CHAPTER 3

Using Google APIs

Virtually all of Google's products are built according to an API-first philosophy. This approach encompasses both Cloud Platform products like Google Compute Engine and consumer-facing products like Google Maps. On Google Cloud Platform, although Google makes it easy to consume products using either the web-based Developers Console or the console-based `gcloud` tool, the real power of the platform is best appreciated by using the core building blocks: the product APIs. In addition, certain developer-targeted products are made available solely through APIs.

API access is subject to access control. Access control comprises authentication and authorization and is collectively referred to as *Auth*. In order to consume an API, an application should be properly authenticated and authorized. The level of access control depends on whether the application is requesting access just to a public API (for example Translate API) or to an API that has access to protected information (for example Cloud Storage). In the first case, the application needs to be authenticated; in the second case, the application needs to be both authenticated and authorized to access the user's data.

Google supports OpenID Connect for authentication and OAuth 2.0 for authorization. OpenID Connect is also known as OAuth for authentication. Google uses the OAuth 2.0 open-standard protocol with Bearer tokens[1] for both web and installed applications. This chapter first covers the essentials of OAuth 2.0 required to access Google APIs. All Google APIs are available as REST APIs, so it is easy to consume them through HTTP(S) requests.

In addition, Google provides application support libraries for many of its APIs in several programming languages. This makes it easier to develop client applications that consume Google APIs and simpler for Google APIs to be deeply integrated with the respective programming language's features and capabilities. For information about the availability of client libraries in your programming language of interest, see `https://developers.google.com/accounts/docs/OAuth2#libraries`. To aid your understanding of both Auth and API access, in this chapter's example you use a relatively simple API from Cloud Platform—the Google Translate API—and access it using both REST APIs and client libraries.

Auth Essentials

Every application that attempts to access Google APIs needs to prove its identity. The level of identification depends on the access scope requested by the application. For example, for APIs like Google Translate that do not access application or users' private data, the level of identification is a simple API key. An application that needs access to protected information must use an OAuth 2.0–based identification process. In addition to that, there are different types of authorization in OAuth 2.0. 3-legged flows are common when requests

[1]Bearer tokens are a type of access tokens. Access tokens represent credentials that provide third-party clients with the necessary rights to access protected information. These tokens are issued by an authorization server that has the approval of the resource owner.

need to be done on behalf of a concrete user. This type of flow normally requires user interaction to obtain access. Because of that, this flow is suitable for applications that have a user interface, like web server or mobile applications. On the other hand, 2-legged flows are used by clients with limited capabilities –e.g.: clients that are not able to store secret keys privately like JavaScript client side applications– or in situations where requests are sent on behalf of applications, hence there is no need for user consent –e.g.: server to server communication. For example, the Prediction API reads data from files stored in Google Cloud Storage and so uses OAuth 2.0 to request access to the API. Conversely, the Translate API does not need to access private data from users or the application itself, so the only authentication mechanism needed is an API key. This is used by Google to measure usage of the API. Let's examine the difference between using an API key and user/application specific OAuth 2.0.

■ **Note** In order to keep tokens, secrets, and keys safe, it is strongly encouraged that you operate over secure connections using SSL. Some endpoints will reject requests if they are run over HTTP.

API Keys

An API key has the following form:

```
AIzaSyCySn7SBWYPCMEM_2CBJgyDGO5qNkiHtTA
```

This key is all you need to authenticate requests against services that do not access users' private data or specific permissions like Directions API, such as the Directions API. Here is an example of how to request directions for the Via Regia—from Moscow to Berlin—using the Directions API[2]:

```
GET https://maps.googleapis.com/maps/api/directions/json?
origin=Moscow&
destination=Santiago%20de%20Compostela&
key=AIzaSyCySn7SBWYPCMEM_2CBJgyDGO5qNkiHtTA
```

The key used in the previous example is not valid. Because of that, if you try to run a request using the previous URL, it fails stating that access is denied for the key provided. To obtain a new API key, do the following:

1. Go to the Developers Console in Google: https://console.developers.google.com.
2. Select a project, or create a new one.
3. Go to Credentials, and create a new API key under Public API access.

[2]Via Regia is a historic road dating back to the Middle Ages that travels from Moscow to Santiago de Compostela (http://en.wikipedia.org/wiki/Via_Regia).

When you do that, you are offered four different options or types of keys to create. Choose the type that fits your needs, depending on the platform or system you are using to access an API:

- Choose a *server key* if your application runs on a server. Keep this private in order to avoid quota theft.

  When you select this method, you can specify the IP addresses of the allowed clients that you expect to connect to this server. You do that by adding a query parameter with the IP address: `userIp=<user-ip-address>`. If access is started by your server—for example, when running a cron job—you can provide a `quotaUser` parameter with a value limited to 40 characters. For example: `quotaUser=myemail@gmail.com`. These two parameters are also used to associate usage of an API with the quota of a specific user.

- Use a *browser key* if your application runs on a web client. When you select this type of key, you must specify a list of allowed `Referers`. Requests coming from URLs that do not match are rejected. You can use wildcards at the beginning or end of each pattern. For example: `www.domain.com`, `*.domain.com`, `*.domain.com/public/*`.

