MWA Software

OAuth2 Client

Issue 1.0,
23 August 2021

McCallum Whyman Associates Ltd

EMail: info@ mccallumwhyman.com, http://www.mccallumwhyman.com

COPYRIGHT

Disclaimer

Although our best efforts have been made to ensure that the information contained within is up-to-date and accurate, no warranty whatsover is offered as to its correctness and readers are responsible for ensuring through testing or any other appropriate procedures that the information provided is correct and appropriate for the purpose for which it is used.

CONTENTS **Page**

1

Introduction

This is the User Guide for MWA Software's OAuth2 Client. This is an RFC 6749 OAuth2 Client written in Object Pascal and provided as a Lazarus package. It is made available under the Lesser GPL. It has been tested using fpc 3.2.0 and Lazarus 2.0 and 2.2 under both Linux and Windows.

The client is intended to be fully featured and supports the following grant types:

- Authorization Code Grant
- Implicit Grant
- Resource Owner Password Credentials Grant
- Client Credentials Grant.

Attention has also been paid to extensibility. The client also provides:

- A means of implementing an Extension Grant, including new grant types.
- Support for new token types
- Support for New Endpoint Parameters, and
- Support for additional error codes.

This OAuth2 Client uses an external User Agent - the System Web Browser - and incorporates in internal http server for handling redirect responses from an Authorization Server.

The package uses the Indy Component library for both an http/https client and an http server. When the https protocol is used the OpenSSL library must also be installed and available for use.

Multithreading support is required for Authorization Code and Implicit Grants.

2

Installation and Setup

A recent copy of the Lazarus IDE and the Free Pascal (fpc) compiler is assumed to be already installed. Both the Indy components and the OAuth2 Client component must then be downloaded and installed. The command line tool “git” is assumed to be available for use in the examples below. However, any other appropriate means of downloading from github may be used instead.

2.1 Indy Component Library

The Indy component library provides a set of communications components, written in Object Pascal and available for use under Lazarus. The library is installed from Github using the following command run in the directory in which you intend to install the Indy library:

```
git clone -b master https://github.com/IndySockets/Indy.git
```

This will install the Indy library in the “Indy” subdirectory.

In the Lazarus IDE you should open the indylaz package in Indy/Lib and compile the package. There is no need to install it at this time.

At the time of writing, the Indy Component Library is available under a modified BSD License or the Mozilla Public License (see https://www.indyproject.org/license/).

2.2 OAuth2 Package

Now download the OAuth2 sources, if you have not already done so:

```
git clone -b main https://github.com/MWASoftware/oauth2client.git
```

This will install the components in the “oauth2” subdirectory.

In the Lazarus IDE you should open the “oauth2_laz” package in the oauth2 subdirectory and install the package into the IDE. This will also install the indy components.

You are now ready to setup a test server and try out the example programs (see section 5).

3

Using the OAuth2 Client

MWA Software's OAuth2 Client is a non-visual component that may be added to any Lazarus project. The examples/gui/oauth2test program provides an example of how the TOAuth2Client component is added to a project's main form, and used.

3.1 Implementation Model

The TOAuth2Client Implementation Model is illustrated below in Figure 1. The Client Application is assumed to include an instance of TOAuth2Client.

The use case is of a Client Application that needs to access some controlled resource (interaction A on the figure). However, in order to so, it needs to provide a proof that it has the right to access the resource by presenting a valid “Access Token”, where the “Access Token” is an identifier issued for a limited period of time and from which access rights can be determined.

The Access Token is issued by an Authorization Server that issues Access Tokens on behalf of the Resource Owner. The OAuth2 Specification identifies several different means by which Access Tokens can be issued to a requesting Client.

- **Client Credentials Grant**: this is achieved by a simple request/response exchange between the Client and the Authorization Server (interaction B). The Client identifies itself by an identifier and shared secret (password) and provided that it is then successfully authenticated the Access Token is returned. This grant type is sufficient for situations where an application, in its own right, can be granted access to the resource.

- **Resource Owner Password Credentials Grant**: this is similar to the above, except that interaction B includes a Resource Owner User Name and Password. The Authorization Server then authenticates both Client and Resource Owner and grants the Access Token on the Resource Owner's behalf. With this type of grant, the Client needs to be trusted “to see” the user's password and to pass it on to the Authorization Server. The User Name and Password is typically obtained from the Resource Owner by (e.g.) a dialog box (interaction F0).

