Python (https://cloud.google.com/python/) Guides

Authenticating users with Cloud Identity-Aware Proxy for Python

Apps running on Google Cloud managed platforms such as App Engine (https://cloud.google.com/appengine/docs/) can avoid managing user authentication and session management by using Identity-Aware Proxy (IAP) (https://cloud.google.com/iap/) to control access to them. IAP can not only control access to the app, but it also provides information about the authenticated users, including the email address and a persistent identifier to the app in the form of new HTTP headers.

Objectives

- Require users of your App Engine app to authenticate themselves by using IAP.
- Access users' identities in the app to display the current user's authenticated email address.

Costs

This tutorial uses the following billable components of Google Cloud:

- App Engine (https://cloud.google.com/appengine/pricing)
- IAP (https://cloud.google.com/iap/pricing)

To generate a cost estimate based on your projected usage, use the pricing calculator (https://cloud.google.com/products/calculator). New Google Cloud users might be eligible for a free trial (https://cloud.google.com/free-trial).

When you finish this tutorial, you can avoid continued billing by deleting the resources you created. For more information, see Cleaning up (#clean-up).

Before you begin

1. Sign in (https://accounts.google.com/Login) to your Google Account.

   If you don't already have one, sign up for a new account (https://accounts.google.com/SignUp).
2. In the Cloud Console, on the project selector page, select or create a Google Cloud project.

   Note: If you don't plan to keep the resources that you create in this procedure, create a project instead of selecting an existing project. After you finish these steps, you can delete the project, removing all resources associated with the project.

   GO TO THE PROJECT SELECTOR PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/PROJECTSELECT
3. Install and initialize the Cloud SDK (https://cloud.google.com/sdk/docs/).

Background

This tutorial uses IAP to authenticate users. This is only one of several possible approaches. To learn more about the various methods to authenticate users, see the Authentication concepts (#authentication_concepts) section.

The Hello ***`user-email-address`*** app

The app for this tutorial is a minimal Hello world App Engine app, with one non-typical feature: instead of "Hello world" it displays "Hello ***`user-email-address`***", where ***`user-email-address`*** is the authenticated user's email address.

This functionality is possible by examining the authenticated information that IAP adds to each web request it passes through to your app. There are three new request headers added to each web request that reaches your app. The first two headers are plain text strings that you can use to identify the user. The third header is a cryptographically signed object with that same information.

- `X-Goog-Authenticated-User-Email`: A user's email address identifies them. Don't store personal information if your app can avoid it. This app doesn't store any data; it just echoes it back to the user.
- `X-Goog-Authenticated-User-Id`: This user ID assigned by Google doesn't show information about the user, but it does allow an app to know that a logged-in user is the same one that was previously seen before.
- `X-Goog-Iap-Jwt-Assertion`: You can configure Google Cloud apps to accept web requests from other cloud apps, bypassing IAP, in addition to internet web requests. If an app is so configured, it's possible for such requests to have forged headers. Instead of using either of the plain text headers previously mentioned, you can use and verify this cryptographically signed header to check that the information was provided by Google. Both the user's email address and a persistent user ID are available as part of this signed header.

If you are certain that the app is configured so that only internet web requests can reach it, and that no one can disable the IAP service for the app, then retrieving a unique user ID takes only a single line of code:

```
user_id = request.headers.get('X-Goog-Authenticated-User-ID')
```

However, a resilient app should expect things to go wrong, including unexpected configuration or environmental issues, so we instead recommend creating a function that uses and verifies the cryptographically signed header. That header's signature cannot be forged, and when verified, can be used to return the identification.

Create the source code

1. Use a text editor to create a file named `main.py`, and paste the following code in it:

   authenticating-users/main.py (https://github.com/GoogleCloudPlatform/getting-started-python/blob/master/authenticating-users/main.py)

