Using Web APIs in DataFlex

Mike Peat, Unicorn InterGlobal Limited
Synergy 2015, Seattle

Web APIs

- Generally:
  - Use REST over HTTP, thus receiving a lot of their input in the URL itself
  - Use JSON for accepting and returning complex data
  - Require some kind of authentication

REST

- Representational State Transfer
- Alternative to SOAP-based web services
- Generally done over HTTP
- Uses a range of HTTP verbs:
  - GET (retrieves stuff, but doesn't change anything)
  - POST (adds stuff)
  - PUT (updates stuff)
  - PATCH (amends stuff)
  - DELETE (I'm sure you can work that one out yourself)

POST vs PUT vs PATCH

- You might ask what are the differences between these three verbs
- I'm sure somebody knows, but I am not going to try to explain
- And I don't care, because I am *using* these APIs, not *designing* them
- I just use what the designers specify

Verbs

- GET and POST are familiar from the DataFlex cHttpTransfer class
  - HttpGetRequest
  - HttpPostRequest
- DataFlex claims it also does PUT (I never had)
- We *had* no mechanism for DELETE and PATCH

New for DataFlex 18.1

- John Tuohy kindly added a new function into the cHttpTransfer class
- **HttpVerbAddrRequest**
- It allows you to *specify* which HTTP verb you want to use
- Uses the *Address* and *Length* of the required content (if any - use zeros if there is none)

HttpVerbAddrRequest function

Get **HttpVerbAddrRequest** {path} {address} {length} ;
                                        {is-file} {HTTP-verb} to {integer}

```
String sData
Integer iOK
Move "some stuff (serialised JSON usually)" to sData
Get HttpVerbAddrRequest "api/v1.0/me/sendmail" ;
        (AddressOf(sData)) (Length(sData)) False "POST" ;
                                                    to iOK
```

JSON

- Passing complex data to/from APIs
- Create a "struct" conforming to the structure of the data being passed (example)
- Compile your program
- Use Sture Andersen's *excellent* VDF XRay tool to generate struct handler packages by scanning the output .PRN file

Parsing and serialising JSON

- "Use" the packages generated by VDF XRay (generally you only have to use the outer struct one, as *that* will "Use" the inner ones)
- Call StringToDoc, then JsonToStruct to parse received JSON strings into struct variables
- Call StructToJson, then DocToString to serialise struct variables to JSON strings

Microsoft Office 365

- Range of APIs:
  - **Outlook**:
    - Mail
    - Calendar
    - Contacts
  - **SharePoint**:
    - Files
- The two categories work somewhat differently
- (There is also a "Discovery" service)

Office 365 Mail API operations

- Get messages GET
- Send message POST
- Reply to message POST
- Forward message POST
- Update message PATCH
- Delete message DELETE
- Move/Copy message POST
- Get attachments GET
- Create attachment POST
- Delete attachment DELETE
- Get folders GET
- Create folder POST
- Update folder PATCH
- Delete folder DELETE
- Move/Copy folder POST

Office 365 Contacts API operations

- Get contacts GET
- Create contact POST
- Update contact PATCH
- Delete contact DELETE
- Get contact folders GET

Office 365 Calendar API operations

- Get events GET
- Create event POST
- Update event PATCH
- Delete event DELETE
- Get attachments GET
- Create attachment POST
- Delete attachment DELETE

- Get calendars GET
- Create calendar POST
- Update calendar PATCH
- Delete calendar DELETE
- Get calendar groups GET
- Create calendar group POST
- Update calendar group PATCH
- Delete calendar group DELETE

Office 365 Files API operations

- Create folder PUT
- Get folder props GET
- List folder contents GET
- Update folder props PATCH
- Copy folder POST
- Delete folder DELETE
- Create/update file PUT
- Download file GET
- Get file properties GET
- Update file props PATCH
- Copy file POST
- Delete file DELETE
- Get drive props GET

Authentication

- These APIs (in some cases) support two mechanisms for authenticating users
  - User ID and Password
  - OAuth 2.0

- (In the case of the Office 365 Files API <u>only</u> OAuth2 is supported… at least so far as I can tell)

User ID and Password

- Simple to use
- Just set the psUsername and psPassword properties of your HttpTransfer object
- Microsoft say it is OK for testing, but not production
- The problem is that at some point the user has to give your application their credentials

OAuth2

- **Is *complicated*!!!** (especially the MS way)
- Basically 4 steps:
    - Register your application with the provider
    - Make a call that invites the user to log in and allow your app access to their data
    - Get an access token based on that consent
    - Use that token in your app's requests for data

Microsoft OAuth2

- Get an Office 365 Developer account
- Go into Microsoft Azure Portal and log in
- Find Active Directory (AAD: Azure Active Directory)
- Navigate to: *your-company* → Applications → Add → Set up its properties (URI, Callback URL)

MS OAuth2 continued...

- Go into "Configure"
- Get the Client ID for the application
- Create a key (aka "*client secret*"... remember that: it is not obvious those are the same thing. And copy that key - you won't see it again!)
- Add application permissions and delegated permissions

MS OAuth2 continued...

- In JavaScript, we open the Microsoft OAuth2 URL in a new browser window, passing it a whole lot of stuff in the query string
- If the user is not logged in to their Microsoft 365 account it will show them a login screen to do that
- Then they will be presented with a screen asking them to give your app access

MS OAuth2 continued...

- If they grant that, Microsoft redirects them to your callback URL
- We wait in a JavaScript timer loop looking at the window's URL until it changes to that
- Then we capture that full URL
- Parse the information from *that* query string
- Get the authorisation code out of that

MS OAuth2 continued...

- Use the authorisation code to request an access token
- (I couldn't get this to work in JavaScript, because of cross-site scripting restrictions, so I had to do this step back in DataFlex)
- Use that access token in the "Authorization" HTTP header of your API requests

Microsoft JSON oddities

- Microsoft's JSON has some awkward names
- Often starting "@odata." (example)
- Such as "@odata.context" or "@odata.id
- We can't have struct element names which match those exactly, so we need to replace them: with "odata_" on the way in and reverse the procedure on the way out

Demonstration

Other APIs

- Microsoft are not the only player in the Web API space (although they have a lot more than I have talked about so far)
- The 800-pound gorilla is Google (Amazon is big too, but in a different way)
- No point in even starting to list Google's range of APIs (see here)

Google Web APIs

- Google APIs only support OAuth 2.0 authentication
- Fortunately their mechanism is much simpler than Microsoft's
- Sign up for a Google Developer account and go to the Developer Console

Google OAuth 2.0

- Create a new Client ID for your app with a redirect URL and where your JavaScript will be
- Copy the client ID and configure the APIs it wants to access
- Configure a consent screen (optional)

Google OAuth 2.0 continued

- Call the Google OAuth2 endpoint in a new browser window, specifying the "scopes" your app wants to access
- Parse the redirect URL that is taken to when the user gives consent
- Extract and store the access token
- Use that token in the query string of your API HTTP requests

Demonstration

That was...

- The result of a **single call**
- To **one** operation
- Of **one** Google API
- Then organising and using the data coming back from that
- Then using the URLs of the documents to access them

So far...

- I have only dipped a toe in the water of the ocean of available APIs (a large set from Microsoft - a *vast* array from Google… and there are others too)
- There really seems to be enormous potential for things you can do with these

End of Presentation

Thank you for watching/listening