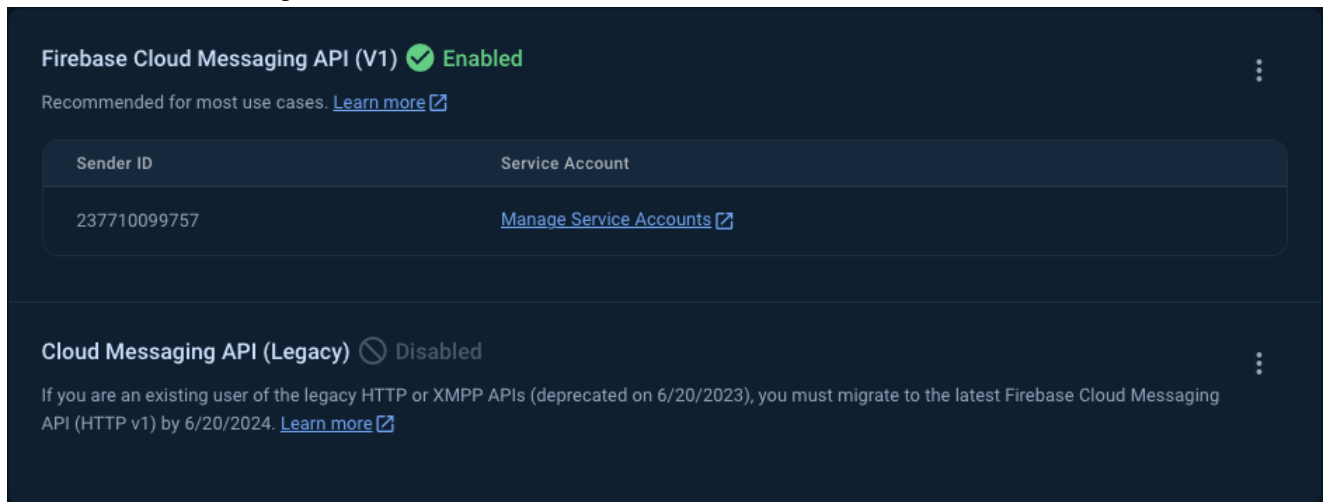


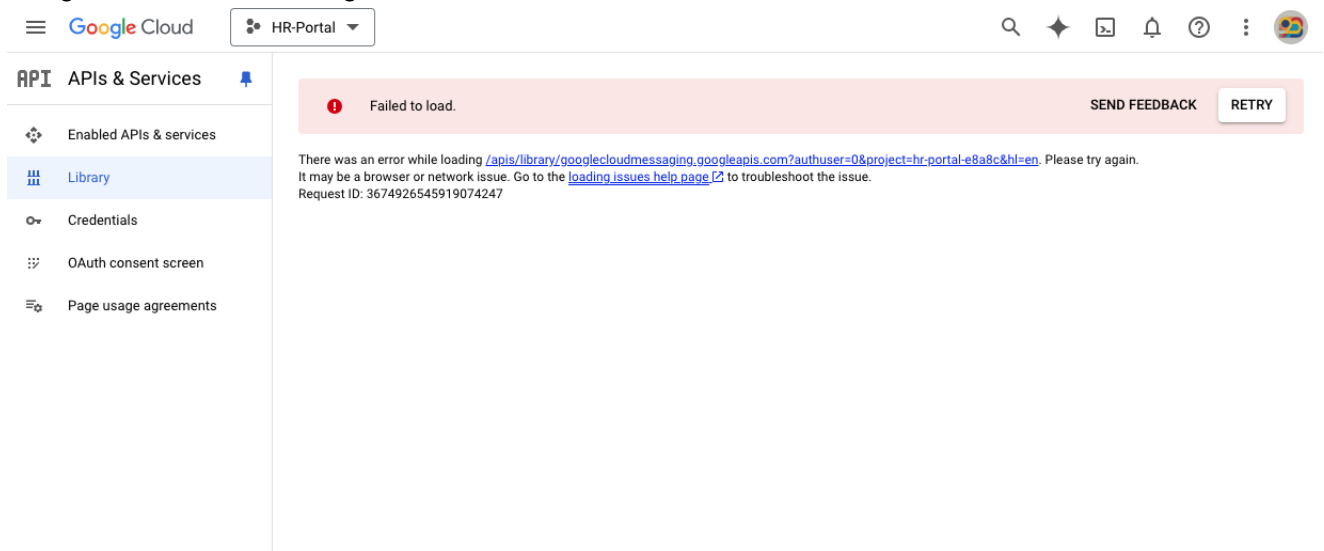
Firebase has brings a new policy for the FCM access token. Now, you need to create a new access token for FCM. You can create a new access token by following the steps below:

Before that see the case:

Firebase Console Image



Google Cloud Console Image



The screenshot shows the Firebase documentation page for migrating from legacy FCM APIs to HTTP v1. The page is part of the 'Run' section under 'FCM'. The left sidebar shows a navigation menu with categories like 'ITERATE', 'ENGAGE', and 'In-App Messaging'. The main content area has a breadcrumb trail: 'Firebase > Documentation > FCM > Run'. The title is 'Migrate from legacy FCM APIs to HTTP v1'. The text explains that apps using deprecated FCM legacy APIs should migrate to the HTTP v1 API by June 20, 2023, with a shutdown beginning on July 22, 2024. It lists three advantages of the HTTP v1 API: better security via access tokens, more efficient customization of messages across platforms, and being more extendable and future-proof for new client platform versions.

