Permissions on Android

Permissions (Overview)

App permissions help support user privacy by protecting access to the following:

- **Restricted data**, such as system state and users' contact information
- **Restricted actions**, such as connecting to a paired device and recording audio

This page provides an overview to how Android permissions work, including a high-level workflow for using permissions, descriptions of different types of permissions, and some best practices for using permissions in your app. Other pages explain how to minimize your app's requests for permissions, declare permissions, request runtime permissions, and restrict how other apps can interact with your app's components.

To view a complete list of Android app permissions, visit the permissions API reference page.

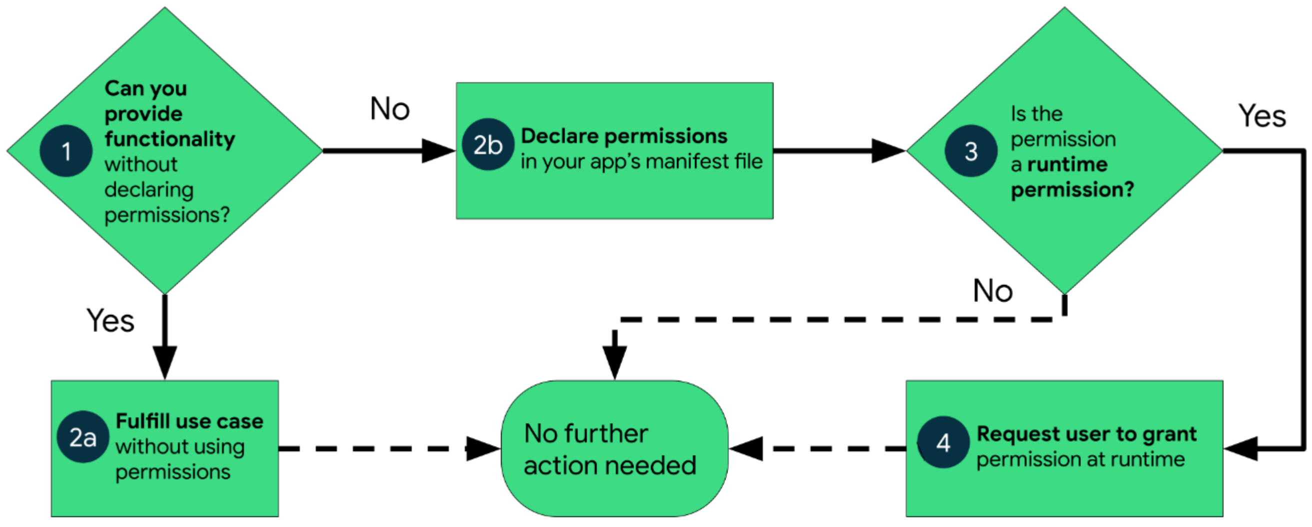
To view some sample apps that demonstrate the permissions workflow, visit the Android permissions samples repository on GitHub.

Workflow for using permissions

If your app offers functionality that might require access to restricted data or restricted actions, determine whether you can get the information or perform the actions without needing to declare permissions. You can fulfill many use cases in your app, such as taking photos, pausing media playback, and displaying relevant ads, without needing to declare any permissions.

If you decide that your app must access restricted data or perform restricted actions to fulfill a use case, declare the appropriate permissions. Some permissions, known as install-time permissions, are automatically granted when your app is installed. Other permissions, known as runtime permissions, require your app to go a step further and request the permission at runtime.

Figure 1 illustrates the workflow for using app permissions:



1 Can you provide functionality without declaring permissions?
No
2b Declare permissions in your app's manifest file
3 Is the permission a runtime permission?
Yes
No
Yes
2a Fulfill use case without using permissions
No further action needed
4 Request user to grant permission at runtime

Figure 1. *High-level workflow for using permissions on Android.*

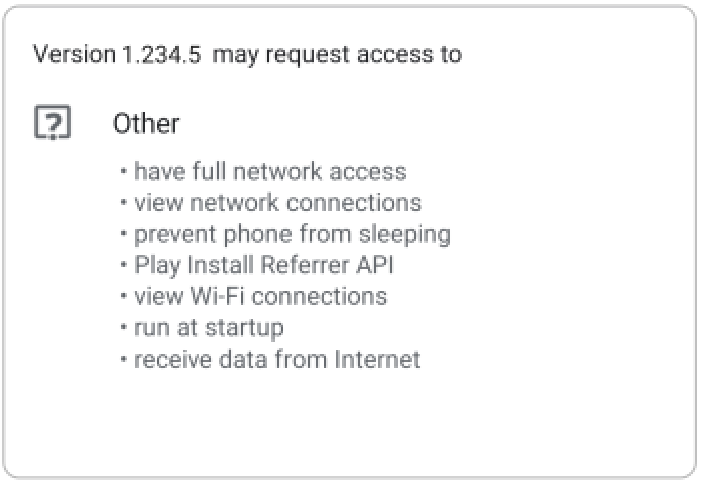
Types of permissions

Android categorizes permissions into different types, including install-time permissions, runtime permissions, and special permissions. Each permission's type indicates the scope of restricted data that your app can access, and the scope of restricted actions that your app can perform, when the system grants your app that permission. The protection level for each permission is based on its type and is shown on the permissions API reference page.

Install-time permissions

Install-time permissions give your app limited access to restricted data or let your app perform restricted actions that minimally affect the system or other apps. When you declare install-time permissions in your app, an app store presents an install-time permission notice to the user when they view an app's details page, as shown in figure 2. The system automatically grants your app the permissions when the user installs your app.

Android includes several sub-types of install-time permissions, including normal permissions and signature permissions.


Version 1.234.5 may request access to

Other

- have full network access
- view network connections
- prevent phone from sleeping
- Play Install Referrer API
- view Wi-Fi connections
- run at startup
- receive data from Internet

Figure 2. *The list of an app's install-time permissions, which appears in an app store.*

Normal permissions

These permissions allow access to data and actions that extend beyond your app's sandbox but present very little risk to the user's privacy and the operation of other apps.