- If you plan to access a Google API from an Android client, use an *Android key*. For this key, you need to specify the list of SHA1 fingerprints and package names corresponding to your application(s). To generate the SHA1 fingerprint of the signature used to create your APK file, use the `keytool` command from the terminal:

  ```
  keytool -exportcert -alias androiddebugkey -keystore <path-to-keystore-file> -list -v
  ```

  When you run your app from your development environment, the key in `~/.android/debug.keystore` is used to sign your APK. The password for this signature is normally "android" or an empty string: "".

  Here is an example of the requested string to identify your application.

  ```
  B6:BB:99:41:97:F1:1F:CF:84:2A:6E:0B:FE:75:78:BE:7E:6C:C5:BB;com.lunchmates
  ```

- Use an *iOS key* if your application runs on an iOS device. When using this key, you need to add the bundle identifier(s) of the whitelisted app(s) to the dedicated field in the API key creation process. For example: `com.gcpbook`.

■ **Note** In Windows machines, `keytool.exe` is usually located under
`C:\Program Files\Java\<jdk-version>\bin\`

Remember that prior to accessing a Google API, you must enable access to it and billing where it applies.

You do that as follows:

1. Go to the Developers Console in Google: https://console.developers.google.com.
2. Select a project, or create a new one.
3. In the left sidebar, Expand APIs and Auth and navigate to APIs.
4. Look for the API you are interested in, and change its status to On.

To enable billing, click on the preferences icon next to your profile in the top right side of the screen. If a project is selected, you see an option to access "project billing settings". From there, you can see the details of the billing account associated with that project. To see all the billing accounts that you registered click on "Billing accounts" from the same preferences menu.

OAuth 2.0

This protocol was created with the intention of providing a way to grant limited access to protected content hosted by third-party services in a standardized and open manner. This protected content can be requested on behalf of either a resource owner or an external application or service. This protocol has been adopted by Google to enable access to its APIs, by providing a way to authenticate and authorize external agents interested in exchanging information with Google APIs.

The following steps describe the complete process of requesting access to specific content:

1. The client requests authorization from the resource owner.
2. The resource owner sends back an authorization grant.
3. The client uses this authorization grant to request an access token to the authorization server.
4. If the process is successful, the authorization server provides the client with an access token.
5. The client accesses protected content, authorizing requests with the access token just acquired.
6. If the token is valid, the client receives the requested information.

This process is very similar to how you obtain access to APIs in Google, although that varies depending on the type of application or system you are building. We cover each of these cases in the following paragraphs.

■ **Note** Given the many steps involved in this process, the chances of making a mistake are high, which has security implications. It is highly recommended that you use one of the available libraries that enable and simplify the fulfillment of this protocol. Google provides a variety of client libraries that work with OAuth 2.0[3] in programming languages like Java, Python, .NET, Ruby, PHP, and JavaScript. The Internet also offers valuable resources related to this topic.

In this chapter, you use `oauth2client`. You can find this library in the Google APIs Client Libraries for Python or through the link to the code repository in GitHub: `https://github.com/google/oauth2client`.

Each of the application types follow different OAuth 2.0 flows (2-legged, 3-legged) and thus require different associated information. In the following sections you see how to operate with each of them.

[3]Google OAuth 2.0 client libraries: `https://developers.google.com/accounts/docs/OAuth2#libraries`.

OAuth 2.0 Application Authentication

You use this kind of authentication when you need to access content on behalf of your application, typically in server-to-server communications: for example, managing internal files stored in Cloud Storage. Because of this, the authorization process does not require the authentication of any specific user in order to obtain an access token. Instead, you use the identity of your application.

Some services in Cloud Platform – like App Engine or Compute Engine – already have associated default credentials that are used to perform requests to the different APIs through the client libraries. If you are calling a Google API from somewhere else, you can still use this functionality by creating a new client ID for your service in Developers Console:

1. Go to the Developers Console in Google: `https://console.developers.google.com`.
2. Select a project, or create a new one.
3. In the left sidebar, Expand APIs and Auth, and navigate to Credentials.
4. Create a new client ID by clicking the button for that purpose.
5. Select the application type based on needs and click on Create.

Now you can generate and download the JSON key associated to this client ID. Place it somewhere private within your system. The client libraries attempt to use this key by looking under the path set in the environmental variable `GOOGLE_APPLICATION_CREDENTIALS`. Set this variable to the path where you stored your key.

Figure 3-1 shows the application authorization process.

