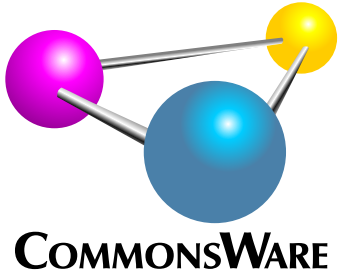
FINAL Version

Elements of Android R

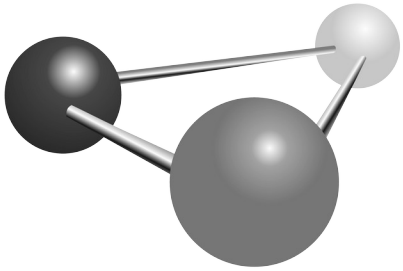
Mark L. Murphy



CommonsWare

Elements of Android R

by Mark L. Murphy



COMMONSWARE

Elements of Android R
by Mark L. Murphy

Printed in the United States of America.

Printing History:
November 2020: FINAL Version

Table of Contents

Headings formatted in ***bold-italic*** have changed since the last version.

Preface

Thanks!

Thanks for your continued interest in Android! Android advances year after year, and 2020's Android 11 (R) continues that pattern. Many developers ignore new Android versions until some concrete problem causes them grief. Hopefully, you are reading this in advance of when Android 11 ships to lots of devices, so you can head off any problems before they turn into customer complaints.

(on the other hand, if you are reading this in *response* to Android 11 customer complaints... sorry!)

And thanks for your interest in this book and CommonsWare's overall line of Android books!

The Book's Prerequisites

This book is designed for developers with 1+ years of Android app development experience. If you are fairly new to Android, please consider reading *Elements of Android Jetpack*, *Exploring Android*, or both, before continuing with this book.

Also note that this book's examples are written in Kotlin.

What's New in the Final Version?

This book is almost unchanged from the previous version.

Warescription

If you purchased the Warescription, read on! If you obtained this book from other channels, feel free to jump ahead.

The Warescription entitles you, for the duration of your subscription, to digital editions of this book and its updates, in PDF, EPUB, and Kindle (MOBI/KF8) formats, plus the ability to read the book online at the Warescription Web site. You also have access to other books that CommonsWare publishes during that subscription period.

Each subscriber gets personalized editions of all editions of each title. That way, your books are never out of date for long, and you can take advantage of new material as it is made available.

However, you can only download the books while you have an active Warescription. There is a grace period after your Warescription ends: you can still download the book until the next book update comes out after your Warescription ends. After that, you can no longer download the book. Hence, **please download your updates as they come out**. You can find out when new releases of this book are available via:

1. The CommonsBlog
2. The CommonsWare Twitter feed
3. Opting into emails announcing each book release — log into the Warescription site and choose Configure from the nav bar
4. Just check back on the Warescription site every month or two

Subscribers also have access to other benefits, including:

- "Office hours" — online chats to help you get answers to your Android application development questions. You will find a calendar for these on your Warescription page.
- A Stack Overflow "bump" service, to get additional attention for a question that you have posted there that does not have an adequate answer.
- A discussion board for asking arbitrary questions about Android app development.

Source Code and Its License

The source code in this book is licensed under the Apache 2.0 License, in case you have the desire to reuse any of it.

Copying source code directly from the book, in the PDF editions, works best with Adobe Reader, though it may also work with other PDF viewers. Some PDF viewers, for reasons that remain unclear, foul up copying the source code to the clipboard when it is selected.

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on *1 November 2024*. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license, visit the Creative Commons Web site

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Acknowledgments

The author would like to thank the Google team responsible for Android 11.

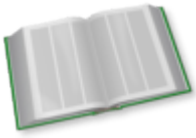
Storage Shifts

Android 10 introduced what Google calls “scoped storage” and what the author of this book called “the death of external storage”.

Android 11 tweaks scoped storage some more, improving things in some areas and causing new and exciting challenges in others.

Recapping What Happened in Android 10

Before we dive into the Android 11 changes to scoped storage, let’s quickly review what happened in Android 10.



You can learn more about scoped storage in Android 10 in the "The Death of External Storage" chapter of *Elements of Android Q*!

Limited Filesystem Access

While apps can still use `getExternalFilesDir()` and other methods on `Context` to work with external and removable storage, everything else has been blocked. Notably, the methods on `Environment` like `getExternalStorageDirectory()` and `getExternalStoragePublicDirectory()` are deprecated. And, if you try to use those directories, you will find that your app lacks access, even if you hold `READ_EXTERNAL_STORAGE` and/or `WRITE_EXTERNAL_STORAGE`.

