

Google API in salsa C++ [seconda parte]

Nel precedente articolo: *Google API in salsa C++ [prima parte]*, abbiamo fatto un gran discorrere degli aspetti teorici che stanno dietro ai meccanismi di autenticazione ed autorizzazione impiegati da Google per le sue API pubbliche. Abbiamo anche visto, con un piccolo tutorial, come con la *Google Developers Console*, sia possibile cominciare ad impostare un nostro progetto che ci consenta di giocare con qualche API di Google.

In questo articolo, lasceremo da parte gli aspetti teorici e ci concentreremo invece sul codice che ci darà le basi per interagire con una *Google API*.

Il progetto

L'intero progetto è disponibile su github al mio repository: `cpp0Auth2RestClient`.

Il codice dell'esempio è stato sviluppato all'interno di un'immagine docker; il che rende l'intero progetto portabile ed immediatamente fruibile da chiunque lo voglia testare.

L'immagine docker di partenza è una Debian 10 mantenuta da Microsoft e precisamente la seguente:

```
mcr.microsoft.com/vscode/devcontainers/base:0-debian-10
```

Si tratta di un'immagine estremamente minimale ma ognuno la può estendere secondo le proprie necessità.

Il progetto è stato sviluppato con Microsoft Visual Studio Code, che come molti di voi sapranno è un IDE molto potente che può essere tranquillamente utilizzato anche in ambiente Linux.

L'immagine docker

Vediamo dunque come si presenta l'immagine docker utilizzata dal progetto, di seguito il dockerfile:

```
[..] # To fully customize the contents of this image, use the
following Dockerfile as a base and add the RUN statement from
this file:
```

```
#
https://github.com/microsoft/vscode-dev-containers/blob/v0.112
.0/containers/debian-10-git/.devcontainer/Dockerfile
FROM mcr.microsoft.com/vscode/devcontainers/base:0-debian-10
[..] # Configure apt and install packages
RUN apt-get update \ # Install C++ tools && apt-get -y install
build-essential cmake cppcheck valgrind automake autogen
libtool libcurl4-openssl-dev\ [..] # Build and install
restclient-cpp library && mkdir /3rdpartylibs && cd
/3rdpartylibs \ && git clone
https://github.com/mrtazz/restclient-cpp.git && cd restclient-
cpp \ && git checkout 0.5.2 \ && ./autogen.sh \ && ./configure
\ && make install [..]
```

Come vedete, le modifiche apportate all'immagine *vanilla* dell'immagine della docker non sono poi molte; vediamo le più importanti:

- apt-get -y install libcurl4-openssl-dev

L'installazione di questo pacchetto è necessaria perchè dobbiamo sviluppare un'applicazione C/C++ che utilizza la libreria libcurl.

Ricordiamoci che la nostra applicazione dovrà fare chiamate HTTPS per invocare le REST API di Google Drive e curl è un tool adatto a questo scopo.

- mkdir /3rdpartylibs && cd /3rdpartylibs \
&& git clone
https://github.com/mrtazz/restclient-cpp.git && cd
restclient-cpp \
&& git checkout 0.5.2 \

```
&& ./autogen.sh \  
&& ./configure \  
&& make install
```

Questo blocco, invece, è necessario per clonare il repository di `restclient-cpp` e per effettuare la build e l'install del tag 0.5.2 della libreria stessa.

Questa libreria ci torna comoda perchè non vogliamo usare `libcurl` direttamente.

`libcurl` è una libreria potentissima, ma anche piuttosto ostica e verbosa; meglio non sacrificare il nostro tempo prezioso: affidiamoci tranquillamente ad un layer che ne astrae la complessità.

SETUP & AVVIO

Bene, a questo punto dovremmo essere pronti; una volta clonato il repository ed avviato Visual Studio Code, potrete aprire la directory del progetto e osservare l'IDE che effettuerà tutti i passi necessari:

1. Download dell'immagine della docker
2. Setup dell'immagine della docker secondo le direttive del dockerfile

Una volta completati questi step, potrete avviare la build del progetto che a questo punto sarà pronto per essere eseguito.

Naturalmente, dovremo prima configurare il file di properties: `rc.properties`.