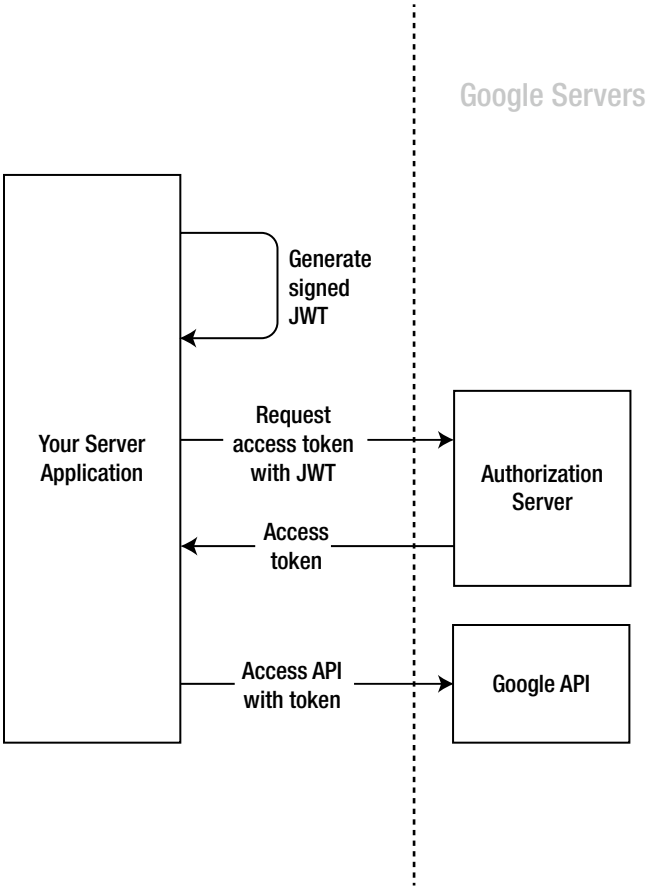

Figure 3-1. *Oauth 2.0 authorization flow for service accounts*

To create the credentials based on the key associated to your account you do the following:

```
from oauth2client.client import GoogleCredentials

credentials = GoogleCredentials.get_application_default()
```

Before making requests to the API, you need make sure that the credentials have the right scope to access the information you are interested in. In this case, you need read permissions:

```
CLOUD_STORAGE_SCOPE = 'https://www.googleapis.com/auth/devstorage.read_only'

if credentials.create_scoped_required():
    credentials = credentials.create_scoped(CLOUD_STORAGE_SCOPE)
```

Now, these credentials have all the necessary information to obtain an access token. The API client does that internally by wrapping the creation of every new request and adding a pre-execution trigger that checks for the existence of an access token. If the access token is invalid or inexistent, the method obtains a new access token; otherwise, it adds the access token to the request as a means of authorization before it is executed. You can create a client representing a concrete Google API that you can use to make requests against it. In this case, we are using the Python client library. For example, if you are interested in listing the files stored on a bucket in Cloud Storage, you do the following:

```
from apiclient.discovery import build

# ...previous code generating credentials

gcs_service = build('storage', 'v1', credentials=credentials)
content = gcs_service.objects().list(bucket='lunchmates_document_dropbox').execute()
print json.dumps(content)
```

■ **Note** You can learn more about the `discovery` and `build` directives at https://cloud.google.com/appengine/articles/efficient_use_of_discovery_based_apis.

If you want to see the full implementation, check the script `api_access_application_authentication.py` under `oauth2` in the `code_snippets` repo. https://github.com/GoogleCloudPlatformBook/code-snippets/tree/master/oauth2.

If you are interested in obtaining an access token manually for testing or other purposes, you can do so by executing the `_refresh()` method from the class `OAuth2Credentials` directly, passing a dummy request: `Http().request`. This internal method is called each time you execute a request—after you authorize your credentials with an instance of `httplib2.Http()`—if there is no access token yet or the access token is invalid. The following snippet generates and prints the obtained access token:

```
credentials._refresh(Http().request)
print credentials.access_token
```

Note that once you have an access token, you can, for instance, perform requests from any system that operates with the HTTP standard. For example, you can execute the previous request using only HTTP:

```
GET https://www.googleapis.com/drive/v2/files?alt=json
Authorization: Bearer <access_token>
```

OAuth 2.0 User Authentication

This type of authentication is used when there is the need to access protected information on behalf of a concrete user. This is common in user facing applications so that users can grant access to the required scopes.

The most common version is the 3-legged OAuth 2.0 user authentication flow, shown in Figure 3-2.

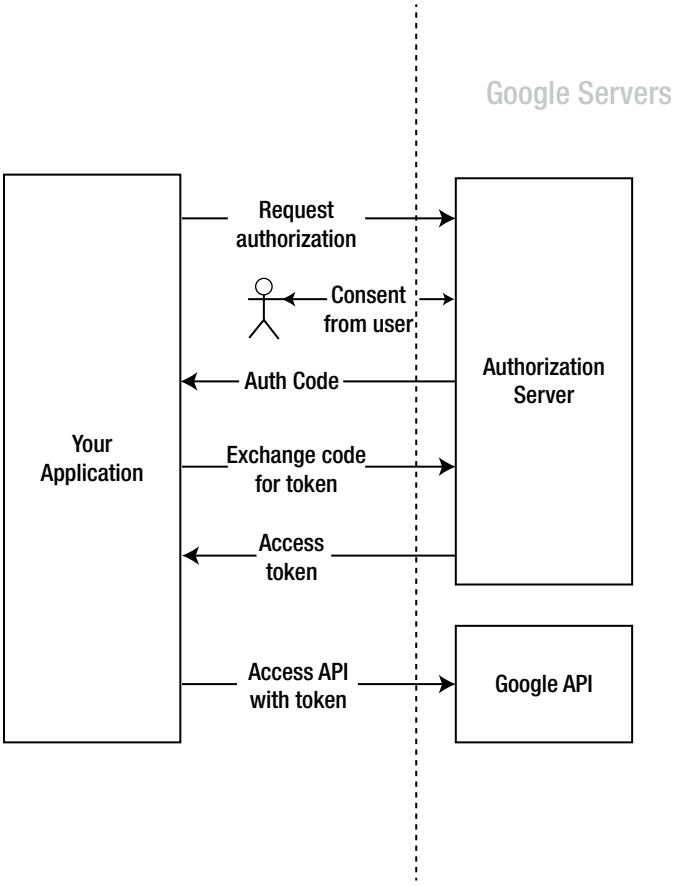

Figure 3-2. *Oauth 2.0 user authentication flow*

As you can see in the figure, this flow asks for user consent. This is because the content is accessed on behalf of that user. The first thing you need to do is obtain the authorization URI to redirect the user, in order for the user to authenticate with their Google credentials and authorize the specified scope:

```
from oauth2client import client

client_secrets_json_path = <path_to_your_client_secrets_file.json>
api_scope = <api_scope_url>
redirect_uri = <redirect_uri_in_client_id>

flow = client.flow_from_clientsecrets(
     client_secrets_json_path,
     scope=api_scope,
     redirect_uri=redirect_uri,
     include_granted_scopes=True)

auth_uri = flow.step1_get_authorize_url()
```

