OpenID Connect

(http://openid.net/certification/)

Google's OAuth 2.0 APIs can be used for both authentication and authorization. This document describes our OAuth 2.0 implementation for authentication, which conforms to the OpenID Connect (http://openid.net/connect/) specification, and is OpenID Certified (http://openid.net/certification/). The documentation found in Using OAuth 2.0 to Access Google APIs (https://developers.google.com/identity/protocols/OAuth2) also applies to this service. If you want to explore this protocol interactively, we recommend the Google OAuth 2.0 Playground (https://developers.google.com/oauthplayground/). To get help on Stack Overflow (http://stackoverflow.com/questions/tagged/google-oauth), tag your questions with 'google-oauth'.

Sign in with Google

(https://developers.google.com/identity/sign-in/) **Note:** If you want to provide a "Sign-in with Google" button for your website or app, we recommend using Google Sign-In (https://developers.google.com/identity/sign-in/), our sign-in client library that is built on the OpenID Connect protocol and provides OpenID Connect formatted ID Tokens.

Setting up OAuth 2.0

Before your application can use Google's OAuth 2.0 authentication system for user login, you must set up a project in the Google API Console (https://console.developers.google.com/) to obtain OAuth 2.0 credentials, set a redirect URI, and (optionally) customize the branding information that your users see on the user-consent screen. You can also use the API Console to create a service account, enable billing, set up filtering, and do other tasks. For more details, see the Google API Console Help (https://developers.google.com/console/help/console).

Obtain OAuth 2.0 credentials

You need OAuth 2.0 credentials, including a client ID and client secret, to authenticate users and gain access to Google's APIs.

To find your project's client ID and client secret, do the following:

1. Select an existing OAuth 2.0 credential or open the Credentials page (https://console.developers.google.com/apis/credentials).
2. If you haven't done so already, create your project's OAuth 2.0 credentials by clicking **Create credentials > OAuth client ID**, and providing the information needed to create the credentials.
3. Look for the **Client ID** in the **OAuth 2.0 client IDs** section. For details, click the client ID.

Set a redirect URI

The redirect URI that you set in the API Console determines where Google sends responses to your authentication requests (#sendauthrequest).

To find the redirect URIs for your OAuth 2.0 credentials, do the following:

1. Open the Credentials page (https://console.developers.google.com/apis/credentials) in the API Console.
2. If you haven't done so already, create your OAuth 2.0 credentials by clicking **Create credentials > OAuth client ID**.

3. After you create your credentials, view or edit the redirect URLs by clicking the client ID (for a web application) in the **OAuth 2.0 client IDs** section.

Customize the user consent screen

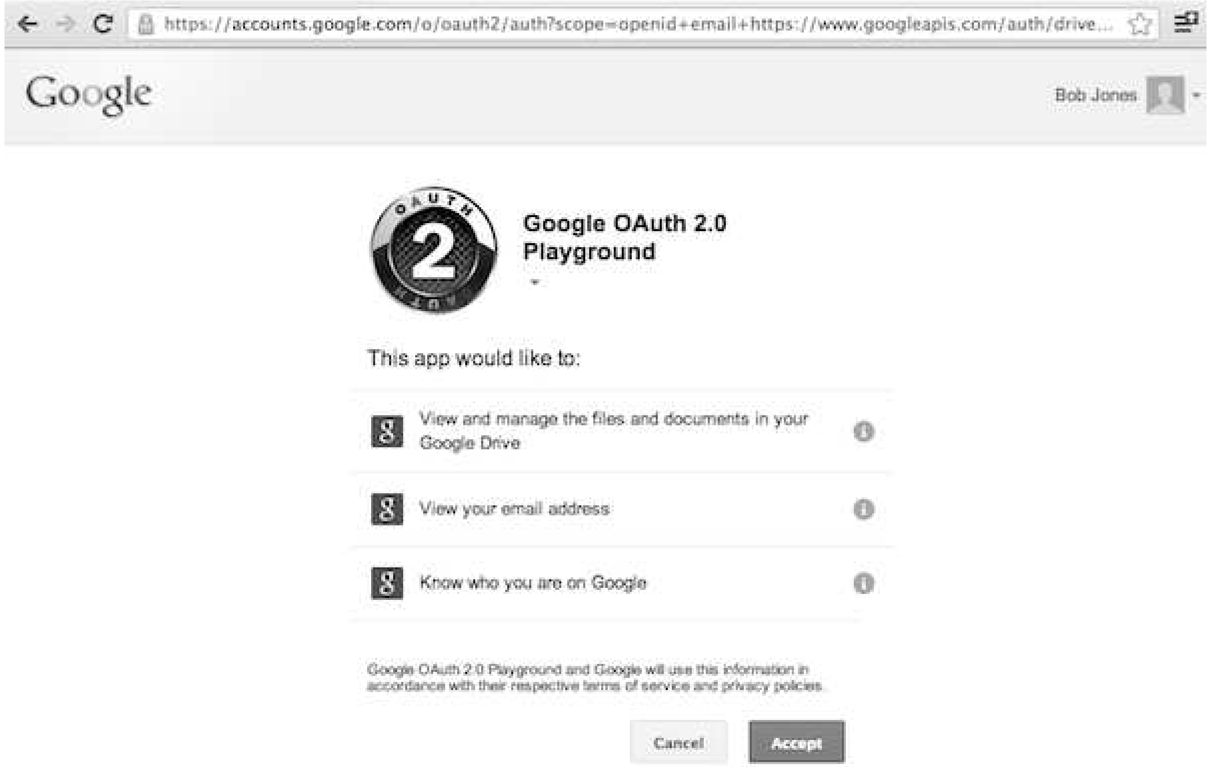
For your users, the OAuth 2.0 authentication experience includes a consent screen that describes the information that the user is releasing and the terms that apply. For example, when the user logs in, they might be asked to give your app access to their email address and basic account information. You request access to this information using the **scope** (#scope-param) parameter, which your app includes in its authentication request (#sendauthrequest). You can also use scopes to request access to other Google APIs.

