OAuth2 and REST Course

OAuth2

General

The OAuth2 mechanism (slides).

Practical - Google OAuth2

1. Create a web app:
    a. Run Studio (v18.2) as Administrator (since creating a new WebApp)
    b. Create a new workspace and a Desktop Web Project
    c. Use OAuth2 library (included on USB stick)
    d. Copy the AppHTML/OAuth2 directory from the library into AppHTML in your workspace
    e. Add **`<script src="OAuth2/OAuth2.js"></script>`** line to index.html
    f. Create a Web View
    g. Add the line **`Delegate Set phoDefaultView to Self`**
    h. Rename the default web form to oScope & label "Scopes:"
    i. Drag in an oOAuth2 object from the OAuth2 section of the Class Palette
    j. Drag in a button and have OnClick send Login of the oOAuth object
2. Sign up as a Google developer (need a Google account)
3. Register your app: https://developers.google.com/ then: https://console.developers.google.com/
4. Go to "Credentials" and Create a Project (choose a name for the project) - choose "OAuth 2.0" and "Web Application"
5. Create credentials:
    a. Copy the client ID to the property wpsClientID in your OAuth object
    b. Copy the client secret to the psClientSecret property
6. Configure Consent Screen: Name, Home page & Logo URL then click Save
7. Select "Web application" and give it a name, a "JavaScript origins" of http://localhost and a "Redirect URI" of http://localhost/GoogleTest/OAuth2/Callback.html
8. Go to https://developers.google.com/identity/protocols/OAuth2WebServer scroll down to "Redirecting to Google's OAuth 2.0 server" then select the HTTP/REST tab
9. Copy the endpoint address (https://accounts.google.com/o/oauth2/v2/auth) to the wpsOAuth2Url property of the object in your program
10. Looking at the list of parameters for HTTP/REST access, the only non-optional one that the OAuth2 component does not deal with automatically is scope, which depends on the specific API(s) we want to access. We will use the Google Drive API, so doing a search on "Google Drive API scope" we should find that "`https://www.googleapis.com/auth/drive`" should work (full access), so we will set the psValue property of the scopes form to that. We use "AddParam" in the OAuth2 object's **`OnBeforeLogin`** event to set it to whatever was entered in the "scopes"

form (we will also send ClearParams first for safety). However we will also add another parameter "**`access_type`**" with a value of "**`offline`**".

11. On that web page (https://developers.google.com/identity/protocols/OAuth2WebServer) scroll down to "Handling the OAuth 2.0 server response and select the HTTP/REST tab
12. There we see that the first field we have to pass in our request for an access token is called "code" - the OAuth2 class defaults this to "token", so in our OAuth2 object we need to add the line **`Set wpsResponseType to "code"`**
13. Scrolling down that page some more, we can see the definition of what a request for an access token looks like - there we can see that:
    a. The server to call is **www.googleapis.com**, which we set the psTokenHost of the OAuth object to
    b. The path on that is **/oauth2/v4/token**, which we set the psTokenPath of the OAuth object to
14. Now let's add some more controls to our web view so we can see what is going on:
    a. Beside the "Login" button add an "Error" form, but **`Set pbVisible to False`**
    b. Add a Web Label with **`Set psCaption to "Not logged in"`**
    c. Add a "Token" form so we can see the access token we will hopefully get
    d. Add an "Expires" form so we can see the time it is valid for
    e. In the OAuth2 object's **`OnLogin`** event add the code:

```
String  sToken
Integer iExp

WebSet psCaption of oLoginState to "Logged in"
WebGet wpsAccessToken           to sToken
WebGet wpiExpiresIn             to iExp
WebSet psValue of oToken        to sToken
WebSet psValue of oExpires      to iExp
WebSet pbVisible of oError      to False
```

    f. In the OAuth2 object's **`OnLoginFail`** event add the code:

```
String sErr sErrDesc

WebSet psCaption of oLoginState to "Login failed"
WebGet wpsErrorCode             to sErr
WebGet wpsErrorDesc             to sErrDesc
WebSet psValue of oError        to ;
                           (sErr + ":" * sErrDesc)
WebSet pbVisible of oError      to True
```

15. With luck and a following wind, we should now be ready to request access, so compile/run the application and when it comes up in the browser, click the login button

Testing our access

1. Once we have that working, we'd like to test that it actually gives us access - exit the debugger and:
    a. Add **`Set_Argument_Size 10000000`** (a big number - ~10Mb in this case) at the top of the view
    b. Drag a cHttpTransferUIG object into the view (if you wish, right-click the class and select "Go To Definition" to see what has been added in that subclass)
    c. Add an "API Call" form so we can enter what to call to test that it works - **`Set psValue to "drive/v2/files"`** (we'll look at where that comes from later)
    d. Add a Procedure **MakeAPICall** to the view - empty for now
    e. Add a "Call API" button to trigger doing that: **`Send CallAPI`** in its **`OnClick`**
    f. Add a Web Edit to display the result we get back (or news of what went wrong!) - **`Set pbFillHeight to True`** so it will have all the rest of the vertical space
    g. In the MakeAPICall procedure, add the following code:

```
String  sPath sResp sToken
Integer iOK iStat

Send ClearHeaders of oHttp
Send Reset of oHttp
Set psRemoteHost of oHttp       to "www.googleapis.com"
Set piRemotePort of oHttp       to rpHttpSSL
Set peTransferFlags of oHttp    to ifSecure
WebGet wpsAccessToken of oOAuth to sToken
WebGet psValue of oCall         to sPath
Get AddHeader of oHttp ;
            "Authorization" ;
            ("Bearer" * sToken) to iOK

Get HttpVerbAddrRequest of oHttp ;
                        sPath ;
                        0 ;
                        0 ;
                        False ;
                        "GET"    to iOK

If iOK Begin
    Get ResponseStatusCode of oHttp to iStat
    Get psData of oHttp to sResp

    If ((iStat >= 200) and (iStat <= 299)) Begin
        WebSet psValue of oResult to sResp
```

```
            End
            Else Begin
                WebSet psValue of oResult to ;
                   ("HTTP Error:" * String(iStat) * sResp)
            End

        End
        Else Begin
            WebSet psValue of oResult to ;
                   ("HTTP GET from" * ;
                   psRemoteHost(oHttp(Self)) + "/" + sPath * ;
                   "failed")
        End
```

