Using OAuth 2.0 for Web Server Applications

This document explains how web server applications use Google API Client Libraries or Google OAuth 2.0 endpoints to implement 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, an application can use OAuth 2.0 to obtain permission from users to store files in their Google Drives.

This OAuth 2.0 flow is specifically for user authorization. It is designed for applications that can store confidential information and maintain state. A properly authorized web server application can access an API while the user interacts with the application or after the user has left the application.

Web server applications frequently also use service accounts (/identity/protocols/oauth2/service-account) to authorize API requests, particularly when calling Cloud APIs to access project-based data rather than user-specific data. Web server applications can use service accounts in conjunction with user authorization.

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 (#libraries).

Client libraries

The language-specific examples on this page use Google API Client Libraries (/api-client-library) to implement OAuth 2.0 authorization. To run the code samples, you must first install the client library for your language.

Google API Client Libraries for server-side applications are available for the following languages:

- Go (https://github.com/googleapis/google-api-go-client)
- Java (/api-client-library/java)
- .NET (/api-client-library/dotnet)
- Node.js (https://github.com/googleapis/google-api-nodejs-client)
- PHP (https://github.com/googleapis/google-api-php-client)
- Python (https://github.com/googleapis/google-api-python-client)
- Ruby (https://github.com/googleapis/google-api-ruby-client)

Important: The Google API client library for JavaScript (/api-client-library/javascript) and Sign In With Google (/identity/gsi/web/guides/overview) are **only** intended to handle OAuth 2.0 in the user's browser. If you want to use JavaScript on the server-side to manage OAuth 2.0 interactions with Google, consider using the Node.js (https://github.com/googleapis/google-api-nodejs-client) library on your back-end platform.

Prerequisites

Enable APIs for your project

Any application that calls Google APIs needs to enable those APIs in the API Console.

To enable an API for your project:

1. Open the API Library (https://console.developers.google.com/apis/library) in the Google API Console.
2. If prompted, select a project, or create a new one.

6. If prompted, read and accept the API's Terms of Service.

Create authorization credentials

Any application that uses OAuth 2.0 to access Google APIs must have authorization credentials that identify the application to Google's OAuth 2.0 server. The following steps explain how to create credentials for your project. Your applications can then use the credentials to access APIs that you have enabled for that project.

1. Go to the Credentials page (https://console.developers.google.com/apis/credentials).
2. Click **Create credentials > OAuth client ID**.
3. Select the **Web application** application type.
4. Fill in the form and click **Create**. Applications that use languages and frameworks like PHP, Java, Python, Ruby, and .NET must specify authorized **redirect URIs**. The redirect URIs are the endpoints to which the OAuth 2.0 server can send responses. These endpoints must adhere to Google's validation rules (#uri-validation).

   For testing, you can specify URIs that refer to the local machine, such as `http://localhost:8080`. With that in mind, please note that all of the examples in this document use `http://localhost:8080` as the redirect URI.

   We recommend that you design your app's auth endpoints (#protectauthcode) so that your application does not expose authorization codes to other resources on the page.

After creating your credentials, download the **client_secret.json** file from the API Console. Securely store the file in a location that only your application can access.

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

Identify access scopes

scopes that your app will need permission to access.

We also recommend that your application request access to authorization scopes via an incremental authorization (#incrementalAuth) process, in which your application requests access to user data in context. This best practice helps users to more easily understand why your application needs the access it is requesting.

The OAuth 2.0 API Scopes (/identity/protocols/oauth2/scopes) document contains a full list of scopes that you might use to access Google APIs.

If your public application uses scopes that permit access to certain user data, it must complete a verification process. If you see **unverified app** on the screen when testing your application, you must submit a verification request to remove it. Find out more about unverified apps (https://support.google.com/cloud/answer/7454865) and get answers to frequently asked questions about app verification (https://support.google.com/cloud/answer/9110914) in the Help Center.

Language-specific requirements

To run any of the code samples in this document, you'll need a Google account, access to the Internet, and a web browser. If you are using one of the API client libraries, also see the language-specific requirements below.