The user consent screen also presents branding information such as your product name, logo, and a homepage URL. You control the branding information in the API Console.

To set up your project's consent screen, do the following:

1. Open the Consent Screen page (https://console.developers.google.com/apis/credentials/consent) in the Google API Console. If prompted, select a project or create a new one.
2. Fill out the form and click **Save**.

The following consent dialog shows what a user would see when a combination of OAuth 2.0 and Google Drive scopes are present in the request. (This generic dialog was generated using the Google OAuth 2.0 Playground (https://developers.google.com/oauthplayground/), so it does not include branding information that would be set in the API Console.)



https://accounts.google.com/o/oauth2/auth?scope=openid+email+https://www.googleapis.com/auth/drive...

Google OAuth 2.0 Playground

This app would like to:

- View and manage the files and documents in your Google Drive
- View your email address
- Know who you are on Google

Google OAuth 2.0 Playground and Google will use this information in accordance with their respective terms of service and privacy policies.

Accessing the service

Google and third parties provide libraries that you can use to take care of many of the implementation details of authenticating users and gaining access to Google APIs. Examples include Google Sign-In (https://developers.google.com/identity/sign-in/) and the Google client libraries (#libraries), which are available for a variety of platforms.

Note: Given the security implications of getting the implementation correct, we strongly encourage you to take advantage of a pre-written library or service (http://accountchooser.net/owners/next-level). Authenticating users properly is important to their and your safety and security, and using well-debugged code written by others is generally a best practice. For more information, see Client libraries (#libraries).

If you choose not to use a library, follow the instructions in the remainder of this document, which describes the HTTP request flows that underly the available libraries.

Authenticating the user

Authenticating the user involves obtaining an ID token and validating it. ID tokens are a standardized feature of OpenID Connect (http://openid.net/connect) designed for use in sharing identity assertions on the Internet.

The most commonly used approaches for authenticating a user and obtaining an ID token are called the "server" flow and the "implicit" flow. The server flow allows the back-end server of an application to verify the identity of the person using a browser or mobile device. The implicit flow is used when a client-side application (typically a JavaScript app running in the browser) needs to access APIs directly instead of via its back-end server.

This document describes how to perform the server flow for authenticating the user. The implicit flow is significantly more complicated because of security risks in handling and using tokens on the client side. If you need to implement an implicit flow, we highly recommend using Google Sign-In (https://developers.google.com/identity/sign-in/).

