Downloading a file using XMLHttp

This recipe will show you how to download a file using the `XMLHttp` object, which we have seen in action in the *Checking whether page links are broken* recipe. Here we will expand on a theme and see how to synchronize our script using the `onreadystatechange` event handler to report on the progress of our download. The code includes the required modifications, and is the same as the one I used in a project a few years ago.

Getting ready

From the **File** menu, navigate to **New** | **Function Library** or use the *Alt* + *Shift* + *N* shortcut. Name the new function library as `Web_Download.vbs`. To use the `AutoItX` COM object, go to https://www.autoitscript.com/site/autoit/downloads/ to download and install AutoIt. This is absolutely necessary in order to ensure that the code given here will work properly with regard to the notifications display.

How to do it...

This recipe will demonstrate how to download the last build JAR file from a remote build machine and deploy it to the local machine. This is very useful to automate daily build updates and trigger automated tests to check the new build for sanity. Please take note that this solution comprises several components and is quite complex to grasp:

ch handles the download operation

ndler class, which listens to the `onreadystatechange` event and handles notifications about the progress

- The `AutoIt` class, which is a utility wrapper for the `AutoItX` COM object
- The `App_GetLastBuild` class, which controls the whole process

Proceed with the following steps:

1. First, we will define the following constants in the function library (of course, it would be better that at least some of these values be stored as `Environment` variables):

```
const S_OK = 0
const APP_PATH = "C:\Program Files\MyApp"
const DOWNLOAD_PATH = "C:\Documents and Settings\admin\My Document
const BUILD_PATH = "http://repository.app:8081/builds/last/"
const TMP_JAR = "App-1.0.0-build1.jar"
const APP_JAR = "App.jar"
const RES_ZIP = "App-1.0.0-build1-resources.zip"
```

> **Note**
>
> The preceding values are for illustration purposes only.

2. The next step is to write the `Http` class to handle the download process. The process is explained as follows:
   - First we will define the fields:

```
class Http
    Public m_objXMLHttp'The XMLHttp objectPrivate m_objHandl
    Private m_strLocalfilename 'Name of local filename
```

```
Private m_strUrl 'Address of download location
```

- Next, we will write the initialization and termination subroutines for the class:

```
private sub class_initialize
        Handler = new StateChangeHandler
        Handler.Http = Me
        XML = createobject("MSXML2.XMLHTTP")
    end sub

    private sub class_terminate
        Handler = Nothing
        XML = Nothing
    end sub
```

- Then, we will write the properties for the class that will provide access to the fields. Note especially the `XMLHttp` property, which is used to assign the `XMLHttp` object to the `m_objXMLHttp` field and also to set the `StateChangeHandler` object to the object's `onreadystatechange` event.

```
public property get XML
        set XML = m_objXML
    end property
    private property let XML(byref objXML)
        set m_objXML = objXML

        if typename(objXML) <> "Nothing" then _
                m_objXML.onreadystatechange = Handler
    end property
```

- Other properties of the class are quite trivial, just being accessors to the fields:

```
public property get LocalFilename
        LocalFilename = m_strLocalfilename
    end property
    private property let LocalFilename(byval strFilename)
        m_strLocalfilename = strFilename
    end property

    public property get Filename
        Filename = createobject("Scripting.FileSystemObject"
    end property

    public property get URL
        URL = m_strUrl
    end property
    private property let URL(byval strUrl)
        m_strUrl = strUrl
    end property

    private property get Handler
        set Handler = m_objHandler
    end property
    private property let Handler(byref objHandler)
        set m_objHandler = objHandler
    end property
```

- The `DownBinFile` method handles the process as shown in the following code:

```
public function DownBinFile(byval strURL, byval strDownloadP
        const adTypeBinary = 1
        const adModeReadWrite = 3
          const adSaveCreateOverwrite = 2

          dim arrTmp, oStream, intStatus, FSO, strInfo

        arrTmp = Split(strURL, "/")
        URL = strURL
        LocalFilename = strDownloadPath & Unescape(arrTmp(UB

        if XML.open("GET", strURL, false) = S_OK then
            XML.send

            set oStream = createobject("ADODB.Stream")
            with oStream
                .type = adTypeBinary
                .mode = adModeReadWrite
                .open
                do Until XML.readyState = 4
                    Wscript.Sleep 500
                Loop
                .write XML.responseBody
                .SaveToFile LocalFilename, adSaveCreateOverw

                strInfo = "Download of file '" & strURL & "'
                set FSO = createobject("Scripting.FileSystem
                if FSO.FileExists(LocalFilename) then
                    strInfo = strInfo & "success."
                    DownBinFile = 0
                else
                    strInfo = strInfo & "failure."
                    DownBinFile = 1
                end if
                with oAutoIt.Object
                    .ToolTip strInfo, 1100, 1000
                    .Sleep 7000
                    .ToolTip("")
                end with
              end with 'ADODB.Stream
              set oStream = Nothing
        else
            XML.abort
            strInfo = "Send download command to server faile
            with oAutoIt.Object
```

```
                call .ToolTip(strInfo, 1100, 1000)
                .Sleep 7000
                call .ToolTip("")
            end with
            exit function
        end if
    end function
end class
```