PHP (#php) Python (#python) Ruby (#ruby) Node.js (#node.js) HTTP/REST (#httprest)

To run the Python code samples in this document, you'll need:

- Python 2.6 or greater
- The pip (https://pypi.org/project/pip/) package management tool.
- The Google APIs Client Library for Python:

```
$ pip install --upgrade google-api-python-client
```

- The Flask Python web application framework.

```
$ pip install --upgrade flask
```

- The `requests` HTTP library.

```
$ pip install --upgrade requests
```

Obtaining OAuth 2.0 access tokens

The following steps show how your application interacts with Google's OAuth 2.0 server to obtain a user's consent to perform an API request on the user's behalf. Your application must have that consent before it can execute a Google API request that requires user authorization.

The list below quickly summarizes these steps:

1. Your application identifies the permissions it needs.
2. Your application redirects the user to Google along with the list of requested permissions.
3. The user decides whether to grant the permissions to your application.
4. Your application finds out what the user decided.
5. If the user granted the requested permissions, your application retrieves tokens needed to make API requests on the user's behalf.

Step 1: Set authorization parameters

parameters on that URL.

The tabs below define the supported authorization parameters for web server applications. The language-specific examples also show how to use a client library or authorization library to configure an object that sets those parameters.

PHP (#php) Python (#python) Ruby (#ruby) Node.js (#node.js) HTTP/REST (#httprest)

The following code snippet uses the `google-auth-oauthlib.flow` module to construct the authorization request.

The code constructs a `Flow` object, which identifies your application using information from the **client_secret.json** file that you downloaded after creating authorization credentials (#creatingcred). That object also identifies the scopes that your application is requesting permission to access and the URL to your application's auth endpoint, which will handle the response from Google's OAuth 2.0 server. Finally, the code sets the optional `access_type` and `include_granted_scopes` parameters.

For example, this code requests read-only, offline access to a user's Google Drive:

```
import google.oauth2.credentials
import google_auth_oauthlib.flow

# Required, call the from_client_secrets_file method to retrieve the cli
# client_secret.json file. The client ID (from that file) and access sco
# also use the from_client_config method, which passes the client config
# appeared in a client secrets file but doesn't access the file itself.)
flow = google_auth_oauthlib.flow.Flow.from_client_secrets_file(
    'client_secret.json',
    scopes=['https://www.googleapis.com/auth/drive.metadata.readonly'])

# Required, indicate where the API server will redirect the user after t
# the authorization flow. The redirect URI is required. The value must e
# match one of the authorized redirect URIs for the OAuth 2.0 client, wh
# configured in the API Console. If this value doesn't match an authoriz
```

```
    # Optional, enable incremental authorization. Recommended as a best
    include_granted_scopes='true',
    # Recommended, state value can increase your assurance that an incom
    # of an authentication request.
    state=sample_passthrough_value,
    # Optional, if your application knows which user is trying to authen
    # parameter to provide a hint to the Google Authentication Server.
    login_hint='hint@example.com',
    # Optional, set prompt to 'consent' will prompt the user for consent
    prompt='consent')
```

The Google authorization server supports the following query string parameters for web server applications:

Parameters	
`client_id`	**Required** The client ID for your application. You can find this value in the API Console Credentials page (https://console.developers.google.com/apis/credentials).
`redirect_uri`	**Required** Determines where the API server redirects the user after the user completes the authorization flow. The value must exactly match one of the authorized redirect URIs for the OAuth 2.0 client, which you configured in your client's API Console Credentials page (https://console.developers.google.com/apis/credentials). If this value doesn't match an authorized redirect URI for the provided `client_id` you will get a `redirect_uri_mismatch` error. Note that the `http` or `https` scheme, case, and trailing slash ('`/`') must all match.
`response_type`	**Required** Determines whether the Google OAuth 2.0 endpoint returns an authorization code

Parameters

	Scopes enable your application to only request access to the resources that it needs while also enabling users to control the amount of access that they grant to your application. Thus, there is an inverse relationship between the number of scopes requested and the likelihood of obtaining user consent. We recommend that your application request access to authorization scopes in context whenever possible. By requesting access to user data in context, via incremental authorization (#incrementalAuth), you help users to more easily understand why your application needs the access it is requesting.
`access_type`	**Recommended** Indicates whether your application can refresh access tokens when the user is not present at the browser. Valid parameter values are **`online`**, which is the default value, and **`offline`**. Set the value to **`offline`** if your application needs to refresh access tokens when the user is not present at the browser. This is the method of refreshing access tokens described later in this document. This value instructs the Google authorization server to return a refresh token *and* an access token the first time that your application exchanges an authorization code for tokens.

