REST clients: Using the Google APIs in Free Pascal

Michaël Van Canneyt

May 3, 2015

Abstract

This article demonstrates how to use Free Pascal components that implement the Google APIs, to access Google's services. Using the Google APIs presumes use of OAuth 2, and the use of REST technologies. The article will also show how you can create your own Google API in Pascal using the Google Discovery service.

1 Introduction

More and more, applications are connected to the Web or implemented solely on the web (Facebook, twitter). Data is fetched from and stored in the web, even for traditionally desktop oriented software such as spreadsheets or word-processing software. Prime examples are Google docs (or Apps) and more recently Microsoft Office365. One could even say that these WEB apis become equally important as the traditional OS and installed software APIS: Software running on Tablets and smartphones are expected to interact with online services.

To be able to interact with these web applications and this data, APIs are needed. This is increasingly done using REST technologies. (REST stands for Representational State Transfer). REST is not a protocol, rather an architectural way to make data available on the web. It rests on 2 pillars:

1. Everything is a resource, acessible through a URI - which almost implies use of the HTTP protocol. In database terms one could say that every record of a table is accessible through its own URI.
2. Data manipulation follows mostly the CRUD (Create Read Update Delete) pattern, much as data in a database. These operations nicely translate to HTTP verbs POST, GET, PUT and DELETE.

Since all this is very much the technology used in web browsers, it is easy to understand and use. The downside of being an architecture is that there is no strict protocol, each application developer can develop his own protocol.

Where XML was the format of choice for messages in SOAP-related APIs, for REST APIs this is replaced mostly with data descriptions in JSON format (JavaScript Object Notation), which has several advantages over XML: it is less verbose (that applies to the specification as well: the JSON specification fits on an A4 page), is less sensitive to whitespace, and has some native notion of data types (string, boolean, number, object, array). Last but not least, it is a subset of Javascript, and as such can be handled natively by any browser.

Obviously all the data on the web needs to be protected from unauthorized access. This protection is increasingly done using OAuth (version 2). OAuth is also implemented using JSON. OAuth in essence relies on the user giving consent to an application (mostly the browser) to use data on his or her behalf.

Figure 1: The Google APIs component palette

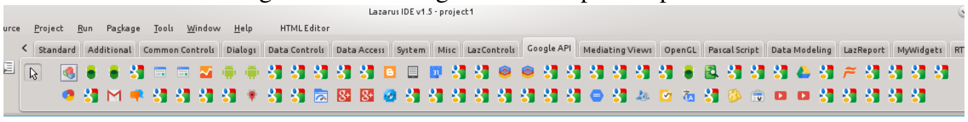
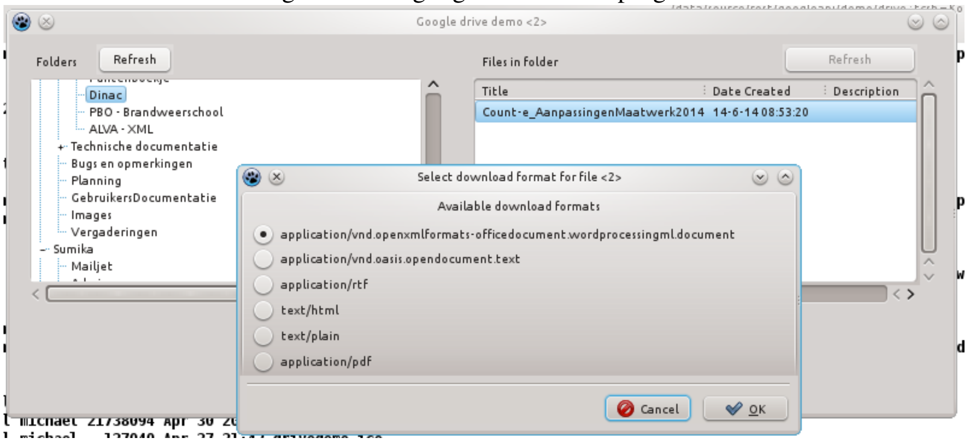

Lazarus IDE v1.5 - project1
urce Project Run Package Tools Window Help HTML Editor
Standard Additional Common Controls Dialogs Data Controls Data Access System Misc LazControls Google API Mediating Views OpenGL Pascal Script Data Modeling LazReport MyWidgets RT

Figure 2: The google drive demo program



Dinac
PBO - Brandweerschool
ALVA - XML
Technische documentatie
Bugs en opmerkingen
Planning
GebruikersDocumentatie
Images
Vergaderingen
Sumika
Mailjet

Count-e_AanpassingenMaatwerk2014 14-6-14 08:53:20

Select download format for file <2>

Available download formats

- [✓ application/vnd.openxmlformats-officedocument.wordprocessingml.document]
- application/vnd.oasis.opendocument.text
- application/rtf
- text/html
- text/plain
- application/pdf

All the technologies needed to perform REST operations and OAuth authorization are available in Free Pascal. It was therefor only a matter of time before a comprehensive set of components became available to easily access web APIs. Indeed, the various Google APIS are now available as components (figure **??** on page **??**) for the lazarus component palette, so they can be dropped on a form, making it really easy to access the Google APIS.

In this article we'll describe how to access the Google APIs using REST in Free Pascal and Lazarus, and demonstrate how they can be used in sample applications such as the Google Drive demo (figure 2 on page 2) To understand better how the various classes are hooked up, they will not be dropped on the component paletter, but we'll create and connect them in code.

Access to Microsoft Office365 is also worked on, but is the subject of a later contribution.

2 Architecture

The Free Pascal implementation of the web APIS makes several assumptions:

1. Transport uses the HTTP(s) protocol. Several TCP/IP socket implementations exist (Synapse, lnet, Indy, the FPC native client). Each developer has his own preference which implementation he uses. So, the REST APIs should work with each of those.

   That means that the HTTP request and response mechanism is abstracted in a new class `TFPWebClient`. Concrete implementations of this abstract class have been made for Synapse and TFPHTTPClient.