Server flow

Make sure you set up your app in the API Console (#appsetup) to enable it to use these protocols and authenticate your users. When a user tries to log in with Google, you need to:

1. Create an anti-forgery state token (#createxsrftoken)
2. Send an authentication request to Google (#sendauthrequest)

3. Confirm the anti-forgery state token (#confirmxsrftoken)
4. Exchange `code` for access token and ID token (#exchangecode)
5. Obtain user information from the ID token (#obtainuserinfo)
6. Authenticate the user (#authuser)

1. Create an anti-forgery state token

You must protect the security of your users by preventing request forgery attacks. The first step is creating a unique session token that holds state between your app and the user's client. You later match this unique session token with the authentication response returned by the Google OAuth Login service to verify that the user is making the request and not a malicious attacker. These tokens are often referred to as cross-site request forgery (CSRF (http://en.wikipedia.org/wiki/Cross-site_request_forgery)) tokens.

One good choice for a state token is a string of 30 or so characters constructed using a high-quality random-number generator. Another is a hash generated by signing some of your session state variables with a key that is kept secret on your back-end.

The following code demonstrates generating unique session tokens.

PHP JAVA PYTHON

You must download the Google APIs client library for PHP (https://code.google.com/p/google-api-php-client/) to use this sample.

```
// Create a state token to prevent request forgery.
// Store it in the session for later validation.
$state = sha1(openssl_random_pseudo_bytes(1024));
$app['session']->set('state', $state);
// Set the client ID, token state, and application name in the HTML while
// serving it.
return $app['twig']->render('index.html', array(
    'CLIENT_ID' => CLIENT_ID,
    'STATE' => $state,
    'APPLICATION_NAME' => APPLICATION_NAME
));
```

2. Send an authentication request to Google

The next step is forming an HTTPS `GET` request with the appropriate URI parameters. Note the use of HTTPS rather than HTTP in all the steps of this process; HTTP connections are refused. You should retrieve the base URI from the Discovery document (#discovery) using the key `authorization_endpoint`. The following discussion assumes the base URI is `https://accounts.google.com/o/oauth2/v2/auth`.

For a basic request, specify the following parameters:

- `client_id`, which you obtain from the API Console (https://console.developers.google.com/).
- `response_type`, which in a basic request should be `code`. (Read more at `response_type` (#response-type).)
- `scope`, which in a basic request should be `openid email`. (Read more at `scope` (#scope-param).)
- `redirect_uri` should be the HTTP endpoint on your server that will receive the response from Google. You specify this URI in the API Console (https://console.developers.google.com/).
- `state` should include the value of the anti-forgery unique session token, as well as any other information needed to recover the context when the user returns to your application, e.g., the starting URL. (Read more at `state` (#state-param).)
- `nonce` a random value generated by your app that enables replay protection when present.
- `login_hint` can be the user's email address or the `sub` string, which is equivalent to the user's Google ID. If you do not provide a `login_hint` and the user is currently logged in, the consent screen includes a request for approval to release the user’s email address to your app. (Read more at `login_hint` (#login-hint).)

- Use the **`openid.realm`** if you are migrating an existing application from OpenID 2.0 to OpenID Connect. For details, see Migrating off of OpenID 2.0 (https://developers.google.com/identity/protocols/OpenID2Migration).
- Use the **`hd`** parameter to optimize the OpenID Connect flow for users of a particular G Suite domain. (Read more at **`hd`** (#hd-param).)

Note: Only the most commonly used parameters are listed above. For a complete list, plus more details about all the parameters, see Authentication URI parameters (#authenticationuriparameters).

Here is an example of a complete OpenID Connect authentication URI, with line breaks and spaces for readability:

```
https://accounts.google.com/o/oauth2/v2/auth?
 client_id=424911365001.apps.googleusercontent.com&
 response_type=code&
 scope=openid%20email&
 redirect_uri=https://oauth2-login-demo.example.com/code&
 state=security_token%3D138r5719ru3e1%26url%3Dhttps://oauth2-login-demo.example.com/myHome&
 login_hint=jsmith@example.com&
 openid.realm=example.com&
 nonce=0394852-3190485-2490358&
 hd=example.com
```

Users are required to give consent if your app requests any new information about them, or if your app requests account access that they have not previously approved.

3. Confirm anti-forgery state token

The response is sent to the **`redirect_uri`** that you specified in the request (#sendauthrequest). All responses are returned in the query string, as shown below:

```
https://oa2cb.example.com/code?state=security_token%3D138r5719ru3e1%26url%3Dhttps://oa2cb.example.com/myHome&code=
```

On the server, you must confirm that the **`state`** received from Google matches the session token you created in Step 1 (#createxsrftoken). This round-trip verification helps to ensure that the user, not a malicious script, is making the request.