Il file contiene le seguenti entries:

```
code=XXX  
api_key=test0Auth2  
client_id=XXX  
client_secret=XXX  
token_server=https://oauth2.googleapis.com/token  
rest_api_url=https://www.googleapis.com/drive/v3  
rest_api_uri=files
```

Una breve spiegazione di cosa rappresentano:

- `code` – l'*authorization code* associato al nostro account che abbiamo ottenuto da Google per operare su un determinato *scope* di una qualche API.
- `api_key` – il nome dell'applicazione che abbiamo creato dalla *Google Developers Console*.
- `client_id` e `client_secret` – le credenziali della nostra applicazione; generate qui.
- `token_server` – l'indirizzo del server di autorizzazione di Google.
- `rest_api_url` – la URL dell'API che abbiamo scelto (Drive).
- `rest_api_uri` – la URI di una risorsa che vogliamo accedere con l'API.

Lanciando l'eseguibile direttamente dalla shell di Visual Studio Code otterremo un risultato simile a questo:

```
vscode →/workspaces/cpp0Auth2RestClient (master X) $ ./restclient
auth_conn_ object created
resource_conn_ object created
sending refresh token request
refresh token response:
{
  "access_token" : "ya29.a0AfH6SMC3tYooDCe0tNztJAK6v3zS3-j0u4mKCjcVI88NdnAAemqFI3t2F-8nBXZg0Gp0Pv7yJSFwEtYHt7JJcYJP",
  "expires_in" : 3599,
  "scope" : "https://www.googleapis.com/auth/drive.readonly",
  "token_type" : "Bearer"
}
authentication token will expire at:Thu Oct  8 21:01:12 2020
call to REST API was successful, dumping response to file...
```

A questo punto, possiamo andare a sbirciare dentro al file `GET_response` che, se tutto è andato per il verso giusto, conterrà la response per l'API: `files`.

```
{ "files" : [ { "id" : "1IEl7r_IQ0YQK09CsA5jxZjdX5AVPWkZP",
"kind" : "drive#file", "mimeType" : "image/png", "name" :
"Istantanea schermo 2019-11-22 (14.50.55).png" }, { "id" :
"1IdbH6qIr8ublzry3gt-yd-RkGEqothZE", "kind" : "drive#file",
"mimeType" : "application/... presentationml.presentation",
"name" : "20 Docker Networking.pptx" }, { "id" : "1a3Ci9L8GAP-
YbTH0Q_yS0FsBkzERXyZc", "kind" : "drive#file", "mimeType" :
"application/... presentationml.presentation", "name" : "18
```

```

    Docker Java & Maven.pptx" }, { "id" :
    "lsR7gzsG0tX_NwoicCUA0V0XN1_gA4jMh", "kind" : "drive#file",
    "mimeType" : "application/... presentationml.presentation",
    "name" : "19 Docker Miscellanea.pptx" }, ... ],
    "incompleteSearch" : false, "kind" : "drive#fileList",
    "nextPageToken" :
    "~!!~AI9FV7QdvBt05Rsbj3oH4yVf01Guew7HV00gBoXN58w7Y ... "
}

```

Se il risultato è troppo grande, Google paginerà la response e quindi occorrerà chiamare l'API con l'opportuno nextPageToken fornito nella pagina corrente.

In questa sede però, non siamo interessati ad approfondire le API di Google Drive; ci interessano i meccanismi base con cui andiamo ad interagire con una generica REST API.

analisi del PROGRAMMA

Il programma è stato realizzato con un fine divulgativo e didattico; per questo motivo e per rendere la lettura più agevole, il codice applicativo è interamente disponibile nel file main.cpp .

Stiamo parlando di circa 300 righe di codice C++ (standard 14).

librerie di terze parti utilizzate dal programma

1. libcurl per il trasporto su HTTPS dell'intera conversazione con Google.

Gli headers di libcurl non sono importati direttamente dal nostro main.cpp.

2. restclient-cpp come interfaccia di alto livello per libcurl.

