Using OAuth 2.0 to Access Google APIs

Google APIs use the OAuth 2.0 protocol for authentication and authorization. Google supports common OAuth 2.0 scenarios such as those for web server, installed, and client-side applications.

To begin, obtain OAuth 2.0 client credentials from the Google Developers Console. Then your client application requests an access token from the Google Authorization Server, extracts a token from the response, and sends the token to the Google API that you want to access. For an interactive demonstration of using OAuth 2.0 with Google (including the option to use your own client credentials), experiment with the OAuth 2.0 Playground.

This page gives an overview of the OAuth 2.0 authorization scenarios that Google supports, and provides links to more detailed content. For details about using OAuth 2.0 for authentication, see OpenID Connect.

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.

Basic steps

All applications follow a basic pattern when accessing a Google API using OAuth 2.0. At a high level, you follow four steps:

1. Obtain OAuth 2.0 credentials from the Google Developers Console.

Visit the Google Developers Console to obtain OAuth 2.0 credentials such as a client ID and client secret that are known to both Google and your application. The set of values varies based on what type of application you are building. For example, a JavaScript application does not require a secret, but a web server application does.

2. Obtain an access token from the Google Authorization Server.

Before your application can access private data using a Google API, it must obtain an access token that grants access to that API. A single access token can grant varying degrees of access to multiple APIs. A variable parameter called `scope` controls the set of resources and operations that an access token permits. During the access-token request, your application sends one or more values in the `scope` parameter.

There are several ways to make this request, and they vary based on the type of application you are building. For example, a JavaScript application might request an access token using a browser redirect to Google, while an application installed on a device that has no browser uses web service requests.

Some requests require an authentication step where the user logs in with their Google account. After logging in, the user is asked whether they are willing to grant the permissions that your application is requesting. This process is called *user consent*.

If the user grants the permission, the Google Authorization Server sends your application an access token (or an authorization code that your application can use to obtain an access token). If the user does not grant the permission, the server returns an error.

It is generally a best practice to request scopes incrementally, at the time access is required, rather than up front. For example, an app that wants to support purchases should not request Google Wallet access until the user presses the “buy” button; see Incremental authorization.

3. Send the access token to an API.

After an application obtains an access token, it sends the token to a Google API in an HTTP authorization header. It is possible to send tokens as URI query-string parameters, but we don't recommend it, because URI parameters can end up in log files that are not completely secure. Also, it is good REST practice to avoid creating unnecessary URI parameter names.

Access tokens are valid only for the set of operations and resources described in the `scope` of the token request. For example, if

an access token is issued for the Google+ API, it does not grant access to the Google Contacts API. You can, however, send that access token to the Google+ API multiple times for similar operations.

4. Refresh the access token, if necessary.

Access tokens have limited lifetimes. If your application needs access to a Google API beyond the lifetime of a single access token, it can obtain a refresh token. A refresh token allows your application to obtain new access tokens.

Note: Save refresh tokens in secure long-term storage and continue to use them as long as they remain valid. Limits apply to the number of refresh tokens that are issued per client-user combination, and per user across all clients, and these limits are different. If your application requests enough refresh tokens to go over one of the limits, older refresh tokens stop working.

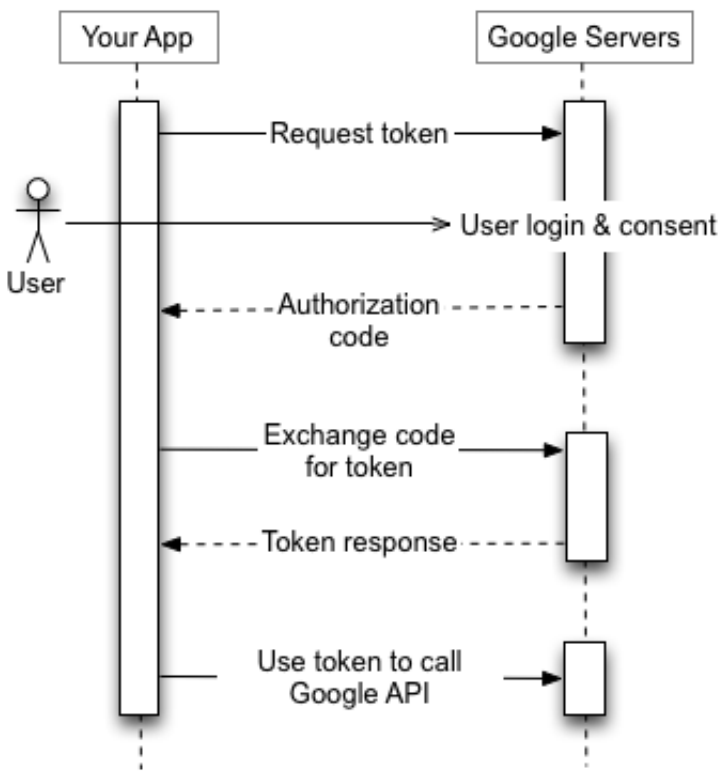
Scenarios

Web server applications

The Google OAuth 2.0 endpoint supports web server applications that use languages and frameworks such as PHP, Java, Python, Ruby, and ASP.NET.