2. Authentication of the HTTP requests happens using OAuth2, but other mechanisms can be implemented as well. Since the OAuth2 protocol involves exchanging tokens with a webserver, it needs a HTTPS transport layer as well.

3. Serialization of objects is done using JSON. Therefor the basic REST object contains a JSON serialization mechanism, based on RTTI.

The upshot of these architectural decisions is that there are several classes involved in the REST implementation:

TFPWebclient to handle HTTP(s) messages. A descendent of this client is needed, which uses a particular TCP/IP suite to actually send the request and read the response. A request is represented by a `TWebRequest` class, the response will come in the form of a `TWebresponse` class.

Requests are executed using the `ExecuteRequest` and `ExecuteSignedRequest` methods: To each `TFPWebclient` instance, a `TRequestSigner` component can be attached: this component is allowed to examine the request and response when they are sent or received. This allows a request to be signed, for instance by adding a `Authorization` header with a `Bearer` token.

TFPOauth2Handler is a class that handles OAuth 2 authentication. Technically, it is a descendent of a `TRequestSigner` that will add the OAuth2 header. This class may use the `TFPWebclient` instance (or a second `TFPWebclient` instance) to execute token exchange requests as part of the OAuth2 flow.

The class can be used in offline mode (for desktop apps) as wel as in online mode (for web applications).

TRestObject This is the basic object that represents a REST resource. It has 2 important methods: `LoadFromJSON` and `SaveToJSON`. These methods use the RTTI to create a JSON representation of the object, or to read the object properties from a JSON representation. It also has an mechanism to record which properties have been changed.

The mechanism to record property changes is needed because many REST APIs allow both PUT and PATCH (or UPDATE) methods. A PUT method generally completely replaces a resource with the new value specified in the request, whereas PATCH modifies the resource by applying the changes in the request to the existing resource.

To be able to support PATCH-like functionality, a mechanism is needed to record which properties have changed, and to send only these properties in a request. This mechanism is implemented using the property Index mechanism. Each property has a unique index, and a property setter which accepts this index (the property getter may or may not use this index). When the property is set, the index is used to record the change. Since the index is unique, the RTTI can then be used to construct a list of property names that were modified.

Armed with these objects, a REST API client can be made: all that needs to be done is create descendents of TRestObject, and load them from a HTTP response. To modify data on the REST server, the object's properties are set, and the object serialized to JSON. This JSON is then sent (with the appropriate HTTP method) to the server. That is basically it.

3 Service descriptions

Google has more than 100 APIs, each of which has more than 1 resource, containing much data structures which can be manipulated. To write objects and serialization code for each of these resources would be a very tedious task indeed. Luckily, this is not necessary. Google offers a Google Service Discovery service, which returns a JSON document that completely describes each of its rest-based APIs. The Google discovery service is described at

```
https://developers.Google.com/discovery/
```

The service consists of 2 parts: it lists the services offered by Google, and it offers a description of the REST API for each of these services. For people acquainted with SOAP implementations: This description is equivalent to the WSDL description document. it is based on JSON schema:

`http://json-schema.org/`

The REST API description also contain information on the authorizations needed to use the resources in the APIs (the so-called authorization scope). It is very important to be aware of these scopes: the user of the API will be asked for consent based on these scopes. More on this below.

The Microsoft REST APIs are based on their OData specification, and OData based services have a similar document (the service document), based on EDMX, described at:

`https://www.odata.org/`

Unfortunately, the EDMX description is still in XML (somewhat awkward if the JSON format is to be used) and not as complete as the Google-provided descriptions, implying that always a certain (very small) amount of manual interpretation is required.

Converters for these 2 formats have been implemented in FPC: The Google Service Discovery REST description document can be converted automatically to a complete Object Pascal implementation of a client for the REST APIs. There are 2 programs available to do this:

- A command-line program which can be used to convert a single API to an object pascal unit. The REST description can be a file with the JSON description of the service, or the program can download a service description from the Google service discovery server.
- A GUI program that allows you to browse and search the Google APIs and convert a selected API to an Object pascal unit.

The Google service discovery itself is a REST API. So, an Object Pascal implementation can be generated for it, using the command-line program. This has been done, and the Google discovery program was constructed using this unit.

The code to convert a Google REST API description to an Object pascal unit is a component (`TDiscoveryJSONToPas`), which can be descended from and modified if need be: if the choices made by the author for the generated code are not to your particular liking, the code can easily be modified to generate different code. It has 3 important methods:

LoadFromStream This method can be used to load a REST service description from a stream containing valid JSON. There are 2 auxiliary methods `LoadFromFile` and `LoadFromJSON` to make life easier.

SaveToStream Calling this will generate the pascal code, and save the resulting code to stream. The `SaveToFile` method will do the same but save directly to file, and will set the unit name of it was not yet set.

Execute can be used to generate the code. The code is then available in the `Source` property.

There are some properties that can be used to control the code: the parent class name for the generated resource classes, the unit name to generate, the indentation can be set and so forth. The component can be found in the googlediscoverytopas unit. A similar component was made to convert the Microsoft OData service discriptions to Pascal code.

A small command-line program (googleapiconv) is available that can be used to generate a pascal source file from a service description file. It can also fetch the description from the Google discovery service, based on the name (and version) of the service, or simply by providing a URL. For example, the following command will generate a unit for the Google calendar API :

```
googleapiconv -s calendar/v3 -o calendar.pp
```

the same can be obtained using

```
googleapiconv -u http://www.googleapis.com/discovery/v1/apis/calendar/v3/rest -o
```

4 A service description breakdown

Since the Google Discovery service is a service, there is a description of itself in the Google discovery service. Therefor, the command-line program was used to create a unit googlediscovery which implements the Google discovery service API.

A Google API breaks down in 4 parts, all of which have a base class in the `googleservice` unit:

TGoogleClient This is a simple component that handles the transport and authorization, it uses a `TFPWebClient` and a `TFPOauth2Handler` to communicate with Google servers.

TGoogleAPI There is a descendent of this component for each Google service API, which handles all calls to the service. It uses a `TGoogleClient` component to handle actual communication.