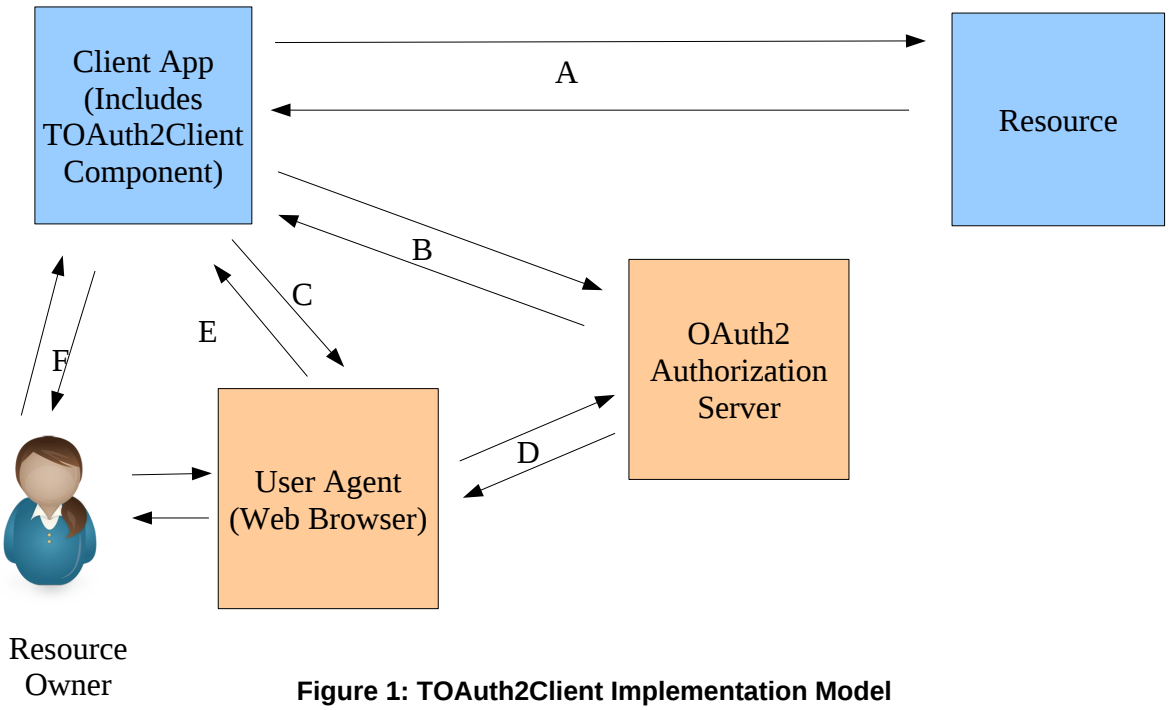


Figure 1: TOAuth2Client Implementation Model

- **Authorization Code Grant**: This grant type is used when the Client is not trusted to handle the Resource Owner's password, or when the Resource Owner authentication process is more complex than can be achieved with a simple password. For example, when two factor authentication is used. In this case, an external User Agent (e.g. a Web Browser) is invoked by the Client (interaction C) and redirected to the Authorization Server (interaction D). The Resource Owner is now in direct contact with the Authorization Server (via the User Agent) with the Client out of the loop.

  Once the Resource Owner has been authenticated and agrees to grant access to the Client, the Authorization Server returns a redirection to the User Agent with an Authorization Token (as part of the redirection URI query string). This redirection redirects the User Agent back to the Client (interaction E); the redirection gives the client access to the Authorization Token. The Client is now able to request an Access Token direct from the Authorization Server using the Authorization Token as a proof that access has been granted (interaction B).

  In some deployments, the Client could be a remote Web based application, or it could be a local application with an embedded web server. The TOAuth2Client implements the latter model and requires that both Client and User Agent are on the same system.

- **Implicit Grant**: This is a simplified version of the Authorization Code Grant which avoids the Client having to assert its identity by passing its Client Secret to the Authorization Server. This avoids the need for Interaction B, but does imply that trust in the client is *implicit* in the fact that the Resource Owner is contacting the Authorization Server via their User Agent. In this case, the Access Token is returned with the redirection (interaction E).

  This grant type is not usually suitable for remote clients. As with the Authorization Code Grant, TOAuth2Client implements this grant type by requiring that both Client and User Agent are on the same system.

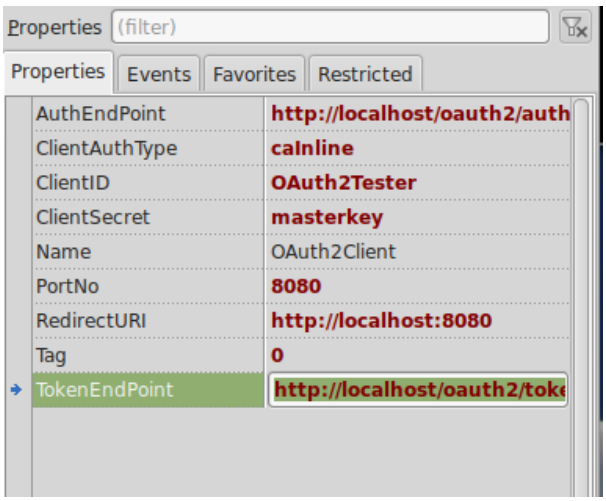
The Resource Owner Password Credentials Grant and the Authorization Code Grant may also issue a Refresh Token. This is a long lived token that can be saved by a client in secure storage and used to get a new Access Token when the current one expires, and by using a variant of Interaction B.

Each Authorization Server will implement its own Authorization Policies. Not all Authorization Server's support all grant types and only the grant types supported by an Authorization Server may be used to obtain an Access Token.

3.2 The TOAuth2Client Properties

In use, each instance of a TOAuth2Client component is typically dedicated for use with a specific OAuth2 Authorization Server, and this is reflected in the list of properties.

The TOAuth2Client properties are illustrated right, showing how they appear in the Lazarus Object Inspector.



The OAuth2 Specification requires that an Authorization Server publishes two endpoints:

- The Authorization Endpoint, and
- The Token Endpoint.

These are used for user authentication and to obtain an access token respectively. The AuthEndPoint and TokenEndPoint need to be set to the URI's for each endpoint respectively.

Note: In production use, these endpoints are typically on a remote server and should be accessed using the https protocol. The above illustration is for a local test environment where it is sufficient to use http.

The OAuth2 Specification permits the client to be authenticated using either an http Basic Authentication Header (caBasic), or inline (caInline) where the Client ID and Secret are included as part of the request body. An individual OAuth2 Server may support either authentication mechanism or both. The ClientAuthType property should be set to the authentication method specified/recommended by the Authorization Server.

OAuth2 Clients are identified to an Authorization Server by a unique Client ID and Client Secret. The Client ID and Secret used to identified the Client Application should be entered into the respective properties.

Note: in production use, the Client Secret should be set at run time and obtained from secure storage (e.g. encrypted in the application source code). It should not be included in the properties loaded from a form as a form resource (lfm file contents) is readily available to an attacker who has access to the compiled client executable.

The TOAuth2Client includes an embedded http server. This is used to receive redirected URI's from a User Agent (an external web browser). The Port No. that the embedded http server waits on is by default 8080. This may be changed if it is thought likely that this Port No. will conflict with other applications. Changing the PortNo property will automatically update the RedirectURI.