The system assigns the normal protection level to normal permissions.

Signature permissions

The system grants a signature permission to an app only when the app is signed by the same certificate as the app or the OS that defines the permission.

Applications that implement privileged services, such as autofill or VPN services, also make use of signature permissions. These apps require service-binding signature permissions so that only the system can bind to the services.

Note: Some signature permissions aren't for use by third-party apps.

The system assigns the signature protection level to signature permissions.

Runtime permissions

Runtime permissions, also known as dangerous permissions, give your app additional access to restricted data or let your app perform restricted actions that more substantially affect the system and other apps. Therefore, you need to request runtime permissions in your app before you can access the restricted data or perform restricted actions. Don't assume that these permissions have been previously granted—check them and, if needed, request them before each access.

When your app requests a runtime permission, the system presents a runtime permission prompt, as shown in figure 3.

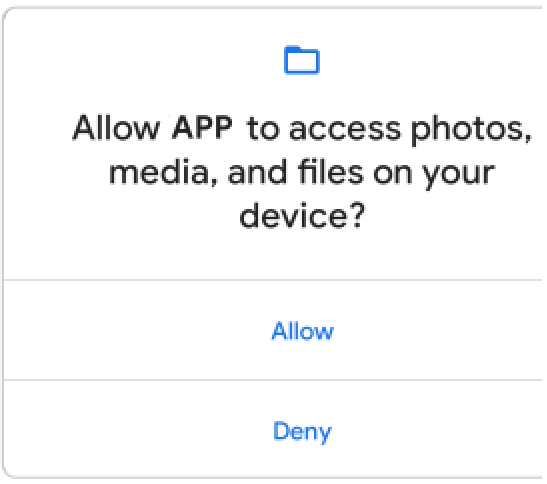

Allow APP to access photos, media, and files on your device?

Figure 3. *The system permission prompt that appears when your app requests a runtime permission.*

Many runtime permissions access *private user data*, a special type of restricted data that includes potentially sensitive information. Examples of private user data include location and contact information.

The microphone and camera provide access to particularly sensitive information. Therefore, the system helps you explain why your app accesses this information.

The system assigns the dangerous protection level to runtime permissions.

Special permissions

Special permissions correspond to particular app operations. Only the platform and OEMs can define special permissions. Additionally, the platform and OEMs usually define special permissions when they want to protect access to particularly powerful actions, such as drawing over other apps.

The **Special app access** page in system settings contains a set of user-toggleable operations. Many of these operations are implemented as special permissions.Learn more about how to request special permissions. The system assigns the appop protection level to special permissions.

Permission groups

Permissions can belong to permission groups. Permission groups consist of a set of logically related permissions. For example, permissions to send and receive SMS messages might belong to the same group, as they both relate to the application's interaction with SMS.

Permission groups help the system minimize the number of system dialogs that are presented to the user when an app requests closely related permissions. When a user is presented with a prompt to grant permissions for an application, permissions belonging to the same group are presented in the same interface. However, permissions can change groups without notice, so don't assume that a particular permission is grouped with any other permission.

Best practices

App permissions build on system security features and help Android support the following goals related to user privacy:

- **Control:** The user has control over the data that they share with apps.
- **Transparency:** The user understands what data an app uses and why the app accesses this data.
- **Data minimization:** An app accesses and uses only the data that's required for a specific task or action that the user invokes.

This section presents a set of core best practices for using permissions effectively in your app. For more details on how you can work with permissions on Android, visit the app permissions best practices page.

Request a minimal number of permissions

When the user requests a particular action in your app, your app should request only the permissions that it needs to complete that action. Depending on how you are using the permissions, there might be an alternative way to fulfill your app's use case without relying on access to sensitive information.

Associate runtime permissions with specific actions

Request permissions as late into the flow of your app's use cases as possible. For example, if your app lets users send audio messages to others, wait until the user has navigated to the messaging screen and has pressed the **Send audio message** button. After the user presses the button, your app can then request access to the microphone.

Consider your app's dependencies

When you include a library, you also inherit its permission requirements. Be aware of the permissions that each dependency requires and what those permissions are used for.

Be transparent

When you make a permissions request, be clear about what you're accessing, why, and what functionalities are affected if permissions are denied, so users can make informed decisions.

Make system accesses explicit

When you access sensitive data or hardware, such as the camera or microphone, provide a continuous indication in your app if the system doesn't already provide these indicators. This reminder helps users understand exactly when your app accesses restricted data or performs restricted actions.

Permissions in system components

Permissions aren't only for requesting system functionality. Your app's system components can restrict which other apps can interact with your app, as described on the page about how to restrict interactions with other apps.

Declare app permissions

As mentioned in the workflow for using permissions, if your app requests app permissions, you must declare these permissions in your app's manifest file. These declarations help app stores and users understand the set of permissions that your app might request.

The process to request a permission depends on the type of permission:

- If the permission is an install-time permission, such as a normal permission or a signature permission, the permission is granted automatically at install time.
- If the permission is a runtime permission or special permission, and if your app is installed on a device that runs Android 6.0 (API level 23) or higher, you must request the runtime permission or special permission yourself.

Caution: Carefully consider which permissions to declare in your app's manifest. Add only the permissions that your app needs. For each permission that your app requests, make sure that it offers clear benefits to users and that the request is done in a way that's obvious to them.

