```
1: ////////////////////////////////
2:  maXbox Starter 52 Second Part
3: ////////////////////////////////
4:
5:
6: Work with WMI System Managment V2
7: ---------------------------------
8: Max Kleiner
9:
10:
11: In the first available part
12: http://www.softwareschule.ch/download/maxbox_starter52.pdf
13: we got an impression about the force concerning WMI. In this second part we go
    deeper with data analysis and filtering.
14:
15: Suppose you need to retrieve records from the Windows event logs for maybe
    security reason. With the following script snippet you can easily read event logs
    but before you run the listing, we should point out that the example script can
    take a long time to run if your event logs contain thousands of records. So we
    have to filter it with a clever query:
16:
17:   isQuery:= Select * from Win32_NTLogEvent Where Logfile ="System" AND EventType
    ="2" AND  User = "NT AUTHORITY\\NETWORK SERVICE";
18:
19: The filter is based on Logfile, EventType and User.
20: The eventtype is divided into:
21:
22:          1 Error
23:
24:          2 Warning()
25:
26:          3 Information()
27:
28:          4 Security audit success
29:
30:          5 Security audit failure
31:
32: For the logfile you can also choose. Events for the System and Application logs
    fall into one of three categories: error, warning, and information. Error events
    are the most serious and cause a red stop sign to appear on the left side of the
    screen. Warning events identify a possible problem, but not as crucial a problem
    as in an error event. Information events are basic notifications, such as
    services starting and stopping, browser elections, and print jobs.
33:
34: The Security log uses two event types: success and failure. These events signal
    whether a user was able to log on or access a resource. You want the system to
    prevent unauthorized users from logging on, so a success event for an
    unauthorized user is a problem.
35: Back to code. To get the query running we need a service and a locator:
36:
37:     var isloc: ISWBemLocator;
38:         isser: ISWBemServices;
39:         iset: ISWbemObjectSet;
40:         ENum: IEnumVariant;
41:         tObj: OleVariant;
42:         isQuery: string;
43:
44:   isloc:= WMIStart;
45:   isser:= WMIConnect(isloc, 'localhost','max',''); //: ISWBemServices;
46:
```

After passing the locator to a connector, we get a service to pass it again to our query to get an object set:

```
  isQuery:= 'Select * from Win32_NTLogEvent Where Logfile = "System" and
EventType= "2" AND User = "NT AUTHORITY\\NETWORK SERVICE"';

  iset:= WMIExecQuery(isser, isQuery); //WbemObjectSet;
```

The query with the user notification has to be escaped with \\ otherwise you get an error! This is because the backslash character is an escape character and if you need to put it in a select statement then you need to put it twice i.e.

```
               "BUILTIN\\Administrators"';
```

So the same query work also if the Operating System is Windows Server 2003 Itainium Edition or other win server systems. Next we call the query in a loop to get the object set as a result set:

```
      it:= 0;
      iset:= WMIExecQuery(isser, isQuery); //WbemObjectSet;
      writeln(botoStr(WMIRowFindFirst(iset, ENum, tObj)));
      repeat
        inc(it)
        PrintF('-systime : %s - Name %s User %s'+#13#10+'%s'+#13#10,
               [tobj.TimeGenerated,tobj.SourceName,tobj.User,tobj.message])
      until not WMIRowFindNext(ENum, tobj); //}
      writeln('System Warnings : '+itoa(it))
      tObj:= unassigned;
```

As we set the select query to the wildcard * we get in a tempobj <tobj> as a variant the fields we are intersted in:

```
"Log File:       " & wbemObject.LogFile       & vbCrLf & _
                 "Record Number:    & wbemObject.RecordNumber
                 "Type:             & wbemObject.Type
                 "Time Generated:   & wbemObject.TimeGenerated
                 "Source:           & wbemObject.SourceName
                 "Category:         & wbemObject.Category
                 "Category String:  & wbemObject.CategoryString
                 "Event:            & wbemObject.EventCode
                 "User:             & wbemObject.User
                 "Computer:         & wbemObject.ComputerName
                 "Message:          & wbemObject.Message
```

In our case are 4 fields of interest: TimeGenerated, SourceName, User and Message. And the result looks like (exctract):

```
-systime : 20160914194051.572325-000 - Name Microsoft-Windows-DNS-Client User NT
AUTHORITY\NETWORK SERVICE
Name resolution for the name v10.vortex-win.data.microsoft.com timed out after
none of the configured DNS servers responded.

-systime : 20160817071325.358494-000 - Name Microsoft-Windows-DNS-Client User NT
AUTHORITY\NETWORK SERVICE
Name resolution for the name win10.ipv6.microsoft.com. timed out after none of
the configured DNS servers responded.

-systime : 20160814150919.620692-000 - Name Microsoft-Windows-DNS-Client User NT
AUTHORITY\NETWORK SERVICE
```

Name resolution for the name fs.microsoft.com timed out after none of the configured DNS servers responded.