)GLECLOUDPLATFORM/GETTING-STARTED-PYTHON/BLOB/MASTER/AUTHENTICATING-USERS/MAIN.PY)

```
import sys

from flask import Flask
app = Flask(__name__)

CERTS = None
AUDIENCE = None


def certs():
    """Returns a dictionary of current Google public key certificates for
    validating Google-signed JWTs. Since these change rarely, the result
    is cached on first request for faster subsequent responses.
    """
    import requests

    global CERTS
    if CERTS is None:
        response = requests.get(
            'https://www.gstatic.com/iap/verify/public_key'
        )
        CERTS = response.json()
    return CERTS


def get_metadata(item_name):
    """Returns a string with the project metadata value for the item_name.
    See https://cloud.google.com/compute/docs/storing-retrieving-metadata for
    possible item_name values.
    """
    import requests

    endpoint = 'http://metadata.google.internal'
    path = '/computeMetadata/v1/project/'
    path += item_name
    response = requests.get(
        '{}{}'.format(endpoint, path),
        headers={'Metadata-Flavor': 'Google'}
    )
    metadata = response.text
    return metadata


def audience():
    """Returns the audience value (the JWT 'aud' property) for the current
    running instance. Since this involves a metadata lookup, the result is
    cached when first requested for faster future responses.
    """
    global AUDIENCE
    if AUDIENCE is None:
        project_number = get_metadata('numeric-project-id')
        project_id = get_metadata('project-id')
        AUDIENCE = '/projects/{}/apps/{}'.format(
            project_number, project_id
        )
    return AUDIENCE


def validate_assertion(assertion):
    """Checks that the JWT assertion is valid (properly signed, for the
    correct audience) and if so, returns strings for the requesting user's
    email and a persistent user ID. If not valid, returns None for each field.
    """
    from jose import jwt

    try:
        info = jwt.decode(
            assertion,
            certs(),
            algorithms=['ES256'],
            audience=audience()
            )
        return info['email'], info['sub']
    except Exception as e:
        print('Failed to validate assertion: {}'.format(e), file=sys.stderr)
        return None, None


@app.route('/', methods=['GET'])
def say_hello():
    from flask import request

    assertion = request.headers.get('X-Goog-IAP-JWT-Assertion')
    email, id = validate_assertion(assertion)
    page = "<h1>Hello {}</h1>".format(email)
    return page
```

This `main.py` file is explained in detail in the Understanding the code (#understanding_the_code) section later in this tutorial.

2. Create another file called `requirements.txt`, and paste the following into it:

   authenticating-users/requirements.txt (https://github.com/GoogleCloudPlatform/getting-started-python/blob/master/authenticating-users/requirements.txt)

   PLATFORM/GETTING-STARTED-PYTHON/BLOB/MASTER/AUTHENTICATING-USERS/REQUIREMENTS.TXT)

   ```
   Flask==1.1.1
   cryptography==2.8
   python-jose[cryptography]==3.0.1
   requests==2.22.0
   ```

   The `requirements.txt` file lists any non-standard Python libraries your app needs App Engine to load for it:

   - `Flask` is the Python web framework used for the app.
   - `cryptography` is a module that provides strong cryptographic functions.
   - `python-jose[cryptography]` provides the JWT checking and decoding function.
   - `requests` retrieves data from web sites.

3. Create a file named `app.yaml` and put the following text in it:

   authenticating-users/app.yaml (https://github.com/GoogleCloudPlatform/getting-started-python/blob/master/authenticating-users/app.yaml)

   GLECLOUDPLATFORM/GETTING-STARTED-PYTHON/BLOB/MASTER/AUTHENTICATING-USERS/APP.YAML)

   ```
   runtime: python37
   ```

   The `app.yaml` file tells App Engine which language environment your code requires.

Understanding the code

This section explains how the code in `main.py` works. If you just want to run the app, you can skip ahead to the Deploy the app (#deploying_the_app) section.

The following code is in the `main.py` file. When a an `HTTP GET` request to the home page is received by the app, the Flask framework invokes the `say_hello` function:

authenticating-users/main.py
(https://github.com/GoogleCloudPlatform/getting-started-python/blob/master/authenticating-users/main.py)

)GLECLOUDPLATFORM/GETTING-STARTED-PYTHON/BLOB/MASTER/AUTHENTICATING-USERS/MAIN.PY)

```
@app.route('/', methods=['GET'])
def say_hello():
    from flask import request

    assertion = request.headers.get('X-Goog-IAP-JWT-Assertion')
    email, id = validate_assertion(assertion)
    page = "<h1>Hello {}</h1>".format(email)
    return page
```

The `say_hello` function gets the JWT assertion header value that IAP added from the incoming request and calls a function to validate that cryptographically signed value. The first value returned (email) is then used in a minimal web page that it creates and returns.