The following code demonstrates confirming the session tokens that you created in Step 1:

PHP JAVA PYTHON

You must download the Google APIs client library for PHP (https://code.google.com/p/google-api-php-client/) to use this sample.

```
// Ensure that there is no request forgery going on, and that the user
// sending us this connect request is the user that was supposed to.
if ($request->get('state') != ($app['session']->get('state'))) {
  return new Response('Invalid state parameter', 401);
}
```

4. Exchange `code` for access token and ID token

The response includes a **`code`** parameter, a one-time authorization code that your server can exchange for an access token and ID token. Your server makes this exchange by sending an HTTPS **`POST`** request. The **`POST`** request is sent to the token endpoint, which you should retrieve from the Discovery document (#discovery) using the key **`token_endpoint`**. The following discussion assumes the endpoint is **`https://www.googleapis.com/oauth2/v4/token`**. The request must include the following parameters in the **`POST`** body:

Field	Description
`code`	The authorization code that is returned from the initial request (#sendauthrequest).

`client_id`	The client ID that you obtain from the API Console (https://console.developers.google.com/), as described in Obtain OAuth 2.0 credentials (#getcredentials).
`client_secret`	The client secret that you obtain from the API Console, as described in Obtain OAuth 2.0 credentials (#getcredentials).
`redirect_uri`	The URI that you specify in the API Console, as described in Set a redirect URI (#setredirecturi).
`grant_type`	This field must contain a value of **`authorization_code`**, as defined in the OAuth 2.0 specification.

The actual request might look like the following example:

```
POST /oauth2/v4/token HTTP/1.1
Host: www.googleapis.com
Content-Type: application/x-www-form-urlencoded

code=4/P7q7W91a-oMsCeLvIaQm6bTrgtp7&
client_id=8819981768.apps.googleusercontent.com&
client_secret={client_secret}&
redirect_uri=https://oauth2-login-demo.example.com/code&
grant_type=authorization_code
```

A successful response to this request contains the following fields in a JSON array:

Field	Description
`access_token`	A token that can be sent to a Google API.
`id_token`	A JWT (http://openid.net/specs/draft-jones-json-web-token-07.html) that contains identity information about the user that is digitally signed by Google.
`expires_in`	The remaining lifetime of the access token.

`token_type`	Identifies the type of token returned. At this time, this field always has the value **`Bearer`**.
`refresh_token` (optional)	This field is only present if **`access_type=offline`** is included in the authentication request (#sendauthrequest). For details, see Refresh tokens (#refresh-tokens).

Note: There is a limit to the number of tokens per Google user account, and any authentication request above this limit might quietly invalidate an outstanding refresh token. For details, see Token expiration (https://developers.google.com/identity/protocols/OAuth2#expiration).

5. Obtain user information from the ID token

An ID Token is a JWT (JSON Web Token), that is, a cryptographically signed Base64-encoded JSON object. Normally, it is critical that you validate an ID token (#validatinganidtoken) before you use it, but since you are communicating directly with Google over an intermediary-free HTTPS channel and using your client secret to authenticate yourself to Google, you can be confident that the token you receive really comes from Google and is valid. If your server passes the ID token to other components of your app, it is extremely important that the other components validate the token (#validatinganidtoken) before using it.

Since most API libraries combine the validation with the work of decoding the base64 and parsing the JSON, you will probably end up validating the token anyway as you access the fields in the ID token.

An ID token's payload

An ID token is a JSON object containing a set of name/value pairs. Here’s an example, formatted for readability:

```
{"iss":"accounts.google.com",
 "at_hash":"HK6E_P6Dh8Y93mRNtsDB1Q",
 "email_verified":"true",
 "sub":"10769150350006150715113082367",
 "azp":"1234987819200.apps.googleusercontent.com",
 "email":"jsmith@example.com",
 "aud":"1234987819200.apps.googleusercontent.com",
 "iat":1353601026,
 "exp":1353604926,
 "nonce": "0394852-3190485-2490358",
 "hd":"example.com" }
```

Google ID Tokens may contain the following fields (known as *claims*):