Parameters

`state`	**Recommended** Specifies any string value that your application uses to maintain state between your authorization request and the authorization server's response. The server returns the exact value that you send as a `name=value` pair in the URL query component (`?`) of the `redirect_uri` after the user consents to or denies your application's access request. You can use this parameter for several purposes, such as directing the user to the correct resource in your application, sending nonces, and mitigating cross-site request forgery. Since your `redirect_uri` can be guessed, using a `state` value can increase your assurance that an incoming connection is the result of an authentication request. If you generate a random string or encode the hash of a cookie or another value that captures the client's state, you can validate the response to additionally ensure that the request and response originated in the same browser, providing protection against attacks such as cross-site request forgery. See the OpenID Connect (/identity/protocols/oauth2/openid-connect#createxsrftoken) documentation for an example of how to create and confirm a `state` token.
`include_granted_scopes`	**Optional** Enables applications to use incremental authorization to request access to additional scopes in context. If you set this parameter's value to `true` and the authorization request is granted, then the new access token will also cover any scopes to which the user previously granted the application access. See the incremental authorization (#incrementalAuth) section for examples.
`enable_granular_consent`	**Optional** Defaults to `true`. If set to `false`, more granular Google Account permissions (https://developers.googleblog.com/2018/10/more-granular-google-account.html)

Parameters

	prefilling the email field in the sign-in form or by selecting the appropriate multi-login session. Set the parameter value to an email address or **sub** identifier, which is equivalent to the user's Google ID.
`prompt`	**Optional** A space-delimited, case-sensitive list of prompts to present the user. If you don't specify this parameter, the user will be prompted only the first time your project requests access. See Prompting re-consent (/identity/protocols/oauth2/openid-connect#re-consent) for more information. Possible values are: `none` — Do not display any authentication or consent screens. Must not be specified with other values. `consent` — Prompt the user for consent. `select_account` — Prompt the user to select an account.

Step 2: Redirect to Google's OAuth 2.0 server

Redirect the user to Google's OAuth 2.0 server to initiate the authentication and authorization process. Typically, this occurs when your application first needs to access the user's data. In the case of incremental authorization (#incrementalAuth), this step also occurs when your application first needs to access additional resources that it does not yet have permission to access.

PHP (#php) Python (#python) Ruby (#ruby) Node.js (#node.js) HTTP/REST (#httprest)

your application to access the requested scopes. The response is sent back to your application using the redirect URL you specified.

Step 3: Google prompts user for consent

In this step, the user decides whether to grant your application the requested access. At this stage, Google displays a consent window that shows the name of your application and the Google API services that it is requesting permission to access with the user's authorization credentials and a summary of the scopes of access to be granted. The user can then consent to grant access to one or more scopes requested by your application or refuse the request.

Your application doesn't need to do anything at this stage as it waits for the response from Google's OAuth 2.0 server indicating whether any access was granted. That response is explained in the following step.

Errors

Requests to Google's OAuth 2.0 authorization endpoint may display user-facing error messages instead of the expected authentication and authorization flows. Common error codes and suggested resolutions are listed below.

`admin_policy_enforced`

The Google Account is unable to authorize one or more scopes requested due to the policies of their Google Workspace administrator. See the Google Workspace Admin help article Control which third-party & internal apps access Google Workspace data (https://support.google.com/a/answer/7281227) for more information about how an administrator may restrict access to all scopes or sensitive and restricted scopes until access is explicitly granted to your OAuth client ID.

`disallowed_useragent`

(https://developer.android.com/reference/android/webkit/WebView). Developers should instead use Android libraries such as Google Sign-In for Android (/identity/sign-in/android) or OpenID Foundation's AppAuth for Android (https://openid.github.io/AppAuth-Android/).

Web developers may encounter this error when an Android app opens a general web link in an embedded user-agent and a user navigates to Google's OAuth 2.0 authorization endpoint from your site. Developers should allow general links to open in the default link handler of the operating system, which includes both Android App Links (https://developer.android.com/training/app-links) handlers or the default browser app. The Android Custom Tabs (https://developer.chrome.com/docs/android/custom-tabs/overview/) library is also a supported option.