`client_secrets_json_path` is the path to the file containing the secrets and other relevant information related to your client ID. Remember that you can download this JSON file at any point from the Developers Console, under APIs & Auth ➤ Credentials.

You can also execute this first step through HTTP:

```
POST https://accounts.google.com/o/oauth2/auth?
access_type=offline&
response_type=code&
client_id=<client_id>&
redirect_uri=<redirect_uri>&
scope=<api_scope>&
included_granted_scopes=true
```

This request accepts the parameters listed in Table 3-1.

Table 3-1. *List of accepted parameters for the authorization endpoint in Google APIs* `https://accounts.google.com/o/oauth2/auth`

Parameter	Description
`response_type`	Determines the expected response. Options are `code` for web server and installed applications or `access_token` for JavaScript client-side applications.
`client_id`	Identifies the client ID used for this request. You can get this value from the client ID used to perform this request in the Developers Console.
`redirect_uri`	Defines the mechanism used to deliver the response. This value must match one of the values listed under Redirect URIs in the client ID in use. In web applications, this URI is called to deliver a response after the authentication phase. It must also contain the scheme and trailing `/`.
`scope`	Determines the API and level of access requested. It also defines the consent screen shown to the user after authorization succeeds.
`state`	Allows any type of string. The value provided is returned on response; its purpose is to provide the caller with a state that can be used to determine the next steps to take.
`access_type`	Determines whether the application needs to access a Google API when the user in question is not present at the time of request. Accepted values are `online` (the default) and `offline`. When using the latter, a refresh token is added to the response in the next step of the process, the result of exchanging the authorization code for an access token.
`approval_prompt`	Accepts `force` or `auto`. If `force` is chosen, the user is presented with all the scopes requested, even if they have been accepted in previous requests.
`login_hint`	Provides the authorization server with extra information that allows it to simplify the authentication process for the user. It accepts an e-mail or a sub-identifier of the user who is being asked for access.
`include_granted_scopes`	If the authorization process is successful and this parameter is set to `true`, it includes any previous authorizations granted by this user for this application.

■ **Note** In scenarios where applications cannot catch redirects to URLs—for example on mobile devices other than Android or iOS—`redirect_uri` can take the following values:

urn:ietf:wg:oauth:2.0:oob: The authorization code is placed in the title tag of the HTML file. The same code is also exposed in a text field where it can be seen and from which it can be copied manually. This approach is useful when the application can load and parse a web page. Note that if you do not want users to see this code, you must close the browser window as soon as the operation has completed. Conversely, if the system you are developing for has limited capabilities, you can instruct the user to manually copy the code and paste it into your application.

urn:ietf:wg:oauth:2.0:oob:auto: This value behaves almost identically to the previous value. This procedure also places the authorization code in the title tag of the HTML page, but instead of showing the code in the body of the HTML, it asks the user to close the window.

This request responds with a redirect to the URI specified under `redirect_uri`, including an `error` or `code` parameters in the query string, depending on whether the authorization process succeeded or failed, respectively. If the authorization succeeds, the redirect is as follows:

```
<redirect_uri>?code=<authorization_code>
```

And this is the redirect if the authorization fails:

```
<redirect_uri>?error=access_denied
```

Now you can use the `code` to obtain an access token with it:

```
from oauth2client import client

code = <auth_code_from_previous_step>
credentials = flow.step2_exchange(code)
...
```

Just as before, you can use the `discovery` classes and `build` directive to instantiate a service representing the API to interact with:

```
from apiclient.discovery import build

gcs_service = build('storage', 'v1', credentials=credentials)
content = gcs_service.objects().list(bucket='lunchmates_document_dropbox').execute()
```

If you prefer to obtain the access token manually through HTTP, you can do so by using the following endpoint:

```
POST https://www.googleapis.com/oauth2/v3/token

Content-Type: application/x-www-form-urlencoded
```

```
client_id=<client_id>&
client_secret=<client_secret>&
code=<code_from_previous_request>&
grant_type=authorization_code&
redirect_uri=<redirect_uri>
```

This includes the following:

- `client_id` is the identifier for the client ID used throughout the process.
- `client_secret` is the secret corresponding to that client ID.
- `code` is the resulting authorization code extracted from the previous request.
- `grant_type` determines the type of authorization.
- `redirect_uri` has to match the specified value during the previous step.

If the request is successful, you should see something like this:

```
{
   "access_token": <access_token>,
   "token_type": "Bearer",
   "expires_in": 3600,
   "refresh_token": <refresh_token>
}
```

Note that you get a property called `refresh_token`. This is because in the first step, when obtaining the authorization code, you set `access_type` to `offline`. This refresh token allows you to obtain renewed access tokens until the user deliberately revokes access to your application.