Claim	Provided	Description
`iss`	always	The Issuer Identifier for the Issuer of the response. Always `https://accounts.google.com` or `accounts.google.com` for Google ID tokens.
`at_hash`		Access token hash. Provides validation that the access token is tied to the identity token. If the ID token is issued with an access token in the server flow, this is always included. This can be used as an alternate mechanism to protect against cross-site request forgery attacks, but if you follow Step 1 (#createxsrftoken) and Step 3 (#confirmxsrftoken) it is not necessary to verify the access token.
`email_verified`		True if the user's e-mail address has been verified; otherwise false.
`sub`	always	An identifier for the user, unique among all Google accounts and never reused. A Google account can have multiple emails at different points in time, but the `sub` value is never changed. Use `sub` within your application as the unique-identifier key for the user.
`azp`		The `client_id` of the authorized presenter. This claim is only needed when the party requesting the ID token is not the same as the audience of the ID token. This may be the case at Google for hybrid apps where a web application and Android app have a different `client_id` but share the same project.
`email`		The user's email address. This may not be unique and is not suitable for use as a primary key. Provided only if your scope included the string "email".

`profile`		The URL of the user's profile page. Might be provided when: • The request scope included the string "profile" • The ID token is returned from a token refresh When **`profile`** claims are present, you can use them to update your app's user records. Note that this claim is never guaranteed to be present.
`picture`		The URL of the user's profile picture. Might be provided when: • The request scope included the string "profile" • The ID token is returned from a token refresh When **`picture`** claims are present, you can use them to update your app's user records. Note that this claim is never guaranteed to be present.
`name`		The user's full name, in a displayable form. Might be provided when: • The request scope included the string "profile" • The ID token is returned from a token refresh When **`name`** claims are present, you can use them to update your app's user records. Note that this claim is never guaranteed to be present.
`aud`	always	Identifies the audience that this ID token is intended for. It must be one of the OAuth 2.0 client IDs of your application.
`iat`	always	The time the ID token was issued, represented in Unix time (integer seconds).
`exp`	always	The time the ID token expires, represented in Unix time (integer seconds).
`nonce`		The value of the nonce supplied by your app in the authentication request. You should enforce protection against replay attacks by ensuring it is presented only once.
`hd`		The hosted G Suite domain of the user. Provided only if the user belongs to a hosted domain.

6. Authenticate the user

After obtaining user information from the ID token, you should query your app's user database. If the user already exists in your database, you should start an application session for that user.

If the user does not exist in your user database, you should redirect the user to your new-user sign-up flow. You may be able to auto-register the user based on the information you receive from Google, or at the very least you may be able to pre-populate many of the fields that you require on your registration form. In addition to the information in the ID token, you can get additional user profile information (#obtaininguserprofileinformation) at our user profile endpoints.

Advanced topics

The following sections describe the Google OAuth 2.0 API in greater detail. This information is intended for developers with advanced requirements around authentication and authorization.

Access to other Google APIs

One of the advantages of using OAuth 2.0 for authentication is that your application can get permission to use other Google APIs (such as YouTube, Google Drive, Calendar, or Contacts) at the same time as you authenticate the user. To do this, include the other scopes that you need in the authentication request (#sendauthrequest) that you send to Google. For example, to add Google Drive access to your authentication request, pass a scope parameter of `openid email https://www.googleapis.com/auth/drive`. The user is prompted appropriately on the consent screen (#consentpageexperience). The access token that you receive back from Google allows you to access all the APIs related to the scopes you requested.

Note: If your application is asking for many scopes, the consent screen contains many lines of text. The more scopes your application requests, the less likely it is that the user will consent, so your application should ask only for the scopes it needs.

Refresh tokens

In your request for API access you can request a refresh token to be returned during the **`code exchange`** (#exchangecode). A refresh token provides your app continuous access to Google APIs while the user is not logged into your application. To request a refresh token, add **`access_type=offline`** to the authentication request (#sendauthrequest).

Considerations:

- Be sure to store the refresh token safely and permanently, because you can only obtain a refresh token the first time that you perform the code exchange flow.
- There are limits on the number of refresh token that are issued—one limit per client/user combination, and another per user across all clients. If your application requests too many refresh tokens, it may run into these limits, in which case older refresh tokens stop working.

For more information, see Offline Access (https://developers.google.com/identity/protocols/OAuth2WebServer#offline) and Using a refresh token (https://developers.google.com/identity/protocols/OAuth2WebServer#refresh).