Add declaration to app manifest

To declare a permission that your app might request, include the appropriate <uses-permission> element in your app's manifest file. For example, an app that needs to access the camera has this line in AndroidManifest.xml:

```
<manifest ...>
    <uses-permission android:name="android.permission.CAMERA"/>
    <application ...>
        ...
    </application>
</manifest>
```

Declare hardware as optional

Some permissions, such as CAMERA, let your app access pieces of hardware that only some Android devices have. If your app declares one of these hardware-associated permissions, consider whether your app can still run on a device that doesn't have that hardware. In most cases, hardware is optional, so it's better to declare the hardware as optional by setting android:required to false in your <uses-feature> declaration, as shown in the following code snippet from an AndroidManifest.xml file:

```
<manifest ...>
    <application>
        ...
    </application>
    <uses-feature android:name="android.hardware.camera"
                  android:required="false" />
<manifest>
```

Caution: If you don't set **android:required** to **false** in your **<uses-feature>** declaration, Android assumes that the hardware is required for your app to run. The system then prevents some devices from being able to install your app.

Determine hardware availability

If you declare hardware as optional, it's possible for your app to run on a device that doesn't have that hardware. To check whether a device has a specific piece of hardware, use the hasSystemFeature() method, as shown in the following code snippet. If the hardware isn't available, gracefully disable that feature in your app.

JAVA(Code)

```java
// Check whether your app is running on a device that has a
// front-facing camera.
if (getApplicationContext().getPackageManager().hasSystemFeature(
        PackageManager.FEATURE_CAMERA_FRONT)) {
    // Continue with the part of your app's workflow that requires a
    // front-facing camera.
} else {
    // Gracefully degrade your app experience.
}
```

Declare permissions by API level

To declare a permission only on devices that support runtime permissions—that is, devices that run Android 6.0 (API level 23) or higher—include the <uses-permission-sdk-23> element instead of the <uses-permission> element.

When using either of these elements, you can set the maxSdkVersion attribute to indicate that devices running a version of Android higher than the specified value don't need a particular permission. This lets you eliminate unnecessary permissions while still providing compatibility for older devices.

For example, your app might show media content, such as photos or videos, that the user created while in your app. In this situation, you don't need to use the READ_EXTERNAL_STORAGE permission on devices that run Android 10 (API level 29) or higher, as long as your app targets Android 10 or higher. However, for compatibility with older devices, you can declare the READ_EXTERNAL_STORAGE permission and set the android:maxSdkVersion to 28.

Request runtime permissions

Every Android app runs in a limited-access sandbox. If your app needs to use resources or information outside of its own sandbox, you can declare a runtime permission and set up a permission request that provides this access. These steps are part of the workflow for using permissions.

Note: Some permissions guard access to system resources that are particularly sensitive or aren't directly related to user privacy. For these special permissions, follow a different process.

If you declare any dangerous permissions, and if your app is installed on a device that runs Android 6.0 (API level 23) or higher, you must request the dangerous permissions at runtime by following the steps in this guide.

If you don't declare any dangerous permissions, or if your app is installed on a device that runs Android 5.1 (API level 22) or lower, the permissions are automatically granted, and you don't need to complete any of the remaining steps on this page.

Basic principles

The basic principles for requesting permissions at runtime are as follows:

- Ask for a permission in context, when the user starts to interact with the feature that requires it.
- Don't block the user. Always provide the option to cancel an educational UI flow, such as a flow that explains the rationale for requesting permissions.
- If the user denies or revokes a permission that a feature needs, gracefully degrade your app so that the user can continue using your app, possibly by disabling the feature that requires the permission.
- Don't assume any system behavior. For example, don't assume that permissions appear in the same *permission group*. A permission group merely helps the system minimize the number of system dialogs that are presented to the user when an app requests closely related permissions.

Workflow for requesting permissions

Before you declare and request runtime permissions in your app, evaluate whether your app needs to do so. You can fulfill many use cases in your app, such as taking photos, pausing media playback, and displaying relevant ads, without needing to declare any permissions.

If you conclude that your app needs to declare and request runtime permissions, complete these steps:

1. In your app's manifest file, declare the permissions that your app might need to request.
2. Design your app's UX so that specific actions in your app are associated with specific runtime permissions. Let users know which actions might require them to grant permission for your app to access private user data.
3. Wait for the user to invoke the task or action in your app that requires access to specific private user data. At that time, your app can request the runtime permission that's required for accessing that data.
4. Check whether the user has already granted the runtime permission that your app requires. If so, your app can access the private user data. If not, continue to the next step.

   You must check whether you have a permission every time you perform an operation that requires that permission.
5. Check whether your app should show a rationale to the user, explaining why your app needs the user to grant a particular runtime permission. If the system determines that your app shouldn't show a rationale, continue to the next step directly, without showing a UI element.

   If the system determines that your app should show a rationale, however, present the rationale to the user in a UI element. In this rationale, clearly explain what data your app is trying to access and what benefits the app can provide to the user if they grant the runtime permission. After the user acknowledges the rationale, continue to the next step.
6. Request the runtime permission that your app requires to access the private user data. The system displays a runtime permission prompt, such as the one shown on the permissions overview page.
7. Check the user's response—whether they chose to grant or deny the runtime permission.
8. If the user granted the permission to your app, you can access the private user data. If the user denied the permission instead, gracefully degrade your app experience so that it provides functionality to the user without the information that's protected by that permission.