The code generator creates 1 descendent of this class in the unit it creates for the service. For the Google discovery service, the class is called `TDiscoveryAPI`. This class contains some class methods that expose metadata about the service: The base URL for the service, what authorization scopes are used, where documentation and icons for this service can be found etc.

This class contains a method called `ServiceCall` which is used by all resources in the API to execute service requests. It will use the client to do the actual HTTP request.

TGoogleResource For each resource exposed by the service, a descendent of this class is generated that has all the methods for that resource, as described in the REST service description. `TGoogleResource` uses an instance of the `TGoogleAPI` class to handle all calls to the service. For the Discovery API, there is 1 resource class `TApisResource`.

TGoogleBaseObject for each data type used in the API, a descendent of this class is used: it is a descendent of `TBaseObject` and handles loading from and saving to JSON. For the Discovery service, there are 2 main datatypes that descend from this base class: `TDirectoryList` for the list of services, and `TRestDescription` which describes a REST api. The latter is used to create a pascal unit.

The JSON serialization mechanism works with an object factory: when it needs to create an object of a certain type (the 'kind' in Google parlance), it looks in the factory to see which class must actually be created. The `TGoogleAPI` object will register all the classes it uses in the factory. It is possible to override the classes in the factory, so that when a class is requested a descendent of the class can be returned.

For instance, the `TDirectoryList` is registered with name `discovery#directoryList`.
To have the API return a `TMyDirectoryList` item whenever it requires a `discovery#directoryList` instance, the `TMyDirectoryList` can simply be registered with the same name:

```
 TMyDirectoryList.RegisterObject;
```

The class must override the `RestKindName` class method for this to work correctly:

```
Function TMyDirectoryList.RestKindName : string;

begin
  Result:='discovery#directoryList';
end;
```

The advantage is that this mechanism allows the programmer to create descendents of the classes in an API which have custom behaviour and properties, rather than modify the service description unit. When data arrives from the server, his private classes will be instantiates instead of the declared stock classes. When the service changes, the API converter can then regenerate the service description unit, and not all customizations are lost.

If the object factory does not contain a definition for a certain class, the serializer will always fall back to instantiating an instance of the declared property type.

5 Using the generated APIs

Armed with these base classes, it is time to start using them to create an actual program.

The Google discovery demo program uses the `googlediscovery` unit to create a small GUI program. Using this GUI program the following actions can be performed:

- View and search in available services.
- Open the documentation of a service in a browser.
- View the JSON rest description of the service.
- Generate a unit based on the REST description of a service.

The main form of the application simply shows a list of services, with a button to (re)fetch the list, and a textbox to filter the list.

The `OnCreate` event is used to set up everything:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  // set up communication.
  FClient:=TGoogleClient.Create(Self);
  FClient.WebClient:=TSynapseWebClient.Create(Self);
  // Register all classes so they can be streamed.
  TDiscoveryAPI.RegisterAPIResources;
  // create the API and hook it up to the Google client.
  FDiscoveryAPI:=TDiscoveryAPI.Create(Self);
  FDiscoveryAPI.GoogleClient:=FClient;
  // The code generator uses its own objects.
  TDiscoveryJSONToPas.RegisterAllObjects;
  UpdateCaption;
end;
```

The first 2 lines set up the communication: in the example code, the synapse descendent of the `TFPWebClient` class is used to handle transport. To be able to stream all objects in the discovery API, all resources needed for the API are registered. Then an instance of the Discovery API is created, and connected to the client component. The last 2 lines handle initialization of the code generator, and update the caption of the Google API.

In general the above mechanism will be the same for all applications that want to communicate with a Google service API.

The `TApisResource` class represents the resources exposed by the discovery API, and was generated by the code generator as follows:

```
TApisResource = Class(TGoogleResource)
Public
  Class Function ResourceName : String; override;
  Class Function DefaultAPI : TGoogleAPIClass; override;
  Function GetRest(_api: string; version: string) : TRestDescription;
  Function List(AQuery : string  = '') : TDirectoryList;
  Function List(AQuery : TApislistOptions) : TDirectoryList;
end;
```

The first 2 methods are for the API's internal bookkeeping for the factory methods. The interesting methods are `List` and `GetRest`: The latter requires the name and the version of the API, and will return a description of the REST API in the `TRestDescription` instance. These 2 parameters are required and are encoded in path of the URI used to access the resource: this is a feature of the API and reflected in the signature of the methods generated by the API.

As seen in the declaration, the `List` method of this resource comes in 2 forms: One accepts a string, the other a structure of type `TApislistOptions`. This pattern can be seen in all resource classes generated by the code generator, and this is a design choice: The Google REST description document describes for each call what optional parameters the call accepts, usually these parameters serve to filter the returned response. These parameters are passed to the API in the query variables encoded in the URL.

This is always translated by the code generator to 2 calls: rather than creating a method that contains all the parameters in its signature, a record is declared that contains each parameter as a field. For the `List` method, these parameters are described in `TApisListOptions`:

```
  TApisListOptions = Record
    _name : string;
    preferred : boolean;
  end;
```

Since the API can change and to allow for custom queries, whenever there are such optional parameters for a method, the method is generated twice, the second form just accepts a query string which is passed on as-is in the URL (which means it must be URL-encoded). Internally, the method using the record just constructs the query from non-empty fields in the record, and calls the latter method.

So how to use the TApisResource and its methods ?

Each API class has methods to create the resource instances used in the API. For the discovery API, there is only 1 resource, so the number of methods is limited:

```
TDiscoveryAPI = Class(TGoogleAPI)
  //Add create function for resources
  Function CreateApisResource(AOwner : TComponent) : TApisResource;
  Function CreateApisResource : TApisResource;
  //Add default on-demand instances for resources
  Property ApisResource : TApisResource Read GetApisInstance;
end;
```

The `CreateApisResource` call will create an instance of `TApisResource`, and hooks it up to itself, so the resource can execute service calls. Optionally, an owner for the resource can be specified (if none is specified, the API instance is the owner). When the `ApisResource` property is read, it will create an instance of `TApisResource` if need be, and keep it in memory till the API is freed.

The same pattern is reused in all the API classes generated by the API code generator.

In the demo application, when the Refresh button or menu item is chosen, the list of available API's is fetched and displayed. This happens using the `List` method of the `TApisResource`.