2. Compile/run the app again; click the login button, then when that has completed, click the Call API button
3. If you have a lot of files in Google Drive, the call may take a few seconds to complete
4. If everything worked, you should see a big pile of JSON data in the Results form

Unravelling JSON

(slides)

Using RESTGen

Service providers like Google generally provide some form of sample JSON to illustrate what you need to send to their service operations and what you should get back from them.

If we do a Google search for - in this case - Google Drive REST API, we should find it quite quickly: https://developers.google.com/drive/v2/reference/. We can now see where the host - www.googleapis.com - and base path - drive/v2/ - we used above came from. Scrolling down to the "list" operation we can see that final part of that path is "files" and the HTTP verb to use is GET. If we click on that "list" operation, we get to a page that describes it in more detail. Further down there is a "Try it!" section - ensure that you switch the toggle "Authorize requests using OAuth 2.0" to "on", when you should be presented with a pop-up list of scopes to authorize. You can then click the "EXECUTE" button farther down the page to see the output, which should be essentially the same as we saw in our Results Web Edit.

- Run the RESTGen program (included on USB stick)
- In the webapp's Results edit, go down to the end of the first item in the "items" array (so just before the second occurance of `"kind": "drive#file"`)
- Position the cursor after the closing "}", but before the preceding comma, then holding down the shift-key scroll to the end of the data and click just after the closing "}" of the last list item, then press "Del", removing all but the first item in the "items" array
- Select all (Ctrl-A) then copy (Ctrl-C)
- Paste into RESTGen's JSON window

We should now have a fair slice of data, but not so much as to overwhelm RESTGen's window.

Give the outer struct a name - something like tGoogleDriveList. Tabbing through to the inner struct prefix window will copy that value to there as well. You can make choices about where the generated packages will go and how they will reference each other - hint: if you put RESTGen on your Studio's "Tools" menu with a parameter of "<workspace>", it will already have sensible defaults for these.

Click the "Generate" button.

If we have done that correctly then the output window should show a list of the package RESTGen has created, plus the fact that there were zero warnings. (Note: the commonest cause of warnings is empty arrays in the sample JSON - if they are empty the program can't guess what type should be in them, so it defaults them to strings, but if it is not an array of strings, you should manually correctly populate the JSON and click "Generate" again until there are no warnings - it will silently overwrite previously generated packages without warning, so be aware of that also.)

A Windows Explorer window should also open showing the packages in their directory.

(Following Sture's convention in his VDFXray program, all of the generated handler objects are called "oStructHandler_*nameOfTheStruct*.)

Making calls to the API

Now we can start to use the data, rather than simply displaying it as raw JSON.

Back in the Studio, we now "Use" the outer struct package (it will "Use" the inner ones in turn) and in our code we can declare a variable of the type name we told it.

In the list of Use statements add another: **`Use ApiStructs\tGoogleDriveFiles.pkg`** (or whatever you called it).

Drag a cWebList onto your view (call it oFilesList), beneath the Results edit, **`Set pbDataAware to False`** and **`Set pbFillHeight to True`**, then add three columns - two cWebColumns and a cWebColumnCheckbox: "Title", "Owner" and "Shared".

Add an OnManualLoadData Procedure containing the following:

```
tGoogleDriveFiles tFiles
UChar[] ucaTest
Integer iOldSize i iMax
String  sJson
```

```
Get pucaData of oHttp to ucaTest
Get_Argument_Size to iOldSize
Set_Argument_Size (iOldSize max (SizeOfArray(ucaTest) + 1000))
Get psData of oHttp to sJson

Send JsonStringToStruct of oStructHandler_tGoogleDriveFiles ;
                                        (&sJson) (&tFiles)

Set_Argument_Size iOldSize

Move (SizeOfArray(tFiles.items) - 1) to iMax

For i from 0 to iMax
    Move i to aTheRows[i].sRowID
    Move tFiles.items[i].title to ;
                aTheRows[i].aCells[0].sValue
    Move tFiles.items[i].ownerNames[0] to ;
                aTheRows[i].aCells[1].sValue
    Move tFiles.items[i].shared to ;
                aTheRows[i].aCells[2].sValue
Loop
```

In our call, in the "success" block (iOK True and iStat in the 200 range), we can now add the line:

```
Send GridRefresh of oFilesList
```

Once we have the data in our struct, we don't need to worry about the size of the data as a single string any more, so we can change the line in our "success" block setting the result object's psValue to just "Success!" and remove the Set_Argument_Size 10000000 at the top of the view as well.