Using OAuth 2.0 for Client-side Applications

The Google OAuth 2.0 endpoint supports JavaScript-centric applications. These applications may access a Google API while the user is present at the application, and this type of application cannot keep a secret.

This article describes how to use OAuth 2.0 when accessing a Google API from a JavaScript application.

Contents

Overview
Forming the URL
Handling the response
Validating the token
Calling a Google API
Incremental authorization

Overview

This scenario begins by redirecting a browser (full page or popup) to a Google URL with a set of query parameters that indicate the type of Google API access the application requires. As in other scenarios, Google handles user authentication and consent, and the result is an access token. Google returns the access token on the fragment of the response, and client side script extracts the access token from the response.

The application may access a Google API after it receives the access token.

> **Note**: Your application should always use HTTPS in this scenario.

Forming the URL

The URL used when authenticating a user is `https://accounts.google.com/o/oauth2/auth`. This endpoint is accessible over SSL, and HTTP connections are refused.

Endpoint	Description
`https://accounts.google.com/o/oauth2/auth`	This endpoint is the target of the initial request. It handles active session lookup, authenticating the user, and user consent.

The set of query string parameters supported by the Google Authorization Server for client-side applications are:

Parameter	Values	Description
`response_type`	`token`	JavaScript applications should use `token`. This tells the Google Authorization Server to return the access token on the fragment.
`client_id`	The client ID	Identifies the client that is making the request. The value passed in this

	you obtain from the Developers Console when you set up your app.	parameter must exactly match the value shown in the Google Developers Console.
`redirect_uri`	One of the `redirect_uri` values listed for this project in the Developers Console.	Determines where the response is sent. The value of this parameter must exactly match one of the values listed for this project in the Google Developers Console (including the http or https scheme, case, and trailing '/').
`scope`	Space-delimited set of permissions that the application requests.	Identifies the Google API access that your application is requesting. The values passed in this parameter inform the consent screen that is shown to the user. There is an inverse relationship between the number of permissions requested and the likelihood of obtaining user consent. For information about available login scopes, see Login scopes. To see the available scopes for all Google APIs, visit the APIs Explorer. It is generally a best practice to request scopes incrementally, at the time access is required, rather than up front. For example, an app that wants to support purchases should not request Google Wallet access until the user presses the “buy” button; see Incremental authorization.
`state`	Any string	Provides any state that might be useful to your application upon receipt of the response. The Google Authorization Server roundtrips this parameter, so your application receives the same value it sent. Possible uses include redirecting the user to the correct resource in your site, nonces, and cross-site-request-forgery mitigations.
`approval_prompt`	`force` or `auto`	Indicates if the user should be re-prompted for consent. The default is `auto`, so a given user should only see the consent page for a given set of scopes the first time through the sequence. If the value is `force`, then the user sees a consent page even if they previously gave consent to your application for a given set of scopes.
`login_hint`	`email address` or `sub` identifier	When your application knows which user it is trying to authenticate, it can provide this parameter as a hint to the Authentication Server. Passing this hint will either pre-fill the email box on the sign-in form or select the proper multi-login session, thereby simplifying the login flow.
`include_granted_scopes`	`true` or `false`	If this is provided with the value `true`, and the authorization request is granted, the authorization will include any previous authorizations granted to this user/application combination for other scopes; see Incremental Authorization.

An example URL is shown below, with line breaks and spaces for readability.

```
https://accounts.google.com/o/oauth2/auth?
 scope=email%20profile&
 state=%2Fprofile&
 redirect_uri=https%3A%2F%2Foauth2-login-demo.appspot.com%2Foauthcallback&
 response_type=token&
 client_id=812741506391.apps.googleusercontent.com
```

Handling the response

Google returns an access token to your application if the user grants your application the permissions it requested. The access token is returned to your application in the fragment as part of the `access_token` parameter. Since a fragment is not returned to the server, client-side script must parse the fragment and extract the value of the `access_token` parameter.

Other parameters included in the response include `expires_in` and `token_type`. These parameters describe the lifetime of the token in seconds, and the kind of token that is being returned. If the `state` parameter was included in the request, then it is also included in the response.