Roughly speaking, there are three alternatives for addressing this limitation.

Alternative #1: Storage Access Framework

For general-purpose content, Google expects you to use the Storage Access Framework:

- `ACTION_OPEN_DOCUMENT` to have the user choose a piece of content
- `ACTION_CREATE_DOCUMENT` to create a new piece of content in a user-chosen location
- `ACTION_OPEN_DOCUMENT_TREE` to have the user choose a "document tree" (e.g., a directory) that you can then use for reading and writing

The actual mechanics of the Storage Access Framework did not change in Android 10, merely its importance.

Alternative #2: `MediaStore`

For apps that work with media and wish to place content in common media locations, `MediaStore` is still an option. However, the behavior of `MediaStore` changed some in Android 10 and again in Android 11 — we will explore that more in the next chapter.

Alternative #3: Opt Out of the Change

You could add `android:requestLegacyExternalStorage="true"` to the `<application>` element in the manifest to say that you want the "legacy" storage model. In other words, `android:requestLegacyExternalStorage="true"` has your app running on Android 10 behave much as it would on Android 9.

Alternatively, simply having a `targetSdkVersion` below 29 would give you the same effect.

Let's Do the Time Warp

Back when Android 10 was still Android Q, we were told that `android:requestLegacyExternalStorage="true"` would no longer work once we raised `targetSdkVersion` to 29.

Somewhere along the line, Google rescinded that, and the author of this book missed that change.

It appears that in the final release of Android 10, Google allowed `android:requestLegacyExternalStorage="true"` regardless of `targetSdkVersion`.

Since Android 11 also honors it — at least until you reach `targetSdkVersion 30` — this gives you a bit more time to adapt to scoped storage.

Extending the Opt-Out

Android 11 also offers `android:preserveLegacyExternalStorage="true"`. This says:

- For users who upgrade the app, opt out of scoped storage
- For users who freshly install the app, use scoped storage as normal

There is no timetable given for if and when `android:preserveLegacyExternalStorage` might no longer be honored, though the documentation states:

> Note that this may not always be respected due to policy or backwards compatibility reasons.

Also, it means that different users *of the same app version* will get different storage behavior, depending on how the user got that app version (fresh install vs. upgrade).

And, of course, since this is new to Android 11, this should have no effect on Android 10 devices.

Given all of that, this seems like an attribute to avoid.

Raw Paths Support

However, even without the opt-out, `READ_EXTERNAL_STORAGE` works again, more or less as it did from Android 4.4 through Android 9. If you request it, and the user grants it, you can traverse external storage as you were used to.

However, there are major caveats:

- You still do not have access to `Android/` and its subdirectories. We will see this limitation again with the Storage Access Framework. Google now considers those per-app external storage directories to be private.
- You still do not have read access to certain other locations, such as

`Documents/` and `Downloads/`. Basically, if it is controlled by `MediaStore`, and you lack read access through `MediaStore`, you also lack read access through raw paths.
- Removable storage does not appear to be supported by this "raw paths" feature.
- The documentation mentions reduced performance. This does not appear to be severe, but it may pose issues for performance-sensitive apps.

Note that while the documentation emphasizes native libraries, read access works fine from Java/Kotlin.

Also, methods like `getExternalStorageDirectory()` and `getExternalStoragePublicDirectory()` on `Environment` are still deprecated. Instead, we are supposed to use `getDirectory()` on `StorageVolume`, which is new to Android 11. As the names suggest, this gives us the root directory for a particular storage volume, whether that is external storage or some removable storage device.

The `RawPaths` sample module in the book's sample project is designed to demonstrate the behavior of `READ_EXTERNAL_STORAGE` across a variety of build scenarios. There are five product flavors, with varying configurations:

Flavor	targetSdkVersion	requestLegacyExternalStorage
alfa	28	true
bravo	29	true
charlie	29	false
delta	30	true
echo	30	false

The UI is a crude file explorer. It shows you a list of files and directories for a particular location, starting with some root:

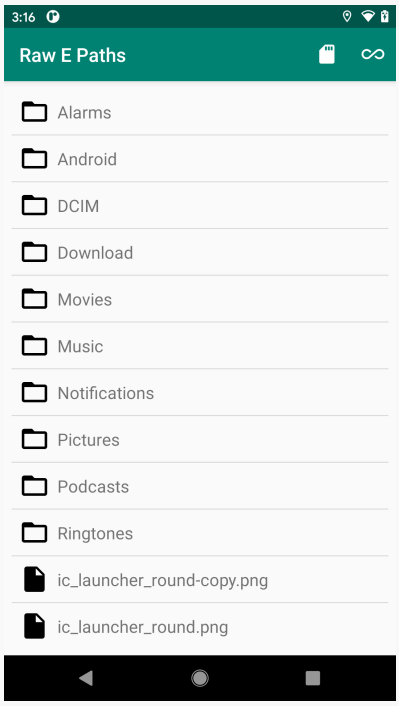


Figure 1: RawPaths, Echo Build, Running on Android 11 DP2

The “SD card” toolbar button will display a checkable submenu with the available storage volumes:

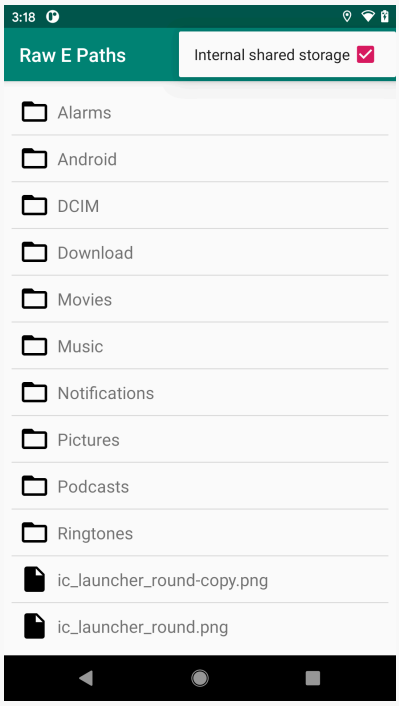


Figure 2: RawPaths, Echo Build, Showing Storage Volume Submenu

If you switch to a different storage volume, that volume’s root directory will be loaded into the list.

Tapping on a file, by default, will bring up a `Toast` showing the CRC32 checksum of the file, used to prove that we have read access to the file’s contents. Tapping on a directory will load that directory’s contents into the list. However, this is a fairly simplistic file explorer, so there is no way to traverse up the directory tree to get back to a root.

Before loading any of this content, though, `MainActivity` will request the `READ_EXTERNAL_STORAGE` permission.

Our viewmodel, `MainMotor`, gets the roster of `StorageVolume` objects from the `StorageManager` system service:

```
val volumes: List<StorageVolume> =
  context.getSystemService(StorageManager::class.java)!!.storageVolumes
```

(from RawPaths/src/main/java/com/commonsware/android/r/rawpaths/MainMotor.kt)

`MainActivity` uses that list to build up the submenu contents. If the user taps on one, `MainActivity` calls a `loadRoot()` function on `MainMotor`. If the app is running on Android 11, that in turn gets the selected `StorageVolume` out of that list and retrieves its directory. On older devices, we just use the deprecated `Environment.getExternalStorageDirectory()` option instead:

```
fun loadRoot(volumeIndex: Int = 0) {
  if (Build.VERSION.SDK_INT < 30) {
    load(Environment.getExternalStorageDirectory())
  } else {
    load(volumes[volumeIndex].directory!!)
  }
}
```

(from RawPaths/src/main/java/com/commonsware/android/r/rawpaths/MainMotor.kt)

That directory is then used by the `load()` function to get the directory's contents and calculate the CRC32 checksums for all files in the directory:

```
fun load(dir: File) {
  _states.postValue(MainViewState.Loading)

  viewModelScope.launch(Dispatchers.IO) {
    try {
      val items = dir.listFiles().orEmpty()
        .sortedBy { it.name }
        .map { file ->
          if (file.isDirectory) {
            FileItem(file, isDirectory = true)
          } else {
            FileItem(file,
              crc32 = CRC32().let { crc ->
                crc.update(file.readBytes())
                crc.value
              })
          }
        }

      _states.postValue(MainViewState.Content(items))
    } catch (t: Throwable) {
      Log.e("RawPaths", "Exception loading $dir", t)
      _states.postValue(MainViewState.Error)
    }
  }
}
```

(from RawPaths/src/main/java/com/commonsware/android/r/rawpaths/MainMotor.kt)

What you will find is:

- On Android 11, both `delta` and `echo` behave the same. `delta` requests the scoped storage opt-out (`android:requestLegacyExternalStorage="true"`), while `echo` does not. Yet, with `READ_EXTERNAL_STORAGE`, we can still access the contents of external storage, though apparently not removable storage and the other restricted locations noted above.
- On Android 10, `android:requestLegacyExternalStorage` has a clear effect. If you use it (`bravo`), `READ_EXTERNAL_STORAGE` works, even though we have `targetSdkVersion` set to 29. If you do not use it (`charlie`), even if the user grants `READ_EXTERNAL_STORAGE`, you do not have filesystem access to external storage.
- On Android 9, before all of these changes took effect, `READ_EXTERNAL_STORAGE` works normally.

