TMS FNC Cloud Pack
DEVELOPERS GUIDE

January 2022

Web: http://www.tmssoftware.com
Email: info@tmssoftware.com

Index

Availability

Supported frameworks and platforms
- VCL Win32/Win64
- FMX Win32/Win64, macOS, iOS, Android, Linux
- LCL Win32/Win64, macOS, iOS, Android, numerous Linux variants including Raspbian
- WEB: Chrome, Edge, Firefox, …

Supported IDE's
- Delphi XE7 and C++ Builder XE7 or newer releases
- Lazarus 1.4.4 with FPC 2.6.4 or newer official releases
- TMS WEB Core for Visual Studio Code 1.3 or newer releases

Important Notice: TMS FNC Cloud Pack requires TMS FNC Core (separately available at the My Products page)

Online references

TMS software website:
http://www.tmssoftware.com

TMS FNC Cloud Pack page:
http://www.tmssoftware.com/site/tmsfnccloudpack.asp

Description

TMS FNC Cloud Pack is a set of components to access and use existing cloud services via REST and that offer this in a combination of 2 parts:

- Core
- Ready-to-use service implementation

This documentation focuses on the core implementation that can be used as a guide to implement any custom service. The "Ready-to-use service implementation" part is explained in a separate documentation (TMS FNC Cloud Pack Services (PDF)).

Getting Started

After installing the TMS FNC Cloud Pack, you'll have access to the unit *TMSFNCCloudOAuth.pas. Depending on the framework, you'll need to prefix the unit with "FMX.", "VCL.", "WEBLib." or "LCL". The samples and explanation in this document is based on FMX, but actually applies to all supported frameworks. There are 2 classes that can be used for implementing a custom service:

- TTMSFNCCloudOAuth (base class for accessing and implementing REST services based on OAuth 2.0 authentication)
- TTMSFNCSimpleCloudOAuth (base class for accessing and implementing REST services without authentication)

Depending on the service, you can inherit from one of the 2 classes. The next chapters explain the difference between the 2 classes and the way to implement/use them.

TTMSFNCCloudOAuth

When inheriting from TTMSFNCCloudOAuth, you'll be presented with abstract virtual methods that need to be implemented in order to handle the OAuth 2.0 authentication flow. Below are the methods that need to be implemented.

```
[dcc32 Warning] UDemo.pas(34): W1020 Constructing instance of 'TTMSFNCCloudSample' containing
abstract method 'TTMSFNCCustomCloudOAuth.GetAuthenticationURL'
[dcc32 Warning] UDemo.pas(34): W1020 Constructing instance of 'TTMSFNCCloudSample' containing
abstract method 'TTMSFNCCustomCloudOAuth.RetrieveAccessToken'
[dcc32 Warning] UDemo.pas(34): W1020 Constructing instance of 'TTMSFNCCloudSample' containing
abstract method 'TTMSFNCCustomCloudOAuth.TestTokens'
[dcc32 Warning] UDemo.pas(34): W1020 Constructing instance of 'TTMSFNCCloudSample' containing
abstract method 'TTMSFNCCustomCloudOAuth.GetTestTokensResult'
```

The class definition looks like the following code snippet.

```
TTMSFNCCloudSample = class(TTMSFNCCloudOAuth)
protected
  function GetAuthenticationURL: string; override;
  procedure RetrieveAccessToken; override;
public
  procedure TestTokens(const ATestTokensRequestResultEvent:
TTMSFNCCloudBaseRequestResultEvent = nil); override;
  function GetTestTokensResult(const ARequestResult:
TTMSFNCCloudBaseRequestResult): Boolean; override;
end;
```

The OAuth 2.0 authentication flow is a three-step process:

1) Construct the authentication URL with client-id, secret and callback URL
2) Use the authentication token to convert it to an access token (and optional refresh token)
3) Test the access token against the service and return true when the access token is valid