Gli headers della libreria, importati dal nostro programma, sono:

```
#include <restclient-cpp/connection.h>
```

```
#include <restclient-cpp/restclient.h>
```

Questi header importano l'header di libcurl:

```
#include <curl/curl.h>
```

3. jsoncpp per il parsing da C++ delle responses in formato json.

L'header della libreria, importato dal nostro programma, è:

```
#include "json/json.h"
```

Nel nostro programma compiliamo direttamente questa libreria i cui sorgenti sono stati piazzati nella directory vendor/json.

Aggiungo a nota che questa eccellente libreria consente di effettuare indifferentemente la serializzazione e la deserializzazione di un dato in formato json.

4. Pako2K/utils per la lettura del file di properties.

L'header importato nel main.cpp:

```
#include "properties_file_reader.h"
```

Anche in questo caso, compiliamo direttamente noi il file sorgente della libreria posto nella directory vendor.

il codice

Analizzando il codice, cercheremo di concentreremo l'attenzione sui concetti principali del programma.

Una piccola premessa: nel codice si utilizzano degli oggetti di tipo RestClient::Connection. È bene precisare che questi oggetti, sebbene il nome possa essere fuorviante, non modellano connessioni persistenti, bensì servono per innescare di volta in volta chiamate HTTP verso un determinato servizio.

Molto bene allora, iniziamo.

Nel main() del programma si costruisce e si utilizza un tipo applicativo chiamato, con poca fantasia: rest_client.

Questo tipo va a modellare l'interazione con una generica API

di Google astraendo la complessità derivata dal protocollo OAuth2.

rappresentazione di rest_client

```
/** rest_client * * rest_client provides basic client
functionalities to call a REST APIs * based on HTTPs protocol
and using OAuth2 like authentication. ...
*/
struct rest_client { //ctor rest_client(std::string
&&properties_file_name); //init method int init(); [...] //the
current configuration (code, client_id, etc)
utils::PropertiesFileReader cfg_; //the connection-object for
the OAuth2 handshake(s)
std::unique_ptr<RestClient::Connection> auth_conn_; //the
connection-object for the actual Google API
std::unique_ptr<RestClient::Connection> resource_conn_;
//refresh token std::string refresh_token_; //access token
std::string access_token_; //access token type (Bearer)
std::string access_token_type_; //access token expiring time-
point std::chrono::system_clock::time_point
access_token_expire_tp_;
};
```

Vediamo cosa rappresentano le variabili di istanza di rest_client :

- `cfg_` – la configurazione corrente del nostro programma; permette di accedere al file di properties: `rc.properties`.
- `auth_conn_` – l'oggetto di tipo `RestClient::Connection` che consente di interagire con il servizio di autorizzazione OAuth2 di Google.
Da questo servizio otterremo sia il refresh token che l'access token.
- `resource_conn_` – l'oggetto di tipo `RestClient::Connection` che consente di interagire con il servizio che espone le API a cui siamo interessati (Drive).
- `refresh_token_` – una stringa che contiene il refresh

token ottenuto dal servizio di autorizzazione. Questo codice, una volta ottenuto non cambierà mai più, per questo motivo una volta ottenuto, lo salveremo dentro un file chiamato senza ambiguità: `refresh_token`.

- `access_token_` – una stringa che contiene l'access token ottenuto dal servizio di autorizzazione. Questo codice, a differenza del refresh token ha una vita molto breve, tipicamente 30 minuti o un'ora.

Se qualcuno riuscisse mai ad intercettare uno dei vostri access token, potrebbe operare con esso relativamente per poco tempo.

- `access_token_type_` – una stringa che contiene la tipologia tecnica del formato dell'access token; quasi sicuramente sarà Bearer.
- `access_token_expire_tp_` – rappresenta un punto nel tempo (futuro), al raggiungimento del quale, l'access token che abbiamo scadrà.

Ci serve per capire se, prima di effettuare una chiamata REST alle API, non dobbiamo prima farci dare un nuovo access token.

I tokens

Che cosa sono i tokens?

Sono oggetti che consentono l'autenticazione di una chiamata REST.

Il meccanismo è molto semplice: un token viene inserito come attributo dell'header HTTP, in questo modo ogni singola richiesta può essere autenticata ed autorizzata dal server.