```
procedure TMainForm.DoFetch;

begin
  // Free any previous list.
  FreeAndNil(FDirectory);
  // Get the new list using a default ApisResource.
  FDirectory:=FDiscoveryAPI.ApisResource.List();
  ShowDiscovery(MIPreferredOnly.Checked,EFilter.Text);
end;
```

The second line of code gets the list of services. For this, it uses the default `ApisResource` property of the `TAPIDiscovery` instance. This first line cleared any previous list. The result of any API methods needs to be freed by the user. Note that the base classes used in the APIs will always free properties of class or dynamic array type when they are destroyed: the user does not need to do this, but the needs to be aware of it.

The `ShowDiscovery` method shows all services in the list. It has 2 arguments that can be used to filter the list: `PreferredOnly` and a filter on text (the title, name, description and labels are filtered on this text).

Google APIs come in different versions, and one of these version is the preferred version, normally this is the version that should be used in new implementations. To cater for this, the demo program has a check menu which can be used to show only the preferred versions of an API.

The `ShowDiscovery` method just fills a listview with the result of the call:

```
procedure TMainForm.ShowDiscovery(PreferredOnly : Boolean;
                                  FilterOn : String);

Var
  DLI : TDirectoryListitems;
  LI : TListItem;
```

```
begin
  FilterOn:=LowerCase(Filteron);
  LVServices.Items.BeginUpdate;
  try
    LVServices.Items.Clear;
    LVServices.Column[1].Visible:=Not PreferredOnly;
    For DLI in FDirectory.Items do
      if ShowItem(DLI) then
        begin
        LI:=LVServices.Items.Add;
        LI.Caption:=DLI.name;
        LI.Data:=DLI;
        With LI.SubItems,DLI do
          begin
          Add(BoolToStr(preferred,'True','False'));
          Add(id);
          Add(title);
          Add(version);
          Add(description);
          Add(discoveryLink);
          Add(discoveryRestUrl);
          Add(documentationLink);
          Add(icons.x16);
          Add(icons.x32);
          Add(DoComma(labels));
          end;
        end;
    UpdateCaption;
  finally
    LVServices.Items.EndUpdate;
  end;
end;
```

The call to `ShowItem` decides whether an item must be shown or not, depending on the options to `ShowDiscovery`.

The rest of the program uses the properties of the `TDirectoryListitems`: the instances are stored in the `Data` property of the listitems in the listview. For instance, to show the documentation of a particular API, the following code is used:

```
function TMainForm.CurrentAPI: TDirectoryListitems;
begin
  If Assigned(LVServices.Selected)
     and Assigned(LVServices.Selected.Data) then
    Result:=TDirectoryListitems(LVServices.Selected.Data)
  else
    Result:=Nil;
end;

procedure TMainForm.AViewHelpExecute(Sender: TObject);
begin
  OpenURL(CurrentAPI.DocumentationLink);
end;
```

Things can hardly get more simple than this.

To view the JSON description of a service, the following code is used:

```
procedure TMainForm.APreViewRestExecute(Sender: TObject);

Var
  DLI : TDirectoryListitems;

begin
  DLI:=CurrentAPI;
  ViewRestAPI(DLI.Name,DLI.DiscoveryRestUrl);
end;

procedure TMainForm.ViewRestAPI(const AName, AURL: String);
Var
  S : TMemoryStream;

begin
  S:=TMemoryStream.Create;
  try
    if HttpGetBinary(AURL,S) then
      begin
      ViewFile(S,sJSON,'REST discovery for '+AName);
      S:=Nil;
      end;
  finally
    S.Free;
  end;
end;
```

The `HttpGetBinary` call is a part of the Synapse TCP/IP suite, and the `ViewFile` call just shows a secondary form with a syntax highlighter.

The other functionality of the program are simple variations on these calls, the code for it will not be presented here. The program can be seen in action in figure 3 on page 11.

6 Setting up OAuth 2 on Google

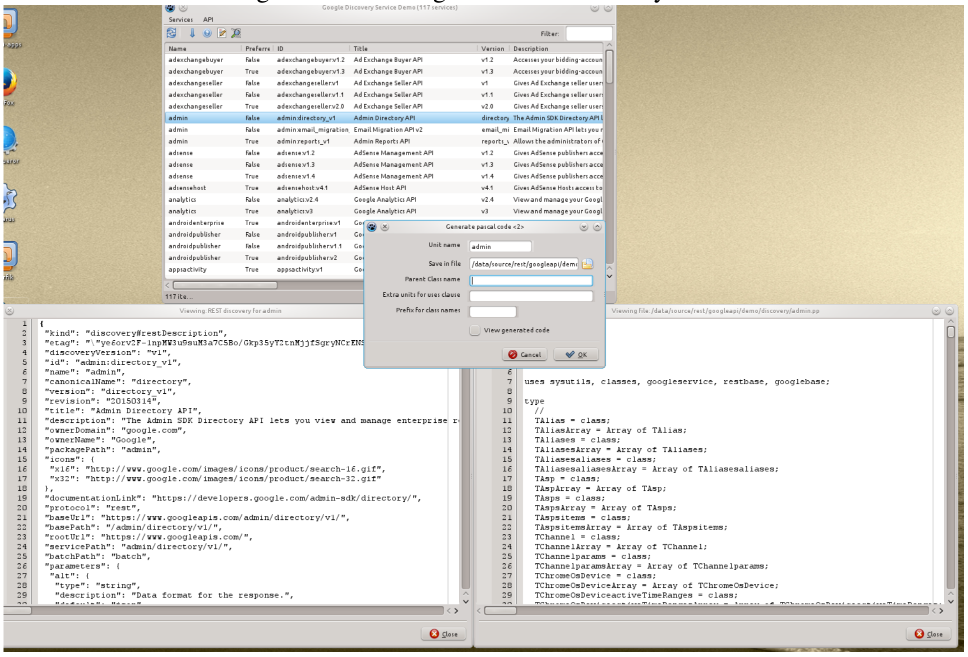
The Google discovery service is publicly available. This means that anyone can call these services, no authentication or authorization is required. Most other services, however, do need authorization to be used. The Google API converter overrides 2 method of `TGoogleAPI`:

```
Class Function APIAuthScopes : TScopeInfoArray;virtual; abstract;
Class Function APINeedsAuth : Boolean ;virtual;
```

The first method, `APIAuthScopes` lists the scopes for which authentication can be requested when an API is used. The second method, `APINeedsAuth`, returns `True` if the API needs authorization. (this is the case when the `APIauthscopes` array is non-empty). The `ServiceCall` method of the `TGoogleAPI` class uses this function to decide whether it must make a signed or unsigned request.

For the Google Discovery service, the `APINeedsAuth` returns `False`. Setting up OAuth 2 for an application requires several steps:

Figure 3: The Google Service Discovery demo



Name	Preferre	ID	Title	Version	Description
adexchangebuyer	False	adexchangebuyer:v1.2	Ad Exchange Buyer API	v1.2	Accesses your bidding-accoun
adexchangebuyer	True	adexchangebuyer:v1.3	Ad Exchange Buyer API	v1.3	Accesses your bidding-accoun
adexchangeseller	False	adexchangeseller:v1	Ad Exchange Seller API	v1	Gives Ad Exchange seller users
adexchangeseller	False	adexchangeseller:v1.1	Ad Exchange Seller API	v1.1	Gives Ad Exchange seller users
adexchangeseller	True	adexchangeseller:v2.0	Ad Exchange Seller API	v2.0	Gives Ad Exchange seller users
admin	False	admin:directory_v1	Admin Directory API	directory	The Admin SDK Directory API l
admin	False	admin:email_migration_	Email Migration API v2	email_mi	Email Migration API lets you r
admin	True	admin:reports_v1	Admin Reports API	reports_v	Allows the administrators of
adsense	False	adsense:v1.2	AdSense Management API	v1.2	Gives AdSense publishers acce
adsense	False	adsense:v1.3	AdSense Management API	v1.3	Gives AdSense publishers acce
adsense	True	adsense:v1.4	AdSense Management API	v1.4	Gives AdSense publishers acce
adsensehost	True	adsensehost:v4.1	AdSense Host API	v4.1	Gives AdSense Hosts access to
analytics	False	analytics:v2.4	Google Analytics API	v2.4	View and manage your Googl
analytics	True	analytics:v3	Google Analytics API	v3	View and manage your Googl
androidenterprise	True	androidenterprise:v1	Go		
androidpublisher	False	androidpublisher:v1	Go		
androidpublisher	False	androidpublisher:v1.1	Go		
androidpublisher	True	androidpublisher:v2	Go		
appsactivity	True	appsactivity:v1	Go		

Generate pascal code <2>

Unit name	admin
Save in file	/data/source/rest/googleapi/demo
Parent Class name	
Extra units for uses clause	
Prefix for class names	

☐ View generated code

Cancel OK

Viewing: REST discovery for admin

```
{
 "kind": "discovery#restDescription",
 "etag": "\"ye6orv2F-1npMW3u9suM3a7C5Bo/Gkp35yY2tnMjjfSgryNCrENS
 "discoveryVersion": "v1",
 "id": "admin:directory_v1",
 "name": "admin",
 "canonicalName": "directory",
 "version": "directory_v1",
 "revision": "20150314",
 "title": "Admin Directory API",
 "description": "The Admin SDK Directory API lets you view and manage enterprise r
 "ownerDomain": "google.com",
 "ownerName": "Google",
 "packagePath": "admin",
 "icons": {
  "x16": "http://www.google.com/images/icons/product/search-16.gif",
  "x32": "http://www.google.com/images/icons/product/search-32.gif"
 },
 "documentationLink": "https://developers.google.com/admin-sdk/directory/",
 "protocol": "rest",
 "baseUrl": "https://www.googleapis.com/admin/directory/v1/",
 "basePath": "/admin/directory/v1/",
 "rootUrl": "https://www.googleapis.com/",
 "servicePath": "admin/directory/v1/",
 "batchPath": "batch",
 "parameters": {
  "alt": {
   "type": "string",
   "description": "Data format for the response.",
```

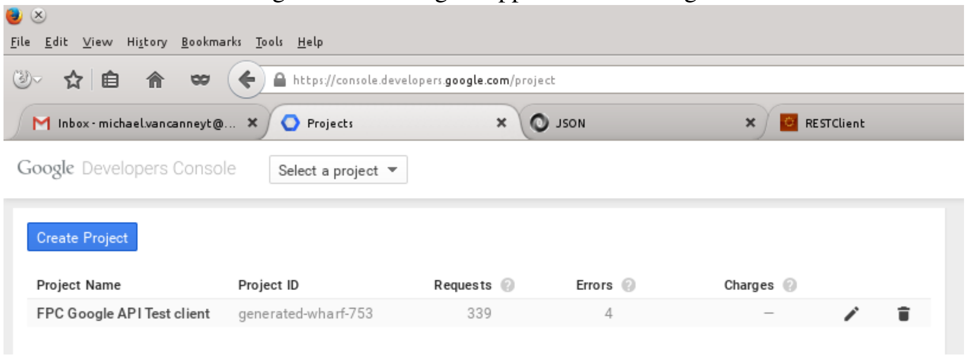
Viewing file: /data/source/rest/googleapi/demo/discovery/admin.pp