To obtain a new access token from a refresh token, perform the following request:

■ **Note** This step also applies to other scenarios and account types.

```
POST https://www.googleapis.com/oauth2/v3/token

Content-Type: application/x-www-form-urlencoded

client_id=<client_id>&
client_secret=<client_secret>&
refresh_token=<refresh_token>&
grant_type=refresh_token
```

In this case, `grant_type` is set to `refresh_token`, and you need to add the refresh token under the parameter `refresh_token`. As before, the expected response is

```
{
   "access_token": <access_token>,
   "token_type": "Bearer",
   "expires_in": 3600
}
```

■ **Pro Tip** In order to obtain new tokens from a refresh token, you must store the latter in your application for as long as you want to have this ability.

■ **Note** As you may have noted, there is no Python implementation for the generation of new tokens from a refresh token. This process is encapsulated and automated in the client libraries. As mentioned previously, you can trigger this process manually in the Python library by calling `credentials._refresh(Http().request)`.

Some applications may need to revoke access to authorized scopes and invalidate tokens when users unsubscribe or delete their account from your system. You can do that by requesting the following:

```
GET https://accounts.google.com/o/oauth2/revoke?token=<access_or_refresh_token>
```

The value for the query parameter can be either an access token or a refresh token. If the revoked access token has an associated `refresh_token`, both are revoked. This request responds with `200 OK` if executed successfully or `400` in case of error.

2-Legged OAuth 2.0 User Authentication

This scenario is aimed to support OAuth 2.0 on applications running on the side of the client. Because of that they are not assumed to have the ability to keep a secret. For example, JavaScript client side applications do not have a way to store a private key, used in the 3-legged user authentication flow.

The flow for this scenario is slightly different from those seen so far. However, the requests that need to be performed to obtain access are consistent with what you used in earlier examples. You begin as in other scenarios by calling to the `/o/oauth2/auth` endpoint. The difference this time is that instead of asking for a code, you directly request an access token. Figure 3-3 shows the process.

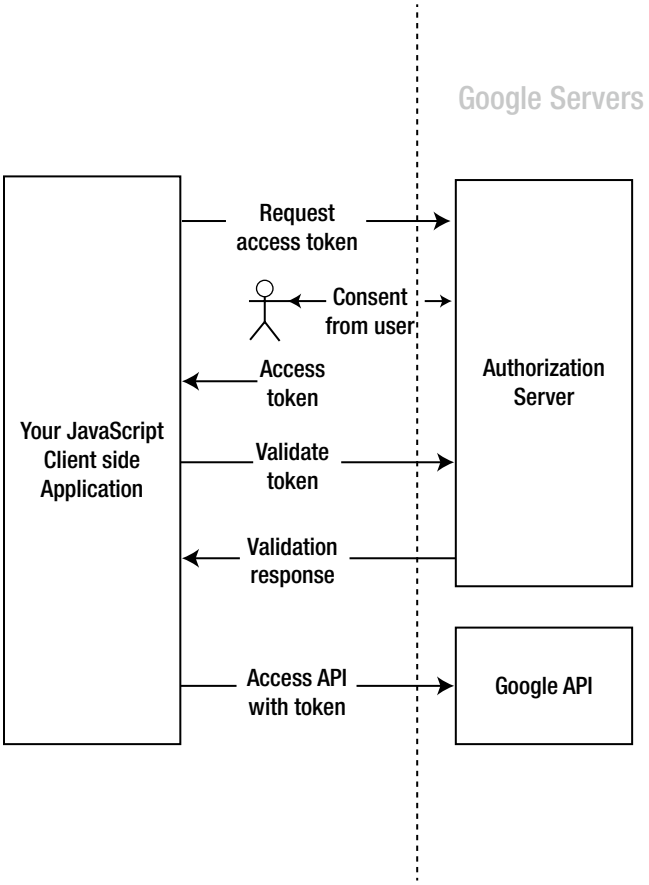

Figure 3-3. *Oauth 2.0 authorization flow for JavaScript client-side applications*

To request the access token, start by calling the authorization server:

```
POST https://accounts.google.com/o/oauth2/auth?
response_type=token&
client_id=<client_id>&
redirect_uri=<redirect_uri>&
scope=<api_scope>&
included_granted_scopes=true
```

The expected response for this request is as follows if the request failed to authorize a combination of user and scope:

```
<redirect_uri>#error=access_denied
```

Or as follows if the process was completed successfully:

```
<redirect_uri>#
access_token=<access_token>&
token_type=Bearer&
expires_in=3600
```

■ **Pro Tip** Your client needs to parse the fragment part of the URL. The sample response contains the minimum set of parameters returned in the fragment, but others may be included.

Before using this token, you must validate it using the `TokenInfo` endpoint, adding the access token in the query string:

```
GET https://www.googleapis.com/oauth2/v1/tokeninfo?access_token=<access_token>
```

A successful response looks like this:

```
{
   "issued_to": <issuer_of_the_token>,
   "audience": <audience>,
   "scope": <requested_scope>,
   "expires_in": 3578,
   "access_type": "online"
}
```

This response includes two parameters that you have not seen before:

- `issued_to`: Specifies to whom the token was issued. This is normally the same as `audience`.
- `audience`: The identifier of the application that is intended to use the token to query Google APIs.

There is one last critical step before using the recently obtained access token: you need to confirm that the value returned under `audience` exactly matches the client ID of your application. As you know very well at this point, you can find this value in the Developers Console.