The RedirectURI is usually registered with an Authorization Server along with a Client ID. In the implementation model used by TOAuth2Client, the client is running on the same system as the User Agent and hence all redirections are local. The RedirectURI will thus always be in the form:

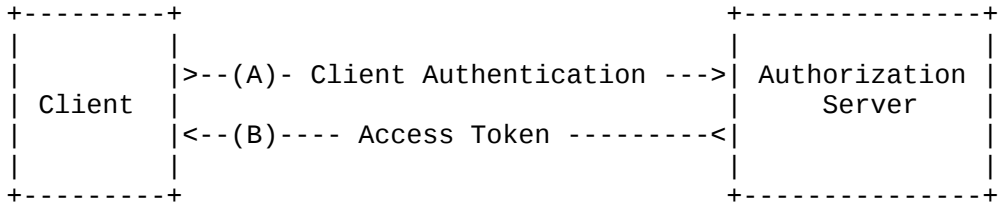
http://localhost:<port no>

It is unlikely that you will want to change this property other than as a result of a change to the Port No. property.

3.3 Client Credentials Grant

RFC 6749: The client can request an access token using only its client credentials (or other supported means of authentication) when the client is requesting access to the protected resources under its control, or those of another resource owner that have been previously arranged with the authorization server.

This grant type is known as the Client Credentials Grant, and has the following process flow:



In the above, the client uses an http/https GET (to the Token Endpoint) to obtain an Access Token from an Authorization Server. It provides its client credentials in order to identify itself and requests an Access Token for zero, one or more “scopes”. The Authorization Server authenticates the client and, provided that the client is successfully authenticated, it returns an Access Token together with an expiry time in seconds. This is the time for which the Access Token is valid. This is typically a relatively short time and which depends on Authorization Server policy.

A new Access Token should be requested each time the client is invoked and whenever the current Access Token expires.

The following TOAuth2Client method is called to request a Client Credentials Grant:

```
procedure GrantClientCredentials(Scope: string;
      var AccessToken: string; var TokenScope: string; var expires_in: integer);
```

When more than one scope is requested then this is given as a single space character separated list of scope names.

This call blocks until a response is received. If an error response is received then an EOAuth2Exception exception is raised (see 3.8).

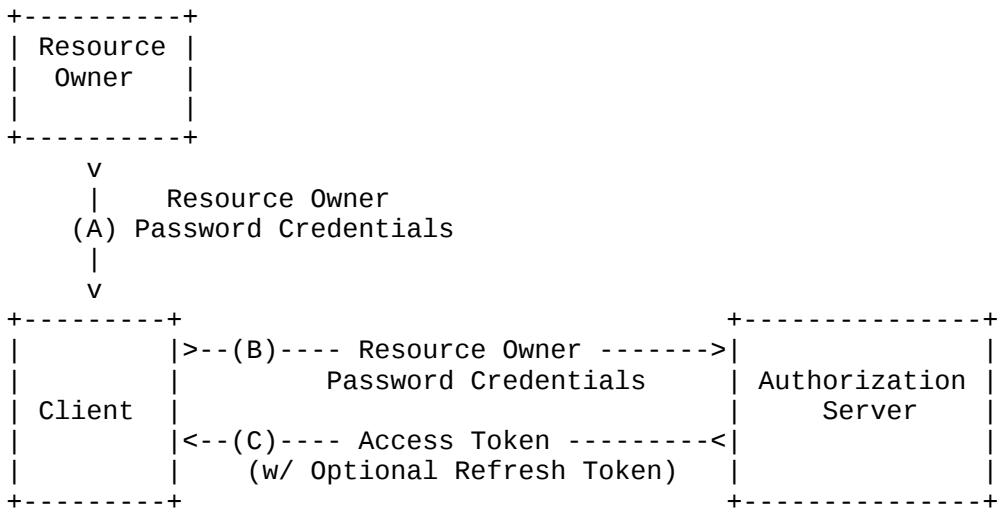
The method returns:

- The Access Token
- The lifetime of the token (expires_in) in seconds, and
- Optionally, the scope (TokenScope) for which the Access Token has been issued.

3.4 Resource Owner Password Credentials Grant

RFC 6749: The resource owner password credentials grant type is suitable in cases where the resource owner has a trust relationship with the client, such as the device operating system or a highly privileged application. It is suitable for clients capable of obtaining the resource owner's credentials (username and password, typically using an interactive form). It is also used to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials to an access token.

This grant request has the following process flow:



In the above, the Resource Owner (typically a person) interacts with the client to provide the User Name and Password that identifies them to the Authorization Server.

The client then uses an http/https GET (to the Token Endpoint)) to obtain an Access Token from the Authorization Server. The GET request provides:

- The Resource Owner's User Name and Password, and
- The client credentials used to identify itself and
- requests an Access Token for zero, one or more “scopes”.

The Authorization Server authenticates the client and the Resource Owner and, provided that the client and Resource Owner are successfully authenticated, it returns an Access Token together with an expiry time in seconds. This is the time for which the Access Token is valid. This is typically a relatively short time and depends on Authorization Server policy.

A new Access Token should be requested each time the client is invoked and whenever the current Access Token expires.

This grant type may also provide a Refresh Token. This has a much longer lifetime and may be saved in secure storage and used by subsequence invocations of the same program to request another Access Token (see 3.7) without needing to first obtain the Resource Owner's User Name and Password.

The following TOAuth2Client method is called to request a Resource Owner Password Credentials Grant:

```
procedure GrantUserPasswordCredentials(Scope: string;
  UserName, Password: string; var AccessToken: string; var RefreshToken: string;
  var TokenScope: string; var expires_in: integer);
```

When more than one scope is requested then this is given as a single space character separated list of scope names, along with the Resource Owner's User Name and Password.

This call blocks until a response is received. If an error response is received then an EOAuth2Exception exception is raised (see 3.8).

