Using OAuth 2.0 for Web Server Applications

You can use the Google API Client Libraries or Google OAuth 2.0 endpoints to create web server applications that use OAuth 2.0 authorization to access Google APIs. OAuth 2.0 allows users to share specific data with an application while keeping their usernames, passwords, and other information private. For example, a web application can use OAuth 2.0 to obtain permission from users to store files in their Google Drives.

This document describes using OAuth 2.0 for user authorization. Web applications frequently also use service accounts when making API calls, particularly when calling Cloud APIs to access project-based data rather than user-specific data. These mechanisms can be used in conjunction with user authorization in a web server application.

Note: Given the security implications of getting the implementation correct, we strongly encourage you to use OAuth 2.0 libraries when interacting with Google's OAuth 2.0 endpoints. It is a best practice to use well-debugged code provided by others, and it will help you protect yourself and your users. For more information, see Client libraries.

Overview

To use OAuth 2.0 in a web application, first create web application credentials for your project in the Developers Console.

Then, when your application needs to access a user's data with a Google API, your application redirects the user to Google's OAuth 2.0 server. The OAuth 2.0 server authenticates the user and obtains consent from the user for your application to access the user's data.

Next, Google's OAuth 2.0 server redirects the user back to your application along with a single-use authorization code. Your application exchanges this authorization code for an access token.

Finally, your application can use the access token to call Google APIs.

When you use a Google API Client Library to handle your application's OAuth 2.0 flow, the client library keeps track of when a stored access token can be used and when the application must re-acquire consent, generates correct redirect URLs, and helps to implement redirect handlers that exchange authorization codes for access tokens. An application that carries out the OAuth 2.0 flow without using a client library must correctly complete the same steps.

Client libraries

The language-specific examples on this page make use of the Google API Client Libraries, which make API authorization with OAuth 2.0 simpler. To run the example code, you must first install the client library for your language.

Client libraries are available for the following languages:

- Go
- Java
- .NET
- Node.js
- PHP
- Python
- Ruby

Creating web application credentials

All web applications that use OAuth 2.0 must have credentials that identify the application to the OAuth 2.0 server. Applications that have these credentials can access the APIs that you enabled for your project.

To obtain web application credentials for your project, complete these steps:

1. Open the Credentials page.
2. If you haven't done so already, create your OAuth 2.0 credentials by clicking **Create new Client ID** under the **OAuth** heading. Next, look for your application's client ID and client secret in the relevant table.
3. You can also create and edit redirect URIs from this page, by clicking a client ID. Redirect URIs are the URIs to your application's auth endpoints, which handle responses from the OAuth 2.0 server. You must change this value from the default example to the URI of your application's auth endpoint before you can use OAuth 2.0. For testing, you can specify URIs that refer to the local machine, such as `http://localhost:8080`. You should design your app's auth endpoints in a way that doesn't expose authorization codes to other resources on the page.

Download the client_secrets.json file and securely store it in a location that only your application can access.

Important: Do not store the client_secrets.json file in a publicly-accessible location, and if you share the source code to your application—for example, on GitHub—store the client_secrets.json file outside of your source tree to avoid inadvertently sharing your client credentials.

Preparing to start the OAuth 2.0 flow

If you are using a Google API client library to handle the OAuth 2.0 flow, configure the client object, which you will use to make OAuth 2.0 requests. If you are handling the flow by directly accessing the OAuth 2.0 endpoints, just take note of the client ID that you created in the previous step and the scopes you need to request.

To configure the client object:

PHP | PYTHON | RUBY | HTTP/REST

Use the client_secrets.json file that you created to configure a client object in your application. When you configure a client object, you specify the scopes your application needs to access, along with the URL to your application's auth endpoint, which will handle the response from the OAuth 2.0 server.

For example, to request read-only access to a user's Google Drive:

```
$client = new Google_Client();
$client->setAuthConfigFile('client_secrets.json');
$client->addScope(Google_Service_Drive::DRIVE_METADATA_READONLY);
$client->setRedirectUri('http://' . $_SERVER['HTTP_HOST'] . '/oauth2callback.php');
```

Your application uses the client object to perform OAuth 2.0 operations, such as generating authorization request URLs and applying access tokens to HTTP requests.

Redirecting to Google's OAuth 2.0 server

When your application needs to access a user's data, redirect the user to Google's OAuth 2.0 server.

PHP | PYTHON | RUBY | HTTP/REST

1. Generate a URL to request access from Google's OAuth 2.0 server:

   ```
   $auth_url = $client->createAuthUrl();
   ```

2. Redirect the user to `$auth_url`:

   ```
   header('Location: ' . filter_var($auth_url, FILTER_SANITIZE_URL));
   ```

Google's OAuth 2.0 server will authenticate the user and obtain consent from the user for your application to access the requested scopes. The response will be sent back to your application using the redirect URL you specified.

Handling the OAuth 2.0 server response

The OAuth 2.0 server responds to your application's access request by using the URL specified in the request.

If the user approves the access request, then the response contains an authorization code. If the user does not approve the request, the response contains an error message. All responses are returned to the web server on the query string, as shown below:

An error response:

```
https://oauth2-login-demo.appspot.com/auth?error=access_denied
```

An authorization code response:

```
https://oauth2-login-demo.appspot.com/auth?code=4/P7q7W91a-oMsCeLvIaQm6bTrgtp7
```

Important: If your response endpoint renders an HTML page, any resources on that page will be able to see the authorization code in the URL. Scripts can read the URL directly, and all resources may be sent the URL in the `Referer` HTTP header. Carefully consider if you want to send authorization credentials to all resources on that page (especially third-party scripts such as social plugins and analytics). To avoid this issue, we recommend that the server first handle the request, then redirect to another URL that doesn't include the response parameters.