An example User Agent flow response is shown below:

```
https://oauth2-login-demo.appspot.com/oauthcallback#access_token=1/fFBGRNJru1FQd44AzqT3Zg&token_type=Bearer&expires_in=3600
```

> **Note**: Other fields may be included in the response, and your application should not treat this as an error. The set shown above is the minimum set.

Below is a JavaScript snippet that parses the response and returns the parameters to the server. This code is hosted at the `https://oauth2-login-demo.appspot.com/oauthcallback` URL.

```
// First, parse the query string
var params = {}, queryString = location.hash.substring(1),
    regex = /([^&=]+)=([^&]*)/g, m;
while (m = regex.exec(queryString)) {
  params[decodeURIComponent(m[1])] = decodeURIComponent(m[2]);
}

// And send the token over to the server
var req = new XMLHttpRequest();
// consider using POST so query isn't logged
req.open('GET', 'https://' + window.location.host + '/catchtoken?' + queryString, true);

req.onreadystatechange = function (e) {
  if (req.readyState == 4) {
     if(req.status == 200){
       window.location = params['state']
   }
  else if(req.status == 400) {
        alert('There was an error processing the token.')
    }
    else {
      alert('something else other than 200 was returned')
    }
  }
};
req.send(null);
```

This code sends the parameters received on the fragment to the server using XMLHttpRequest and writes the access token to local storage in the browser. The latter is an optional step, and depends on whether or not the application requires other JavaScript code to make calls to a Google API. Also note that this code sends the parameters to the /accepttoken endpoint, and they are sent over an HTTPS channel.

Error response

The Google Authorization Server returns an error if the user did not grant your application the permissions it requested. The error is returned in the fragment.

An example error response is shown below:

```
https://oauth2-login-demo.appspot.com/oauthcallback#error=access_denied
```

Validating the token

Tokens received on the fragment **MUST** be explicitly validated. Failure to verify tokens acquired this way makes your application more vulnerable to the confused deputy problem.

You can validate a token by making a web service request to an endpoint on the Google Authorization Server and performing a string match on the results of that web service request.

TokenInfo validation

Verifying a token using the Google Authorization Server endpoint is relatively simple. Your application includes the access token in the `access_token` parameter for the following endpoint:

Endpoint	Description
`https://www.googleapis.com/oauth2/v1/tokeninfo`	Accepts an access token and returns information about that access token including which application was it issued to, the intended target of the token, the scopes the user consented to, the remaining lifetime of the token, and the user ID.

Below is an example of such a request:

```
https://www.googleapis.com/oauth2/v1/tokeninfo?access_token=1/fFBGRNJru1FQd44AzqT3Zg
```

The TokenInfo endpoint will respond with a JSON array that describes the token or an error. Below is a table of the fields included in the non-error case:

Field	Description
`audience`	The application that is the intended target of the token.
`scope`	The space-delimited set of scopes that the user consented to.
`userid`	This field is only present if the `profile` scope was present in the request. The value of this field is an immutable identifier for the logged-in user, and may be used when creating and managing user sessions in your application. This identifier is the same regardless of the client ID. This provides the ability to correlate profile information across multiple applications in the same organization.
`expires_in`	The number of seconds left in the lifetime of the token.

On the wire, the response looks similar to the following:

```
{
  "audience":"8819981768.apps.googleusercontent.com",
  "user_id":"123456789",
```

```
  "scope":"profile email",
  "expires_in":436
}
```

> **Note:** When verifying a token, it is critical to ensure the `audience` field in the response exactly matches the client ID that you obtained in the Developers Console. It is absolutely vital to perform this step, because it is the mitigation for the confused deputy issue.

If the token has expired, has been tampered with, or the permissions revoked, the Google Authorization Server will respond with an error. The error surfaces as a `400` status code, and a JSON body as follows:

```
{"error":"invalid_token"}
```

By design, no additional information is given as to the reason for the failure.