The authorization sequence begins when your application redirects a browser to a Google URL; the URL includes query parameters that indicate the type of access being requested. Google handles the user authentication, session selection, and user consent. The result is an authorization code, which the application can exchange for an access token and a refresh token.

The application should store the refresh token for future use and use the access token to access a Google API. Once the access token expires, the application uses the refresh token to obtain a new one.



For details, see Using OAuth 2.0 for Web Server Applications.

Installed applications

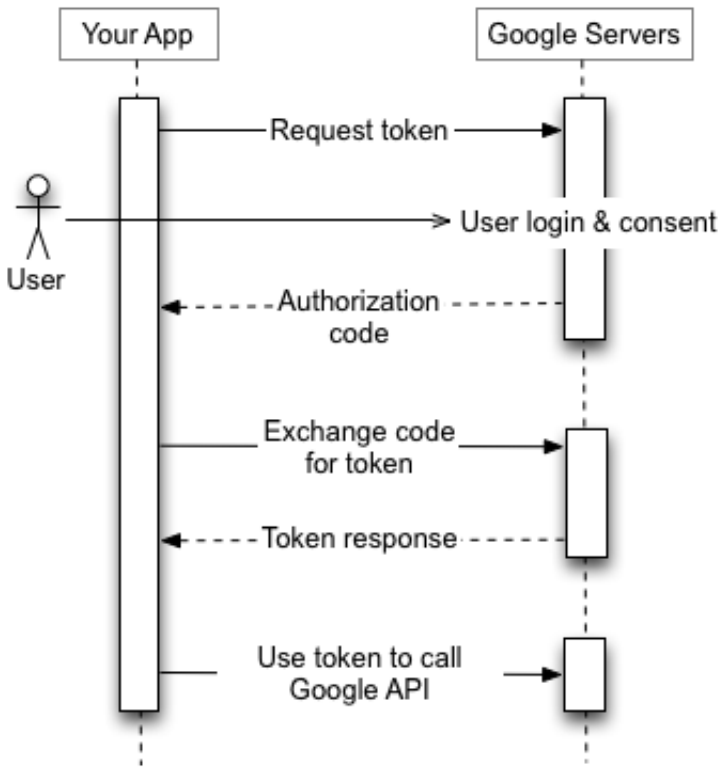
The Google OAuth 2.0 endpoint supports applications that are installed on devices such as computers, mobile devices, and tablets. When you create a client ID through the Google Developers Console, specify that this is an Installed application, then select Android, Chrome, iOS, or "Other" as the application type.

The process results in a client ID and, in some cases, a client secret, which you embed in the source code of your application. (In this context, the client secret is obviously not treated as a secret.)

The authorization sequence begins when your application redirects a browser to a Google URL; the URL includes query parameters that indicate the type of access being requested. Google handles the user authentication, session selection, and user consent. The result is an authorization code, which the application can exchange for an access token and a refresh token.

The application should store the refresh token for future use and use the access token to access a Google API. Once the access

token expires, the application uses the refresh token to obtain a new one.