After the web server receives the authorization code, it can exchange the authorization code for an access token.

PHP PYTHON RUBY HTTP/REST

To exchange an authorization code for an access token, use the `authenticate` method:

```
$client->authenticate($_GET['code']);
```

You can retrieve the access token with the `getAccessToken` method:

```
$access_token = $client->getAccessToken();
```

Calling Google APIs

PHP PYTHON RUBY HTTP/REST

Use the access token to call Google APIs by completing the following steps:

1. If you need to apply an access token to a new `Google_Client` object—for example, if you stored the access token in a user session—use the `setAccessToken` method:

   ```
   $client->setAccessToken($access_token);
   ```

2. Build a service object for the API that you want to call. You build a a service object by providing an authorized `Google_Client` object to the constructor for the API you want to call. For example, to call the Drive API:

   ```
   $drive_service = new Google_Service_Drive($client);
   ```

3. Make requests to the API service using the interface provided by the service object. For example, to list the files in the authenticated user's Google Drive:

   ```
   $files_list = $drive_service->files->listFiles(array())->getItems();
   ```

Complete example

The following example prints a JSON-formatted list of files in a user's Google Drive after the user authenticates and gives consent for the application to access the user's Drive files.

PHP PYTHON RUBY HTTP/REST

To run this example:

1. In the Developers Console, add the URL of the local machine to the list of redirect URLs. For example, add `http://localhost:8080`.
2. Create a new directory and change to it. For example:

```
mkdir ~/php-oauth2-example
cd ~/php-oauth2-example
```

3. Clone the Google API Client Library for PHP:

```
git clone -b v1-master https://github.com/google/google-api-php-client
```

4. Create the files index.php and oauth2callback.php with the content below.
5. Run the example with a web server configured to serve PHP. If you use PHP 5.4 or newer, you can use PHP's built-in test web server:

```
php -S localhost:8080 ~/php-oauth2-example
```

index.php

```
<?php
require_once 'google-api-php-client/autoload.php';

session_start();

$client = new Google_Client();
$client->setAuthConfigFile('client_secrets.json');
$client->addScope(Google_Service_Drive::DRIVE_METADATA_READONLY);

if (isset($_SESSION['access_token']) && $_SESSION['access_token']) {
  $client->setAccessToken($_SESSION['access_token']);
  $drive_service = new Google_Service_Drive($client);
  $files_list = $drive_service->files->listFiles(array())->getItems();
  echo json_encode($files_list);
} else {
  $redirect_uri = 'http://' . $_SERVER['HTTP_HOST'] . '/oauth2callback.php';
  header('Location: ' . filter_var($redirect_uri, FILTER_SANITIZE_URL));
}
```

oauth2callback.php

```
<?php
require_once 'google-api-php-client/autoload.php';

session_start();
$client = new Google_Client();
$client->setAuthConfigFile('client_secrets.json');
$client->setRedirectUri('http://' . $_SERVER['HTTP_HOST'] . '/oauth2callback.php');
$client->addScope(Google_Service_Drive::DRIVE_METADATA_READONLY);

if (! isset($_GET['code'])) {
  $auth_url = $client->createAuthUrl();
  header('Location: ' . filter_var($auth_url, FILTER_SANITIZE_URL));
} else {
  $client->authenticate($_GET['code']);
  $_SESSION['access_token'] = $client->getAccessToken();
  $redirect_uri = 'http://' . $_SERVER['HTTP_HOST'] . '/';
  header('Location: ' . filter_var($redirect_uri, FILTER_SANITIZE_URL));
}
```

Incremental authorization

In the OAuth 2.0 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 `profile` scope to perform basic sign-in, and then later request the `https://www.googleapis.com/auth/drive.file` scope at the time of the first request to save a mix.

Using the procedures described in Using OpenID Connect and Using OAuth 2.0 to Access Google APIs would normally result in your app having to manage two different access tokens.

To avoid this complexity, you can include previously granted scopes in your authorization requests. For example:

PHP | PYTHON | RUBY | HTTP/REST

```
$client->setIncludeGrantedScopes(true);
```

When you make an authorization request with granted scopes included, the Google authorization server rolls the authorization request together with all the previous authorizations granted to the requesting user from the requesting app.

Offline access

In some cases, your application might need to access a Google API when the user is not present. Examples of this include backup services and applications that make Blogger posts exactly at 8am on Monday morning. This style of access is called `offline`, and web server applications can request `offline` access from a user. The normal and default style of access is called `online`.

PHP | PYTHON | RUBY | HTTP/REST

If your application needs offline access to a Google API, set the API client's access type to `offline`:

```
$client->setAccessType("offline");
```

After a user grants offline access to the requested scopes, you can continue to use the API client to access Google APIs on the user's behalf when the user is offline. The client object will refresh the access token as needed.

Revoking a token

In some cases a user may wish to revoke access given to an application. A user can revoke access by visiting Account Settings. It is also possible for an application to programmatically revoke the access given to it. Programmatic revocation is important in instances where a user unsubscribes or removes an application. In other words, part of the removal process can include an API request to ensure the permissions granted to the application are removed.

PHP | PYTHON | RUBY | HTTP/REST

To programmatically revoke a token, call `revokeToken()`:

```
$client->revokeToken();
```

Note: Following a successful revocation response, it might take some time before the revocation has full effect.

Last updated January 7, 2016.