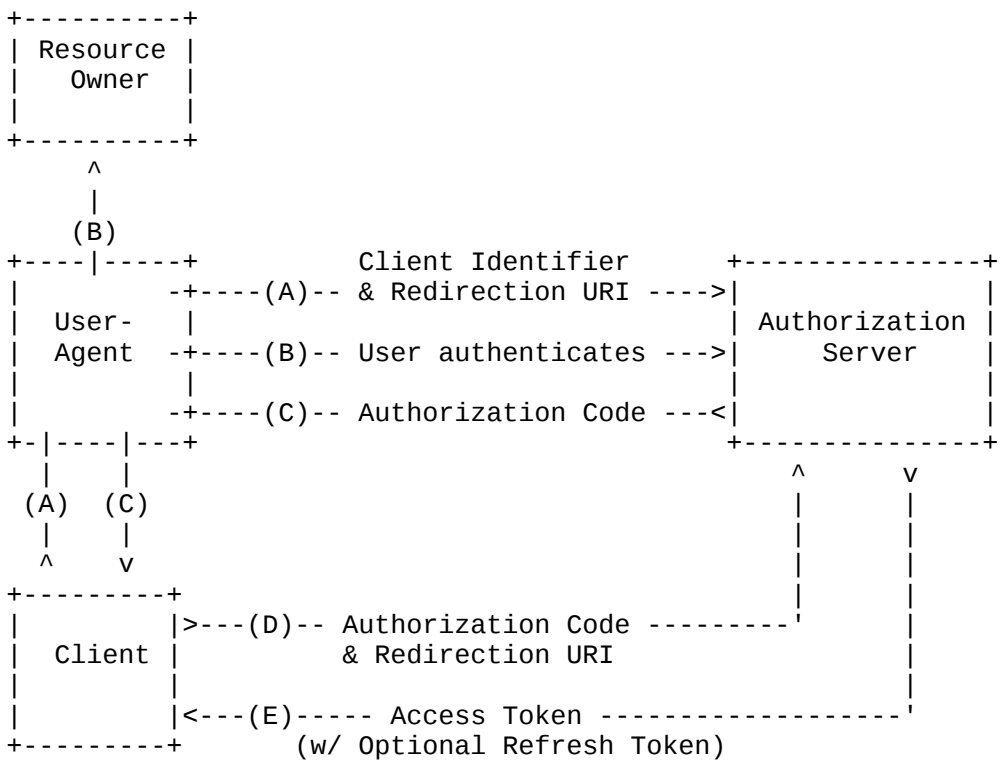
The method returns:

- The Access Token
- The lifetime of the token (expires_in) in seconds,
- Optionally, a Refresh Token, and
- Optionally, the scope (TokenScope) for which the Access Token has been issued.

3.5 Authorization Code Grant

RFC 6749: The authorization code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients. Since this is a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.

This grant request has the following process flow:



The grant request is initiated by TOAuth2Client by invoking the Resource Owner's User Agent.

The TOAuth2Client implementation model uses an external User Agent running on the same system as the OAuth2 Client. That is it invokes the system default Web Browser as the Resource Owner's User Agent and directs the User Agent to open a URI for a location on the Authorization Server (the Authorization Endpoint). This URI also includes a query string that identifies:

- The client (by the client id alone).
- Zero, one or more “scopes” for which an Access Token is requested, and
- The Client's “Redirect URI”.

The Redirect URI will be the value of the TOAuthClient's RedirectURI property and must match a RedirectURI registered with the Authorization Server and for the client as identified by its Client ID.

The Authorization Server will respond by returning some kind of “logon page” for the Resource Owner to identify and authenticate itself, and to approve the client's request for access to the controlled resource. The authentication may proceed over more than one web page and includes, where necessary, two factor authentication.

Once the Authorization Server has successfully authenticated the Resource Owner, it returns a response that includes a “Location” header redirecting the browser to the RedirectURI. The redirection includes a query string that provides an authentication code.

The TOAuth2Client runs an embedded http server in order to receive the redirection and extracts the authentication code from the query string. TOAuth2Client implements this http server using the Indy component library. The server is active for only as long as is necessary to receive the redirection.

Note: An http only server is used on the assumption that both client and user agent are on the same system and operate within a common trust domain.

The client now uses a similar http/https exchange to that used for the Resource Owner Password Grant in order to obtain an Access Code (and expiry time) from the Authorization Server (Token Endpoint). In this case, the Client identifies itself using its Client ID and Client Secret and includes the Authorization Code received in the query string instead of the User Name and Password.

The Authorization Server's response may also include a Refresh Token. This has a much longer lifetime and may be saved in secure storage and used by subsequence invocations of the same program to request another Access Token (see 3.7).

The use of an external user agent is typical for OAuth2 and in general is preferable to using an embedded user agent. The issues over the choice of an embedded versus an external web browser here are discussed at length in RFC 6749 section 9.

3.5.1 Methods

Both blocking and non-blocking TOAuth2Client methods are provided for an Authorization Grant:

3.5.1.1 GrantAuthorizationCode

The GrantAuthorizationCode method implements a blocking call:

```
procedure GrantAuthorizationCode(Scope: string; var AccessToken,
      RefreshToken: string; var TokenScope: string; var expires_in: integer;
      timeout: cardinal=INFINITE);
```

When more than one scope is requested then this is given as a single space character separated list of scope names.

A call to GrantAuthorizationCode may include a timeout (in milliseconds). The method exits with an EOAuth2ClientError exception (see 3.8) if no response is received within the specified timeout.

The method otherwise blocks until a response is received. If an error response is received then an EOAuth2Exception exception is raised (see 3.8).

The method returns:

- The Access Token
- The lifetime of the token (expires_in) in seconds,
- Optionally, a Refresh Token, and
- Optionally, the scope (TokenScope) for which the Access Token has been issued.

3.5.1.2 GrantAuthorizationCodeAsync

The GrantAuthorizationCodeAsync method implements the non-blocking call:

```
procedure GrantAuthorizationCodeAsync(Scope: string);
procedure CancelGrantRequest;
```

This method is intended for use in GUI applications and ensures that such applications remain responsive during the process flow. A separate thread is used to implement the process flow.

The grant request may be cancelled at any time by calling the CancelGrantRequest method.

When more than one scope is requested then this is given as a single space character separated list of scope names.

The method returns immediately after initiating the process.