```

uses sysutils, classes, googleservice, restbase, googlebase;

type
  //
  TAlias = class;
  TAliasArray = Array of TAlias;
  TAliases = class;
  TAliasesArray = Array of TAliases;
  TAliasesaliases = class;
  TAliasesaliasesArray = Array of TAliasesaliases;
  TAsp = class;
  TAspArray = Array of TAsp;
  TAsps = class;
  TAspsArray = Array of TAsps;
  TAspsitems = class;
  TAspsitemsArray = Array of TAspsitems;
  TChannel = class;
  TChannelArray = Array of TChannel;
  TChannelparams = class;
  TChannelparamsArray = Array of TChannelparams;
  TChromeOsDevice = class;
  TChromeOsDeviceArray = Array of TChromeOsDevice;
  TChromeOsDeviceactiveTimeRanges = class;
```

Figure 4: Defining an application in Google



File
Edit
View
History
Bookmarks
Tools
Help
https://console.developers.google.com/project
Inbox - michael.vancanneyt@...
Projects
JSON
RESTClient
Google Developers Console
Select a project
Create Project

Project Name	Project ID	Requests	Errors	Charges
FPC Google API Test client	generated-wharf-753	339	4	–

1. The developer needs a Google account.

2. In the Google developer console, the application must be registered. The Google developer console is available at:

   `https://console.developers.google.com/project`

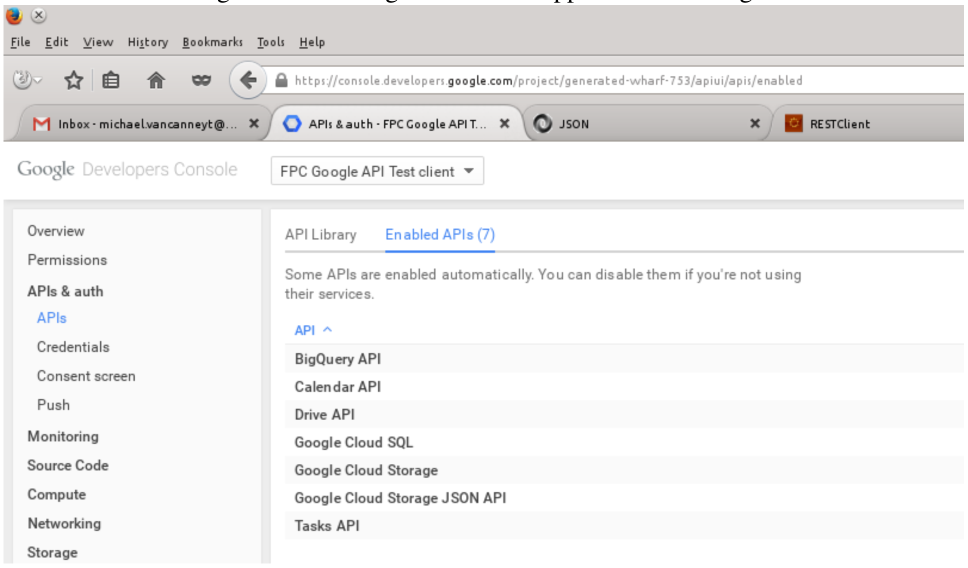
   The console currently looks as figure 4 on page 11, where one application is defined.

3. When the application is defined, access to Google APIs must be set up. This is done under the `API's and Auth` - APIs section of the console. The programmer needs to declare to Google which APIs his application will use.

   If this is not done correctly, then any calls to an API will fail, even if the authorized user has given consent that the application may do so.

   When authorizing, the Google authentication service will present a consent screen, which will list all actions that an application is allowed to perform on behalf of the

Figure 5: Selecting APIs for the application in Google


File Edit View History Bookmarks Tools Help
https://console.developers.google.com/project/generated-wharf-753/apiui/apis/enabled
Inbox - michael.vancanneyt@...
APIs & auth - FPC Google API T...
JSON
RESTClient
Google Developers Console
FPC Google API Test client
Overview
Permissions
APIs & auth
APIs
Credentials
Consent screen
Push
Monitoring
Source Code
Compute
Networking
Storage
API Library
Enabled APIs (7)

Some APIs are enabled automatically. You can disable them if you're not using their services.

API
BigQuery API
Calendar API
Drive API
Google Cloud SQL
Google Cloud Storage
Google Cloud Storage JSON API
Tasks API

user. The list of actions reflects what APIs the programmer has selected here. The API selection is shown in figure 5 on page 12.

4. As a last step, a pair of keys must be generated for the application. This is done under the `API's and Auth` - Credentials section of the console. One key is a unique identifier for the application (the client ID), the other is a secret key (a password). These are used when asking for user consent: they are sent to Google authorization server when the application needs permission to fetch data from a user.

   Through the OAuth 2 protocol flow, the application will then end up with a new token (the access token) that it will use to ask permissions to acces data on behalf of the user.

   The credentials configuration is shown in figure 6 on page 13. Depending on what kind of application you are developing, different settings must be used.

   For desktop applications, the 'native client application' type must be chosen, and `urn:ietf:wg:oauth:2.0:oob` or `http://localhost` must be chosen as the Redirect URI.

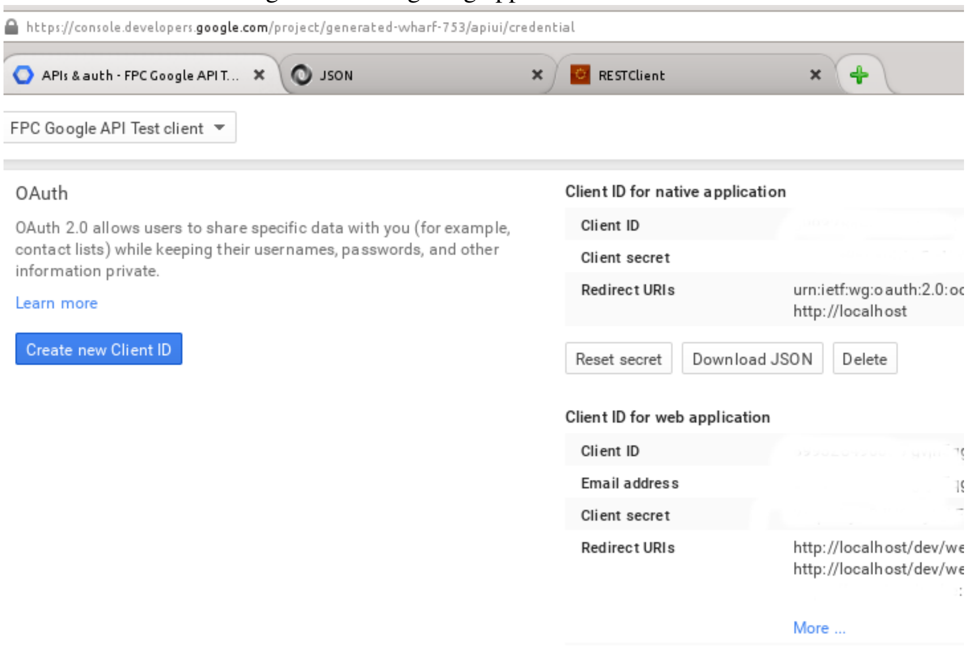
   In the Google Developer Console a new Client ID and Client Secret can then be generated.

The client ID and Secret must be used in the code of your application - preferably scrambled somehow. Google offers a JSON download of this data, the contents of this file must be kept secret. If this data becomes public, then another programmer can impersonate your application and start downloading or, worse, wreak havoc on the user's data (and you will get the blame for it).

7 Setting up OAuth 2 in the application

Once the server part was set up, the application part can be configured. To demonstrate this, the Google Calendar demo program is used. It makes use of the googlecalendar unit, generated with the API code generator. The calendar demo shows a list of calendars of the

Figure 6: Configuring application credentials



https://console.developers.google.com/project/generated-wharf-753/apiui/credential

APIs & auth - FPC Google API T... | JSON | RESTClient

FPC Google API Test client

OAuth

OAuth 2.0 allows users to share specific data with you (for example, contact lists) while keeping their usernames, passwords, and other information private.

Learn more

Create new Client ID

Client ID for native application

Client ID	[illegible]
Client secret	[illegible]
Redirect URIs	urn:ietf:wg:oauth:2.0:oo http://localhost

Reset secret | Download JSON | Delete

Client ID for web application

Client ID	[illegible]
Email address	[illegible]
Client secret	[illegible]
Redirect URIs	http://localhost/dev/we http://localhost/dev/we

More ...

user, and allows to view the items in the calendar. The list of calendars and events in the calendar are simple listboxes, to make it simpler to understand.

The application starts in much the same way as the discovery service demo:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  // Set up Google client.
  FClient:=TGoogleClient.Create(Self);
  FClient.WebClient:=TSynapseWebClient.Create(Self);
  FClient.WebClient.RequestSigner:=FClient.AuthHandler;
  FClient.AuthHandler.WebClient:=FClient.WebClient;
  FClient.AuthHandler.Config.AccessType:=atOffLine;
  // We want to enter a code.
  FClient.OnUserConsent:=@DoUserConsent;
  // Create a calendar API and connect it to the client.
  FCalendarAPI:=TCalendarAPI.Create(Self);
  FCalendarAPI.GoogleClient:=FClient;
  // Register calendar resources.
  TCalendarAPI.RegisterAPIResources;
  LoadAuthConfig;
end;
```