You can now use the access token to access the Google API you are targeting through one of the client libraries offered for this purpose or simple HTTP. For example, you can use the `curl` command:

```
curl -H "Authorization: Bearer <access_token>" https://www.googleapis.com/drive/v2/
files?alt=json
```

If the token has expired or was revoked, the request to `TokenInfo` will respond with an error `400` and a body similar to the following:

```
{
   "error": "invalid_token",
   "error_description": "Invalid Value"
}
```

Translate API

Google Translate (https://translate.google.com) is an online service that automatically translates text from one language to another (for example, English to Italian). Google Translate supports dozens of languages and hundreds of language pairs for translations. The Google Translate API is a RESTful API that lets developers programmatically translate text using either server-side or client-side applications. This API can also detect the language of input text.

The first step in using any Google API, including the Translate API, is to enable it for a project. The Translate API is a paid service and hence needs the API key for a successful transaction. You perform these steps using the web-based Developers Console at https://console.developers.google.com.

Following are the high-level steps required to use the Translate API:

1. Open a browser window to URL https://console.developers.google.com.
2. Select a project, or create a new one.
3. Switch on the Translate API using the following steps:
   a. In the left sidebar, Expand APIs and Auth.
   b. Select APIs.
   c. Change the status of the Google Translate API to On.
4. Click Credentials, and create a new API key under Public API Access. The OAuth client ID is required for APIs that need access to user's data.
5. Enable Billing, if required, as follows.
   a. In the left sidebar, select Billing and Settings.
   b. In the Billing section, click Enable Billing.
   c. Fill in the required details, and click Submit and Enable Billing.

Accessing Translate REST API

The following key phrases are relevant to the Translate API:

- *Source text*: The input text provided to the Translate API for translation
- *Source language*: The language of the input text, as declared by the application developer
- *Target language*: The language into which the source text needs to be translated

The following three methods make up the Translate API:

- `Translate`: Translates source text (from source language) into target language
- `Languages`: Lists the source and target languages supported by the translate methods
- `Detect`: Detects the source language of the input text

All of these methods are available as REST APIs. Translate REST APIs are different from cloud platform REST APIs in the way they provide access to a service, whereas cloud platform REST APIs provide access to a resource. As a result, the API provides a single URI that acts as a service endpoint, is accessible using the HTTP GET method, and accepts service requests as query parameters.

The base URL for requesting service from the Translate API is https://www.googleapis.com/language/translate/v2?PARAMETERS, where PARAMETERS is the list of keywords and values that applies to the query. v2 in the URL refers to version 2 and is the current version of the Translate API. Listing 3-1 constructs a translation query using this base URL template and replacing PARAMETERS with actual query names and values.

Listing 3-1. HTTP Query for a Translation Request

```
https://www.googleapis.com/language/translate/v2?
key=INSERT-YOUR-KEY&q=good%20morning&source=en&target=it
```

This example includes four query words:

- key: Translate API is a paid service, and in order to use it, you need to activate billing and obtain an API access key. The key is also a means to identify your application.
- q: Identifies the source text that needs to be translated into the target language. The text is URL encoded to represent special characters like spaces.
- source: A language code that identifies the source language of the input text as declared by the application developer.
- target: A language code that identifies the target language of the translation request.

Suppose the query in Listing 3-1 succeeds. The Translate API returns a 200 OK HTTP status code along with a simple JSON object-based reply, as shown in Listing 3-2.

Listing 3-2. HTTP Response for the Translation Query in Listing 3-1

```
200 OK

{
   "data": {
      "translations": [
         {
            "translatedText": "buongiorno"
         }
      ]
   }
}
```

In this query invocation, the API is called by specifying the source language. However, the source language specification is optional; when the query does not specify a source language, the API figures it out and does the translation. The Translate API charges an additional (nominal) fee for source-language detection in addition to the translation fee.

The example in Listing 3-3 and Listing 3-4 shows the query syntax without a specified source language, along with the corresponding JSON response. Note that the response object highlights that the source language is not specified by stating the detected input language.

Listing 3-3. HTTP Query for a Translation Request without the Source Language

```
GET https://www.googleapis.com/language/translate/v2?
key=INSERT-YOUR-KEY&target=it&q=Hello%20universe
```

Listing 3-4. HTTP Response for the Translation Request in Listing 3-3

```
200 OK

{
   "data": {
       "translations": [
           {
               "translatedText": "ciao universo",
               "detectedSourceLanguage": "en"
           }
       ]
   }
}
```

The Translate API also supports batch mode, with which the client can make translation requests consisting of a list of input text. The size of the HTTP request data is limited, due to the use of the `GET` or `POST` HTTP method. The request when using `GET` should be fewer than 2,000 characters including the input text. The request size when using `POST` should be fewer than 5,000 characters including the input text. To use the HTTP `POST` method, the client needs to set an HTTP method override as part of the `POST` request to be accepted by the Translate API, called `X-HTTP-Method-Override: GET`. The example in Listing 3-5 and Listing 3-6 shows a sample batch query using the HTTP `GET` method and the JSON response returned by the Translate API.