authenticating-users/main.py
(https://github.com/GoogleCloudPlatform/getting-started-python/blob/master/authenticating-users/main.py)

)GLECLOUDPLATFORM/GETTING-STARTED-PYTHON/BLOB/MASTER/AUTHENTICATING-USERS/MAIN.PY)

```
def validate_assertion(assertion):
    """Checks that the JWT assertion is valid (properly signed, for the
    correct audience) and if so, returns strings for the requesting user's
    email and a persistent user ID. If not valid, returns None for each field.
    """
    from jose import jwt

    try:
        info = jwt.decode(
            assertion,
            certs(),
            algorithms=['ES256'],
            audience=audience()
            )
        return info['email'], info['sub']
    except Exception as e:
        print('Failed to validate assertion: {}'.format(e), file=sys.stderr)
        return None, None
```

The `validate_assertion` function uses the `jwt.decode` function from the third-party `jose` library to verify that the assertion is properly signed, and to extract the payload information from the assertion. That information is the authenticated user's email address and a persistent unique ID for the user. If the assertion cannot be decoded, this function returns `None` for each of those values and prints a message to log the error.

Validating a JWT assertion requires knowing the public key certificates of the entity that signed the assertion (Google in this case), and the audience the assertion is intended for. For an App Engine app, the audience is a string with Google Cloud project identification information in it. This function gets those certificates and the audience string from the functions preceding it.

authenticating-users/main.py (https://github.com/GoogleCloudPlatform/getting-started-python/blob/master/authenticating-users/main.py)

)GLECLOUDPLATFORM/GETTING-STARTED-PYTHON/BLOB/MASTER/AUTHENTICATING-USERS/MAIN.PY)

```
def audience():
    """Returns the audience value (the JWT 'aud' property) for the current
    running instance. Since this involves a metadata lookup, the result is
    cached when first requested for faster future responses.
    """
    global AUDIENCE
    if AUDIENCE is None:
        project_number = get_metadata('numeric-project-id')
        project_id = get_metadata('project-id')
        AUDIENCE = '/projects/{}/apps/{}'.format(
            project_number, project_id
        )
    return AUDIENCE
```

You can look up the Google Cloud project's numeric ID and name and put those in the source code yourself, but the `audience` function does that for you by querying the standard metadata service made available to every App Engine app. Because the metadata service is external to the app code, that result is saved in a global variable that is returned without having to look metadata up in subsequent calls.

authenticating-users/main.py (https://github.com/GoogleCloudPlatform/getting-started-python/blob/master/authenticating-users/main.py)

)GLECLOUDPLATFORM/GETTING-STARTED-PYTHON/BLOB/MASTER/AUTHENTICATING-USERS/MAIN.PY)

```
def get_metadata(item_name):
    """Returns a string with the project metadata value for the item_name.
    See https://cloud.google.com/compute/docs/storing-retrieving-metadata for
    possible item_name values.
    """
    import requests

    endpoint = 'http://metadata.google.internal'
    path = '/computeMetadata/v1/project/'
    path += item_name
    response = requests.get(
        '{}{}'.format(endpoint, path),
        headers={'Metadata-Flavor': 'Google'}
    )
    metadata = response.text
    return metadata
```

The App Engine metadata service (and similar metadata services for other Google Cloud computing services) looks like a web site and is queried by standard web queries. However, it isn't actually an external site, but an internal feature that returns requested information about the running app, so it is safe to use `http` instead of `https` requests. It's used to get the current Google Cloud identifiers needed to define the JWT assertion's intended audience.

authenticating-users/main.py (https://github.com/GoogleCloudPlatform/getting-started-python/blob/master/authenticating-users/main.py)

)GLECLOUDPLATFORM/GETTING-STARTED-PYTHON/BLOB/MASTER/AUTHENTICATING-USERS/MAIN.PY)

```
def certs():
    """Returns a dictionary of current Google public key certificates for
    validating Google-signed JWTs. Since these change rarely, the result
    is cached on first request for faster subsequent responses.
    """
    import requests

    global CERTS
    if CERTS is None:
        response = requests.get(
            'https://www.gstatic.com/iap/verify/public_key'
        )
        CERTS = response.json()
    return CERTS
```

Verification of a digital signature requires the public key certificate of the signer. Google provides a web site that returns all of the currently used public key certificates. These results are cached in case they're needed again in the same app instance.