Calling a Google API

After your application obtains an access token, you can use the token to make calls to a Google API on behalf of a given user account or service account. To do this, include the access token in a request to the API by including either an `access_token` query parameter or an `Authorization: Bearer` HTTP header. When possible, the HTTP header is preferable, because query strings tend to be visible in server logs. In most cases you can use a client library to set up your calls to Google APIs (for example, when calling the People API).

You can try out all the Google APIs and view their scopes at the OAuth 2.0 Playground.

Examples

A call to the `people.get` endpoint (the People API) using the `access_token` query string parameter might look like the following, though you'll need to specify your own access token:

```
GET https://www.googleapis.com/plus/v1/people/userId?access_token=1/fFBGRNJru1FQd44AzqT3Zg
```

Here is a call to the same API for the authenticated user (`me`) using the `Authorization: Bearer` HTTP header:

```
GET /plus/v1/people/me HTTP/1.1
Authorization: Bearer 1/fFBGRNJru1FQd44AzqT3Zg
Host: googleapis.com
```

You can try out with the `curl` command-line application. Here's an example using the HTTP header option (preferred):

```
curl -H "Authorization: Bearer 1/fFBGRNJru1FQd44AzqT3Zg" https://www.googleapis.com/plus/v1/people/me
```

Or, alternatively, the query string parameter option:

```
curl https://www.googleapis.com/plus/v1/people/me?access_token=1/fFBGRNJru1FQd44AzqT3Zg
```

Incremental authorization

In the OAuth protocol, your app requests authorization to access resources which are identified by scopes, and assuming the user is authenticated and approves, your app receives short-lived access tokens which let it access those resources, and (optionally) refresh tokens to allow long-term access.

It is considered a best user-experience practice to request authorization for resources at the time you need them. For example, an app that lets people sample music tracks and create mixes might need very few resources at sign-in time, perhaps nothing more than the name of the person signing in. However, saving a completed mix would require access to their Google Drive. Most people would find it natural if they only were asked for access to their Google Drive at the time the app actually needed it.

In this case, at sign-in time the app might request the scope `https://www.googleapis.com/auth/plus.login` to perform a basic social-login function, and then later request the scope `https://www.googleapis.com/auth/drive.file` at the time of the first request to save a mix.

Using the procedures described in Using OAuth 2.0 for Login and Using OAuth 2.0 to Access Google APIs would normally result in your app having to manage two different access tokens. If you wish to avoid this complexity, you may add an extra parameter to the Authentication URI that you send to `https://accounts.google.com/o/oauth2/auth` as the first step of any OAuth 2.0 flow. The parameter is `include_granted_scopes` and the allowed values are `true` and `false` (the default is `false`). When the value is `true`, the effect is that if your scope authorization request is granted, the Google authorization server will roll this authorization together with all the previous authorizations granted to the requesting user from the requesting app. The URI you construct might end up looking this this (line breaks and space inserted for readability):

```
https://accounts.google.com/o/oauth2/auth?
  scope=https://www.googleapis.com/auth/drive.file&
  state=security_token%3D138r5719ru3e1%26url%3Dhttps://oa2cb.example.com/myHome&
  redirect_uri=https%3A%2F%2Fmyapp.example.com%2Fcallback&
  response_type=code&
  client_id=8127352506391.apps.googleusercontent.com&
  approval_prompt=force&
  include_granted_scopes=true
```

Let's call the resulting authorization the "combined authorization"; the following apply:

- You can use the access tokens you get to access the resources corresponding to any of the scopes that are rolled into the combined authorization.
- When you use the refresh token for a combined authorization, the new access tokens represent the combined authorization and can be used for any of its scopes.
- The combined authorization includes any previously granted authorizations even if they were requested from different clients. For example, if you requested the `https://www.googleapis.com/auth/plus.login` scope from a desktop app, and then issued the request in the example URI above for the same user from a mobile app, and it was granted, the combined authorization would include both scopes.
- When you revoke a token which represents a combined authorization, all of the authorizations are revoked simultaneously; this means that if you retain a token for one of the previous authorizations, it will stop working.

Last updated November 14, 2014.