3.5.2 Obtaining the Access Token with a Non-blocking Call

Two TOAuth2Client events are defined for use with GrantAuthorizationCodeAsync (and ImplicitGrantAsync - see 3.5.2):

```
type
  TOnErrorResponse = procedure(Sender: TObject; E: Exception) of object;
  TOnAccessToken = procedure(Sender: TObject; AccessToken, RefreshToken,
        TokenScope: string; expires_in: integer) of object

  property OnAccessToken: TOnAccessToken read FOnAccessToken write FOnAccessToken;
  property OnErrorResponse: TOnErrorResponse read FOnErrorResponse
           write FonErrorResponse;
```

On success, the OnAccessToken event is called and provides

- The Access Token
- The lifetime of the token (expires_in) in seconds,
- Optionally, a Refresh Token, and
- Optionally, the scope (TokenScope) for which the Access Token has been issued.

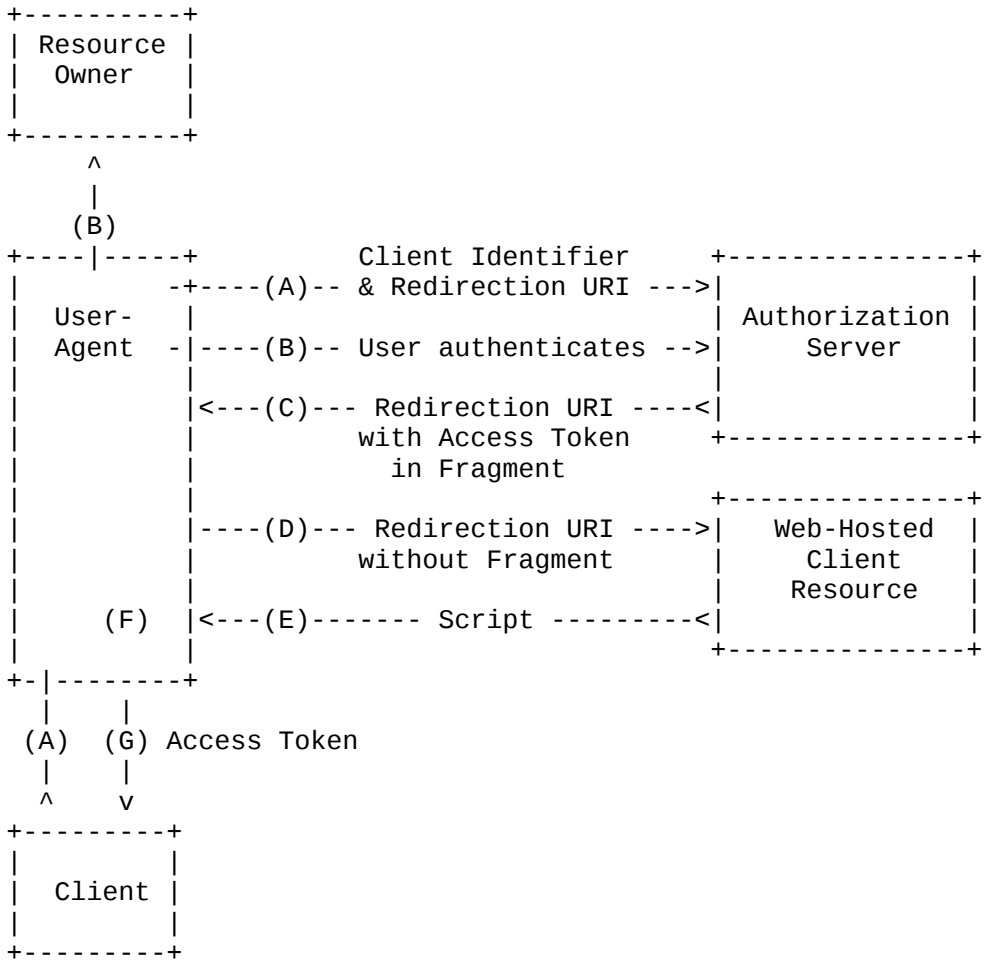
If an error occurred during the process flow, then the OnErrorResponse event is called providing the exception that reported the error. If this event handler is not defined then any errors are silently ignored.

3.6 Implicit Grant

RFC 6749: The implicit grant type is used to obtain access tokens (it does not support the issuance of refresh tokens) and is optimized for public clients known to operate a particular redirection URI. These clients are typically implemented in a browser using a scripting language such as JavaScript.

The implicit grant type does not include client authentication, and relies on the presence of the resource owner and the registration of the redirection URI. Because the access token is encoded into the redirection URI, it may be exposed to the resource owner and other applications residing on the same device

This grant request has the following process flow:



The grant request is initiated by TOAuth2Client by invoking the Resource Owner's User Agent.

The TOAuth2Client implementation model uses an external User Agent running on the same system as the OAuth2 Client. That is it invokes the system default Web Browser as the Resource

Owner's User Agent and directs the User Agent to open a URI for a location on the Authorization Server (the Authorization Endpoint). This URI also includes a query string that identifies:

- The client (by the client id alone).
- Zero, one or more “scopes” for which an Access Token is requested, and
- The Client's “Redirect URI”.

The Redirect URI will be the value of the TOAuthClient's RedirectURI property and must match a RedirectURI registered with the Authorization Server and for the client as identified by its Client ID.

The Authorization Server will return some kind of “logon page” for the Resource Owner to identify and authenticate itself, and to approve the client's request for access to the controlled resource. The authentication may proceed over more than one web page and includes, where necessary, two factor authentication. Once the Authorization Server has successfully authenticated the Resource Owner it returns a response that includes a “Location” header redirecting the browser to the RedirectURI. The redirection includes a URI fragment that provides an Access Token and expiry time.

TOAuth2Client runs an embedded http server in order to receive the redirection acting as the “Web Hosted Client Resource” in the above process flow. TOAuth2Client implements this http server using the Indy component library. The server is active for only as long as is necessary to receive the redirection.

Note: An http only server is used on the assumption that both client and user agent are on the same system and operate within a common trust domain.

When the http server receives the redirection, it does not receive the Access Token at this time. This is because the Access Token was provided as part of a URI fragment (i.e. after a '#' character in the redirection URI), and this part is retained by the User Agent. In order to recover the Access Token, the client's http server response is a web page consisting of a Javascript routine. This executes on the User Agent, and converts the fragment into a query string, which is then returned to the Client's http server, by another redirection.

The Access Token and expiry time may now be extracted from the query string and a success response returned to the Resource Owner's User Agent.