[Link to the Google Recommendation](#)

Solutions:

1. Use Google Oath2 (Easy Way)
2. Manually Rest API Implementation

Use Google Oath2 (Easy Way)

You can use Google Oath2 to get the new access token. Here are the steps:

1. Go to your project in the Firebase Console.
2. In project settings, go to the Service accounts tab.
3. Generate a new private key.
4. Download the private key file.

Now, you can use the private key file to get the new access token. Here is the code to get the new access token:

Note this is dart code and i'm using googleapis_auth package to get the access token from the private key file. You can use the same code in any

language by using the `googleapis_auth` package. Check you language support for `googleapis_auth` package.

```
import 'package:http/http.dart' as http;
import 'package:googleapis_auth/auth_io.dart';

Future<AccessCredentials> obtainCredentials() async {
  final Map<String, dynamic> json = await readAccountFile();
  var accountCredentials = ServiceAccountCredentials.fromJson({
    "private_key_id": json['private_key_id'],
    "private_key": json['private_key'],
    "client_email": json['client_email'],
    "client_id": json['client_id'],
    "type": "service_account"
  });
  final scopes = ['https://www.googleapis.com/auth/cloud-platform'];
  var client = http.Client();
  AccessCredentials credentials =
    await obtainAccessCredentialsViaServiceAccount(
      accountCredentials, scopes, client);

  client.close();
  print(
    "OAuth2 Credential -----\\n${credentials.toJson()}\\n-----\\n-----");
  return credentials;
}

// Service Account File
Future<Map<String, dynamic>> readAccountFile() {
  try {
    final String jsonString =
      File(_serviceAccessFilePath).readAsStringSync();
    return Future.value(jsonDecode(jsonString));
  } on Exception catch (e) {
    return Future.error(e);
  }
}
```

- How to use the above code [SEND OTP]:

```
Future<void> sendOTP() async {
  final credential = await obtainCredentials();
  final deviceToken =
    'cNGXU81nSG6AMb8YYj1Ny4:APA91bGZpAgw_gp6bl6kfgN4wIx CZLR0t1duQiv-
    B43X5ggev10BFuG59r6UpL0wvWm5ubq27DM5ehPq5-Ykhvf3VbpiMGZk6wae25oG1mqVMw-
```