Figure 1 illustrates the workflow and set of decisions associated with this process:

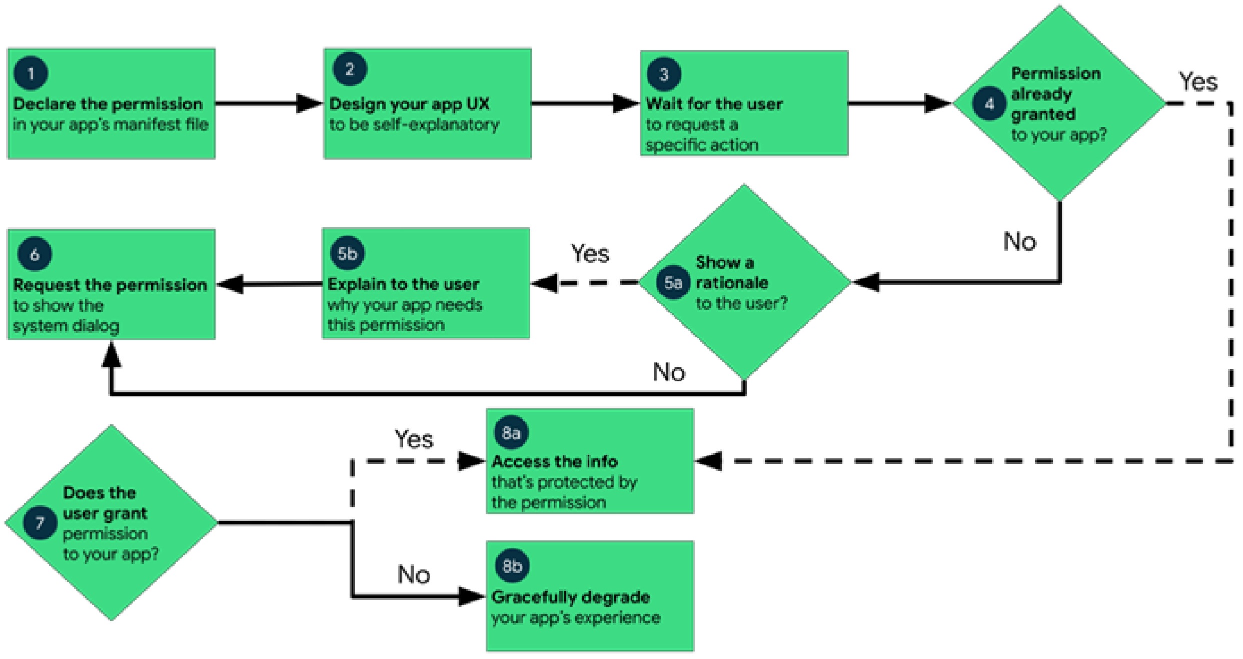

1 Declare the permission in your app's manifest file
2 Design your app UX to be self-explanatory
3 Wait for the user to request a specific action
4 Permission already granted to your app?
Yes
No
5a Show a rationale to the user?
Yes
No
5b Explain to the user why your app needs this permission
6 Request the permission to show the system dialog
7 Does the user grant permission to your app?
Yes
No
8a Access the info that's protected by the permission
8b Gracefully degrade your app's experience

Figure 1. *Diagram that shows the workflow for declaring and requesting runtime permissions on Android.*

Determine whether your app was already granted the permission

To check whether the user already granted your app a particular permission, pass that permission into the ContextCompat.checkSelfPermission() method. This method returns either PERMISSION_DENIED or PERMISSION_GRANTED, depending on whether your app has the permission.

Explain why your app needs the permission

The permissions dialog shown by the system when you call requestPermissions() says what permission your app wants, but doesn't say why. In some cases, the user might find that puzzling. It's a good idea to explain to the user why your app wants the permissions before you call requestPermissions().

Research shows that users are much more comfortable with permissions requests if they know why the app needs them, such as whether the permission is needed to support a core feature of the app or for advertising. As a result, if you're only using a fraction of the API calls that fall under a permission group, it helps to explicitly list which of those permissions you're using and why. For example, if you're only using coarse location, let the user know this in your app description or in help articles about your app.

Under certain conditions, it's also helpful to let users know about sensitive data access in real time. For example, if you're accessing the camera or microphone, it's a good idea to let the user know by using a notification icon somewhere in your app, or in the notification tray (if the application is running in the background), so it doesn't seem like you're collecting data surreptitiously.

Note: Starting in Android 12 (API level 31), privacy indicators notify the user whenever applications access the microphone or camera.

Ultimately, if you need to request a permission to make something in your app work, but the reason isn't clear to the user, find a way to let the user know why you need the most sensitive permissions.

If the ContextCompat.checkSelfPermission() method returns PERMISSION_DENIED, call shouldShowRequestPermissionRationale(). If this method returns true, show an educational UI to the user. In this UI, describe why the feature that the user wants to enable needs a particular permission.

Additionally, if your app requests a permission related to location, microphone, or camera, consider explaining why your app needs access to this information.

Request permissions

After the user views an educational UI, or the return value of shouldShowRequestPermissionRationale() indicates that you don't need to show an educational UI, request the permission. Users see a system permission dialog, where they can choose whether to grant a particular permission to your app.

To do this, use the RequestPermission contract, included in an AndroidX library, where you allow the system to manage the permission request code for you. Because using the RequestPermission contract simplifies your logic, it is the recommended solution when possible. However, if needed you can also manage a request code yourself as part of the permission request and include this request code in your permission callback logic.

Allow the system to manage the permission request code

To allow the system to manage the request code that's associated with a permissions request, add dependencies on the following libraries in your module's build.gradle file:

- androidx.activity, version 1.2.0 or later
- androidx.fragment, version 1.3.0 or later

You can then use one of the following classes:

- To request a single permission, use RequestPermission.
- To request multiple permissions at the same time, use RequestMultiplePermissions.

The following steps show how to use the RequestPermission contract. The process is nearly the same for the RequestMultiplePermissions contract.

1. In your activity or fragment's initialization logic, pass in an implementation of ActivityResultCallback into a call to registerForActivityResult().
   The ActivityResultCallback defines how your app handles the user's response to the permission request.

   Keep a reference to the return value of registerForActivityResult(), which is of type ActivityResultLauncher.
2. To display the system permissions dialog when necessary, call the launch() method on the instance of ActivityResultLauncher that you saved in the previous step.

   After launch() is called, the system permissions dialog appears. When the user makes a choice, the system asynchronously invokes your implementation of ActivityResultCallback, which you defined in the previous step.

