

Google+ Platform

- ▶ Quickstart
- ▶ PhotoHunt Sample
- ▶ API Reference
- ▶ Features
- ▼ Android
 - Getting started
 - Sign in
 - Share
 - +1 button
 - Over-the-air installs
 - App activities
 - People and profiles
 - ▶ Google Play services
- ▶ iOS
- ▶ Websites
- ▶ Hangout Apps
- Downloads
- Best Practices
- Branding Guidelines
- ▶ Community & Support
- Release Notes
- Google APIs Console
- Developer Policies
- Terms of Service

Google+ Sign-in for Android

Allow users to securely sign in to your app and create a more powerful and engaging experience. Google+ Sign-in also lets you grow your app engagement by letting your signed-in users create [interactive posts](#) to invite their friends to use your app. By connecting your users with Google, you will soon be able to influence your appearance in Google Play through trusted social recommendations that show your users their friends who are already using your app.

The **Google+ Sign-In** button authenticates the user and manages the OAuth 2.0 flow, which simplifies your integration with the Google APIs. Signing in is required for your app to create [interactive posts](#), manage [moments](#), and fetch [profile and people information](#).

A user always has the option to [revoke access](#) to an application at any time.

Important: Your use of the Google+ Sign-In button is subject to the [Google+ Platform Button Policies](#) and [Google+ Platform Terms of Service](#).

[Before you begin](#)
[Add the Google+ Sign-In button to your app](#)
[Cross-platform single sign on](#)
[Sign out the user](#)
[Revoking access tokens and disconnecting the app](#)
[Customizing your sign-in button](#)
[Localization](#)
[Server-side access for your app](#)
[Next steps](#)


Before you begin

You must [create a Google APIs Console project and initialize the PlusClient object](#).

Add the Google+ Sign-In button to your app

1. Add the [SignInButton](#) in your application's layout:

```
<com.google.android.gms.common.SignInButton
    android:id="@+id/sign_in_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

2. Initialize `mPlusClient` with the requested visible activities in your [Activity.onCreate](#)  handler.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    mPlusClient = new PlusClient.Builder(this, this, this)
        .setVisibleActivities("http://schemas.google.com/AddActivity", "http://sc
        .build();
}
```

3. In the Android activity, register your button's `OnClickListener` to sign in the user when clicked:

```
findViewById(R.id.sign_in_button).setOnClickListener(this);
```

4. After the user has clicked the sign-in button, you should start to resolve any connection errors held in `mConnectionResult`. Possible connection errors include prompting the user to select an account, and granting access to your app.

```
@Override
public void onClick(View view) {
    if (view.getId() == R.id.sign_in_button && !mPlusClient.isConnected()) {
        if (mConnectionResult == null) {
            mConnectionProgressDialog.show();
        } else {
            try {
```

```

        mConnectionResult.startResolutionForResult(this, REQUEST_CODE_RESOLVE);
    } catch (SendIntentException e) {
        // Try connecting again.
        mConnectionResult = null;
        mPlusClient.connect();
    }
}
}
}
}

```

5. When the user has successfully signed in, your `onConnected` handler will be called. At this point, you are able to retrieve the user's account name or make authenticated requests.

```

@Override
public void onConnected(Bundle connectionHint) {
    mConnectionProgressDialog.dismiss();
    Toast.makeText(this, "User is connected!", Toast.LENGTH_LONG).show();
}

```

Cross-platform single sign on

When a user runs your app for the first time on the Android device, the `PlusClient.onConnect()` method automatically checks if the user previously granted authorization to your app if the clients meet the following requirements:

- The OAuth 2.0 client IDs must be in the same [Google APIs Console](#) project.
- The OAuth scopes must be the same in both clients.
- The list of app activities that your app requests access to write ([PlusClient.setVisibleActivities\(\)](#) and [data-requestvisibleactions](#)) must match.

If the user signed in to your web app previously, then `onConnect()` automatically succeeds and `onConnected()` is invoked immediately. You can proceed to access Google APIs to retrieve the user's info and bypass the need for the user to sign in to your app again.

Sign out the user

You can add a sign out button to your app. Create a button in your app to act as your sign out button. Attach an [onClickListener](#) to the button and configure the `onClick` method to disconnect the `PlusClient`:

```

@Override
public void onClick(View view) {
    if (view.getId() == R.id.sign_out_button) {
        if (mPlusClient.isConnected()) {
            mPlusClient.clearDefaultAccount();
            mPlusClient.disconnect();
            mPlusClient.connect();
        }
    }
}
}

```

This code clears which account is connected to the app. To sign in again, the user would choose their account again.

Revoking access tokens and disconnecting the app

To comply with the terms of the [Google+ developer policies](#), you must offer users that signed in with Google the ability to disconnect from your app. If the user deletes their account, you must delete the information that your app obtained from the Google APIs.