Listing 3-5. Batch Query Request

```
GET https://www.googleapis.com/language/translate/v2?
key=INSERT-YOUR-KEY&source=en&target=it&q=Good%20Afternoon&q=Good%20Evening
```

Listing 3-6. Batch Query Response

```
200 OK

{
   "data": {
       "translations": [
           {
               "translatedText": "buon pomeriggio"
           },
           {
               "translatedText": "buonasera"
           }
       ]
   }
}
```

Discovering Languages Supported by Translate API

The Translate API adds support for new languages and translations between new language pairs on a regular basis. Hence it is useful to know the list of languages supported by the API at any point in time. You can find this by using the API's `languages` subcommand. The language query can be invoked in two forms: with and without the target language specified. Listing 3-7 shows the URI template for making this request.

Listing 3-7. URI Template for Retrieving Language Codes Supported by the Translate API

```
https://www.googleapis.com/language/translate/v2/languages?PARAMETERS
```

You can use this URI template to retrieve the language codes supported by the Translate API; see Listing 3-8. The only required parameter for this API invocation is the key and corresponding value pair.

Listing 3-8. Retrieving the Language Codes Supported by the Translate API

```
https://www.googleapis.com/language/translate/v2/languages?key=INSERT-YOUR-KEY
```

If this succeeds, the server responds with a HTTP `200 OK` message along with a JSON object that lists all the languages supported by the Translate API, as shown in Listing 3-9.

Listing 3-9. Language Query Response

```
200 OK

{
   "data": {
      "languages": [
          {
              "language": "en"
          },
          {
              "language": "it"
          },
          ...
          {
              "language": "zh-TW"
          }
      ]
   }
}
```

This example assumes that you want the results returned in English—that is, the language codes are listed in English. Perhaps you want the list of languages to be returned in another language, such as traditional Chinese. Listing 3-10 shows the example URI to achieve this.

Listing 3-10. Retrieving the Translate API Supported Language Codes in Italian

```
https://www.googleapis.com/language/translate/v2/languages?key=INSERT-YOUR-KEY&target=it
```

If this API succeeds, the server responds with a HTTP `200 OK` message along with a JSON object that list the supported languages codes and their translation in the requested target language. Listing 3-11 shows the result.

Listing 3-11. Language Query Response in a Specified Target Language

```
200 OK

{
  "data": {
      "languages": [
          {
               "language": "af",
               "name": "Afrikaans"
           },
           {
                "language": "sq",
                "name": "Albanese"
            },
            . . .
            {
                 "language": "zu",
                 "name": "Zulu"
            }
      ]
   }
}
```

In some situations, it may not be possible to determine the source language of the input text, or the source language specified may not be correct or reliable. To handle these situations, the Translate API provides a method to predict the source language of the input text. You request this service by using the API's `detect` subcommand. Listing 3-12 shows the URI query template.

Listing 3-12. URI Query Template to Detect the Source Language

```
https://www.googleapis.com/language/translate/v2/detect?PARAMETERS
```

Listing 3-13 uses this URI template to detect the source language of an input string. The only required parameters for this API invocation are the key and corresponding value pair along with the query string.

Listing 3-13. Using the Detect Feature of the Translate API

```
https://www.googleapis.com/language/translate/v2/detect?
key=INSERT-YOUR-KEY&q=Este+mes+es+marzo
```

If the request succeeds, the server responds with a `200 OK` HTTP status code along with a set of values in a JSON object, as shown in Listing 3-14.

Listing 3-14. Language-Detection Query Response for Spanish Input

```
200 OK

{
   "data": {
        "detections": [
            [
                {
                    "language": "es",
                    "isReliable": false,
                    "confidence": 0.015852576
                }
            ]
        ]
   }
}
```

`confidence` is an optional parameter with a floating-point value between 0 and 1; *optional* means this parameter is not always returned. The closer the value is to 1, the higher the confidence in the language detection. `isReliable` has been deprecated, and Google plans to remove it in the near future. Hence you should not use this value to make decisions.

You may wonder why the confidence score is so low and whether it is reliable enough to be used in your applications. We wondered about that, too, and did another test using a simple English sentence; see Listing 3-15 and Listing 3-16.

Listing 3-15. Language-Detection Query Using an English Sentence

```
https://www.googleapis.com/language/translate/v2/detect?
key=YOUR_API_KEY&q=this+is+a+simple+english+sentence
```

Listing 3-16. Language-Detection Query Response for English Input

```
200 OK

{
   "data": {
        "detections": [
            [
                {
                    "language": "en",
                    "isReliable": false,
                    "confidence": 0.025648016
                }
            ]
        ]
   }
}
```

From this result, you can see that the English language-detection confidence score is low as well. This may indicate that the Translate API is strict about confidence scores, and hence even a low score provides usable results.

Just as in the `translate` command, you can pass in several query text inputs. The Translate API returns a list of detected languages.

Accessing Translate API using Client Programs

Although the HTTP-based Translate API is easy to use, Google also makes it simple to consume the Translate API from client applications. This is facilitated by the Google APIs client library (https://cloud.google.com/translate/v2/libraries). The Translate API is part of the Google APIs client library, and as of this writing the client library is available for six programming languages and is being developed for three more.

Let's take a brief look at the Python example provided as part of the Google APIs client library (https://code.google.com/p/google-api-python-client/source/browse/samples/translate). We only examine the relevant lines from the program—Listing 3-17 is a code snippet, not a complete program. To use this snippet, you need to install the Google API client library for Python.