REFRESH token

Otterremo questo token una volta sola *spendendo* l'*authorization code* che abbiamo ottenuto da Google.

Se dovessimo per qualche motivo perdere il refresh token, dovremo nuovamente ottenere da Google l'*authorization code* e riusarlo per generare un nuovo refresh token.

Questo token non è usato direttamente come attributo HTTP delle chiamate REST.

Viene usato, invece, per effettuare il “refresh” periodico dell’access token.

Diamo quindi un’occhiata al codice preposto ad ottenere il refresh token.

```
/** post_confirm_permissions_request
 *
 * Performs an HTTP POST call over authentication connection in
 * order to obtain
 * the refresh token.
 * [...]
 */
int post_confirm_permissions_request() { std::string post_data
= build_confirm_permissions_request(); RestClient::Response
rest_res = auth_conn_->post("", post_data); if(rest_res.code
== 200) { return
process_confirm_permissions_response(rest_res); } else {
std::cerr << "error getting refresh token and authentication
token, http-code:" << rest_res.code << std::endl; std::cerr <<
"http-body:" << rest_res.body << std::endl; return -1; }
} /** build_confirm_permissions_request
 *
 * Builds the POST data used over authentication connection in
 * order to obtain
 * the refresh token.
 * [...]
 */
std::string build_confirm_permissions_request() {
std::ostringstream os; os << "code=" << GET_PROPERTY(cfg_,
"code") << '&' << "client_id=" << GET_PROPERTY(cfg_,
"client_id") << '&' << "client_secret=" << GET_PROPERTY(cfg_,
"client_secret") << '&' << "redirect_uri=http://localhost&" <<
"grant_type=authorization_code"; return os.str();
} /** process_confirm_permissions_response
 *
 * Upon successful POST call over authentication connection,
 * this method parses the response, * retrieves the refresh token
 * and saves it to the refresh_token file.
 * [...]
 */
int process_confirm_permissions_response(const
```

```

RestClient::Response &response) { /* example of response: {
"access_token": "...", "expires_in": 3599, "refresh_token":
"...", "scope":
"https://www.googleapis.com/auth/drive.readonly",
"token_type": "Bearer" } */ Json::Value auth_res;
std::stringstream istr(response.body); istr >> auth_res;
refresh_token_ = auth_res.get("refresh_token", "").asString();
[...] //flush refresh token to "refresh_token" file so it can
be reused in future runs std::ofstream
refresh_token_ofs("refresh_token", std::ofstream::trunc);
refresh_token_ofs << refresh_token_; std::cout << "confirm
permissions response:" << std::endl << auth_res; return 0;
}

```

Questo blocco di tre metodi copre la chiamata POST verso il servizio di autorizzazione per la “conferma delle autorizzazioni” (*Confirm permissions request*).

Questo tipo di richiesta serve nella sostanza a farsi dare il refresh token; che ricordiamolo, non cambierà mai.

Notiamo che nella response ci viene fornito anche un access token.

ACCESS token

L’access token è l’altro tipo di token che possiamo chiedere al servizio di autorizzazione.

Questo token a differenza del refresh token ci serve per autenticare ogni singola chiamata REST.

Vediamo quindi il codice per ottenerlo:

```

/** post_refresh_access_token_request
* * Performs an HTTP POST call over authentication connection
in order to obtain
* an access token.
* [...]
*/
int post_refresh_access_token_request() { std::string
post_data = build_refresh_token_request();
RestClient::Response rest_res = auth_conn_->post("",
post_data); if(rest_res.code == 200) { return
process_refresh_token_response(rest_res); } else { std::cerr