Note: Your app *cannot* customize the dialog that appears when you call **launch()**. To provide more information or context to the user, change your app's UI so that it's easier for users to understand why a feature in your app needs a particular permission. For example, you might change the text in the button that enables the feature.
Also, the text in the system permission dialog references the permission group associated with the permission that you requested. This permission grouping is designed for system ease-of-use, and your app shouldn't rely on permissions being within or outside of a specific permission group.

The following code snippet shows how to handle the permissions response:

JAVA(Code)

```java
// Register the permissions callback, which handles the user's response to the
// system permissions dialog. Save the return value, an instance of
// ActivityResultLauncher, as an instance variable.
private ActivityResultLauncher<String> requestPermissionLauncher =
    registerForActivityResult(new RequestPermission(), isGranted -> {
        if (isGranted) {
            // Permission is granted. Continue the action or workflow in your
            // app.
        } else {
            // Explain to the user that the feature is unavailable because the
            // feature requires a permission that the user has denied. At the
            // same time, respect the user's decision. Don't link to system
            // settings in an effort to convince the user to change their
            // decision.
        }
    });
```

And this code snippet demonstrates the recommended process to check for a permission and to request a permission from the user when necessary:

JAVA(Code)

```java
if (ContextCompat.checkSelfPermission(
        CONTEXT, Manifest.permission.REQUESTED_PERMISSION) ==
        PackageManager.PERMISSION_GRANTED) {
    // You can use the API that requires the permission.
    performAction(...);
} else if (ActivityCompat.shouldShowRequestPermissionRationale(
        this, Manifest.permission.REQUESTED_PERMISSION)) {
    // In an educational UI, explain to the user why your app requires this
    // permission for a specific feature to behave as expected, and what
    // features are disabled if it's declined. In this UI, include a
    // "cancel" or "no thanks" button that lets the user continue
    // using your app without granting the permission.
    showInContextUI(...);
} else {
    // You can directly ask for the permission.
    // The registered ActivityResultCallback gets the result of this request.
    requestPermissionLauncher.launch(
            Manifest.permission.REQUESTED_PERMISSION);
}
```

Manage the permission request code yourself

As an alternative to allowing the system to manage the permission request code, you can manage the permission request code yourself. To do so, include the request code in a call to requestPermissions().

Note: Your app cannot customize the dialog that appears when you call **requestPermissions()**. The text in the system permission dialog references a permission group, but this permission grouping is designed for system ease-of-use. Your app shouldn't rely on permissions being within or outside of a specific permission group.

The following code snippet demonstrates how to request a permission using a request code:

JAVA(Code)

```java
if (ContextCompat.checkSelfPermission(
        CONTEXT, Manifest.permission.REQUESTED_PERMISSION) ==
        PackageManager.PERMISSION_GRANTED) {
    // You can use the API that requires the permission.
    performAction(...);
} else if (ActivityCompat.shouldShowRequestPermissionRationale(
        this, Manifest.permission.REQUESTED_PERMISSION)) {
    // In an educational UI, explain to the user why your app requires this
    // permission for a specific feature to behave as expected, and what
    // features are disabled if it's declined. In this UI, include a
    // "cancel" or "no thanks" button that lets the user continue
    // using your app without granting the permission.
    showInContextUI(...);
} else {
    // You can directly ask for the permission.
    requestPermissions(CONTEXT,
            new String[] { Manifest.permission.REQUESTED_PERMISSION },
            REQUEST_CODE);

}
```

After the user responds to the system permissions dialog, the system then invokes your app's implementation of onRequestPermissionsResult(). The system passes in the user response to the permission dialog, as well as the request code that you defined, as shown in the following code snippet:

JAVA(Code)

```
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions,
        int[] grantResults) {
    switch (requestCode) {
        case PERMISSION_REQUEST_CODE:
            // If request is cancelled, the result arrays are empty.
            if (grantResults.length > 0 &&
                    grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                // Permission is granted. Continue the action or workflow
                // in your app.
            }  else {
                // Explain to the user that the feature is unavailable because
                // the feature requires a permission that the user has denied.
                // At the same time, respect the user's decision. Don't link to
                // system settings in an effort to convince the user to change
                // their decision.
            }
            return;
        }
        // Other 'case' lines to check for other
        // permissions this app might request.
    }
}
```

Request location permissions

When you request location permissions, follow the same best practices as for any other runtime permission. One important difference when it comes to location permissions is that the system includes multiple permissions related to location. Which permissions you request, and how you request them, depend on the location requirements for your app's use case.

Foreground location

If your app contains a feature that shares or receives location information only once, or for a defined amount of time, then that feature requires foreground location access. Some examples include the following:

- Within a navigation app, a feature lets users get turn-by-turn directions.
- Within a messaging app, a feature lets users share their current location with another user.

The system considers your app to be using foreground location if a feature of your app accesses the device's current location in one of the following situations:

- An activity that belongs to your app is visible.
- Your app is running a foreground service. When a foreground service is running, the system raises user awareness by showing a persistent notification. Your app retains access when it's placed in the background, such as when the user presses the Home button on their device or turns their device's display off.
  On Android 10 (API level 29) and higher, you must declare a foreground service type of location, as shown in the following code snippet. On earlier versions of Android, it's recommended that you declare this foreground service type.

```
<!-- Recommended for Android 9 (API level 28) and lower. -->
<!-- Required for Android 10 (API level 29) and higher. -->
<service
    android:name="MyNavigationService"
    android:foregroundServiceType="location" ... >
    <!-- Any inner elements go here. -->
</service>
```

You declare a need for foreground location when your app requests either the ACCESS_COARSE_LOCATION permission or the ACCESS_FINE_LOCATION permission, as shown in the following snippet:

```
<manifest ... >
  <!-- Include this permission any time your app needs location information. -->
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

  <!-- Include only if your app benefits from precise location access. -->
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

Background location

An app requires background location access if a feature within the app constantly shares location with other users or uses the Geofencing API. Several examples include the following:

- Within a family location sharing app, a feature lets users continuously share location with family members.
- Within an IoT app, a feature lets users configure their home devices such that they turn off when the user leaves their home and turn back on when the user returns home.

The system considers your app to be using background location if it accesses the device's current location in any situation other than the ones described in the foreground location section. The background location accuracy is the same as the foreground location accuracy, which depends on the location permissions that your app declares.

On Android 10 (API level 29) and higher, you must declare the ACCESS_BACKGROUND_LOCATION permission in your app's manifest to request background location access at runtime. On earlier versions of Android, when your app receives foreground location access, it automatically receives background location access as well.