Deploying the app

Now you can deploy the app and then enable IAP to require users to authenticate before they can access the app.

1. In your terminal window, go to the directory containing the `app.yaml` file, and deploy the app to App Engine:

   ```
   gcloud app deploy
   ```

2. When prompted, select a nearby region.
3. When asked if you want to continue with the deployment operation, enter `Y`.

   Within a few minutes, your app is live on the internet.
4. View the app:

   ```
   gcloud app browse
   ```

   In the output, copy ***`web-site-url`***, the web address for the app.
5. In a browser window, paste ***`web-site-url`*** to open the app.

   No email is displayed because you're not yet using IAP so no user information is sent to the app.

Enable IAP

Now that an App Engine instance exists, you can protect it with IAP:

1. In the Google Cloud Console, go to the **Identity-Aware Proxy** page.

   GO TO IDENTITY-AWARE PROXY PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/SECURITY/IAP)
2. Because this is the first time you've enabled an authentication option for this project, you see a message that you must configure your OAuth consent screen before you can use

IAP.

Click **Configure Consent Screen**.

3. On the **OAuth Consent Screen** tab of the **Credentials** page, complete the following fields:
    - In the **Application name** field, enter `IAP Example`.
    - In the **Support email** field, enter your email address.
    - In the **Authorized domain** field, enter the hostname portion of the app's URL, for example, `iap-example-999999.appspot.com`. Press the `Enter` key after entering the hostname in the field.
    - In the **Application homepage link** field, enter the URL for your app, for example, `https://iap-example-999999.appspot.com/`.
    - In the **Application privacy policy line** field, use the same URL as the homepage link for testing purposes.
4. Click **Save**. When prompted to create credentials, you can close the window.
5. In the Cloud Console, go to the **Identity-Aware Proxy** page.

    GO TO IDENTITY-AWARE PROXY PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/SECURITY/IAP)

6. To refresh the page, click **Refresh** . The page displays a list of resources you can protect.
7. In the **IAP** column, click to turn on IAP for the app.
8. In your browser, go to ***`web-site-url`*** again.
9. Instead of the web page, there is a login screen to authenticate yourself. When you log in, you're denied access because IAP doesn't have a list of users to allow through to the app.

Add authorized users to the app

1. In the Cloud Console, go to the Identity-Aware Proxy page.

    GO TO IDENTITY-AWARE PROXY PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/SECURITY/IAP)

2. Select the checkbox for the App Engine app, and then click **Add Member**.
3. Enter `allAuthenticatedUsers`, and then select the **Cloud IAP/IAP-Secured Web App User** role.
4. Click **Save**.

Now any user that Google can authenticate can access the app. If you want, you can restrict access further by only adding one or more people or groups as members:

- Any Gmail or G Suite email address
- A Google Group email address
- A G Suite domain name

Access the app

1. In your browser, go to ***`web-site-url`***.
2. To refresh the page, click **Refresh** ⟳.
3. On the login screen, log in with your Google credentials.

   The page displays a "Hello ***`user-email-address`***" page with your email address.

   If you still see the same page as before, there might be an issue with the browser not fully updating new requests now that you enabled IAP. Close all browser windows, reopen them, and try again.

Authentication concepts

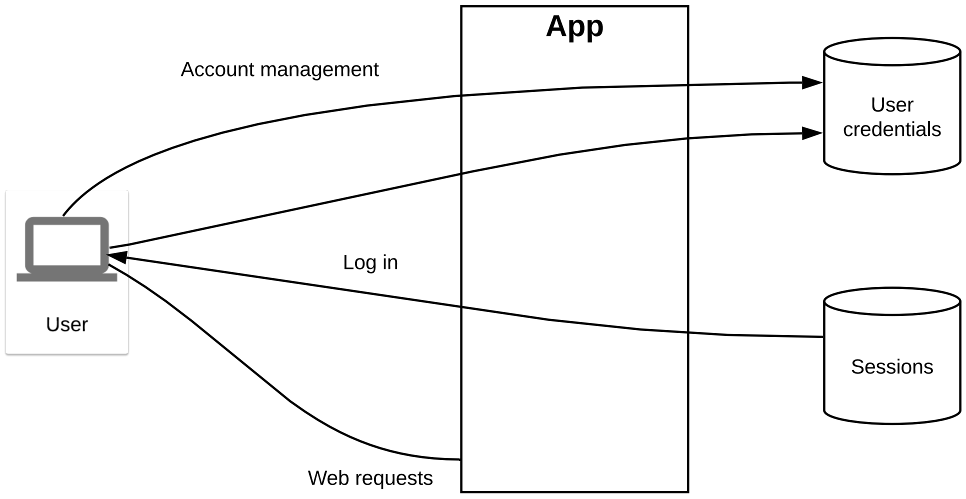
There are several ways an app can authenticate its users and restrict access to only authorized users. Common authentication methods, in decreasing level of effort for the app, are listed in the following sections.