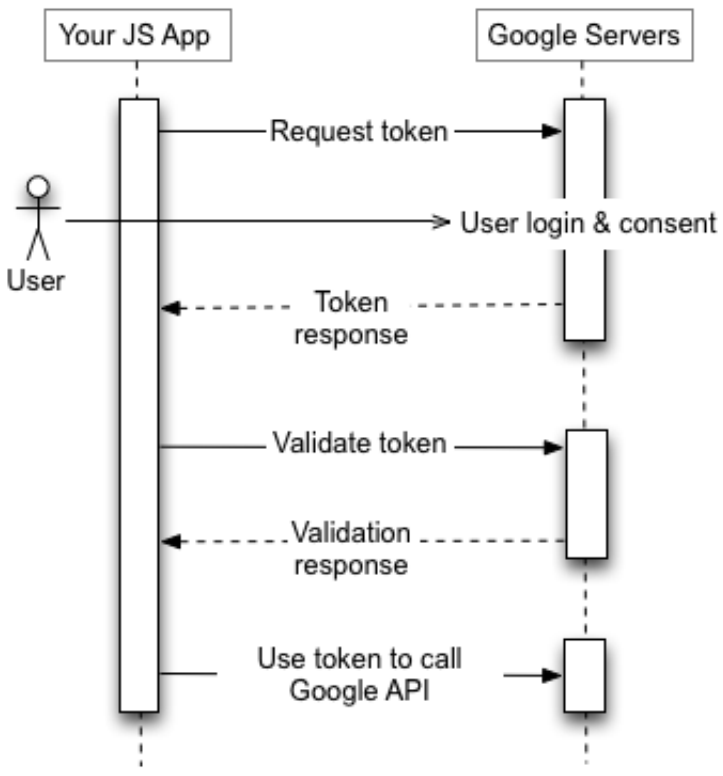
For details, see Using OAuth 2.0 for Installed Applications.

Client-side (JavaScript) applications

The Google OAuth 2.0 endpoint supports JavaScript applications that run in a browser.

The authorization sequence begins when your application redirects a browser to a Google URL; the URL includes query parameters that indicate the type of access being requested. Google handles the user authentication, session selection, and user consent.

The result is an access token, which the client should validate before including it in a Google API request. When the token expires, the application repeats the process.



For details, see Using OAuth 2.0 for Client-side Applications.

Applications on limited-input devices

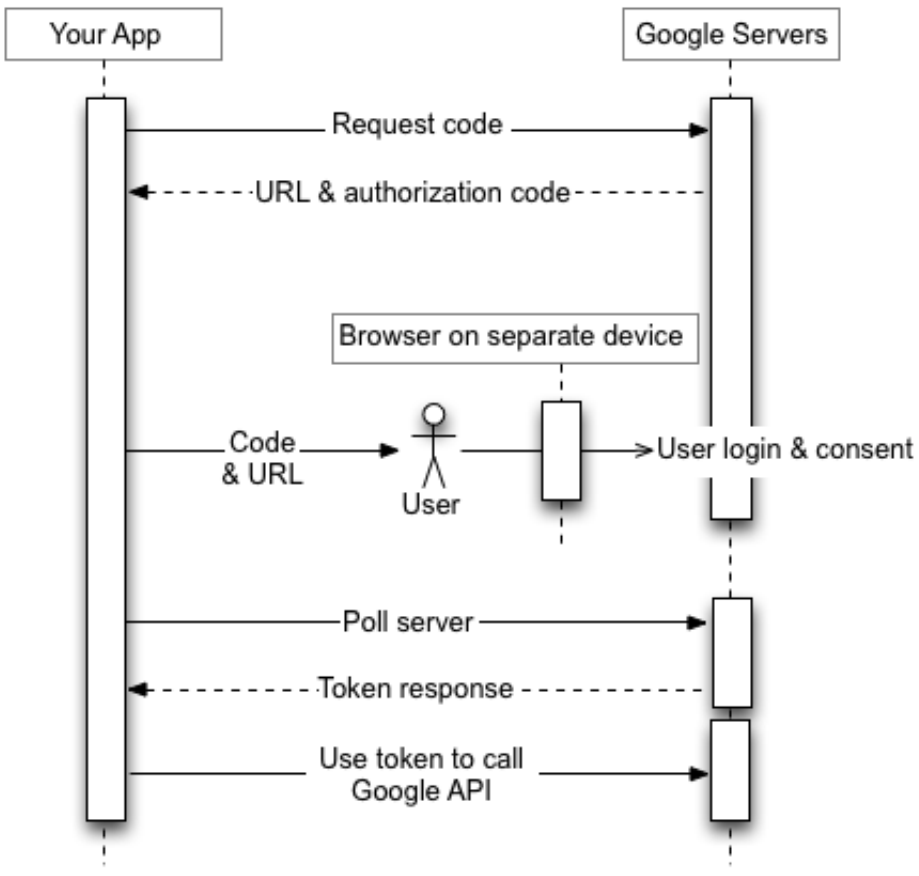
The Google OAuth 2.0 endpoint supports applications that run on limited-input devices such as game consoles, video cameras, and printers.

The authorization sequence begins with the application making a web service request to a Google URL for an authorization code. The response contains several parameters, including a URL and a code that the application shows to the user.

The user obtains the URL and code from the device, then switches to a separate device or computer with richer input capabilities. The user launches a browser, navigates to the specified URL, logs in, and enters the code.