Listing 3-17. Translate API Python Example Provided as Part of the Google APIs' Client Library

```
from apiclient.discovery import build
def main():
    service = build('translate', 'v2', developerKey='INSERT_YOUR_KEY')
    print service.translations().list(
        source='en',
        target='fr',
        q=['flower', 'car']
    ).execute()

if __name__ == '__main__':
    main()
```

Any Python programmer should be able to easily understand the standard Python parts in this code snippet. Hence we discuss only the distinctive parts. At the base level, the Google API client library builds a `service` object. This is facilitated by using the `discovery` class from the module `apiclient`. Specifically, you need the `build` method. The `build` method requires three parameters: the API name, API version, and API key. Once the `service` object is built, you can use the methods inside it. Again, depending on the API name provided to build, the methods available are different.

This example uses the `translations()` method to translate from English to French. The query text is provided as a standard Python list containing two items. When the `execute` method is called, the client library constructs the equivalent REST API using these values, makes the HTTP `GET` request, and returns either the result or an error from this API invocation.

More methods are available in this class. They detect the source language and get the list of languages supported by the Translate API. Listing 3-18 shows a more detailed example that uses most of the Translate API capabilities. This example reads the Unix English dictionary and translates it into all possible languages supported by the Translate API. We have added line numbers for easy reference.

Listing 3-18. Python Program to Translate the Unix English Dictionary into Multiple Languages

```
1 #!/usr/bin/env python
2
3 __author__ = 'sptkrishnan@gmail.com (S. P. T. Krishnan)'
4
5 from apiclient.discovery import build
6 import json
7
8
```

```
def main():
    # create a build object
    service = build('translate', 'v2', developerKey='INSERT_YOUR_KEY')

    # STEP 1 - Get the list of languages supported by Translate API
    languages = service.languages().list(target='en').execute()

    # [debug] print the JSON dump of HTTP response
    # print json.dumps(languages, sort_keys=True, indent=4, separators=(',', ': '))

    # Unpack the JSON object from HTTPResponse, extracts the language name,
    # code and creates a dictionary object from it.
    langdict = {}
    for key, value in languages.iteritems():
        for x in value:
            for y, z in x.iteritems():
                if y == 'name':
                    name = z
                elif y == 'language':
                    language = z
                    langdict[name] = language

    # [debug] print the language name and code langdict object
    # for name, code in sorted(langdict.iteritems()):
        # print name, code

    unidict = {}
    filename = '/usr/share/dict/words'
    filehandle = open(filename, 'r')
    for line in filehandle:
        line = line.rstrip('\n')
        for name in langdict.keys():
            if langdict[name] == 'en':
                continue
            else:
                translation = service.translations().list(source='en',
                target=langdict[name], q=line).execute()
            for key, value in translation.iteritems():
                for x in value:
                    for y, z in x.iteritems():
                        if line == z:
                            pass
                            # print line, '(' + name + ')', 'x'
                        else:
                            print line, '(' + name + ')', z


if __name__ == '__main__':
    main()
```

In this example, lines 1–8 and 55–56 are standard Python. They use the `json` module to programmatically process the Translate API return values. Lines 10–11 construct the `service` object, and lines 13–14 get the list of languages supported by the Translate API. By explicitly specifying the target language to be English, you get the language codes and also the language names. Lines 16–17 are a `debug print` statement to print the JSON response from the Translate API.

You parse the HTTP response and create a Linux data structure in lines 19–19. Once this code block completes execution, you have a Python dictionary data structure that maps the language code to its name in English. Lines 31–33 are a `debug print` statement that prints the contents of this dictionary data structure.

The primary purpose of this program is to read the English dictionary (assuming the default language is English) from a Unix system and translate each word into all the languages supported by the Translate API. This is achieved in lines 36–44. The dictionary file is opened, and words are read from each line. (Note that each line contains a single word.) Next the supported language dictionary is iterated over, and each word is passed to the `translations` method along with a single target language. It doesn't make sense to translate from English to English, so you skip that combination.

There are 235,886 words in the English dictionary on our Mac system running Yosemite. Note that the Translate API has daily API limits; see https://cloud.google.com/translate/v2/pricing. If you intend to run this program, you are advised to have a delay between invocations or to use a subset of the word list per day. Listing 3-19 shows the program output for a few words.

Listing 3-19. Partial Output from the English Dictionary Translation Example

```
abaca (Maltese) manilla
abaca (Portuguese) abacá
abaca (Japanese) アバカ
abaca (Spanish) abacá
abaca (Chinese (Simplified)) 蕉
abacate (Tamil) அபகடே என்றும்
abacate (Finnish) Abacate
abacate (Serbian) абацате
abacate (Galician) aguacate
abacate (Yiddish) אַבאַקאַטע
```

As you can see, the Translate API is able to translate between many language pairs. We leave it as an exercise for you to extend this program to save the output into a file, such as a CSV. You can offset the cost of using the Translate API by applying the $300 credit available from https://cloud.google.com.

Summary

In this chapter, you have learned about a fundamental concept that is applicable to all Cloud Platform products: Auth. Auth, which stands for authentication and authorization, is the gatekeeper of Cloud Platform. It is also responsible for safeguarding users' data and allowing access only from approved third-party applications. Cloud Platform Auth implements the authentication, authorization, and accounting (AAA) protocol.

You also learned about the Translate API, which can translate between 90 pairs of languages. We selected this API because it is relatively simple and showcases how to use a standalone API in Google Cloud Platform. You saw how to use this API both via HTTP and using a client library. This chapter has given you a good introduction to Auth and how to use it to access a Google API.