher::DisconnectFromEventHandler : Sent DELETE http://192.168.186.104:81/default/eventhandlers/9d35496f-d3e7-4134-801b-2e4c02fda88b?appid=app  
Wed Oct 15 10:43:33.656 2014 | Library     | Debug2  | EventDispatcher::InternalProcessEventsThread : Leaving InternalProcessEventsThread()  
Wed Oct 15 10:43:33.657 2014 | Library     | Debug2  | EventDispatcher::DisconnectFromEventHandler : Received 204 No Content  
Wed Oct 15 10:43:33.657 2014 | Library     | Debug2  | EventDispatcher::DisconnectFromEventHandler : EventHandler disconnected  
Wed Oct 15 10:43:33.658 2014 | Library     | Debug2  | EventDispatcher::InternalRunThread : Leaving InternalRunThread()