3.6.1 Methods

Both blocking and non-blocking TOAuth2Client methods are provided for an Implicit Grant.

3.6.1.1 ImplicitGrant

The ImplicitGrant method implements the blocking call:

```
procedure ImplicitGrant(Scope: string;
  var AccessToken: string; var TokenScope: string; var expires_in: integer;
  timeout: cardinal=INFINITE); overload;
```

When more than one scope is requested then this is given as a single space character separated list of scope names.

A call to ImplicitGrant may include a timeout (in milliseconds). The method exits with an EOAuth2ClientError exception (see 3.8) if no response is received within the specified timeout.

The method otherwise blocks until a response is received. If an error response is received then an EOAuth2Exception is raised (see 3.8).

The method returns:

- The Access Token
- The lifetime of the token (expires_in) in seconds, and
- Optionally, the scope (TokenScope) for which the Access Token has been issued.

3.6.1.2 ImplicitGrantAsync

The ImplicitGrantAsync method implements the non-blocking call:

```
procedure ImplicitGrantAsync(Scope: string);
procedure CancelGrantRequest;
```

This method is intended for use in GUI applications and ensures that such applications remain responsive during the process flow. A separate thread is used to implement the process flow.

The grant request may be cancelled at any time by calling the CancelGrantRequest method.

When more than one scope is requested then this is given as a single space character separated list of scope names.

The method returns immediately after initiating the process.

The Access Token is returned using the TOnAccessToken Event Handler (see 3.5.2).

If an error occurred during the process flow, then the OnErrorResponse event is called (see 3.5.2) providing the exception that reported the error. If this event handler is not defined then any errors are silently ignored.

3.7 Using a Refresh Token

A Refresh Token returned with a Resource Owner Password Credentials Grant or an Authorization Code Grant may be used to obtain a new Access Token by a call to the RefreshAccessToken method:

```
procedure RefreshAccessToken(Scope, RefreshToken: string;
  var AccessToken: string; var TokenScope: string;
  var NewRefreshToken: string; var expires_in: integer);
```

When more than one scope is requested then this is given as a single space character separated list of scope names. The list of scopes may be either empty (implying the original scope) or a subset of the originally requested scopes.

This call blocks until a response is received. If an error response is received then an EOAuth2Exception exception is raised (see 3.8).

The method returns:

- The Access Token
- The lifetime of the token (expires_in) in seconds,
- Optimally, a replacement Refresh Token, and

- Optionally, the scope (TokenScope) for which the Access Token has been issued.

The number of times that a Refresh Token can be used and its validity is dependent on the Authorization Server policy. If the call fails then the original grant request must be repeated in order to ensure ongoing access to the resource.

The availability of a replacement Refresh Token is also Authorization Server dependent. The NewRefreshToken argument is empty if no replacement Refresh Token is available.

3.8 Error Handling

Two exception types are defined for use with TOAuth2Client:

- EOAuth2ClientError, and
- EOAuth2Exception

The former is used for general client side errors, while the latter is used to report error responses, either http errors or error responses from the Authorization Server.

EOAuth2Exception has the following properties :

```
    property StatusCode: integer read FStatusCode;
    property ErrorCode: TOAuth2Errors read FErrorCode;
    property Error: string read FError;
    property Description: string read GetDescription;
    property ErrorURI: string read GetErrorURI;
```

Where:

- StatusCode is the HTTP Status Code for the associated response.
- ErrorCode is the OAuth2 Authorization Server error code expressed as an enumerated type.
- Error is the actual text error identifier provided by the Authorization Server.
- Description is a free text error message describing the error.
- ErrorURI is a reference to a web page, if available, giving further information about the error.

See the oauth2errors unit for the declaration of TOAuth2Errors and the semantic of each member of this enumerated type.

3.9 Providing Customised Responses

The TOAuth2Client embedded web server only returns simple plain text responses (e.g. “Access Authorized”). These responses can be customised by defining an OnGetBrowserResponseBody event handler for TOAuth2Client. This is defined as:

```
  TBrowserResponseType = (rtSuccess,rtError,rtIgnored,rtRedirect);

  TOnGetBrowserResponseBody = procedure(Sender: TObject;
        ResponseType: TBrowserResponseType; Contents: TMemoryStream) of object;

  property OnGetBrowserResponseBody: TOnGetBrowserResponseBody
       read FOnGetBrowserResponseBody write FonGetBrowserResponseBody;
```

This event handler is called immediately before a response is sent and allows the calling program to specify an arbitrarily complex web page to be set as a response. This is provided as the contents TStream and must be a complete web page “<html> .. </html>”.

For example:

```
procedure TForm1.OAuth2ClientGetBrowserResponseBody(Sender: TObject;
  ResponseType: TBrowserResponseType; Contents: TMemoryStream);
begin
  if ResponseType = rtSuccess then
    Contents.LoadFromFile('path to my file');
end;
```

Four Response Types are defined and a different web page may provided for each one:

- rtSuccess: used to report that Access has been authorised and and Access Token received.
- rtError: used to report that the Authorization Server has returned an error message. Note the error message itself is not included for security reasons.
- rtIgnored: used to report that an unexpected response was ignored.
- rtRedirect: used only for Implicit Grant when a Javascript routine is returned.

If, on return, the contents stream is empty then the default response text is used.

An rtRedirect response should usually be ignored by this event handler unless you need to modify the Javascript routine or the message displayed when Javascript is not enabled.

4

Extensibility

The TOAuth2Client supports extensibility as defined by RFC 6749 in respect of:

- New Access Token Types
- New Endpoint Parameters
- Extension Grants
- Additional Error Codes.

However, new Authorization Grant types are not currently supported due to the process flow for a new authorization grant being unknown.

4.1 Access Token Types and Endpoint Parameters

New Access Token Types and/or new Endpoint parameters are realised by subclassing the TTokenResponse or the TBearerTokenResponse (itself subclassed from TTokenResponse). Both classes may be found in the oauth2tokens unit.

The TBearerTokenResponse class implements the “bearer” access token type defined in RFC 749.