`org_internal`

The OAuth client ID in the request is part of a project limiting access to Google Accounts in a specific Google Cloud Organization (https://cloud.google.com/resource-manager/docs/cloud-platform-resource-hierarchy#organizations). For more information about this configuration option see the User type (https://support.google.com/cloud/answer/10311615#user-type) section in the Setting up your OAuth consent screen help article.

`invalid_client`

The OAuth client secret is incorrect. Review the OAuth client configuration (/identity/protocols/oauth2/javascript-implicit-flow#creatingcred), including the client ID and secret used for this request.

`invalid_grant`

When refreshing an access token (#offline) or using incremental authorization (#incrementalAuth), the token may have expired or has been invalidated. Authenticate the

The `redirect_uri` passed in the authorization request does not match an authorized redirect URI for the OAuth client ID. Review authorized redirect URIs in the Google API Console Credentials page (https://console.developers.google.com/apis/credentials).

The `redirect_uri` parameter may refer to the OAuth out-of-band (OOB) flow that has been deprecated and is no longer supported. Refer to the migration guide (/identity/protocols/oauth2/resources/oob-migration) to update your integration.

`invalid_request`

There was something wrong with the request you made. This could be due to a number of reasons:

- The request was not properly formatted
- The request was missing required parameters
- The request uses an authorization method that Google doesn't support. Verify your OAuth integration uses a recommended integration method

Step 4: Handle 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. The authorization code or error message that is returned to the web server appears on the query string, as shown below:

An error response:

```
https://oauth2.example.com/auth?error=access_denied
```

see the authorization code in the URL. Scripts can read the URL directly, and the URL in the **`Referer`** HTTP header may be sent to any or all resources on the page.

Carefully consider whether 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.

Sample OAuth 2.0 server response

You can test this flow by clicking on the following sample URL, which requests read-only access to view metadata for files in your Google Drive:

`https://accounts.google.com/o/oauth2/v2/auth?`
`scope=https%3A//www.googleapis.com/auth/drive.metadata.readonly&`
`access_type=offline&`
`include_granted_scopes=true&`
`response_type=code&`
`state=`*`state_parameter_passthrough_value`***`&`**
`redirect_uri=`*`https%3A//oauth2.example.com/code`***`&`**
`client_id=`*`client_id`* (https://accounts.google.com/o/oauth2/v2/auth?scope=https%3A//www.goo

After completing the OAuth 2.0 flow, you should be redirected to `http://localhost/oauth2callback`, which will likely yield a `404 NOT FOUND` error unless your local machine serves a file at that address. The next step provides more detail about the information returned in the URI when the user is redirected back to your application.

Step 5: Exchange authorization code for refresh and access tokens

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

```
flow = google_auth_oauthlib.flow.Flow.from_client_secrets_file(
    'client_secret.json',
    scopes=['https://www.googleapis.com/auth/drive.metadata.readonly'],
    state=state)
flow.redirect_uri = flask.url_for('oauth2callback', _external=True)

authorization_response = flask.request.url
flow.fetch_token(authorization_response=authorization_response)

# Store the credentials in the session.
# ACTION ITEM for developers:
#     Store user's access and refresh tokens in your data store if
#     incorporating this code into your real app.
credentials = flow.credentials
flask.session['credentials'] = {
    'token': credentials.token,
    'refresh_token': credentials.refresh_token,
    'token_uri': credentials.token_uri,
    'client_id': credentials.client_id,
    'client_secret': credentials.client_secret,
    'scopes': credentials.scopes}
```

Errors

When exchanging the authorization code for an access token you may encounter the following error instead of the expected response. Common error codes and suggested resolutions are listed below.

`invalid_grant`

The supplied authorization code is invalid or in the wrong format. Request a new code by restarting the OAuth process (#creatingclient) to prompt the user for consent again.