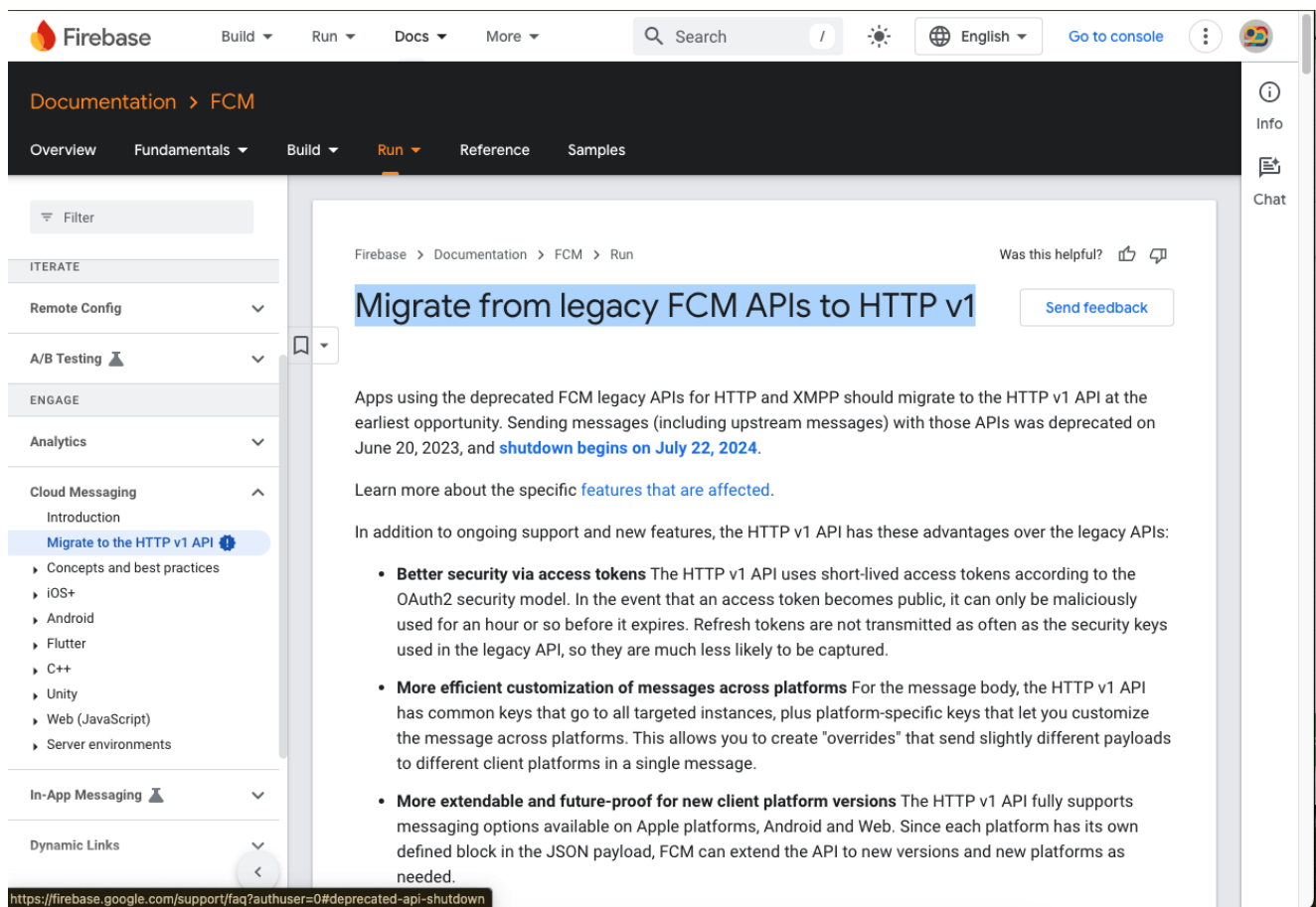
```

5Bhv5Xo14DJ3_SaR2V1DhLVowMou9';
    final serviceURL = Uri.parse(
        'https://fcm.googleapis.com/v1/projects/hr-portal-
e8a8c/messages:send');

    final client = http.Client();
    final response = await client.post(serviceURL,
        headers: {
            'Authorization': 'Bearer ${credential.accessToken.data}',
            'Content-Type': 'application/json',
        },
        body: jsonEncode({
            'message': {
                'notification': {
                    'title': 'OTP',
                    'body': 'Your OTP is 123456',
                },
                'data': {
                    'category': 'leave',
                    'type': 'accepted',
                    'title': 'Only ABCD ',
                    'body': 'Nothing-forground',
                    'click_action': 'FLUTTER_NOTIFICATION_CLICK',
                    'ticket-id': null
                },
                'token':
                    deviceToken,
            },
        })),
    );

    print(response.body);
    client.close();
}

```



For More Details: Visit the
[Google Oath2](#)

Manually Rest API Implementation

You can also get the new access token by using the **Rest API** and the **private key** file along with **RSA algorithm**. Here are the steps:

Google Identity

Authentication

Authorization

More

Search

English

Authorization

OAuth 2.0

Android

Web

Google Account Linking

Filter

Overview

Cross-client Identity

OAuth 2.0 Scopes

OAuth 2.0 Policies

Access to Google APIs for Server-side Web Apps

for JavaScript Web Apps

for Mobile & Desktop Apps

for TV & Device Apps

for Service Accounts

Prepare your app for production

Comply with OAuth 2.0 policies

Submit for brand verification

Sensitive scope verification

Restricted scope verification

Additional considerations for Google Workspace

Resources

Best practices

How to handle granular permission

Home > Products > Google Identity > Authorization > OAuth 2.0

Was this helpful?

Send feedback

Using OAuth 2.0 for Server to Server Applications

★ Important:

If you are working with Google Cloud Platform, unless you plan to build your own client library, use service accounts and a Cloud Client Library instead of performing authorization explicitly as described in this document. For more information, see [Authentication Overview](#) in the Google Cloud Platform documentation.

The Google OAuth 2.0 system supports server-to-server interactions such as those between a web application and a Google service. For this scenario 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, so users aren't directly involved. This scenario is sometimes called "two-legged OAuth," or "2LO." (The related term "three-legged OAuth" refers to scenarios in which your application calls Google APIs on behalf of end users, and in which user consent is sometimes required.)

Typically, an application uses a service account when the application uses Google APIs to work with its own data rather than a user's data. For example, an application that uses Google Cloud Datastore for data persistence would use a service account to authenticate its calls to the Google Cloud Datastore API.

Google Workspace domain administrators can also [grant service accounts domain-wide authority](#) to access user data on behalf of users in the domain.

This document describes how an application can complete the server-to-server OAuth 2.0 flow by using either

Info

Chat

API

For More Details: Visit the [Migration](#)

PROFESSEUR : M.DA ROS

◆ 6 / 6 ◆

BTS SIO BORDEAUX - LYCÉE GUSTAVE EIFFEL