Hey, What About Writing?

You have write access to certain raw paths... even without `WRITE_EXTERNAL_STORAGE`.

However, the exact list of these locations is undocumented. As such, these locations are unreliable — device manufacturers might elect to change the behavior here.

But, based on experiments and some Google comments you should be able to write to:

- `Download/`
- `DCIM/`
- `Movies/`
- `Music/`

...and perhaps others.

SAF Restrictions

Since the beginning, the Storage Access Framework has been billed as the way for the app to get access to whatever content the user wants to work with.

In Android 11, that is no longer the case, as the OS will prevent the user from

accessing the user's content in scenarios that Google does not like.

ACTION_OPEN_DOCUMENT_TREE

In the Android 11 version of this UI, the user can navigate into a directory and click a "Use this folder" button at the bottom to choose it:

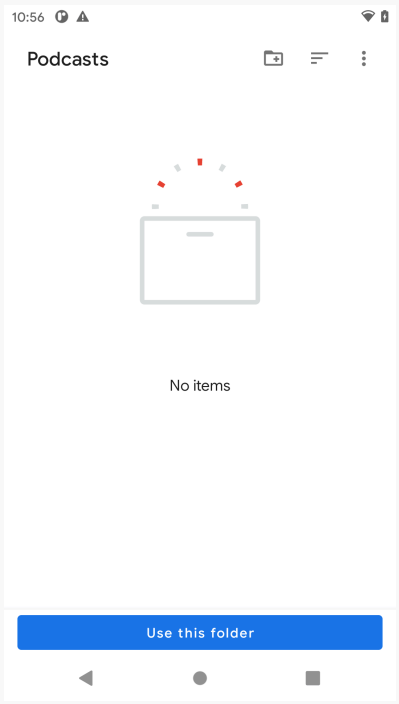


Figure 3: SAF UI, Showing "Use this folder" Button

However, that button will be grayed out in some situations.

Overall External Storage Root

The user cannot grant an app rights to work with the root of external storage, precluding apps from placing new directories there:

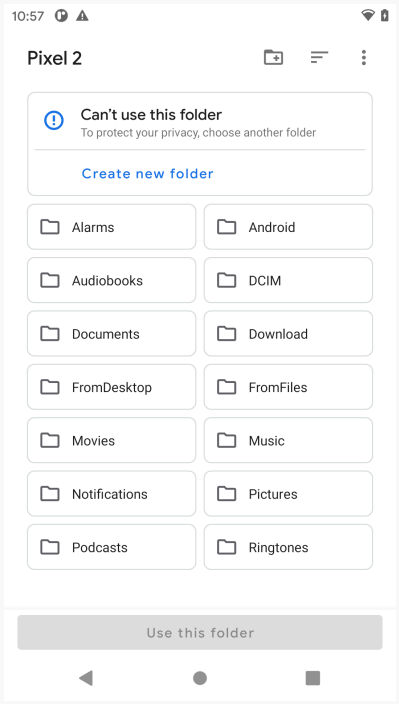


Figure 4: SAF UI, Showing Disabled "Use this folder" Button for External Storage Root

`Download/`

The same holds true for `Download/`:

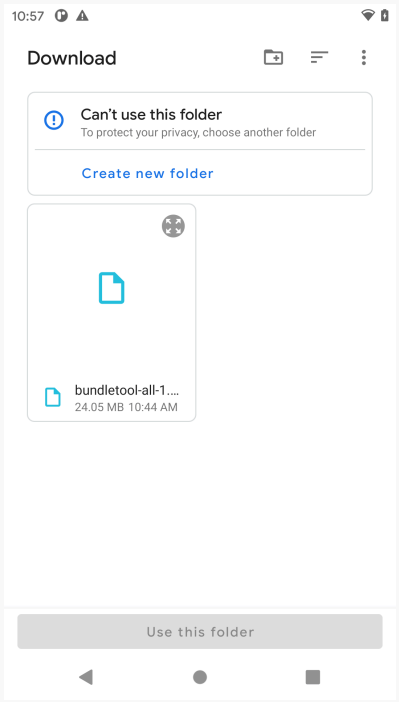


Figure 5: SAF UI, Showing Disabled "Use this folder" Button for `Download/`

However, if the user creates a subdirectory of `Download/`, such as via their USB cable or the stock Files app, they can choose that subdirectory.