Applying this on our service implementation:

```
{ TTMSFNCCloudSample }

function TTMSFNCCloudSample.GetAuthenticationURL: string;
var
  url: string;
begin
  url := '?redirect_uri=' +
TTMSFNCUtils.URLEncode(Authentication.CallBackURL)
    + '&response_type=code'
    + '&client_id=' + Authentication.ClientID;

  url := 'https://myservice.com/o/oauth2/auth' + url;

  Result := url;
end;
```

```
function TTMSFNCCloudSample.GetTestTokensResult(
  const ARequestResult: TTMSFNCCloudBaseRequestResult): Boolean;
var
  o: TJSONValue;
  s: string;
begin
  Result := False;
  s := ARequestResult.ResultString;
  if s <> '' then
  begin
    o := TTMSFNCUtils.ParseJSON(s);
    if Assigned(o) then
    begin
      Result := not Assigned(TTMSFNCUtils.GetJSONValue(o, 'error'));
      o.Free;
    end;
  end;
end;

procedure TTMSFNCCloudSample.RetrieveAccessToken;
begin
  Request.Clear;
  Request.Name := 'RETRIEVE ACCESS TOKEN';
  Request.Host := 'https://myservice.com';
  Request.Path := '/o/oauth2/token';
  Request.Query :=  'client_id=' + Authentication.ClientID
  + '&client_secret=' + Authentication.Secret
  + '&redirect_uri=' + Authentication.CallBackURL
  + '&code=' + Authentication.AuthenticationToken
  + '&grant_type=authorization_code';
  Request.Method := rmPOST;
  ExecuteRequest(DoRetrieveAccessToken);
end;

procedure TTMSFNCCloudSample.TestTokens(
  const ATestTokensRequestResultEvent: TTMSFNCCloudBaseRequestResultEvent);
begin
  Request.Clear;
  Request.Name := 'TEST TOKENS';
  Request.Host := 'https://www.myservice.com';
  Request.Path := '/oauth2/v1/tokeninfo';
  Request.Query := 'access_token=' + Authentication.AccessToken;
  Request.Method := rmGET;
  ExecuteRequest(ATestTokensRequestResultEvent);
end;
```

After these steps, the service is then in the connected state. To start the process of connecting a service, call the Connect method.

```
procedure TMyCloudServiceForm.FormCreate(Sender: TObject);
var
  c: TTMSFNCCloudSample;
begin
  c := TTMSFNCCloudSample.Create;
  c.Authentication.ClientID := 'My Client ID';
  c.Authentication.Secret := 'My Secret';
  c.Authentication.CallBackURL := 'http://localhost:8000';
  c.OnConnected := DoConnected;
  c.Connect;
end;

procedure TMyCloudServiceForm.DoConnected(Sender: TObject);
begin
  TTMSFNCUtils.Log('Connected !');
end;
```

The OnConnected event is triggered as soon as the service successfully connects via the three-step authentication process. The tokens that are retrieved are automatically saved, and the next time the application runs, the tokens are loaded and the connection process automatically handles this.

After the service implementation is done, and successfully connected, the requests can be made to retrieve information from the service.

TTMSFNCSimpleCloudOAuth

The TTMSFNCSimpleCloudOAuth is a class that can be used in the same way as the TTMSFNCCloudOAuth class but excluding the authentication flow process. You can directly start writing requests to the service you wish to implement. This can, for example, be a file download, or JSON data retrieval, that doesn't require authentication. Below is a sample that demonstrates this.

```
procedure TTMSFNCCloudSample.DoRetrieveTestDownloadResult(
  const ARequestResult: TTMSFNCCloudBaseRequestResult);
begin
  TTMSFNCUtils.Log('File Downloaded !');
end;

procedure TTMSFNCCloudSample.TestDownload;
begin
  Request.ClearHeaders;
  Request.Host := 'http://www.tmssoftware.net';
  Request.Name := 'TEST FILE DOWNLOAD';
  Request.Path := '/public/test.zip';
  Request.PostData := '';
  Request.Method := rmGET;
  Request.ResultType := rrtFile;
  Request.ResultFile :=
TTMSFNCUtils.AddBackslash(TTMSFNCUtils.GetDocumentsPath) + 'test.zip';

  ExecuteRequest(DoRetrieveTestDownloadResult);
end;
```

Supported REST methods

The TMS FNC Cloud Pack core implementation supports the following REST methods:

- GET
- POST
- PUT
- PATCH
- UPDATE
- DELETE

Additionally, the PUT and POST methods data can be constructed using a post data builder. This can be combined with multi-part form data if necessary. The sample below is demonstrating a service that requires multi-part form data for uploading a file.

```
PostDataBuilder.Clear;
PostDataBuilder.AddFormData('userfile', '', True,
ExtractFileName(ARequestResult.DataString), 'application/octet-stream');
s := PostDataBuilder.Build;

Request.Clear;
Request.Name := 'UPLOAD FILE';
Request.Host := Service.BaseURL;
Request.Path := '/upload' + FBasePath + id + '?access_token=' +
Authentication.AccessToken;
```

```
Request.Method := rmPUTMULTIPART;
Request.PostData := s;
Request.UploadFile := ARequestResult.DataUpload;
ExecuteRequest(DoRequestUploadFile);
```

Asynchronous

All requests are handled asynchronously, we wanted to make sure that each action, whether it's retrieving a list of files, or uploading/download a file, is running seamless and doesn't interfere with the main thread. This always makes your application responsive and allows you to perform other tasks while waiting for the download/upload to finish, or the list of files to be loaded.