The code differs only in the setup of the authentication handler: The webclient's `RequestSigner` property is set to the Google Client `AuthHandler` property. When the webclient needs to sign a request (Basically, it adds an authorization handler), it checks the AuthHandler. This will check if an access token is available. For the authentication handler to be able to do its work, 2 properties must be set:

```
FClient.AuthHandler.Config.AccessType:=atOffLine;
FClient.OnUserConsent:=@DoUserConsent;
```

The first line tells the authentication handler class that the application is an offline application. The second line registers an event handler: for an offline application, this event handler is called if user consent is needed. The last line of the OnCreate event handler loads the configuration from an ini file; we'll get back to this.

Now, when the application needs to do a service call to a Google service, the authentication handler will check if it has an access token. If it does not, and it does not have a refresh token (with which it can ask an access token from Google), it will call the `DoUserConsent` event handler.

When the event handler is called, several things must be done:

1. The event handler gets an URL to a Google authentication server, which must be displayed in a browser.
2. Google authentication server will then ask the user to log in (if he or she is not yet logged in) and will ask permission for your application to access the calendar.
3. The browser will then display an authorization code, which must be entered by the user in the program.
4. Once the user has entered the authorization code, the event handler may return, passing the code back to the authorization handling component.

In the calendar demo application, there is a groupbox with an edit control and 2 buttons (OK and cancel). In this edit control the user can enter the authorization code returned by Google. Initially, this groupbox is invisible.

The code for this event handler looks as follows:

```
Procedure TMainForm.DoUserConsent(Const AURL: String;
                                  Out AAuthCode: String);

begin
  // Make the code entry visible.
  GBAccess.Visible:=True;
  EAccessCode.Text:='<enter code here>';
  FAccessState:=acsWaiting;
  // Show the URL in the browser
  OpenUrl(AURL);
  // Wait for the user to enter the code
  While (FAccessState=acsWaiting) do
    Application.ProcessMessages;
  // If the user has entered the code, return it
  if FAccessState=acsOK then
    AAuthCode:=EAccessCode.Text;
  GBAccess.Visible:=False;
end;
```

The code starts by showing the button and edit control. It then repeatedly runs the application message loop to wait for the user to enter the authorization code. The OK and Cancel buttons simply set a state, which is picked up in the loop:

```
procedure TMainForm.BSetAccessClick(Sender: TObject);
begin
  FAccessState:=acsOK;
end;
```

```
procedure TMainForm.BCancelClick(Sender: TObject);
begin
  FAccessState:=acsCancel;
end;
```

Obviously, it would be annoying for the user to have to login and enter this code each time the application is used. Fortunately, this is not necessary: the application saves the tokens it received after the first calls to the server.

```
procedure TMainForm.SaveRefreshToken;

Var
  ini:TIniFile;

begin
  // We save the refresh token for later use.
  With FClient.AuthHandler.Session do
  if RefreshToken<>'' then
    begin
    ini:=TIniFile.Create('Google.ini');
    try
      With ini do
        begin
        WriteString('Session','RefreshToken',RefreshToken);
        WriteString('Session','AccessToken',AccessToken);
        WriteString('Session','TokenType',AuthTokenType);
        WriteDateTime('Session','AuthExpires',AuthExpires);
        WriteInteger('Session','AuthPeriod',AuthExpiryPeriod);
        end;
    finally
      Ini.Free;
    end;
    end;
end;
```

The above code shows what properties of the user session must be saved. For safety reasons, it is better not to save the AccessToken in a desktop application. But for instance in a CGI web application, the access token can better be saved, to avoid having to exchange the refresh token for an access token each time the Google service needs to be called.