3. The `Http` class here refers to `StateChangeHandler`, which in turn uses the `AutoItX` COM object to display a notification to inform about the progress of the download process.

   The `Exec` method is defined as `Public Default` so that it is automatically triggered when the object is referenced. As an instance of this object is assigned to the `onreadystatechange` event of the `Http` request object, every time `readystate` changes, this function is performed to display the updated data on the download process in the notification area on the taskbar.

```
class StateChangeHandler
    public m_objHttp

    public default function Exec()
        dim strInfo, intDelay

        intDelay = 0
        strInfo = "State changed: " & Http.XML.readyState & vbNewL
        Select Case Http.XML.readyState
            Case "3"
                strInfo = strInfo & vbNewLine & "Please wait..."
            Case "4"
                strInfo = strInfo & vbNewLine & "Finished. Total "
                intDelay = 1500
            Case else
        End Select
        'with AutoIt
        with oAutoIt.Object
            .ToolTip strInfo, 1100, 1000
             .Sleep 500+intDelay
             .ToolTip("")
         end with
    end function

    public property get Http
        set Http = m_objHttp
    end property
    public property let Http(byval objHttp)
        set m_objHttp = objHttp
    end property
end class
```

4. The `AutoItX` COM object is wrapped by the `AutoIt` class for easier use:

```
class AutoIt
    private m_oAutoIt

    public default property Get Object
        set Object = m_oAutoIt
    end property
    private property let Object(byval AutoItX)
        set m_oAutoIt = AutoItX
    end property

    private sub class_initialize
        Object = createobject("AutoItX3.Control")
    end sub
    private sub class_terminate
        Object = Nothing
    end sub
end class
```

5. Finally, the next `App_GetLastBuild` class controls the whole process. The whole process is explained as follows:

   - First, we define the fields as follows:

```
class App_GetLastBuild
    private oHttp
    private Status
    private FSO
    private FoldersToDelete 'Local folders to delete'
    private ResourcesZIP
    private OrigJar
    private DestJar
    private BuildPath
    private ExtractPath
```

   - Then, we define a method that will assign these fields the value:

```
public function SetArgs()
        FoldersToDelete = Array(APP_PATH & "\images", APP_PA
        ResourcesZIP = RES_ZIP
        OrigJar = TMP_JAR
        DestJar = APP_JAR
        ExtractPath = APP_PATH
        BuildPath = BUILD_PATH
    end function
```

   - Next, we define the `Exec` method as default; a method that will control the whole process:

```
public default function Exec()
        call SetArgs()

        '1) Delete local folders
        DeleteFolders(FoldersToDelete)

        '2) Download resources zip
        Status = oHttp.DownBinFile(BUILD_PATH & ResourcesZIP
        if Status <> 0 then exit function
        'Extract the resources
        call ExtractZipFile(DOWNLOAD_PATH & ResourcesZIP, AP
        'Delete resources zip
        FSO.DeleteFile(DOWNLOAD_PATH & ResourcesZIP)

        '3) Delete main GUI jar
        FSO.DeleteFile(APP_PATH & "\lib\" & DestJar)

        '4) Download the updated GUI jar
        Status = oHttp.DownBinFile(BUILD_PATH & OrigJar, DOW
        if Status <> 0 then exit function
        'Copy the updated GUI jar
        call FSO.CopyFile(DOWNLOAD_PATH & OrigJar, APP_PATH
        'Rename GUI jar
        call FSO.MoveFile(APP_PATH & "\lib\" & OrigJar, APP_

        'Delete the downloaded GUI jar
        FSO.DeleteFile(DOWNLOAD_PATH & OrigJar)
    end function
```

- We then define the method to delete folders (for cleanup purposes before downloading):

```
public function DeleteFolders(byval arrFolders)
        dim ix

        for ix = 0 To UBound(arrFolders)
            if FSO.FolderExists(arrFolders(ix)) then
                FSO.DeleteFile(arrFolders(ix) & "\*.*")
                FSO.DeleteFolder(arrFolders(ix))
            end if
        next
    end function
```