```

```

<< "error refreshing token, http-code:" << rest_res.code <<
std::endl; std::cerr << "http-body:" << rest_res.body <<
std::endl; return -1; }
} /** build_refresh_access_token_request
* * Builds the POST data used over authentication connection
in order to obtain
* the access token.
* [...]
*/
std::string build_refresh_access_token_request() {
std::ostringstream os; os << "client_id=" <<
GET_PROPERTY(cfg_, "client_id") << '&' << "client_secret=" <<
GET_PROPERTY(cfg_, "client_secret") << '&' << "refresh_token="
<< refresh_token_ << '&' << "grant_type=refresh_token"; return
os.str();
} /** process_refresh_access_token_response
* * Upon successful POST call over authentication connection,
this method parses the response,
* retrieves the access token and saves it to the access_token_
instance variable.
* It also computes the time point at which the obtained access
token will expire.
* Such computation is done taking in consideration the
expires_in attribute of the response,
* that is expressed in seconds.
* [...]
*/
int process_refresh_access_token_response(const
RestClient::Response &response) { /* example of response: {
"access_token": "...", "expires_in": 3599, "scope":
"https://www.googleapis.com/auth/drive.readonly",
"token_type": "Bearer" } */ Json::Value auth_res;
std::istream istr(response.body); istr >> auth_res;
access_token_ = auth_res.get("access_token", "").asString();
access_token_type_ = auth_res.get("token_type",
 "").asString(); access_token_expire_tp_ =
std::chrono::system_clock::now() +
std::chrono::duration<int>(auth_res.get("expires_in",
0U).asUInt()); std::cout << "refresh token response:" <<
std::endl << auth_res << std::endl; std::time_t tt =
std::chrono::system_clock::to_time_t(auth_token_expire_tp_);

```

```
std::cout << "access token will expire at:" << ctime(&tt);  
return 0;  
}
```

I tre metodi preposti alla richiesta dell'access token sono del tutto analoghi a quelli per ottenere il refresh token.

Ovviamente, l'access token può essere richiesto solo se si è in possesso del refresh token.

Notiamo inoltre che il codice calcola il *time point* della scadenza dell'access token appena ricevuto.

Questo servirà per capire se ad un dato momento sarà il caso di richiedere un nuovo access token.

INVOCHIAMO la nostra API, finalmente!

Sembra che a questo punto, ottenuti sia il refresh token che l'access token, abbiamo davvero tutti gli ingredienti per poter cominciare ad interagire con l'API a cui eravamo interessati sin dal principio.

Vediamo prima come invocare l'API dal punto di vista dell'utente di `rest_client`.

```
int main()  
{ //we create a rest client rest_client rc("rc.properties");  
//we init it if(rc.init()) { std::cerr << "error init  
rest_client, exiting..." << std::endl; return 1; } //the  
optional query string to apply to the REST API call  
std::string query_string("key="); query_string +=  
GET_PROPERTY(rc.get_cfg(), "api_key"); //the response in json  
format from the REST API Json::Value json_response; //the HTTP  
GET call for the REST API int http_res = rc.get(query_string,  
[&](RestClient::Response &get_res) -> int { std::istream  
istr(get_res.body); istr >> json_response; return 0; });  
if(http_res != 200) { std::cerr << "call to REST API failed,  
exiting..." << std::endl; return 1; } else { std::cout <<  
"call to REST API was successful, dumping response to file..."  
<< std::endl; } //flush REST API response to file named:  
GET_response std::ofstream GET_response_ofs("GET_response",  
std::ofstream::trunc); GET_response_ofs << json_response;  
return 0;  
}
```

Bene, il codice *utente* è tutto sommato abbastanza facile da leggere:

1. si crea l'oggetto `rc` di tipo `rest_client` e gli vengono associate le properties del file: `rc.properties`.
2. si inizializza `rc`.
3. si crea una query string `query_string` con il valore `"key=${api_key}"`.
4. si crea l'oggetto `json_response` di tipo `Json::Value` che conterrà la response della chiamata REST all'API.
5. si invoca l'API mediante una GET HTTP; ci salviamo il codice HTTP di response dentro `http_rese` il body HTTP dentro `json_response`.
6. stampiamo alcuni messaggi informativi e alla fine salviamo la response ricevuta dentro ad un file chiamato: `GET_response`.

la GET HTTP

L'URI finale della GET HTTP dell'API: files di Drive, comprensiva di query string, è fatta così:

```
https://www.googleapis.com/drive/v3/files?key=test0Auth2
```

Naturalmente, la GET dovrà essere arricchita con l'opportuno attributo HTTP a contenere l'access token.