Calling Google APIs

1. Build a service object for the API that you want to call. You build a service object by calling the `googleapiclient.discovery` library's `build` method with the name and version of the API and the user credentials: For example, to call version 3 of the Drive API:

```
from googleapiclient.discovery import build

drive = build('drive', 'v2', credentials=credentials)
```

2. Make requests to the API service using the interface provided by the service object (https://github.com/googleapis/google-api-python-client/blob/master/docs/start.md#building-and-calling-a-service). For example, to list the files in the authenticated user's Google Drive:

```
files = drive.files().list().execute()
```

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 metadata.

PHP (#php) Python (#python) Ruby (#ruby) Node.js (#node.js) HTTP/REST (#httprest)

This example uses the Flask (https://palletsprojects.com/p/flask/) framework. It runs a web application at `http://localhost:8080` that lets you test the OAuth 2.0 flow. If you go to that URL, you should see four links:

- **Revoke current credentials:** This link points to a page that revokes (#tokenrevoke) permissions that the user has already granted to the application.
- **Clear Flask session credentials:** This link clears authorization credentials that are stored in the Flask session. This lets you see what would happen if a user who had already granted permission to your app tried to execute an API request in a new session. It also lets you see the API response your app would get if a user had revoked permissions granted to your app, and your app still tried to authorize a request with a revoked access token.

Note: To run this code locally, you must have followed the directions in the prerequisites (#prerequisites) section, including setting **`http://localhost:8080`** as a valid redirect URI for your credentials and downloading the **client_secret.json** file for those credentials to your working directory.

```
# -*- coding: utf-8 -*-

import os
import flask
import requests

import google.oauth2.credentials
import google_auth_oauthlib.flow
import googleapiclient.discovery

# This variable specifies the name of a file that contains the OAuth 2.0
# information for this application, including its client_id and client_s
CLIENT_SECRETS_FILE = "client_secret.json"

# This OAuth 2.0 access scope allows for full read/write access to the
# authenticated user's account and requires requests to use an SSL conne
SCOPES = ['https://www.googleapis.com/auth/drive.metadata.readonly']
API_SERVICE_NAME = 'drive'
API_VERSION = 'v2'

app = flask.Flask(__name__)
```

```
@app.route('/test')
def test_api_request():
  if 'credentials' not in flask.session:
    return flask.redirect('authorize')

  # Load credentials from the session.
  credentials = google.oauth2.credentials.Credentials(
      **flask.session['credentials'])

  drive = googleapiclient.discovery.build(
      API_SERVICE_NAME, API_VERSION, credentials=credentials)

  files = drive.files().list().execute()

  # Save credentials back to session in case access token was refreshed.
  # ACTION ITEM: In a production app, you likely want to save these
  #              credentials in a persistent database instead.
  flask.session['credentials'] = credentials_to_dict(credentials)

  return flask.jsonify(**files)


@app.route('/authorize')
def authorize():
  # Create flow instance to manage the OAuth 2.0 Authorization Grant Flo
  flow = google_auth_oauthlib.flow.Flow.from_client_secrets_file(
      CLIENT_SECRETS_FILE, scopes=SCOPES)

  # The URI created here must exactly match one of the authorized redire
  # for the OAuth 2.0 client, which you configured in the API Console. I
  # value doesn't match an authorized URI, you will get a 'redirect_uri_
  # error.
  flow.redirect_uri = flask.url_for('oauth2callback', _external=True)

  authorization_url, state = flow.authorization_url(
      # Enable offline access so that you can refresh an access token wi
      # re-prompting the user for permission. Recommended for web server
```

```
@app.route('/oauth2callback')
def oauth2callback():
  # Specify the state when creating the flow in the callback so that it
  # verified in the authorization server response.
  state = flask.session['state']

  flow = google_auth_oauthlib.flow.Flow.from_client_secrets_file(
      CLIENT_SECRETS_FILE, scopes=SCOPES, state=state)
  flow.redirect_uri = flask.url_for('oauth2callback', _external=True)

  # Use the authorization server's response to fetch the OAuth 2.0 token
  authorization_response = flask.request.url
  flow.fetch_token(authorization_response=authorization_response)

  # Store credentials in the session.
  # ACTION ITEM: In a production app, you likely want to save these
  #              credentials in a persistent database instead.
  credentials = flow.credentials
  flask.session['credentials'] = credentials_to_dict(credentials)

  return flask.redirect(flask.url_for('test_api_request'))


@app.route('/revoke')
def revoke():
  if 'credentials' not in flask.session:
    return ('You need to <a href="/authorize">authorize</a> before ' +
            'testing the code to revoke credentials.')

  credentials = google.oauth2.credentials.Credentials(
    **flask.session['credentials'])

  revoke = requests.post('https://oauth2.googleapis.com/revoke',
      params={'token': credentials.token},
      headers = {'content-type': 'application/x-www-form-urlencoded'})

  status_code = getattr(revoke, 'status_code')
  if status_code == 200:
```

```
  return ('Credentials have been cleared.<br><br>' +
          print_index_table())


def credentials_to_dict(credentials):
  return {'token': credentials.token,
          'refresh_token': credentials.refresh_token,
          'token_uri': credentials.token_uri,
          'client_id': credentials.client_id,
          'client_secret': credentials.client_secret,
          'scopes': credentials.scopes}

def print_index_table():
  return ('<table>' +
          '<tr><td><a href="/test">Test an API request</a></td>' +
          '<td>Submit an API request and see a formatted JSON response.
          '    Go through the authorization flow if there are no stored
          '    credentials for the user.</td></tr>' +
          '<tr><td><a href="/authorize">Test the auth flow directly</a><
          '<td>Go directly to the authorization flow. If there are store
          '    credentials, you still might not be prompted to reauthori
          '    the application.</td></tr>' +
          '<tr><td><a href="/revoke">Revoke current credentials</a></td>
          '<td>Revoke the access token associated with the current user
          '    session. After revoking credentials, if you go to the tes
          '    page, you should see an <code>invalid_grant</code> error.
          '</td></tr>' +
          '<tr><td><a href="/clear">Clear Flask session credentials</a><
          '<td>Clear the access token currently stored in the user sessi
          '    After clearing the token, if you <a href="/test">test the
          '    API request</a> again, you should go back to the auth flo
          '</td></tr></table>')


if __name__ == '__main__':
  # When running locally, disable OAuthlib's HTTPs verification.
  # ACTION ITEM for developers:
  #     When running in production *do not* leave this option enabled.
  os.environ['OAUTHLIB_INSECURE_TRANSPORT'] = '1'
```