Option	Advantages	Disadvantages
App authentication	• App can run on any platform, with or without an internet connection • Users don't need to use any other service to manage authentication	• App must manage user credentials securely, guard against disclosure • App must maintain session data for logged-in users • App must provide user registration, password changes, password recovery

Option	Advantages	Disadvantages
OAuth2	• App can run on any internet-connected platform, including a developer workstation • App doesn't need user registration, password changes, or password recovery functions. • Risk of user information disclosure is delegated to other service • New login security measures handled outside the app	• Users must register with the identity service • App must maintain session data for logged-in users
IAP	• App doesn't need to have any code to manage users, authentication, or session state • App has no user credentials that might be breached	• App can only run on platforms supported by the service. Specifically, certain Google Cloud services that support IAP, such as App Engine.

App-managed authentication

With this method, the app manages every aspect of user authentication on its own. The app must maintain its own database of user credentials and manage user sessions, and it needs to provide functions to manage user accounts and passwords, check user credentials, as well as issue, check, and update user sessions with each authenticated login. The following diagram illustrates the app-managed authentication method.



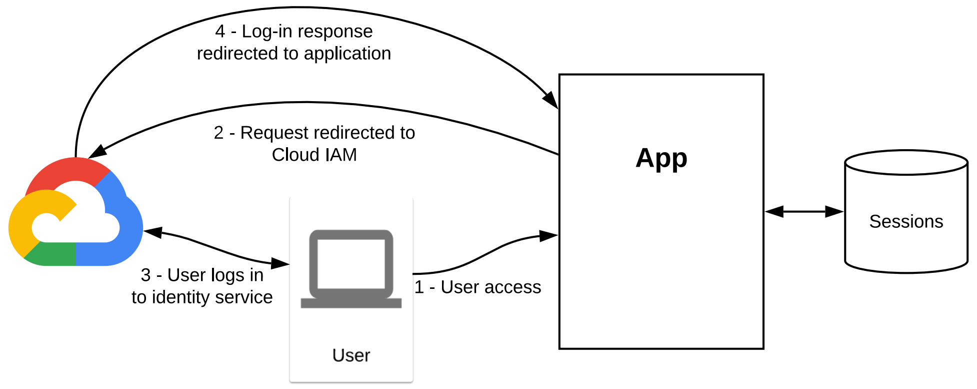
As shown in the diagram, the app must store user credentials so it can provide a way for a user to log in, after which it creates and maintains information about the user's session. When the user makes a request to the app, the request must include session information that the app is responsible for verifying.

The main advantage of this approach is that it is self-contained and under the control of the app. The app doesn't even need to be available on the internet. The main disadvantage is that the app is now responsible for providing all account management functionality and protecting all sensitive credential data.

External authentication with OAuth2

A good alternative to handling everything within the app is to use an external identity service, such as Google, that handles all user account information and functionality and is responsible for safeguarding sensitive credentials. When a user tries to log in to the app the request is redirected to the identity service, which authenticates the user and then redirect the request back to the app with necessary authentication information provided. For more information, see Authenticating as an end user (https://cloud.google.com/docs/authentication/end-user).

The following diagram illustrates the external authentication with the OAuth2 method.