```
<manifest ... >
  <!-- Required only when requesting background location access on
      Android 10 (API level 29) and higher. -->
  <uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
</manifest>
```

Note: The Google Play Store has a location policy concerning device location, restricting background location access to apps that need it for their core functionality and meet related policy requirements.

Handle permission denial

If the user denies a permission request, your app should help users understand the implications of denying the permission. In particular, your app should make users aware of the features that don't work because of the missing permission. When you do so, keep the following best practices in mind:

- **Guide the user's attention.** Highlight a specific part of your app's UI where there's limited functionality because your app doesn't have the necessary permission. Examples of what you could do include the following:
    - Show a message where the feature's results or data would have appeared.
    - Display a different button that contains an error icon and color.
- **Be specific.** Don't display a generic message. Instead, make clear which features are unavailable because your app doesn't have the necessary permission.
- **Don't block the user interface.** In other words, don't display a full-screen warning message that prevents users from continuing to use your app at all.

Tip: Your app should encourage the best user experience possible, even after permission denials. For example, if microphone access is denied, you should still promote full usability of text functionality.

At the same time, your app should respect the user's decision to deny a permission. Starting in Android 11 (API level 30), if the user taps **Deny** for a specific permission more than once during your app's lifetime of installation on a device, the user doesn't see the system permissions dialog if your app requests that permission again. The user's action implies "don't ask again." On previous versions, users saw the system permissions dialog each time your app requested a permission, unless they had previously selected a "don't ask again" checkbox or option.

If a user denies a permission request more than once, this is considered a permanant denial. It's very important to only prompt users for permissions when they need access to a specific feature, otherwise you might inadvertently lose the ability to re-request permissions.

In certain situations, the permission might be denied automatically, without the user taking any action. (A permission might be *granted* automatically as well.) It's important to not assume anything about automatic behavior. Each time your app needs to access functionality that requires a permission, check that your app is still granted that permission.

To provide the best user experience when asking for app permissions, also see App permissions best practices.

Inspect denial status when testing and debugging

To identify whether an app has been permanently denied permissions (for debugging and testing purposes), use the following command:

```
adb shell dumpsys package PACKAGE_NAME
```

Where *PACKAGE_NAME* is the name of the package to inspect.

The output of the command contains sections that look like this:

```
...
runtime permissions:
  android.permission.POST_NOTIFICATIONS: granted=false, flags=[
USER_SENSITIVE_WHEN_GRANTED|USER_SENSITIVE_WHEN_DENIED]

 android.permission.ACCESS_FINE_LOCATION: granted=false, flags=[
USER_SET|USER_FIXED|USER_SENSITIVE_WHEN_GRANTED|USER_SENSITIVE_WHEN_DENIE
D]

 android.permission.BLUETOOTH_CONNECT: granted=false, flags=[
USER_SENSITIVE_WHEN_GRANTED|USER_SENSITIVE_WHEN_DENIED]
...
```

Permissions that have been denied once by the user are flagged by USER_SET. Permissions that have been denied permanently by selecting **Deny** twice are flagged by USER_FIXED.

To ensure that testers see the request dialog during testing, reset these flags when you're done debugging your app. To do this, use the command:

```
adb shell pm clear-permission-flags PACKAGE_NAME PERMISSION_NAME user-set user-fixed
```

PERMISSION_NAME is the name of the permission you want to reset.

To view a complete list of Android app permissions, visit the permissions API reference page.

One-time permissions

Starting in Android 11 (API level 30), whenever your app requests a permission related to location, microphone, or camera, the user-facing permissions dialog contains an option called **Only this time**, as shown in figure 2. If the user selects this option in the dialog, your app is granted a temporary *one-time permission*.

Your app can then access the related data for a period of time that depends on your app's behaviour and the user's actions:

- While your app's activity is visible, your app can access the data.
- If the user sends your app to the background, your app can continue to access the data for a short period of time.
- If you launch a foreground service while the activity is visible, and the user then moves your app to the background, your app can continue to access the data until the foreground service stops.

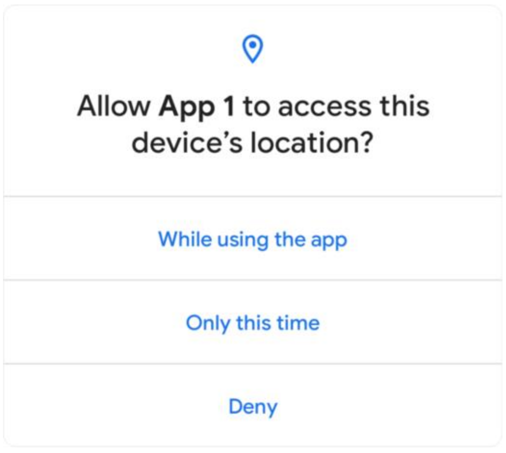


Figure 2. *System dialog that appears when an app requests a one-time permission.*

App process terminates when permission revoked

If the user revokes the one-time permission, such as in system settings, your app can't access the data, regardless of whether you launched a foreground service. As with any permission, if the user revokes your app's one-time permission, your app's process terminates.

When the user next opens your app and a feature in your app requests access to location, microphone, or camera, the user is prompted for the permission again.

Note: If your app already follows best practices when it requests runtime permissions, you don't need to add or change any logic in your app to support one-time permissions.

Reset unused permissions

Android provides several ways to reset unused runtime permissions to their default, denied state:

- An API where you can proactively remove your app's access to an unused runtime permission.
- A system mechanism that automatically resets the permissions of unused apps.

Remove app access

On Android 13 (API level 33) and higher, you can remove your app's access to runtime permissions that your app no longer requires. When you update your app, perform this step so that users are more likely to understand why your app continues to request specific permissions. This knowledge helps build user trust in your app.

To remove access to a runtime permission, pass the name of that permission into revokeSelfPermissionOnKill(). To remove access to a group of runtime permissions at the same time, pass a collection of permission names into revokeSelfPermissionsOnKill(). The permission removal process happens asynchronously and kills all processes associated with your app's UID.

Note: For system settings to show that your app doesn't access data in a particular permission group, you must remove access to **all** permissions in that permission group. In this case, it can be helpful to call **revokeSelfPermissionsOnKill()** and pass in multiple permissions within the permission group.

For the system to remove your app's access to the permissions, all processes tied to your app must be killed. When you call the API, the system determines when it's safe to kill these processes. Usually, the system waits until your app spends an extended period of time running in the background instead of the foreground.

To inform the user that your app no longer requires access to specific runtime permissions, show a dialog the next time the user launches your app. This dialog can include the list of permissions.

Auto-reset permissions of unused apps

If your app targets Android 11 (API level 30) or higher and isn't used for a few months, the system protects user data by automatically resetting the sensitive runtime permissions that the user had granted your app. Learn more in the guide about app hibernation.

Request to become the default handler if necessary

Some apps depend on access to sensitive user information related to call logs and SMS messages. If you want to request the permissions specific to call logs and SMS messages and publish your app to the Play Store, you must prompt the user to set your app as the *default handler* for a core system function before requesting these runtime permissions.
For more information on default handlers, including guidance on showing a default handler prompt to users, see the guide about permissions used only in default handlers.

Grant all runtime permissions for testing purposes

To grant all runtime permissions automatically when you install an app on an emulator or test device, use the -g option for the adb shell install command, as demonstrated in the following code snippet:

```
adb shell install -g PATH_TO_APK_FILE
```

Additional resources

For additional information about permissions, read these articles:

- Permissions overview
- App permissions best practices

To learn more about requesting permissions, review the permissions samples
You can also complete this codelab that demonstrates privacy best practices.

Request special permissions

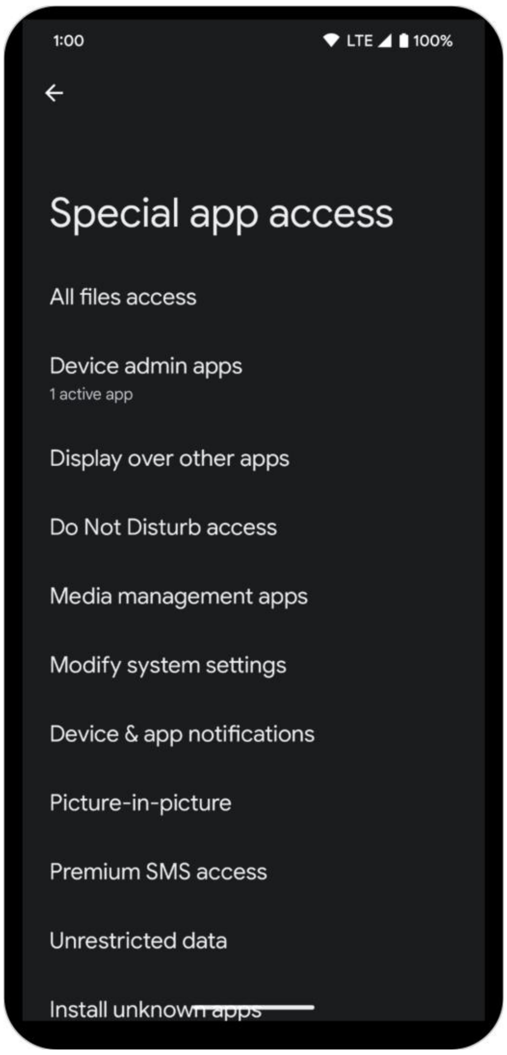
A *special permission* guards access to system resources that are particularly sensitive or not directly related to user privacy. These permissions are different than install-time permissions and runtime permissions.

Some examples of special permissions include:

- Scheduling exact alarms.
- Displaying and drawing over other apps.
- Accessing all storage data.

Apps that declare a special permission are shown in the **Special app access** page in system settings (figure 1). To grant a special permission to the app, a user must navigate to this page: **Settings > Apps > Special app access**.

Note: Special permissions should be only used in specific use cases, and there may be policy implications to adding them in your app,


1:00 LTE 100%

Special app access

All files access

Device admin apps
1 active app

Display over other apps

Do Not Disturb access

Media management apps

Modify system settings

Device & app notifications

Picture-in-picture

Premium SMS access

Unrestricted data

Install unknown apps

***Figure 1**: The **Special app access** screen in system settings.*

Workflow

To request a special permission, do the following:

1. In your app's manifest file, declare the special permissions that your app might need to request.
2. Design your app's UX so that specific actions in your app are associated with specific special permissions. Let users know which actions might require them to grant permission for your app to access private user data.
3. Wait for the user to invoke the task or action in your app that requires access to specific private user data. At that time, your app can request the special permission that's required for accessing that data.
4. Check whether the user has already granted the special permission that your app requires. To do so, use each permission's custom checking function. If granted, your app can access the private user data. If not, continue to the next step. Note: You must check whether you have the permission every time you perform an operation that requires that permission.
5. Present a rationale to the user in a UI element that clearly explains what data your app is trying to access and what benefits the app can provide to the user if they grant the special permission. In addition, since your app sends users to system settings to grant the permission, also include brief instructions that explain how users can grant the permission there. The rationale UI should provide a clear option for the user to opt-out of granting the permission. After the user acknowledges the rationale, continue to the next step.
6. Request the special permission that your app requires to access the private user data. This likely involves an intent to the corresponding page in system settings where the user can grant the permission. Unlike runtime permissions, there is no popup permission dialog.
7. Check the user's response – whether they chose to grant or deny the special permission – in the onResume() method.
8. If the user granted the permission to your app, you can access the private user data. If the user denied the permission instead, gracefully degrade your app experience so that it provides functionality to the user without the information that's protected by that permission.

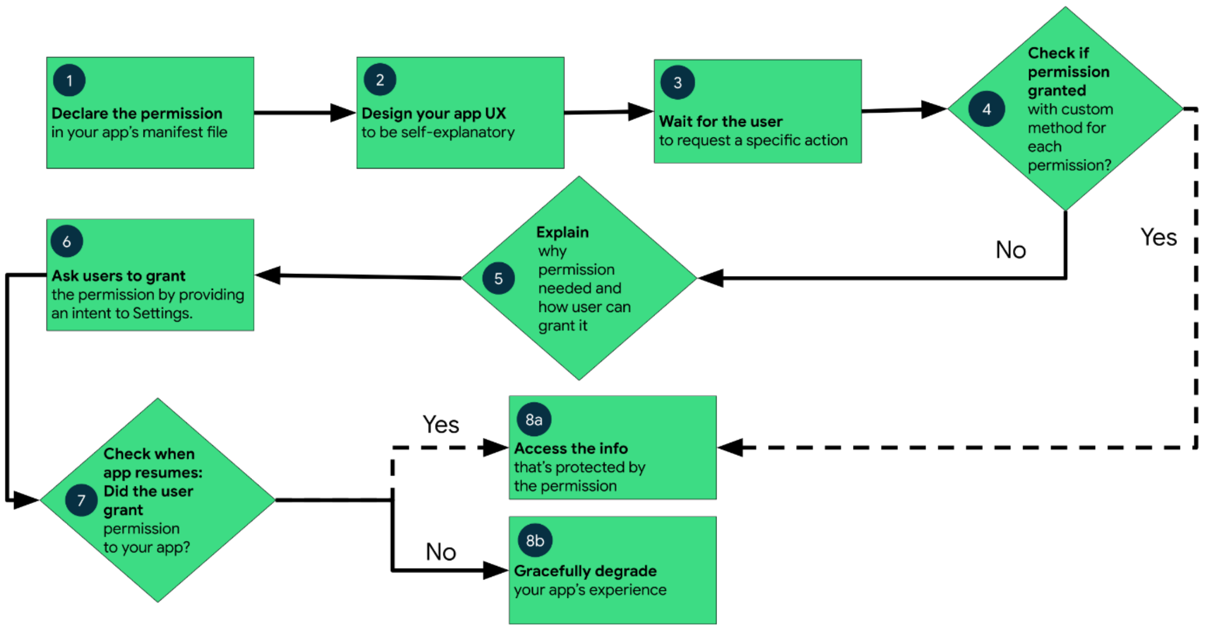

1 Declare the permission in your app's manifest file
2 Design your app UX to be self-explanatory
3 Wait for the user to request a specific action
4 Check if permission granted with custom method for each permission?
Yes
No
5 Explain why permission needed and how user can grant it
6 Ask users to grant the permission by providing an intent to Settings.
7 Check when app resumes: Did the user grant permission to your app?
Yes
No
8a Access the info that's protected by the permission
8b Gracefully degrade your app's experience

Figure 2: *Workflow for declaring and requesting special permissions on Android.*

Request special permissions

Unlike runtime permissions, the user must grant special permissions from the **Special App Access** page in system settings. Apps can send users there using an intent, which pauses the app and launches the corresponding settings page for a given special permission. After the user returns to the app, the app can check if the permission has been granted in the onResume() function.

The following sample code shows how to request the SCHEDULE_EXACT_ALARMS special permission from users:

```
val alarmManager = getSystemService<AlarmManager>()!!
when {
   // if permission is granted, proceed with scheduling exact alarms…
   alarmManager.canScheduleExactAlarms() -> {
        alarmManager.setExact(...)
   }
   else -> {
        // ask users to grant the permission in the corresponding settings page
        startActivity(Intent(ACTION_REQUEST_SCHEDULE_EXACT_ALARM))
   }
}
```

Sample code to check the permission and handle user decisions in onResume():

```
override fun onResume() {
   // ...

   if (alarmManager.canScheduleExactAlarms()) {
        // proceed with the action (setting exact alarms)
        alarmManager.setExact(...)
   }
   else {
        // permission not yet approved. Display user notice and gracefully degrade
        your app experience.
        alarmManager.setWindow(...)
   }
}
```

Best practices and tips

The following sections provide some best practices and considerations when requesting special permissions.

Each permission has its own check method

Special permissions operate differently than runtime permissions. Instead, refer to the permissions API reference page and use the custom access check functions for each special permission. Examples include AlarmManager#canScheduleExactAlarms() for the SCHEDULE_EXACT_ALARMS permission and Environment#isExternalStorageManager() for the MANAGE_EXTERNAL_STORAGE permission.

Request in-context

Similar to runtime permissions, apps should request special permissions in-context when the user requests a specific action that requires the permission. For example, wait to request the SCHEDULE_EXACT_ALARMS permission until the user schedules an email to be sent at a specific time.

Explain the request

Provide a rationale before redirecting to system settings. Since users leave the app temporarily to grant special permissions, show an in-app UI before you launch the intent to the **Special App Access** page in system settings. This UI should clearly explain why the app needs the permission and how the user should grant it on the settings page.