Prompting re-consent

You can prompt the user to re-authorize your app by adding the **`prompt=consent`** parameter to the authentication request (#sendauthrequest). When **`prompt=consent`** is included, the consent screen is displayed every time the user logs into your app. For this reason, include **`prompt=consent`** only when necessary.

For more about the **`prompt`** parameter, see prompt (#prompt) in the URI parameter table.

Authentication URI parameters

The following table gives more complete descriptions of the parameters accepted by Google's OAuth 2.0 authentication API.

If `profile` is present, the ID token might (but is not guaranteed to) include a `profile` claim.

If `email` is present, the ID token includes `email` and `email_verified` claims.

In addition to these OpenID-specific scopes, your scope argument can also include other scope strings. All scope strings must be space-separated. For example, if you wanted per-file access to a user's Google Drive, your scope might be `openid profile email https://www.googleapis.com/auth/drive.file`.

For information about available login scopes, see Login scopes (https://developers.google.com/+/api/oauth#login-scopes). To see the available scopes for all Google APIs, visit the APIs Explorer (https://developers.google.com/apis-explorer/#p/).

Parameter	Required	Description
`client_id`	(Required)	The client ID string that you obtain from the API Console (https://console.developers.google.com/), as described in Obtain OAuth 2.0 credentials (#getcredentials).
`response_type`	(Required)	If the value is `code`, launches a Basic flow, requiring a `POST` to the token endpoint to obtain the tokens. If the value is `token id_token` or `id_token token`, launches an Implicit flow, requiring the use of Javascript at the redirect URI to retrieve tokens from the URI `#fragment`.
`scope`	(Required)	The scope value must begin with the string `openid` and then include `profile` or `email` or both.
`nonce`	(Required)	A random value generated by your app that enables replay protection.
`redirect_uri`	(Required)	Determines where the response is sent. The value of this parameter must exactly match one of the values that you set in the Google API Console (https://console.developers.google.com/) (including the HTTP or HTTPS scheme, case, and trailing '/', if any).
`state`	(Optional, but strongly	An opaque string that is round-tripped in the protocol; that is to say, it is returned as a URI parameter in the Basic flow, and in the URI `#fragment` in the Implicit flow.

	recommended)	The `state` can be useful for correlating requests and responses. Because 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 (#createxsrftoken) or encode the hash of some client state (e.g., a cookie) in this `state` variable, you can validate the response to additionally ensure that the request and response originated in the same browser. This provides protection against attacks such as cross-site request forgery.
`prompt`	(Optional)	A space-delimited list of string values that specifies whether the authorization server prompts the user for reauthentication and consent. The possible values are: • `none` The authorization server does not display any authentication or user consent screens; it will return an error if the user is not already authenticated and has not pre-configured consent for the requested scopes. You can use `none` to check for existing authentication and/or consent. • `consent` The authorization server prompts the user for consent before returning information to the client. • `select_account` The authorization server prompts the user to select a user account. This allows a user who has multiple accounts at the authorization server to select amongst the multiple accounts that they may have current sessions for. If no value is specified and the user has not previously authorized access, then the user is shown a consent screen.
`display`	(Optional)	An ASCII string value for specifying how the authorization server displays the authentication and consent user interface pages. The following values are specified, and accepted by the Google servers, but do not have any effect on its behavior: `page`, `popup`, `touch`, and `wap`.
`login_hint`	(Optional)	When your app knows which user it is trying to authenticate, it can provide this parameter as a hint to the authentication server. Passing this hint suppresses the account chooser and either pre-fill the email box on the sign-in form, or select the proper session (if the user is using multiple sign-in (https://support.google.com/accounts/answer/1721977)), which can help you avoid problems that occur if your app logs in the wrong user account. The value can be either an email address or the `sub` string, which is equivalent to the user's Google ID.