All the blocking grant request calls have overloaded variants that replace the returned values with a TTokenResponse object. These are:

```
procedure GrantClientCredentials(Scope: string; Response: TTokenResponse);
procedure GrantUserPasswordCredentials(Scope: string; UserName, Password: string;
      Response: TTokenResponse);
procedure GrantAuthorizationCode(Scope: string; Response: TTokenResponse;
      timeout: cardinal=INFINITE);
procedure ImplicitGrant(Scope: string;  Response: TTokenResponse;
      timeout: cardinal=INFINITE);
```

The general idea in which they are used should be clear from (e.g.) the standard version of GrantClientCredentials given below:

```
procedure TOAuth2Client.GrantClientCredentials(Scope: string;
  var AccessToken: string; var TokenScope: string;
  var expires_in: integer);
var Response: TBearerTokenResponse;
begin
  Response := TBearerTokenResponse.Create;
  try
    GrantClientCredentials(Scope,Response);
    AccessToken := Response.access_token;
    expires_in := Response.expires_in;
    TokenScope := Response.scope;
  finally
    Response.Free;
  end;
end;
```

The above creates a TBearerTokenResponse object to receive the response data and the method extracts the required data from the object before freeing it.

A similar approach is used when implementing new token types and/or endpoint parameters. The difference will be in the response data and how it is used.

The non-blocking calls have similar overloaded variants i.e.

```
procedure GrantAuthorizationCodeAsync(Scope: string;
     ResponseClass: TTokenResponseClass);
procedure ImplicitGrantAsync(Scope: string; ResponseClass: TTokenResponseClass);
```

Note that in this case the class name is used rather than providing an existing object. This is because the TOAuth2Client will create and own the token object. An extended version of the TOnAccessToken event is used to return the response i.e.

```
TOnAccessTokenExt = procedure(Sender: TObject; Response: TTokenResponse) of object;

property OnAccessTokenExt: TOnAccessTokenExt read FOnAccessTokenExt
                      write FonAccessTokenExt;
```

4.1.1 Declaring a TBearerTokenResponse subclass

When a new Access Token is defined that is an extension of the default “bearer” token then it should be declared as a subclass of TBearerTokenResponse. This allows all existing Endpoint parameters to be supported plus any new ones.

For example, if a new access token “customer” is defined and which includes all “bearer” token endpoint parameters plus a new endpoint parameter “customer_name” (i.e. the customer name). TBearerTokenResponse should be subclassed as follows:

```
  TCustomerResponseToken = class(TBearerTokenResponse)
  private
    FName: string;
  public
    constructor Create; override;
  published
    property customer_name: string read FName write FName;
             {property name is case sensitive}
  end;

constructor TCustomerResponseToken.Create;
begin
  inherited Create;
  SetTokenTypeName('customer');
end;
```

A TCustomerResponseToken may be used whenever a TTokenResponse is required. In the above:

- The call to SetTokenTypeName is used to set the name of the new token type.
- The published property “customer_name” is given an identical name to the new endpoint parameter - this is essential - and may be accessed to get the returned value of this endpoint parameter. If the endpoint parameter name is identical to a Pascal reserved word then it must be prefixed by an ampersand (&).

If there are validation rules that must be applied to the new endpoint parameter, then the protected method:

```
procedure ValidateResponse; override;
```

may be declared and a new validation rule added after calling the inherited method.

4.1.2 Subclassing TTokenResponse

TTokenResponse must be subclassed when the endpoint parameters for the new token type are different from the bearer token endpoint parameters. Otherwise, the above example holds expect that the list of published properties must include all endpoint parameters with each property's type defined as appropriate.

4.2 Extension Grants

New grant types may be implemented using the TOAuth2Client method ExtensionGrant. This is declared as:

```
procedure ExtensionGrant(GrantType: string; Params: TStrings;
               Response: TTokenResponse);
```

This is a blocking call made to the Authorization Server's Token End Point. The methods arguments are:

- GrantType: The new grant type name.
- Params: A list of parameters to the grant type as a list of “<name>=<value>” pairs in the TStrings object. Note that these must not be URIEncoded.

- Response: A subclassed TTokenResponse object that is used to receive a successful response. The TTokenResponse object will be whatever is the appropriate token type for the grant type.

If an error response is returned then an EOAuth2Exception is raised (see 3.8).

4.3 Additional Error Codes

The EOAuth2Exception type is defined in 3.8. If a new error type is returned then the ErrorCode property is set to oeUnknown and the “Error” property may be inspected to get the actual error type name.

5

Using the Example Programs

The oauth2/examples subdirectory contains two examples for GUI and non-gui usage of the OAuth2 Client, respectively. A test OAuth2 server is also provided in oath2/examples/server. This test server is intended to work with the example applications.

5.1 Installing the Test OAuth2 Server

The test server is written in PHP and uses an OAuth2 Server Library for PHP described at

https://bshaffer.github.io/oauth2-server-php-docs/

and downloadable from github.

The test server assumes a typically LAMP (Linux, Apache, Mysql and PHP) environment and the working assumption is that it is running on the same system as the OAuth2 Client. A remote and/or Windows computer may also be used, provided that Apache, mysql and PHP have been correctly installed and set up. An alternative to Apache may also be used. However, the instructions below assume the use of the Apache Web Server, and, by default, this is a local server.

The test server uses a backend mysql database.

A setup script (oath2/examples/setup.sh) is provided to automate the setup of the test server on a Linux system. The same steps must be manually performed when using a different operating system.

The setup script:

- Defines a proposed database name, username and password. These can all be modified before running the script if they are inappropriate for your environment.
- Installs the oauth2-server-php scripts under oauth2/examples/server if they do not already exist. The 'git' command line utility is used to do this i.e.:

```
git clone https://github.com/bshaffer/oauth2-server-php.git -b master
```

- Creates the mysql database and a database user, and grants this database user full access to the database.

- Uses the oath2/examples/schema.sql script to create the database schema and load some essential test data.