Google applies the following validation rules to redirect URIs in order to help developers keep their applications secure. Your redirect URIs must adhere to these rules. See RFC 3986 section 3 (https://tools.ietf.org/html/rfc3986#section-3) for the definition of domain, host, path, query, scheme and userinfo, mentioned below.

Validation rules

Scheme (https://tools.ietf.org/html/rfc3986#section-3.1)	Redirect URIs must use the HTTPS scheme, not plain HTTP. Localhost URIs (including localhost IP address URIs) are exempt from this rule.
Host (https://tools.ietf.org/html/rfc3986#section-3.2.2)	Hosts cannot be raw IP addresses. Localhost IP addresses are exempted from this rule.
Domain (https://tools.ietf.org/html/rfc1034)	• Host TLDs (Top Level Domains (https://tools.ietf.org/id/draft-liman-tld-names-00.html)) must belong to the public suffix list (https://publicsuffix.org/list/). • Host domains cannot be "`googleusercontent.com`". • Redirect URIs cannot contain URL shortener domains (e.g. `goo.gl`) unless the app owns the domain. Furthermore, if an app that owns a shortener domain chooses to redirect to that domain, that redirect URI must either contain "`/google-callback/`" in its path or end with "`/google-callback`".
Userinfo (https://tools.ietf.org/html/rfc3986#section-3.2.1)	Redirect URIs cannot contain the userinfo subcomponent.
Path (https://tools.ietf.org/html/rfc3986#section-3.3)	Redirect URIs cannot contain a path traversal (also called directory backtracking), which is represented by an "`/..`" or "`\..`" or their URL encoding.
Query (https://tools.ietf.org/html/rfc3986#section-3.4)	Redirect URIs cannot contain open redirects (https://tools.ietf.org/html/rfc6749#section-10.15).
Fragment	Redirect URIs cannot contain the fragment component.

