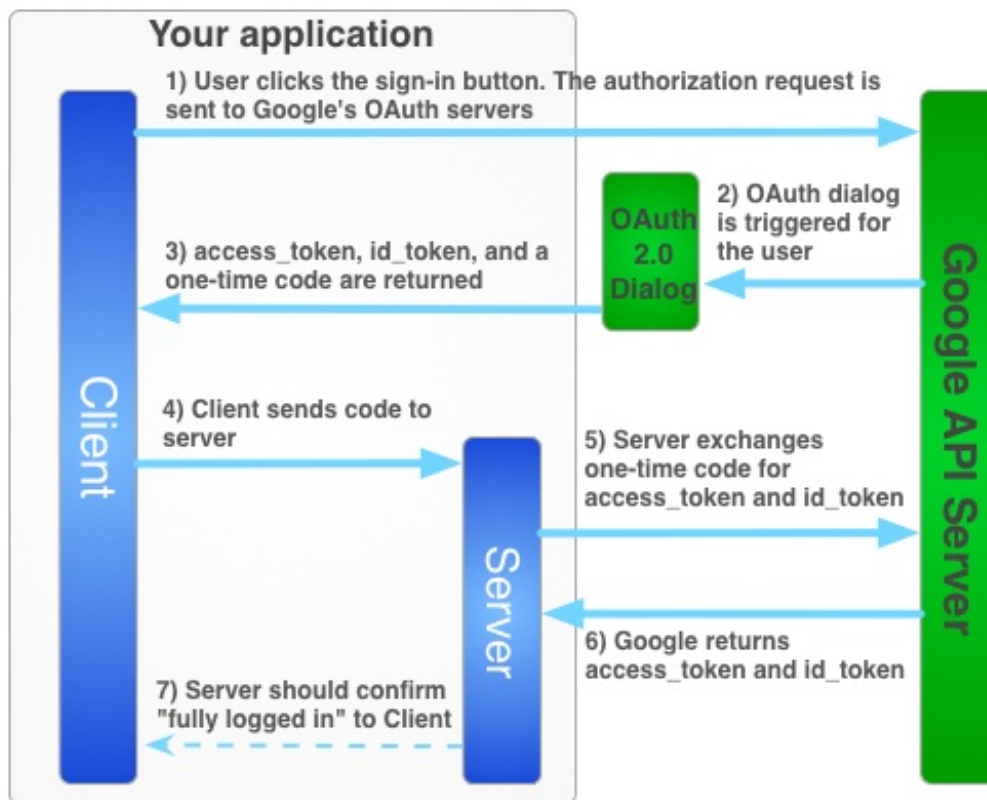


# Google Sign-In for server-side apps

[developers.google.com/identity/sign-in/web/server-side-flow](https://developers.google.com/identity/sign-in/web/server-side-flow)

To use Google services on behalf of a user when the user is offline, you must use a hybrid server-side flow where a user authorizes your app on the client side using the JavaScript API client and you send a special *one-time authorization code* to your server. Your server exchanges this one-time-use code to acquire its own access and refresh tokens from Google for the server to be able to make its own API calls, which can be done while the user is offline. This one-time code flow has security advantages over both a pure server-side flow and over sending access tokens to your server.

The sign-in flow for obtaining an access token for your server-side application is illustrated below.



From your application, the user clicks the sign-in button in your client app. This sends an authorization request to Google's authorization servers (1).

If your user hasn't authorized this app yet, the request triggers the OAuth 2.0 dialog, which pops up for the user (2) and asks them to grant your application the permissions listed by your scopes. If the user does so, the access\_token, id\_token, and a one-time code are returned to your client (3).

You can then send the one-time code from the sign-in button to your server (4). When the server has the code, the server can exchange it for an access\_token (5, 6) that can be stored

locally on the server side. The server can then make Google API calls independently of the client.

A final, optional step, involves sending a message from your server to your client, confirming that the user is now "fully logged in" (7).

One-time codes have several security advantages. With codes, Google provides tokens directly to your server without any intermediaries. Although we don't recommend leaking codes, they are very hard to use without your client secret. **Keep your client secret secret!**

## Implementing the one-time-code flow

---

The Google Sign-In button provides both an *access token* and an *authorization code*. The code is a one-time code that your server can exchange with Google's servers for an access token.