- Creates the config.php file. This is used to pass the DB Name, User name and password to the php scripts. Using the defaults, this file contains:

  ```
  <?php
  $dsn      = 'mysql:dbname=oauth2_db;host=localhost';
  $username = 'oauth2';
  $password = '4uZdvKaM';
  ```

- Creates an example set of apache configuration directives for copying to the apache default virtual server configuration. An example of this file is:

  ```
  Alias "/oauth2/"  "/home/tony/oauth2-test/examples/server/endpoints/"
  <Directory "/home/tony/oauth2-test/examples/server/">
  php_value engine on
  allow from all
  </Directory>
  ```

  These directives will tell apache that /oauth2/ is an alias for the directory in which the endpoint scripts are located and enables the PHP engine and access to this directory. In practice, your server may require additional configuration directives. For example, if “open_basedir” restrictions in place that the above directory path will also need to be added to the list of open_basedir directories.

  Once the apache configuration has been updated, the apache server should be restarted/reloaded.

5.2 The Example GUI Application

The example GUI Applications is located in oath2/examples/gui and demonstrates:

- The Authorization Code Grant
- The Implicit Grant
- Authorization Code and Implicit Grant cancellation
- The Resource Owner Password Credentials Grant
- The Client Credentials Grant.
- Token Refresh
- Resource Access.

By default, the demonstration assumes that the Authorization and Token endpoints are on the locahost. If a remote server is used then either the Endpoint properties (see 3.2) and the ResourceURL variable in the mainframe unit must be changed to point to the remote server endpoints (before compilation), or the program is run with an AUTHSERVER environment variable set to the server's domain name and protocol.

The demonstration program will check to see if the AUTHSERVER environment variable is defined and, if so, updates the endpoints and ResourceURL variable to point to the remote server. For example, if

export AUTHSERVER=https://myserver.com

is defined, then the endpoints are updated, replacing http://localhost with the above.

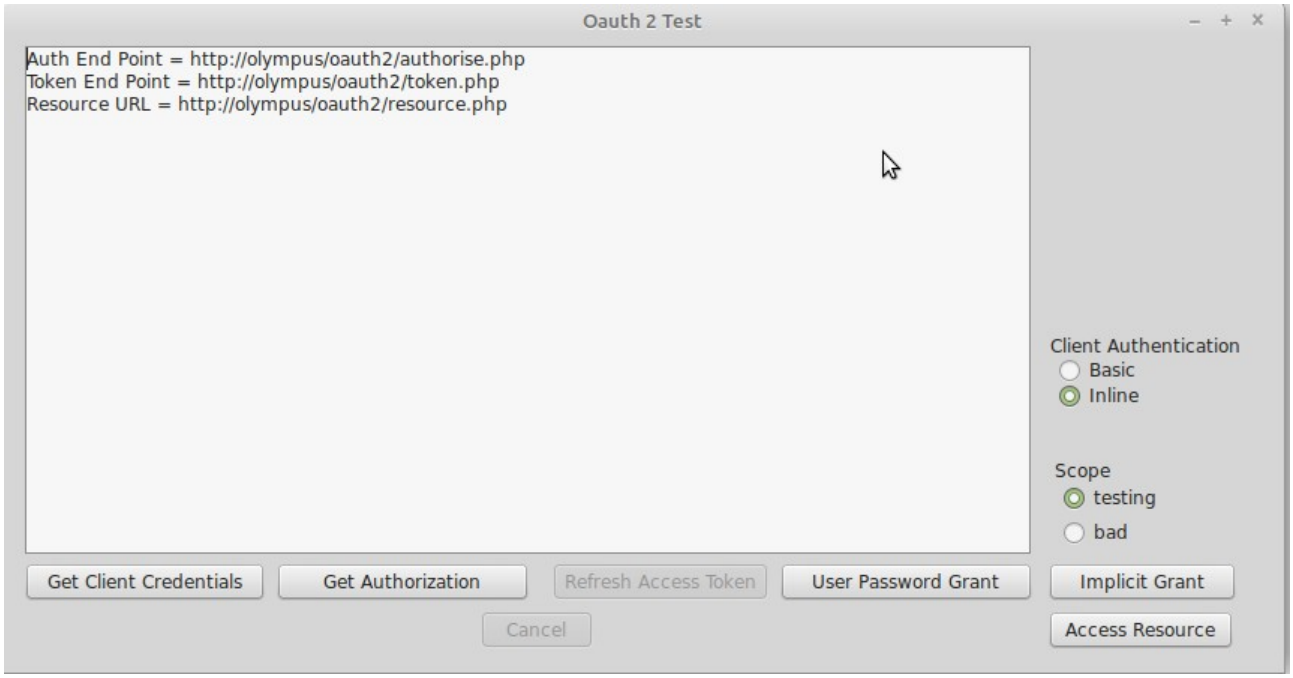


Figure 2: Example use of the OAuth2Client

On startup, the application main window should look as above. The endpoints and ResourceURL are all reported. In this case, demonstrating the use of the AUTHSERVER environment variable to use a test server called “olympus”.

The OAuth2 test Server supports both basic and inline authentication. Either may be selected.

The “testing” scope is the only legitimate scope accepted by the test server. Selecting “bad” can be used to demonstrate error handling.

Otherwise, each grant type may be performed by clicking on the appropriate button. Note that Authorization and Implicit grants will invoke the system web browser. For the test, there is no user authentication via the web browser. It is sufficient to prompt the user to confirm or reject the authorization request. All responses are recorded in the TMemo window.

Once an access token has been granted, a simulated controlled resource may be accessed by clicking on the “Access Resource” button. This resource requires a valid access token and returns a simple text response. The response is JSON encoded and is parsed before reporting in the TMemo. The code for parsing the token is worth viewing as an example for parsing JSON.

5.3 The Example non-GUI (Console Mode) Application

The example non-GUI application is located in oauth2/examples/nongui. This application supports the use of the AUTHSERVER environment variable as above. When invoked, it runs each grant type in turn, reporting the results to the system console. The blocking versions of the Authorization and Implicit grant methods are used here.

The grants are then re-run for the “bad” scope in order to demonstrate error handling.

Note that the program has a dependency on LCLBase via the Indy library. However, it is acceptable to define the use of the “nogui” interface in the project options.