Validation rules	
	digits) • Null characters (an encoded NULL character, e.g., `%00`, `%C0%80`)

Incremental authorization

In the OAuth 2.0 protocol, your app requests authorization to access resources, which are identified by scopes. It is considered a best user-experience practice to request authorization for resources at the time you need them. To enable that practice, Google's authorization server supports incremental authorization. This feature lets you request scopes as they are needed and, if the user grants permission for the new scope, returns an authorization code that may be exchanged for a token containing all scopes the user has granted the project.

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 `openid` and `profile` scopes 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.

To implement incremental authorization, you complete the normal flow for requesting an access token but make sure that the authorization request includes previously granted scopes. This approach allows your app to avoid having to manage multiple access tokens.

The following rules apply to an access token obtained from an incremental authorization:

- The token can be used to access resources corresponding to any of the scopes rolled into the new, combined authorization

another scope to the same application via a mobile client, the combined authorization would include both scopes.

- If you revoke a token that represents a combined authorization, access to all of that authorization's scopes on behalf of the associated user are revoked simultaneously.

Caution: choosing to include granted scopes will automatically add scopes previously granted by the user to your authorization request. A warning or error page may be displayed if your app is not currently approved to request all scopes that may be returned in the response. See Unverified apps (https://support.google.com/cloud/answer/7454865) for more information.

The language-specific code samples in Step 1: Set authorization parameters (#creatingclient) and the sample HTTP/REST redirect URL in Step 2: Redirect to Google's OAuth 2.0 server (#redirecting) all use incremental authorization. The code samples below also show the code that you need to add to use incremental authorization.

PHP (#php) Python Ruby (#ruby) Node.js (#node.js) HTTP/REST (#httprest)
(#python)

In Python, set the `include_granted_scopes` keyword argument to `true` to ensure that an authorization request includes previously granted scopes. It is very possible that `include_granted_scopes` will not be the *only* keyword argument that you set, as shown in the example below.

```
authorization_url, state = flow.authorization_url(
    # Enable offline access so that you can refresh an access token with
    # re-prompting the user for permission. Recommended for web server a
    access_type='offline',
    # Enable incremental authorization. Recommended as a best practice.
    include_granted_scopes='true')
```

Refreshing an access token (offline access)

access token as needed as long as you configure that object for offline access.

- If you are not using a client library, you need to set the `access_type` HTTP query parameter to `offline` when redirecting the user to Google's OAuth 2.0 server (#redirecting). In that case, Google's authorization server returns a refresh token when you exchange an authorization code (#exchange-authorization-code) for an access token. Then, if the access token expires (or at any other time), you can use a refresh token to obtain a new access token.

Requesting offline access is a requirement for any application that needs to access a Google API when the user is not present. For example, an app that performs backup services or executes actions at predetermined times needs to be able to refresh its access token when the user is not present. The default style of access is called `online`.

Server-side web applications, installed applications, and devices all obtain refresh tokens during the authorization process. Refresh tokens are not typically used in client-side (JavaScript) web applications.

PHP (#php) Python Ruby (#ruby) Node.js (#node.js) HTTP/REST (#httprest) (#python)

In Python, set the `access_type` keyword argument to `offline` to ensure that you will be able to refresh the access token without having to re-prompt the user for permission. It is very possible that `access_type` will not be the *only* keyword argument that you set, as shown in the example below.

```
authorization_url, state = flow.authorization_url(
    # Enable offline access so that you can refresh an access token with
    # re-prompting the user for permission. Recommended for web server a
    access_type='offline',
    # Enable incremental authorization. Recommended as a best practice.
    include_granted_scopes='true')
```

After a user grants offline access to the requested scopes, you can continue to use

access by visiting Account Settings (https://myaccount.google.com/permissions). See the Remove site or app access section of the Third-party sites & apps with access to your account (https://support.google.com/accounts/answer/3466521#remove-access) support document for more information.

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, removes an application, or the API resources required by an app have significantly changed. In other words, part of the removal process can include an API request to ensure the permissions previously granted to the application are removed.

PHP (#php) Python (#python) Ruby (#ruby) Node.js (#node.js) HTTP/REST (#httprest)

To programmatically revoke a token, make a request to `https://oauth2.googleapis.com/revoke` that includes the token as a parameter and sets the `Content-Type` header:

```
requests.post('https://oauth2.googleapis.com/revoke',
    params={'token': credentials.token},
    headers = {'content-type': 'application/x-www-form-urlencoded'})
```

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