4 - Log-in response redirected to application
2 - Request redirected to Cloud IAM
App
Sessions
3 - User logs in to identity service
User
1 - User access

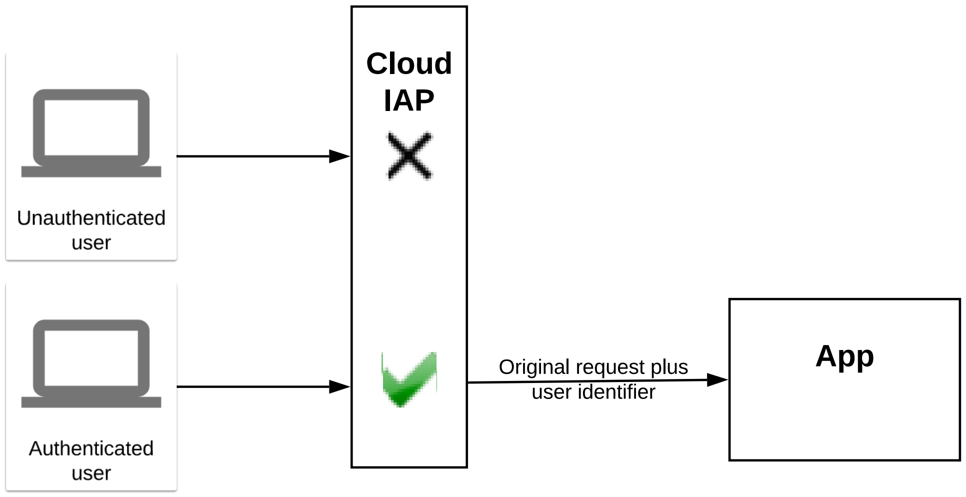
The flow in the diagram begins when the user sends a request to access the app. Instead of responding directly, the app redirects the user's browser to Google's identity platform, which displays a page to log in to Google. After successfully logging in, the user's browser is directed back to the app. This request includes information that the app can use to look up information about the now authenticated user, and the app now responds to the user.

This method has many advantages for the app. It delegates all account management functionality and risks to the external service, which can improve login and account security without the app having to change. However, as is shown in the preceding diagram, the app must have access to the internet to use this method. The app is also responsible for managing sessions after the user is authenticated.

Identity-Aware Proxy

The third approach, which this tutorial covers, is to use IAP to handle all authentication and session management with any changes to the app. IAP intercepts all web requests to your app, blocks any that haven't been authenticated, and passes others through with user identity data added to each request.

The request handling is shown in the following diagram.



Requests from users are intercepted by IAP, which blocks unauthenticated requests. Authenticated requests are passed on to the app, provided that the authenticated user is in the list of allowed users. Requests passed through IAP have headers added to them identifying the user who made the request.

The app no longer needs to handle any user account or session information. Any operation that needs to know a unique identifier for the user can get that directly from each incoming web request. However, this can only be used for computing services that support IAP, such as App Engine and load balancers. You cannot use IAP on a local development machine.

Cleaning up

To avoid incurring charges to your Google Cloud Platform account for the resources used in this tutorial:

The easiest way to eliminate billing is to delete the project that you created for the tutorial.

To delete the project:

Caution: Deleting a project has the following effects:

- **Everything in the project is deleted.** If you used an existing project for this tutorial, when you delete it, you also delete any other work you've done in the project.
- **Custom project IDs are lost.** When you created this project, you might have created a custom project ID that you want to use in the future. To preserve the URLs that use the project ID, such

as an `appspot.com` URL, delete selected resources inside the project instead of deleting the whole project.

If you plan to explore multiple tutorials and quickstarts, reusing projects can help you avoid exceeding project quota limits.

1. In the Cloud Console, go to the **Manage resources** page.

   GO TO THE MANAGE RESOURCES PAGE (HTTPS://CONSOLE.CLOUD.GOOGLE.COM/IAM-ADMIN/PRC

2. In the project list, select the project you want to delete and click **Delete** 🗑 .
3. In the dialog, type the project ID, and then click **Shut down** to delete the project.

Except as otherwise noted, the content of this page is licensed under the Creative Commons Attribution 4.0 License (https://creativecommons.org/licenses/by/4.0/)*, and code samples are licensed under the Apache 2.0 License* (https://www.apache.org/licenses/LICENSE-2.0)*. For details, see our Site Policies* (https://developers.google.com/terms/site-policies)*. Java is a registered trademark of Oracle and/or its affiliates.*

Last updated December 5, 2019.