For each event, the logs show the date and time when the event occurred, as well as the event source and the user. The source is the service, device driver, or application that wrote the event to the log. A source can subdivide the events it writes into multiple categories to let you easily find messages. Each event has an event ID, which helps MSt Product Support troubleshoot problems.

In the end we free the olevariant with:

```
       tObj:= unassigned;
```

The script can be found:

http://www.softwareschule.ch/examples/766_wmi_management2.txt

Now a few hints to late binding with WMI. A Variant variable is always initialized to Unassigned. You can assign almost any kind of value to the variable, and it will keep track of the type and value. To learn the type of a Variant, call the VarType function.
The VarType function returns an integer representing the current data type for a Variant variable <VariantVariable>.

When you use variant in an expression, the script automatically converts the other value in the expression to a Variant and returns a Variant result. You can assign that result to a statically typed variable, provided the Variant's type is compatible with the destination variable.

The most common use for Variants is to write an OLE automation client. You can assign an IDispatch interface to a Variant variable, and use that variable to call functions the interface declares. The compiler does not know about these functions, so the function calls are not checked for correctness until runtime. For example, you can create an OLE client to also print the log events of WMI installed on your system, as shown in the following code.

```
var
    wmiLocator: OLEVariant;
    wmiServices: OLEVariant;
    wmiObject: OLEVariant;
    wmiProp: OLEVariant;

    wmiLocator:= CreateOleObject('WbemScripting.SWbemLocator');
    wMIServices:= WmiLocator.ConnectServer('localhost', 'root\CIMV2', '', '');
```

Note: if you pass a user name to a localhost you get the exception:
{$Exception: SWbemLocator: User credentials cannot be used for local connections.}

Next step is to call the query:

```
    WMIProperty:= 'SerialNumber'; WMIClass:= 'Win32_BIOS';
    WmiObjectSet:= wMIService.ExecQuery(Format('Select %s from %s',
                          [WMIProperty, WMIClass]),'WQL',wbemFlagForwardOnly);
```