- Next, we define a method to uncompress a ZIP file:

```
public function ExtractZipFile(byval strZIPFile, byval strEx
        dim objShellApp
        dim WsShell
        dim objZippedFiles

        set objShellApp = createobject("Shell.Application")
        set WsShell = createobject("Wscript.Shell")

        set objZippedFiles = objShellApp.NameSpace(strZIPFil

        objShellApp.NameSpace(strExtractToPath).CopyHere(obj

        'Free Objects
        set objZippedFiles = Nothing
        set objShellApp = Nothing
    end function
```

- Then we define the initialization and termination subroutines:

```
private sub class_initialize
        set FSO = createobject("Scripting.FileSystemObject")
        set oHttp = new Http
    end sub
    private sub class_terminate
        set FSO = Nothing
        set oHttp = Nothing
    end sub
end class
```

6. In `Action1`, the following code will launch the download process:

```
dim oAutoIt
dim oDownload

set oAutoIt = new AutoIt
set oDownload = new App_GetLastBuild

oDownload.Exec

set oDownload = Nothing
Set oAutoIt = nothing
```

How it works...

As mentioned in the previous section, the solution involves a complex architecture using VBScript classes and several COM objects (`AutoItX`, `FileSystemObject`, `ADODB.Stream`, and so on). A detailed explanation of this architecture is provided here.

Let us start with the main process in `Action1` and then delve into the intricacies of our more complex architecture. The first step involves the instantiation of two of our custom classes, namely, `AutoIt` and `App_GetLastBuild`. After our objects are already loaded and initialized, we call `objDownload.Exec`, which triggers and controls the whole download scenario. In this method we do the following:

- Initialize the class fields.

- Delete the local folders.
- Download the binary resources file. We check if the returned status is OK; if it is otherwise, we exit the function.
- After the download process ends, we extract the contents of the ZIP file to the application path, and then delete the ZIP file.

We then start the process for the main JAR file as follows:

- Delete the old file.
- Download the JAR file. We check if the returned status is OK; if it is otherwise, we exit the function.
- After the download process ends, we copy the file to the target location and rename it (assuming the last build main JAR file always carries the same name, regardless of the version).

Now, let us examine what happens behind the scenes after the two classes are instantiated, as mentioned in this section.

The `AutoIt` class automatically instantiates the `AutoItX.Control` object through its `Private Sub Class_Initialize` subroutine. The `App_GetLastBuild` class, through its own `Sub Class_Initialize`, automatically creates these objects, namely, a `Scripting.FileSystemObject` object and an instance of our `Http` custom class.

Let us take a close look at the `Sub Class_Initialize` of the `Http` class:

```
    private sub class_initialize
        Handler = new StateChangeHandler
        Handler.Http = Me
        XML = createobject("MSXML2.XMLHTTP")
    end sub
```

Here we can see a strange thing. Our `Http` object creates an instance of the `StateChangeHandler` class and immediately assigns the `Http` property of the `Handler` object a reference to itself `)Handler.Http = me`). That is, the parent object (`Http`) has a reference to the child object (`StateChangeHandler`) and the latter has a reference to the parent object stored in its `Http` property. The purpose of this seemingly strange design is to enable interoperability between both objects.

Finally, an instance of `XMLHttp` is created. As seen in our recipe, on checking for broken links, this object provides the services we need to manage communication with a web server through the HTTP protocol. Here, however, there is something extra because we want to notify the end user about the progress of the download. Let us take a closer look at the way this instantiation is handled:

```
  private property let XML(byref objXML)
        set m_objXML = objXML

        if typename(objXML) <> "Nothing" then _
              m_objXML.onreadystatechange = Handler
    end property
```

We pass the `XMLHttp` object created in our `Sub Class_Initialize` subroutine and check if it is a real object (which it always should be because of the way we have designed our code). Then, we implicitly assign our handler's default method to the `XMLHttp` event `onreadystatechange`. Recall that a public default function in a VBScript class is executed whenever an object instance of the class is referenced without explicitly calling a method or property using the dot operator. This way, whenever the `readystate` property of the `XMLHttp` object changes, the default `Exec` method of the `StateChangeHandler` object is automatically executed, and a notification about the status of the process is displayed using the `AutoIt` COM object. Just one thing is missing from our code, a check of the `Http` status after `XMLHttp.send` and later on.