This also gives the advantage that you can start multiple requests at once, and not having to wait until a specific request has finished. Of course, some requests require additional data that has been retrieved via another request, but we have made the request execution and data retrieval as flexible as possible. Below is a sample that shows the way the request is being made, and how the result is captured.

```
procedure TRequestForm.DoRequestExecute(const ARequestResult:
TTMSFNCCloudBaseRequestResult);
begin
  //File Download Finished
end;

procedure TRequestForm.DownloadFile;
begin
  c :=SimpleCloudOAuthInstance;
  try
    c.Request.Clear;
    c.Request.Host := 'http://myhost.com';
    c.Request.Path := '/download';
    c.Request.Query := 'fileid=123';
    c.Request.Name := 'Download File';
    c.Request.Method := rmGET;
    c.ExecuteRequest(DoRequestExecute);
  finally
  end;
end;
```

There is even a small bonus when targetting FMX, VCL and WEB. You can catch the result via anonymous methods as well!

```
procedure TRequestForm.DownloadFile;
begin
  c := SimpleCloudOAuthInstance;
  try
    c.Request.Clear;
    c.Request.Host := 'http://myhost.com';
    c.Request.Path := '/download';
    c.Request.Query := 'fileid=123';
    c.Request.Name := 'Download File';
    c.Request.Method := rmGET;
    c.ExecuteRequest(
      procedure(const ARequestResult: TTMSFNCCloudBaseRequestResult)
      begin
        //File Download Finished
      end
    );
  finally

  end;
end;
```

Utility methods

When writing your own custom service, you can look at the already existing services, that are described in a separate documentation, on how to construct and handle the request. There is already a lot a functionality built-in, such as handling special Unicode characters, converting file to base64, JSON escaping, URL encoding/decoding and many more. The *TMSFNCUtils.pas unit is available in the TMS FNC Core, which wraps a lot of this functionality.

Sample

To illustrate how to implement your own service, we take the source code from accessing a Google service, such as Google Drive. Google Drive has a common layer in where the authentication flow and access token generation are handled to allow other services from Google implement the API's on top of this layer.

The first step is to generate the authentication URL.

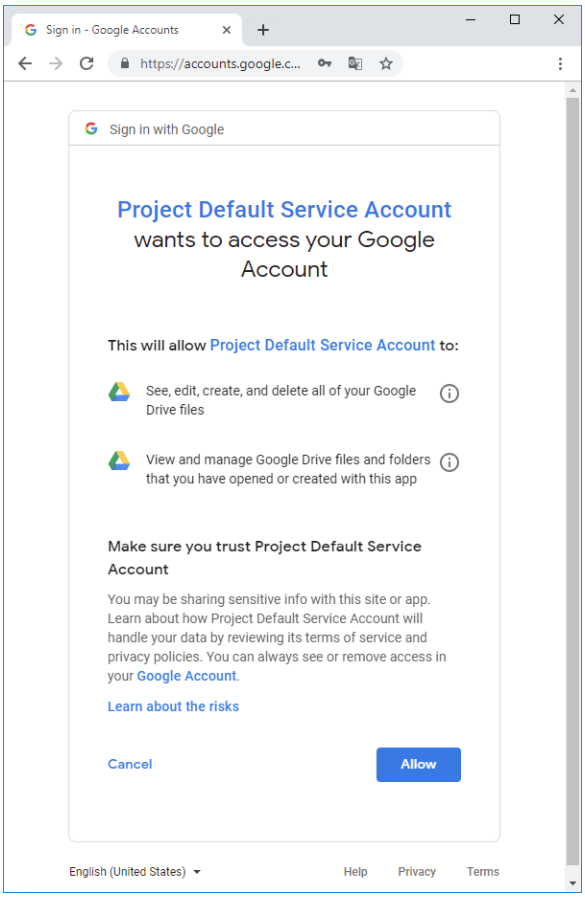
```
function TTMSFNCCustomCloudGoogle.GetAuthenticationURL: string;
begin
  Result := InitializeAuthenticationURL(Self);
end;

function InitializeAuthenticationURL(const ACloudBase: TTMSFNCCloudOAuth):
string;
var
  url, sc: string;
begin
  sc := TTMSFNCCloudOAuthOpen(ACloudBase).GetScopes('+', True);
  url :=
      '?scope=' + sc
    + '&state=profile'
    + '&redirect_uri='+
TTMSFNCUtils.URLEncode(ACloudBase.Authentication.CallBackURL)
```

```
    + '&response_type=code'
    + '&client_id=' + ACloudBase.Authentication.ClientID
    + '&approval_prompt=force'
    + '&access_type=offline'
    + '&hl=en';

  url := 'https://accounts.google.com/o/oauth2/auth' + url;

  Result := url;
end;
```