The following code shows a simple example of calling the `PlusClient.revokeAccessAndDisconnect` method and responding to the `onAccessRevoked` event.

```

// Prior to disconnecting, run clearDefaultAccount().
mPlusClient.clearDefaultAccount();

mPlusClient.revokeAccessAndDisconnect(new OnAccessRevokedListener() {
    @Override
    public void onAccessRevoked(ConnectionResult status) {
        // mPlusClient is now disconnected and access has been revoked.
        // Trigger app logic to comply with the developer policies
    }
}

```

```
});
```

In the `onAccessRevoked` callback, you can respond to the event and trigger any appropriate logic in your app or your backend code. For more information, see the [deletion rules](#) in the developer policies.

Customizing your sign-in button

You can design a custom sign-in button to fit with your app's designs. The

[com.google.android.gms.common.SignInButton](#) class offers you the ability to change the color theme and the dimensions of the button. If you require additional customization, you can define a custom button:

1. Review the [branding guidelines and download icons and images](#) to use in your button.
2. Add the button to your application layout.

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:text="@string/common_signin_button_text_long" />
```

This example loads [localized text from the SDK](#). You could also use

`getString(com.google.android.gms.R.string.common_signin_button_text_long)` within your activity's `onCreate` method to get the label for the button text. However you implement your sign-in button, you must follow the [branding guidelines](#).

Localization

The SDK provides localized strings for the [com.google.android.gms.common.SignInButton](#) button and these are automatically available to users of your app. To view a full list of languages, you can examine the following directory in the SDK: `<android-sdk-folder>/extras/google/google_play_services/libproject/google-play-services_lib/res/`. In that location, you will find directories named `values-<langcode>`.

Server-side access for your app

If you use [PlusClient.connect\(\)](#), your app is authenticating the user on the client side only. The `PlusClient` is able to access the Google APIs while the user is actively using your app. If you wish for your servers to be able to make Google API calls on behalf of users or while they are offline, your server requires an access token and a refresh token.

To obtain an access token and refresh token for your server, you can request a **one-time authorization code** that your server exchanges for these two tokens. You request the one-time code by using the [GoogleAuthUtil.getToken\(\)](#) method and specifying a unique scope string that includes the OAuth 2.0 client ID of your gserver.

After you successfully connect the user, you can call `GoogleAuthUtil.getToken()` which gives you a string with the code.

In the following example, you would replace: `+ <server-client-id>` with your server's OAuth 2.0 client ID. `+ <scope1>` `<scope2>` with a space-delimited list of scopes. `+ <app-activity1>` `<app-activity2>` with a space-delimited list of app activity types.

```
Bundle appActivities = new Bundle();
appActivities.putString(GoogleAuthUtil.KEY_REQUEST_VISIBLE_ACTIVITIES,
    "<app-activity1> <app-activity2>");
String scopes = "oauth2:server:client_id:<server client-id>:api_scope:<scope1> <scope2>";
String code = null;
try {
    code = GoogleAuthUtil.getToken(
        this, // Context context
        mPlusClient.getAccountName(), // String accountName
        scopes, // String scope
        appActivities // Bundle bundle
    );
} catch (IOException transientEx) {
    // network or server error, the call is expected to succeed if you try again later.
    // Don't attempt to call again immediately - the request is likely to
    // fail, you'll hit quotas or back-off.
    ...
}
```

```

    return;
} catch (UserRecoverableAuthException e) {
    // Recover
    code = null;
} catch (GoogleAuthException authEx) {
    // Failure. The call is not expected to ever succeed so it should not be
    // retried.
    ...
    return;
} catch (Exception e) {
    throw new RuntimeException(e);
}

```

After you obtain the one-time authorization, send it to your server to immediately exchange for the server's access and refresh tokens. This process at this point is similar to the [process for web clients](#). Your server can then use the Google API Client libraries to exchange the one-time authorization code and then store the tokens. This one-time use code increases the security over a standard OAuth 2.0 flow; however, you need to exchange it for the tokens immediately to ensure the security of the tokens. You should send the one-time authorization code as securely as possible: use HTTPS and send it as POST data or in the request headers.

If you do not require offline access, you can retrieve the access token and send it to your server over a secure connection. You can obtain the access token directly using [GoogleAuthUtil.getToken\(\)](#) by specifying the scopes without your server's OAuth 2.0 client ID. For example:

```

String accessToken = null;
try {
    accessToken = GoogleAuthUtil.getToken(this,
        mPlusClient.getAccountName(),
        "oauth2:" + TextUtils.join(' ', SCOPES));
} catch (IOException transientEx) {
    // network or server error, the call is expected to succeed if you try again later.
    // Don't attempt to call again immediately - the request is likely to
    // fail, you'll hit quotas or back-off.
    ...
    return;
} catch (UserRecoverableAuthException e) {
    // Recover
    accessToken = null;
} catch (GoogleAuthException authEx) {
    // Failure. The call is not expected to ever succeed so it should not be
    // retried.
    return;
} catch (Exception e) {
    throw new RuntimeException(e);
}

```

To learn more, see: + [Android Authorization methods](#). + [Cross client identity](#)

Next steps

- [Create interactive posts](#)
- [Manage Google+ moments](#)
- [Fetch people information](#)

Except as otherwise [noted](#), the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#).

Last updated May 31, 2013.



Google

[Terms of Service](#)

[Privacy Policy](#)

[Jobs](#)

[Report a bug](#)

English