`access_type`	(Optional)	The allowed values are **`offline`** and **`online`**. The effect is documented in Offline Access (https://developers.google.com/identity/protocols/OAuth2WebServer#offline); if an access token is being requested, the client does not receive a refresh token unless **`offline`** is specified.
`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 (https://developers.google.com/accounts/docs/OAuth2WebServer#incrementalAuth). Note that you cannot do incremental authorization with the Installed App flow.
`openid.realm`	(Optional)	**`openid.realm`** is a parameter from the OpenID 2.0 protocol, not from OAuth 2.0. It is used in OpenID 2.0 requests to signify the URL-space for which an authentication request is valid. Use **`openid.realm`** if you are migrating an existing application from OpenID 2.0 to OpenID Connect. For more details, see Migrating off of OpenID 2.0 (https://developers.google.com/identity/protocols/OpenID2Migration).
`hd`	(Optional)	The **`hd`** (hosted domain) parameter streamlines the login process for G Suite hosted accounts. By including the domain of the G Suite user (for example, **`mycollege.edu`**), you can indicate that the account selection UI should be optimized for accounts at that domain. To optimize for G Suite accounts generally instead of just one domain, use an asterisk: **`hd=*`**. Don't rely on this UI optimization to control who can access your app, as client-side requests can be modified. Be sure to validate (#validatinganidtoken) that the returned ID token (#obtainuserinfo) has an **`hd`** claim value that matches what you expect (e.g. **`mycolledge.edu`**). Unlike the request parameter, the ID token claim is contained within a security token from Google, so the value can be trusted.

Validating an ID token

You need to validate all ID tokens on your server unless you know that they came directly from Google. For example, your server must verify as authentic any ID tokens it receives from your client apps.

The following are common situations where you might send ID tokens to your server:

- Sending ID tokens with requests that need to be authenticated. The ID tokens tell you the particular user making the request and for which client that ID token was granted.
- Sending ID tokens that contain OpenID 2.0 identifiers (`openid_id`) that need to be mapped to the Google ID (`sub`).

ID tokens are sensitive and can be misused if intercepted. You must ensure that these tokens are handled securely by transmitting them only over HTTPS and only via POST data or within request headers. If you store them on your server, you must also store them securely.

One thing that makes ID tokens useful is that fact that you can pass them around different components of your app. These components can use an ID token as a lightweight authentication mechanism authenticating the app and the user. But before you can use the information in the ID token or rely on it as an assertion that the user has authenticated, you **must** validate it.

Validation of an ID token requires several steps:

1. Verify that the ID token is properly signed by the issuer. Google-issued tokens are signed using one of the certificates found at the URI specified in the `jwks_uri` field of the discovery document (#discovery).
2. Verify that the value of `iss` in the ID token is equal to `https://accounts.google.com` or `accounts.google.com`.
3. Verify that the value of `aud` in the ID token is equal to your app's client ID.
4. Verify that the expiry time (`exp`) of the ID token has not passed.
5. If you passed a hd parameter (#hd-param) in the request, verify that the ID token has a `hd` claim that matches your G Suite hosted domain.

Steps 2 to 5 involve only string and date comparisons which are quite straight forward, so we won't detail them here.

The first step is more complex, and involves cryptographic signature checking. For debugging purposes, you can use Google's `tokeninfo` endpoint. Suppose your ID token's value is `XYZ123`. Then you would dereference the URI `https://www.googleapis.com/oauth2/v3/tokeninfo?id_token=XYZ123`. If the token is valid, the response would be its decoded JSON form.

This involves an HTTP round trip, introducing latency and the potential for network breakage. The `tokeninfo` endpoint is useful for debugging but for production purposes, retrieve Google's public keys from the keys endpoint and perform the validation locally. You should retrieve the keys URI from the Discovery document (#discovery) using the key `jwks_uri`.

Since Google changes its public keys only infrequently (on the order of once per day), you can cache them and, in the vast majority of cases, perform local validation much more efficiently than by using the `tokeninfo` endpoint. This requires retrieving and parsing certificates, and making the appropriate crypto calls to check the signature. Fortunately, there are well-debugged libraries available in a wide variety of languages to accomplish this (see jwt.io (https://jwt.io)).

Obtaining user profile information

To obtain additional profile information about the user, you can use the access token (which your application receives during the authentication flow (#authenticatingtheuser)) and the OpenID Connect (http://openid.net/specs/openid-connect-core-1_0.html) standard:

1. To be OpenID-compliant, you must include the `openid profile` (https://developers.google.com/+/api/oauth#profile) scope in your authentication request (#sendauthrequest).

   If you want the user's email address to be included, you can optionally request the `openid email` (https://developers.google.com/+/api/oauth#email) scope. To specify both `profile` and `email`, you can include the following parameter in your authentication request URI:

   ```
   scope=openid%20email%20profile
   ```

2. Add your access token to the authorization header and make an HTTPS `GET` request to the userinfo endpoint, which you should retrieve from the Discovery document (#discovery) using the key `userinfo_endpoint`. The response includes information about the user, as described in `people.getOpenIdConnect` (https://developers.google.com/+/api/latest/people/getOpenIdConnect). Users may choose to supply or withhold certain fields, so you might not get information for every field to which your scopes request access.