The URL is a concatenation of The ClientID, CallBackURL, some default parameters and a set of scopes, which is crucial to allow the user to identify which services is accessing which information. For Google Drive, the scopes are added in the constructor:

```
  Scopes.Clear;
  Scopes.Add('https://www.googleapis.com/auth/drive');
  Scopes.Add('https://www.googleapis.com/auth/drive.file');
  Scopes.Add('https://www.googleapis.com/auth/userinfo.profile');
```

When calling the Connect method, and the Access Token is not yet retrieved, or no longer valid, a browser is shown which allows identifying and authorizing the application which is requesting access to your files/folders and account information. Below is a screenshot of the browser and the scopes that are requested via the authorization URL.



After Clicking on the “Allow” button, the Application is redirected back to your application which runs an HTTP Server listening to the Callback URL & Port set via Authentication.CallBackURL. As soon as the HTTP Server catches the OAuth 2.0 redirect callback URL, it parses the URL and generates an authentication token. The next step is to take the authentication token and convert it to an access token:

```
procedure TTMSFNCCustomCloudGoogle.RetrieveAccessToken;
begin
  InitializeRetrieveAccessTokenRequest(Self);
  ExecuteRequest({$IFDEF LCLWEBLIB}@{$ENDIF}DoRetrieveAccessToken);
end;

procedure InitializeRetrieveAccessTokenRequest(const ACloudBase:
TTMSFNCCloudOAuth);
begin
  ACloudBase.Request.Clear;
  ACloudBase.Request.Name := 'RETRIEVE ACCESS TOKEN';
  ACloudBase.Request.Host := 'https://accounts.google.com';
  ACloudBase.Request.Path := '/o/oauth2/token';
  ACloudBase.Request.Query :=  'client_id=' +
ACloudBase.Authentication.ClientID
  + '&client_secret=' + ACloudBase.Authentication.Secret
  + '&redirect_uri=' + ACloudBase.Authentication.CallBackURL
  + '&code=' + ACloudBase.Authentication.AuthenticationToken
  + '&grant_type=authorization_code';
  ACloudBase.Request.Method := rmPOST;
end;
```

In the access token request, you’ll notice that the secret, authentication token, and callback URL are required to identify your application request and make sure the service returns the correct access token. The request is executed and automatically handled by the core layer in TMS FNC Cloud Pack. There is no need to manually parse the access token, unless the service deviates from the default OAuth 2.0 authentication flow.

After retrieving the access token, the service core layer is automatically performing a test to validate the access token and grant access to service API’s. The test needs to be handled by your service implementation. For Google Drive, the test involves calling a simple tokeninfo API endpoint to validate the tokens, but for other services, it could be retrieving the account information, or testing a retrieval of files/folders.

```
procedure TTMSFNCCustomCloudGoogle.TestTokens(const
ATestTokensRequestResultEvent: TTMSFNCCloudBaseRequestResultEvent = nil);
begin
  InitializeTestTokensRequest(Self);
  ExecuteRequest(ATestTokensRequestResultEvent);
end;

procedure InitializeTestTokensRequest(const ACloudBase: TTMSFNCCloudOAuth);
begin
  ACloudBase.Request.Clear;
  ACloudBase.Request.Name := 'TEST TOKENS';
  ACloudBase.Request.Host := 'https://www.googleapis.com';
  ACloudBase.Request.Path := '/oauth2/v1/tokeninfo';
  ACloudBase.Request.Query := 'access_token=' +
ACloudBase.Authentication.AccessToken;
  ACloudBase.Request.Method := rmGET;
end;
```

After executing the test tokens request, the service returns a JSON response which is unique for each service. For Google Drive, this is checking if the returned JSON does not have an error tag.

```
function TTMSFNCCustomCloudGoogle.GetTestTokensResult(
  const ARequestResult: TTMSFNCCloudBaseRequestResult): Boolean;
begin
  Result := InitializeTestTokensResult(Self, ARequestResult)
end;

function InitializeTestTokensResult(const ACloudBase: TTMSFNCCloudOAuth;
const ARequestResult: TTMSFNCCloudBaseRequestResult): Boolean;
var
  o: TJSONValue;
  s: string;
begin
  Result := False;
  s := ARequestResult.ResultString;
  if s <> '' then
  begin
    o := TTMSFNCUtils.ParseJSON(s);
    if Assigned(o) then
    begin
      Result := not Assigned(TTMSFNCUtils.GetJSONValue(o, 'error'));
      o.Free;
    end;
  end;
end;
```

The result is a Boolean (true/false). When the result is a true, the service is successfully authenticated and the application can start accessing various API's.