Meanwhile, the application polls a Google URL at a specified interval. After the user approves access, the response from the

Google server contains an access token and refresh token. The application should store the refresh token for future use and use the access token to access a Google API. Once the access token expires, the application uses the refresh token to obtain a new one.



For details, see Using OAuth 2.0 for Devices.

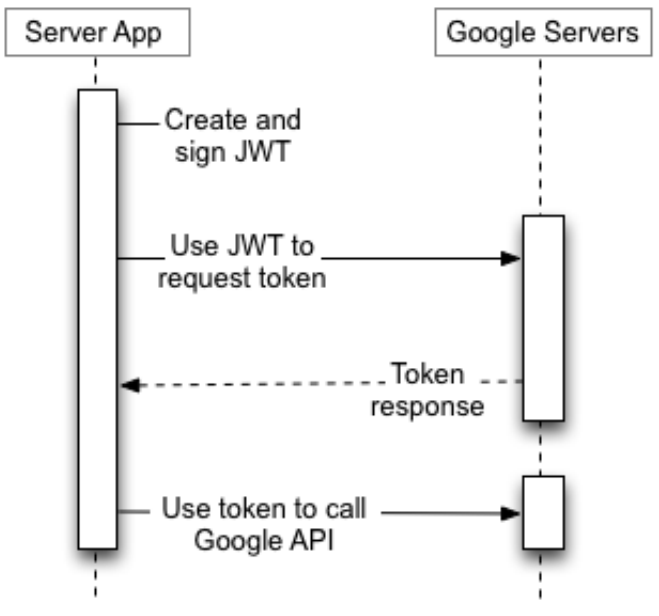
Service accounts

Google APIs such as the Prediction API and Google Cloud Storage can act on behalf of your application without accessing user information. In these situations your application needs to prove its own identity to the API, but no user consent is necessary. Similarly, in enterprise scenarios, your application can request delegated access to some resources.

For these types of server-to-server interactions you need a **service account**, which is an account that belongs to your application instead of to an individual end-user. Your application calls Google APIs on behalf of the service account, and user consent is not required. (In non-service-account scenarios, your application calls Google APIs on behalf of end-users, and user consent is sometimes required.)

Note: These service-account scenarios require applications to create and cryptographically sign JSON Web Tokens (JWTs). We strongly encourage you to use a library to perform these tasks. If you write this code without using a library that abstracts token creation and signing, you might make errors that would have a severe impact on the security of your application. For a list of libraries that support this scenario, see the service-account documentation.

A service account's credentials, which you obtain from the Google Developers Console, include a generated email address that is unique, a client ID, and at least one public/private key pair. You use the client ID and one private key to create a signed JWT and construct an access-token request in the appropriate format. Your application then sends the token request to the Google OAuth 2.0 Authorization Server, which returns an access token. The application uses the token to access a Google API. When the token expires, the application repeats the process.



For details, see the service-account documentation.

Note: **Although you can use service accounts in applications that run from a Google Apps domain, service accounts are not members of your Google Apps account and aren't subject to domain policies set by Google Apps administrators. For example, a policy set in the Google Apps admin console to restrict the ability of Apps end users to share documents outside of the domain would not apply to service accounts.**

Token expiration

You should write your code to anticipate the possibility that a granted token might no longer work. A token might stop working for one of these reasons:

- The user has revoked access.
- The token has not been used for six months.
- The user changed passwords and the token contains Gmail, Calendar, Contacts, or Hangouts scopes.
- The user account has exceeded a certain number of token requests.

There is currently a limit of 25 refresh tokens per user account per client. If the limit is reached, creating a new token automatically invalidates the oldest token without warning. This limit does not apply to service accounts.

There is also a larger limit on the total number of tokens a user account or service account can have across all clients. Most normal users won't exceed this limit but a developer's test account might.

If you need to authorize multiple programs, machines, or devices, one workaround is to limit the number of clients that you authorize per user account to 15 or 20. If you are a Google Apps admin, you can create additional admin users and use them to authorize some of the clients.

Client libraries

The following client libraries integrate with popular frameworks, which makes implementing OAuth 2.0 simpler. More features will be added to the libraries over time.

- Google API Client Library for Java
- Google API Client Library for Python
- Google API Client Library for Go
- Google API Client Library for .NET
- Google API Client Library for Ruby
- Google API Client Library for PHP
- Google API Client Library for JavaScript
- Google Toolbox for Mac OAuth 2.0 Controllers

Last updated January 4, 2016.