but thats not right way, remember, we drive with variants so pass just strings to eval!

```
    WmiObjectSet:= wMIService.ExecQuery(isQuery);
    writeln('Set returns: '+vartostr(WmiObjectSet.count))
```

```
134:
135: However I am getting a generic failure on the "WmiObjectSet.Item(0)" line. Is
     this method of doing things supported, or must I always use the "For Each" method
     with the collection that is returned from WMI queries? Yes you must or change to
     early binding before!
136:
137: The Item is separated from the class by a . character. The instance is identified
     by specifying a property value separated from the property name by a = character.
     The entire path then looks like \\SYSTEM\NAMESPACE:CLASS.PROPNAME="PROPVALUE".
     Again, the system and namespace may be defaulted but the class name must be
     specified.
138:
139:     tObj:= WmiObjectSet.item('Win32_NetworkAdapterConfiguration.Index=1');
140:
141: The script doesn't know anything about the Version property or any other method
     or property of the WMI OLE client. Instead, it compiles your property and method
     references into calls to the IDispatch interface. You lose the benefit of compile-
     time checks, but you gain the flexibility of runtime binding.
142:  If you want to keep the benefits of type safety, you will need a type library
     from the vendor of the OLE automation server as we did in our example.
143:
144: With the conncector you do also have a possibility to remote query a machine.
145: Note: Authenticated users cannot, by default, log events to the Application log
     on a remote computer. As a result, the example described in this topic will fail
     if you use the UNCServerName property of the NTEventLogEventConsumer class and
     specify a remote computer as its value.
146:
147: At last you can check this by the Event Viewer Tool:
148: The Event Viewer is a tool you use to examine the three NT event logs: System,
     Security, and Application. The event logs are in the directory
     \winntroot\system32\config, where winntroot is the directory that houses NT. The
     three log files are sysevent.evt, secevent.evt, and appevent.evt. You cannot use
     a regular text editor to view these files.
149:
150: ------------------------------------------------
151:  Example of SQL Stored Procedure Code with WMI:
152:  -- COLLECT DESIRED DATA
153:
154: EXEC  @rc = master.dbo.sp_OAMethod @wmiServices, 'InstancesOf', @wmiObjectSet
     OUTPUT, 'Win32_NetworkAdapterConfiguration'
155:
156: EXEC  @wmiObjectCount= master.dbo.sp_OAGetProperty @wmiObjectSet,'Count',
     @wmiObjectCount OUTPUT
157:
158: SELECT @loopIdx = 0
159:
160: WHILE @loopIdx < @wmiObjectCount - 1 BEGIN
161:
162:    EXEC  @rc= master.dbo.sp_OAMethod @wmiObjectSet,'Item', @wmiObject OUTPUT,
     @loopIdx
163:
164:    IF @rc <> 0 BEGIN
165:
166:    EXEC  @rc= master.dbo.sp_OAGetErrorInfo @wmiObjectSet, @oleSource OUTPUT,
     @oldDesc OUTPUT
167:
168:    END ELSE BEGIN
169:
170:    EXEC  @rc= master.dbo.sp_OAGetProperty @wmiObject,'Caption',
     @wmiNetAdapterName OUTPUT
```

```
171:
172:    EXEC  @rc= master.dbo.sp_OAGetProperty @wmiObject,'IPAddress',
    @wmiNetAdapterIP OUTPUT
173:
174:    END
175:
176:    SELECT  @loopIdx = @loopIdx + 1
177:
178: END
179:
180: -- CLEANUP
181:
182: EXEC master.dbo.sp_OADestroy @wmiServices
183:
184: EXEC master.dbo.sp_OADestroy @wmiLocator
185:
186:
187:
188: --------------Variant Test Function-----------------
189: var objIE_WMI: variant;
190:
191:     // Show type of a variant
192: procedure ShowBasicVariantType(varVar: Variant);
193: var
194:   typStr : string;
195:   basicType  : Integer;
196:
197: begin
198:   // Get the Variant basic type :
199:   // this means excluding array or indirection modifiers
200:   basicType:= VarType(varVar) and VarTypeMask;
201:
202:   // Set a string to match the type
203:   case basicType of
204:     varEmpty     : typStr:= 'varEmpty';
205:     varNull      : typStr:= 'varNull';
206:     varSmallInt  : typStr:= 'varSmallInt';
207:     varInteger   : typStr:= 'varInteger';
208:     varSingle    : typStr:= 'varSingle';
209:     varDouble    : typStr:= 'varDouble';
210:     varCurrency  : typStr:= 'varCurrency';
211:     varDate      : typStr:= 'varDate';
212:     varOleStr    : typStr:= 'varOleStr';
213:     varDispatch  : typStr:= 'varDispatch';
214:     varError     : typStr:= 'varError';
215:     varBoolean   : typStr:= 'varBoolean';
216:     varVariant   : typStr:= 'varVariant';
217:     varUnknown   : typStr:= 'varUnknown';
218:     varByte      : typStr:= 'varByte';
219:     varWord      : typStr:= 'varWord';
220:     varLongWord  : typStr:= 'varLongWord';
221:     varInt64     : typStr:= 'varInt64';
222:     varStrArg    : typStr:= 'varStrArg';
223:     varString    : typStr:= 'varString';
224:     varAny       : typStr:= 'varAny';
225:     varTypeMask  : typStr:= 'varTypeMask';
226:   end;
227:   // Show the Variant type
228:   ShowMessage('Variant type is '+typeString);
229:  end;
```

```
230:
231:    Procedure TestVariant3;
232:      var myVar : Variant;
233:    begin
234:      // Assign various values to a Variant
235:      // and then show the resulting Variant type
236:      ShowMessage('Variant value = not yet set');
237:      ShowBasicVariantType(myVar);
238:
239:      // Simple value
240:      myVar := 123;
241:      ShowMessage('Variant value = 123');
242:      ShowBasicVariantType(myVar);
243:
244:      // Calculated value using a Variant and a constant
245:      myVar := myVar + 456;
246:      ShowMessage('Variant value = 123 + 456');
247:      ShowBasicVariantType(myVar);
248:
249:      myVar := 'String '+IntToStr(myVar);
250:      ShowMessage('Variant value = String 579');
251:      ShowBasicVariantType(myVar);
252:    end;
253:
254:
255: Physically, the CIM resides in the %SystemRoot%\system32\wbem\Repository\FS\
    directory and consists of the following four files:
256:
257:     index.btr. Binary-tree (btree) index file.
258:     index.map. Transaction control file.
259:     objects.data. CIM repository where managed resource definitions are stored.
260:     objects.map. Transaction control file.
```