The Discovery document

The OpenID Connect protocol requires the use of multiple endpoints for authenticating users, and for requesting resources including tokens, user information, and public keys.

To simplify implementations and increase flexibility, OpenID Connect allows the use of a "Discovery document," a JSON document found at a well-known location containing key-value pairs which provide details about the OpenID Connect provider's configuration, including the URIs of the authorization, token, userinfo, and public-keys endpoints. The Discovery document for Google's OpenID Connect service may be retrieved from:

```
https://accounts.google.com/.well-known/openid-configuration
```

To use Google's OpenID Connect services, you should hard-code the Discovery-document URI (**`https://accounts.google.com/.well-known/openid-configuration`**) into your application. Your application fetches the document, then retrieves endpoint URIs from it as needed. For example, to authenticate a user, your code would retrieve the value of the **`authorization_endpoint`** key and use its value (**`https://accounts.google.com/o/oauth2/auth`** in the example below) as the base URI for authentication requests that are sent to Google.

Here is an example of such a document; the field names are those specified in OpenID Connect Discovery 1.0 (http://openid.net/specs/openid-connect-discovery-1_0.html#ProviderMetadata) (refer to that document for their meanings). The values are purely illustrative and might change, although they are copied from from a recent version of the actual Google Discovery document:

```
{
 "issuer": "https://accounts.google.com",
 "authorization_endpoint": "https://accounts.google.com/o/oauth2/v2/auth",
 "token_endpoint": "https://www.googleapis.com/oauth2/v4/token",
 "userinfo_endpoint": "https://www.googleapis.com/oauth2/v3/userinfo",
 "revocation_endpoint": "https://accounts.google.com/o/oauth2/revoke",
 "jwks_uri": "https://www.googleapis.com/oauth2/v3/certs",
 "response_types_supported": [
  "code",
  "token",
  "id_token",
  "code token",
  "code id_token",
  "token id_token",
  "code token id_token",
  "none"
 ],
 "subject_types_supported": [
  "public"
 ],
 "id_token_signing_alg_values_supported": [
  "RS256"
 ],
 "scopes_supported": [
  "openid",
  "email",
  "profile"
 ],
 "token_endpoint_auth_methods_supported": [
  "client_secret_post",
  "client_secret_basic"
 ],
 "claims_supported": [
  "aud",
  "email",
  "email_verified",
  "exp",
  "family_name",
  "given_name",
  "iat",
  "iss",
  "locale",
  "name",
  "picture",
  "sub"
 ],
 "code_challenge_methods_supported": [
  "plain",
  "S256"
 ]
}
```

You may be able to avoid an HTTP round-trip by caching the values from the Discovery document. Standard HTTP caching headers are used and should be respected.

Client libraries

The following client libraries make implementing OAuth 2.0 simpler by integrating with popular frameworks:

- Google APIs Client Library for Java (https://code.google.com/p/google-api-java-client/wiki/OAuth2)
- Google APIs Client Library for Python (https://developers.google.com/api-client-library/python/guide/aaa_oauth)
- Google APIs Client Library for .NET (https://developers.google.com/api-client-library/dotnet/guide/aaa_oauth)
- Google APIs Client Library for Ruby (//code.google.com/p/google-api-ruby-client/wiki/OAuth2)
- Google APIs Client Library for PHP (//code.google.com/p/google-api-php-client/wiki/OAuth2)

- OAuth 2.0 Library for Google Web Toolkit (//code.google.com/p/gwt-oauth2)
- Google Toolbox for Mac OAuth 2.0 Controllers (//code.google.com/p/gtm-oauth2/)

OpenID Connect compliance

Google's OAuth 2.0 authentication system supports the required features (http://openid.net/specs/openid-connect-core-1_0.html#ServerMTI) of the OpenID Connect Core (http://openid.net/specs/openid-connect-core-1_0.html) specification. Any client which is designed to work with OpenID Connect should interoperate with this service (with the exception of the OpenID Request Object (http://openid.net/specs/openid-connect-core-1_0.html#RequestObject)).

Zuletzt aktualisiert: Juli 12, 2018