The following sample code demonstrates how to do the one-time-code flow.

Authenticating Google Sign-In with one-time-code flow requires you to:

### Step 1: Create a client ID and client secret

---

To create a client ID and client secret, create a Google API Console project, set up an OAuth client ID, and register your JavaScript origins:

1. Go to the [Google API Console](#) .
2. From the project drop-down, select an existing [project](#) , or create a new one by selecting **Create a new project**.  
**Note:** Use a single project to hold all platform instances of your app (Android, iOS, web, etc.), each with a different Client ID.
3. In the sidebar under "APIs & Services", select **Credentials**, then select the **OAuth consent screen** tab.
  1. Choose an **Email Address**, specify a **Product Name**, and press **Save**.
4. In the **Credentials** tab, select the **Create credentials** drop-down list, and choose **OAuth client ID**.
5. Under **Application type**, select **Web application**.

Register the origins from which your app is allowed to access the Google APIs, as follows. An origin is a unique combination of protocol, hostname, and port.

1. In the **Authorized JavaScript origins** field, enter the origin for your app. You can enter multiple origins to allow for your app to run on different protocols, domains, or subdomains. You cannot use wildcards. In the example below, the second URL could be a production URL.

```
http://localhost:8080
https://myproductionurl.example.com
```

2. The **Authorized redirect URI** field does not require a value. Redirect URIs are not used with JavaScript APIs.
3. Press the **Create** button.
6. From the resulting **OAuth client dialog box**, copy the **Client ID** . The Client ID lets your app access enabled Google APIs.

## Step 2: Include the Google platform library on your page

---

Include the following scripts that demonstrate an anonymous function that inserts a script into the DOM of this `index.html` web page.

```
<!-- The top of file index.html -->
<html itemscope itemtype="http://schema.org/Article">
<head>
  <!-- BEGIN Pre-requisites -->
  <script src="//ajax.googleapis.com/ajax/libs/jquery/1.8.2/jquery.min.js">
  </script>
  <script src="https://apis.google.com/js/client:platform.js?onload=start" async defer>
  </script>
  <!-- END Pre-requisites -->
```

## Step 3: Initialize the GoogleAuth object

---

Load the auth2 library and call `gapi.auth2.init()` to initialize the `GoogleAuth` object. Specify your client ID and the scopes you want to request when you call `init()` .

```
<!-- Continuing the <head> section -->
<script>
  function start() {
    gapi.load('auth2', function() {
      auth2 = gapi.auth2.init({
        client_id: 'YOUR_CLIENT_ID.apps.googleusercontent.com',
        // Scopes to request in addition to 'profile' and 'email'
        //scope: 'additional_scope'
      });
    });
  }
</script>
</head>
<body>
  <!-- ... -->
</body>
</html>
```

## Step 4: Add the sign-in button to your page

---

Add the sign-in button to your web page, and attach a click handler to call `grantOfflineAccess()` to start the one-time-code flow.

```
<!-- Add where you want your sign-in button to render -->
<!-- Use an image that follows the branding guidelines in a real app -->
<button id="signinButton">Sign in with Google</button>
<script>
  $('#signinButton').click(function() {
    // signInCallback defined in step 6.
    auth2.grantOfflineAccess().then(signInCallback);
  });
</script>
```

## Step 5: Sign in the user

---

The user clicks the sign-in button and grants your app access to the permissions that you requested. Then, the callback function that you specified in the `grantOfflineAccess().then()` method is passed a JSON object with an authorization code. For example:

```
{"code": "4/yU4cQZTMnnMtetyFcIWNItG32eKxxgXXX-Z4yyJJJo.4qHskT-
UtugceFc0ZR0NyF4z7U4UmAI"}
```

## Step 6: Send the authorization code to the server

---

The `code` is your one-time code that your server can exchange for its own access token and refresh token. You can only obtain a refresh token after the user has been presented an authorization dialog requesting offline access. You must store the refresh token that you retrieve for later use because subsequent exchanges will return `null` for the refresh token. This flow provides increased security over your standard OAuth 2.0 flow.

Access tokens are always returned with the exchange of a valid authorization code.