La GET è invocata dall'utente con il codice che segue:

```
{ rest_client rc("rc.properties"); ... //the response in json
format from the REST API Json::Value json_response; //the HTTP
GET call for the REST API int http_res = rc.get(query_string,
[&](RestClient::Response &get_res) -> int { std::stringstream
istr(get_res.body); istr >> json_response; return 0; }); //200
means success if(http_res != 200) { std::cerr << "call to REST
API failed" << std::endl; return 1; } else { std::cout <<
"call to REST API was successful" << std::endl; } ... }
```

Come potete vedere, la GET HTTP è realizzata dal metodo `get()` di `rest_client`.

Il codice è realizzato in modo tale che l'utente non deve preoccuparsi dei dettagli di come deve avvenire

l'autenticazione OAuth2.

Notiamo inoltre, che la `get()` accetta una lambda come secondo argomento.

Oltre a consentire all'utente l'integrazione con il proprio contesto, la lambda evita al chiamante di dover allocare una variabile di tipo `RestClient::Response`.

Inoltre, in linea di principio, il valore di ritorno della lambda stessa, in un contesto reale, potrebbe essere utilizzato dal metodo `get()` per implementare una qualche logica.

l'implementazione della GET HTTP

Per finire, vediamo quindi l'implementazione del metodo `get()` di `rest_client`:

```
/** get * * Performs an HTTP GET call over resource connection
using * an optional parameter: query_string. * You can pass to
the second parameter: cb, an argument denoting * a lambda that
takes the response * of the HTTP GET call. */
template<typename callback>
int get(const std::string &query_string, callback cb) {
    if(refresh_access_token_if_expired()) { std::cerr << "failed
to get access token, aborting GET call..." << std::endl;
    return -1; } std::string uri("/"); uri += GET_PROPERTY(cfg_,
"rest_api_uri"); if(query_string != "") { uri += '?'; uri +=
query_string; } RestClient::HeaderFields reqHF;
reqHF["Authorization"] = access_token_type_ + ' ' +
access_token_; resource_conn_->SetHeaders(reqHF);
RestClient::Response res; res = resource_conn_->get(uri);
if(res.code != 200) { std::cout << "warning: call result code
was:" << res.code << std::endl; } cb(res); return res.code;
}
```

Il metodo `get()` effettua alcune cose dietro le quinte.

Per prima cosa è in grado di capire se è il caso di richiedere un nuovo access token prima di effettuare la GET.

Questa funzionalità si realizza mediante il metodo privato `refresh_access_token_if_expired()`.

Non voglio annoiarvi proponendovi qui l'implementazione, ma se

siete curiosi la potete ispezionare per conto vostro. Successivamente, se la `get()` è riuscita ad ottenere un access token, provvede a inserire lo stesso nell'header HTTP della GET che sta per effettuare. Questo avviene impostando l'attributo: `Authorization` associandolo al valore dell'access token corrente. A questo punto la GET è pronta per essere imbustata attraverso `libcurl`.

conclusioni

Bene, le spiegazioni teoriche della prima parte unite al codice pratico della seconda, dovrebbero avervi dato una piccola infarinatura sull'argomento OAuth2.

Ci siamo resi conto di cosa significa aver a che fare con una REST API che necessita di essere autenticata mediante OAuth2.

Tipicamente, interazioni di questo tipo avvengono con linguaggi di più alto livello rispetto al C++.

Java, C#, Python e altri possiedono sicuramente già pronti, svariati toolset che consentono di sviluppare rapidamente conversazioni HTTP con molti dei protocolli di autenticazione contemporanei.

Come abbiamo visto, in C++ la cosa è certamente realizzabile, ma occorre effettuare ricerche e trovare i vari pezzi che fanno al caso nostro.

Questo è generalmente vero per tutte le problematiche che deve affrontare uno sviluppatore backend.

Non sono un esperto di Go, ma mi piacerebbe poter fare un raffronto in termini di framework necessari e di linee di codice totali che occorrono per realizzare le stesse funzionalità che abbiamo visto in questo articolo per il C++.

Un saluto!

<https://italiancoders.it/google-api-in-salsa-c-seconda-parte/>