In the `OnCreate` handler of the application, an attempt is made to load these tokens from the ini file, together with the client ID and secret:

```
procedure TMainForm.LoadAuthConfig;

Var
  ini:TIniFile;

begin
  ini:=TIniFile.Create('Google.ini');
  try
    With FClient.AuthHandler.Config,Ini do
      begin
      // Registered application needs calendar scope
      ClientID:=ReadString('Credentials','ClientID','');
      ClientSecret:=ReadString('Credentials','ClientSecret','');
      AuthScope:=ReadString('Credentials','Scope',
                          'https://www.googleapis.com/auth/calendar');
      // We are offline.
      RedirectUri:='urn:ietf:wg:oauth:2.0:oob';
      end;
    With FClient.AuthHandler.Session,Ini do
      begin
      // Session data
      RefreshToken:=ReadString('Session','RefreshToken','');
      AccessToken:=ReadString('Session','AccesToken','');
      AuthTokenType:=ReadString('Session','TokenType','');
      AuthExpires:=ReadDateTime('Session','AuthExpires',0);
      AuthExpiryPeriod:=ReadInteger('Session','AuthPeriod',0);
      end;
  finally
    Ini.Free;
  end;
end;
```

If the application is run the first time, the session tokens are empty, and the user consent event will be called. Later runs of the application will load the refresh and access token from the ini file, and the user no longer needs to log in or enter an access code. Unless he revokes the right of the application to work on his/her behalf.

Needless to say, these tokens should be stored in a safe way. The above method of loading the client secret and ID was implemented done for convenience, but is not recommended for use in real life applications: the Client ID and Client Secret must be secret, and an .ini file is not suitable in such case.

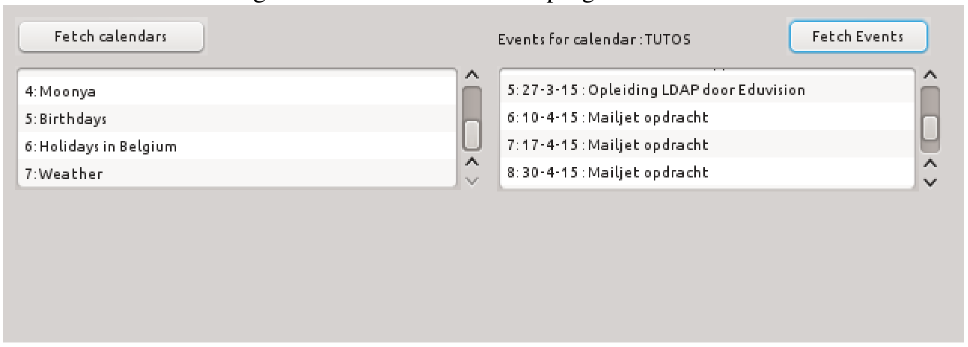
The code to fetch the events and calendar list is in fact very similar to the code used in the discovery demo, but we'll show it anyway:

```
procedure TMainForm.BFetchCalendarsClick(Sender: TObject);

var
  Entry: TCalendarListEntry;
  Resource : TCalendarListResource;
  EN : String;

  i:integer;
begin
  LBCalendars.Items.Clear;
  FreeAndNil(CalendarList);
  Resource:=Nil;
  try
    Resource:=FCalendarAPI.CreateCalendarListResource;
    CalendarList:=Resource.list('');
    SaveRefreshToken;
    if assigned(calendarList) then
      for i:= 0 to Length(calendarList.items)-1 do
        begin
        Entry:=calendarList.items[i];
        EN:=Entry.Summary;
```

Figure 7: The calendar demo program in action



4: Moonya
5: Birthdays
6: Holidays in Belgium
7: Weather

5: 27-3-15 : Opleiding LDAP door Eduvision
6: 10-4-15 : Mailjet opdracht
7: 17-4-15 : Mailjet opdracht
8: 30-4-15 : Mailjet opdracht

```
        if EN='' then
          EN:=Entry.id+' ('+Entry.description+')';
        LBCalendars.Items.AddObject(IntToStr(i)+': '+EN,Entry);
        end;
     BFetchEvents.Enabled:=LBCalendars.Items.Count>0;
  finally
    FreeAndNil(Resource);
  end;
end;
```

The code differs on 2 accounts:

- The code does not make use of the default calendarlist resource of the Calendar API. Instead it uses the `CreateCalendarListResource` to create a private instance, which is freed at the end of the routine.
- After the call to list the calendars is made, the `SaveRefreshToken` routine is called explicitly to save the refresh token. This can also be handled in different ways: The `TOAuth2Handler` has an event called `OnAuthSessionChange` which is called whenever the session information changes, and the `Store` property which can be set to a component that handles storing and retrieving of session and configuration variables.

The result of all this code can be seen in figure 7 on page 17:

8 Conclusion

REST is an important architecture in web APIs. When done right, it makes developing a client library very easy. Google offers a complete REST api for its services, and Lazarus and Free Pascal contain now a complete client-library implementation for the Google APIs, as has been demonstrated here. This client library is somewhat young and certainly subject to improvement, but is certainly usable. Not all aspects have been treated here: much can be said about authentication, getting information about the logged in user, saving sesion state information. A similar library for the Microsoft Office365 environment is in the works. All these topics will be treated in a future contribution.

The author is indebted to Ludo Brands and the late Reinier Olieslagers for their ideas on how to handle serialization and implementation of the OAuth 2 protocol.