The following script defines a callback function for the sign-in button. When a sign-in is successful, the function stores the access token for client-side use and sends the one-time code to your server on the same domain.

```

<!-- Last part of BODY element in file index.html -->
<script>
function signInCallback(authResult) {
    if (authResult['code']) {

        // Hide the sign-in button now that the user is authorized, for example:
        $('#signinButton').attr('style', 'display: none');

        // Send the code to the server
        $.ajax({
            type: 'POST',
            url: 'http://example.com/storeauthcode',
            // Always include an `X-Requested-With` header in every AJAX request,
            // to protect against CSRF attacks.
            headers: {
                'X-Requested-With': 'XMLHttpRequest'
            },
            contentType: 'application/octet-stream; charset=utf-8',
            success: function(result) {
                // Handle or verify the server response.
            },
            processData: false,
            data: authResult['code']
        });
    } else {
        // There was an error.
    }
}
</script>

```

## Step 7: Exchange the authorization code for an access token

---

On the server, exchange the auth code for access and refresh tokens. Use the access token to call Google APIs on behalf of the user and, optionally, store the refresh token to acquire a new access token when the access token expires.

If you requested profile access, you also get an ID token that contains basic profile information for the user.

For example:

### Java

```

// (Receive authCode via HTTPS POST)

if (request.getHeader('X-Requested-With') == null) {
    // Without the `X-Requested-With` header, this request could be forged. Aborts.
}

// Set path to the Web application client_secret_*.json file you downloaded from the
// Google API Console: https://console.developers.google.com/apis/credentials

```

```

// You can also find your Web application client ID and client secret from the
// console and specify them directly when you create the
GoogleAuthorizationCodeTokenRequest
// object.
String CLIENT_SECRET_FILE = "/path/to/client_secret.json";

// Exchange auth code for access token
GoogleClientSecrets clientSecrets =
    GoogleClientSecrets.load(
        JacksonFactory.getDefaultInstance(), new FileReader(CLIENT_SECRET_FILE));
GoogleTokenResponse tokenResponse =
    new GoogleAuthorizationCodeTokenRequest(
        new NetHttpTransport(),
        JacksonFactory.getDefaultInstance(),
        "https://www.googleapis.com/oauth2/v4/token",
        clientSecrets.getDetails().getClientId(),
        clientSecrets.getDetails().getClientSecret(),
        authCode,
        REDIRECT_URI) // Specify the same redirect URI that you use with your
web
                        // app. If you don't have a web version of your app, you
can
                        // specify an empty string.
        .execute();

String accessToken = tokenResponse.getAccessToken();

// Use access token to call API
GoogleCredential credential = new GoogleCredential().setAccessToken(accessToken);
Drive drive =
    new Drive.Builder(new NetHttpTransport(), JacksonFactory.getDefaultInstance(),
credential)
        .setApplicationName("Auth Code Exchange Demo")
        .build();
File file = drive.files().get("appfolder").execute();

// Get profile info from ID token
GoogleIdToken idToken = tokenResponse.parseIdToken();
GoogleIdToken.Payload payload = idToken.getPayload();
String userId = payload.getSubject(); // Use this value as a key to identify a user.
String email = payload.getEmail();
boolean emailVerified = Boolean.valueOf(payload.getEmailVerified());
String name = (String) payload.get("name");
String pictureUrl = (String) payload.get("picture");
String locale = (String) payload.get("locale");
String familyName = (String) payload.get("family_name");
String givenName = (String) payload.get("given_name");

```

## Python

```

from apiclient import discovery
import httplib2
from oauth2client import client

# (Receive auth_code by HTTPS POST)

# If this request does not have `X-Requested-With` header, this could be a CSRF
if not request.headers.get('X-Requested-With'):
    abort(403)

# Set path to the Web application client_secret*.json file you downloaded from the
# Google API Console: https://console.developers.google.com/apis/credentials
CLIENT_SECRET_FILE = '/path/to/client_secret.json'

# Exchange auth code for access token, refresh token, and ID token
credentials = client.credentials_from_clientsecrets_and_code(
    CLIENT_SECRET_FILE,
    ['https://www.googleapis.com/auth/drive.appdata', 'profile', 'email'],
    auth_code)

# Call Google API
http_auth = credentials.authorize(httplib2.Http())
drive_service = discovery.build('drive', 'v3', http=http_auth)
appfolder = drive_service.files().get(fileId='appfolder').execute()

# Get profile info from ID token
userid = credentials.id_token['sub']
email = credentials.id_token['email']

```