Stuff in `Android/`

While the user can choose the `Android/` directory on external or removable storage, the user cannot access anything inside of it. In the other cases, the user could see directories that could not be selected, but in this case, the contents simply do not show up:

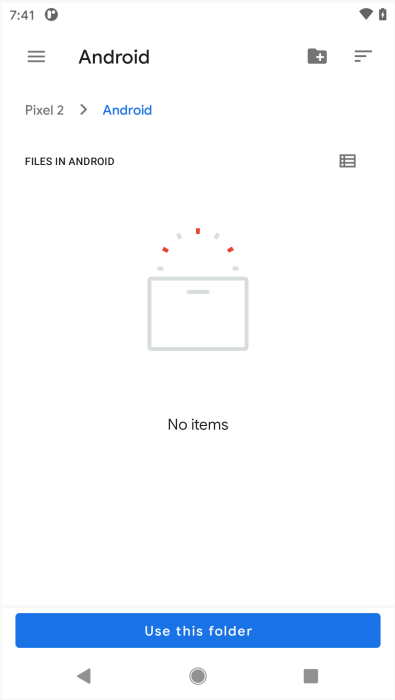


Figure 6: SAF UI, Showing "Empty" `Android/` Directory

The Other SAF Actions

`ACTION_OPEN_DOCUMENT` and `ACTION_CREATE_DOCUMENT` share the last restriction of `ACTION_OPEN_DOCUMENT_TREE`: the user cannot choose or create a file within `getExternalFilesDir()` and kin of some other app. Otherwise, these actions seem unaffected.

Unfortunately, the `ACTION_OPEN_DOCUMENT` limitation means that an app's files are inaccessible by the user except through that app or by copying the files elsewhere using a device-supplied file manager.

“All Files Access”

Android 11 offers an “All Files Access” capability. The idea is that if your app requests the `MANAGE_EXTERNAL_STORAGE` permission, and the user grants it, that you would have unfettered access to most of external and removable storage. There are a few caveats:

1. `MANAGE_EXTERNAL_STORAGE` is not a `dangerous` permission. Instead, it is one of those specialized permissions, like `SYSTEM_ALERT_WINDOW`, where the user needs to go into “Special app access” area of the Settings app to grant the permission.
2. You still do not have access to `Android/data/` on external storage or any removable volume. This limits the utility of `MANAGE_EXTERNAL_STORAGE`, particularly for backup/restore apps.
3. Google hints that this permission will be restricted on the Play Store. The result probably will be akin to select other permissions, where you have to fill out a form and get explicit approval to use this permission. Otherwise, your app might be banned from the Play Store. It might get banned anyway, due to a bot, as Google’s policy violation detection bots seem to be unreliable, to the detriment of too many app developers.
4. Google has indicated that apps cannot request this permission until 2021, presumably because they do not have the policy violation detection bots fully set up yet.

In the `RawPaths` sample module, the activity has a toolbar button with the infinity symbol:

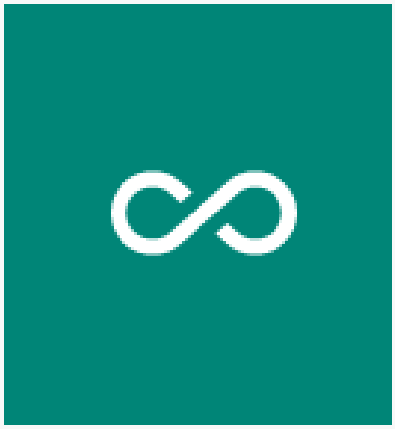


Figure 7: Raw Paths Infinity Toolbar Button

On Android 11 devices, tapping that will request the `MANAGE_EXTERNAL_STORAGE` permission, by means of starting an `ACTION_MANAGE_APP_ALL_FILES_ACCESS_PERMISSION` activity. This takes a `Uri` pointing to your application, using the `package:` scheme:

```
    if (item.itemId == R.id.allAccess) {
      if (Build.VERSION.SDK_INT >= 30) {
        if (hasAllFilesPermission()) {
          Toast.makeText(this, R.string.already_permission, Toast.LENGTH_LONG)
            .show()
        }

        val uri = Uri.parse("package:${BuildConfig.APPLICATION_ID}")

        startActivity(
          Intent(
            Settings.ACTION_MANAGE_APP_ALL_FILES_ACCESS_PERMISSION,
            uri
          )
        )
      } else {
        Toast.makeText(this, R.string.sorry, Toast.LENGTH_LONG).show()
      }
```

(from RawPaths/src/main/java/com/commonsware/android/r/rawpaths/MainActivity.kt)

This will bring up a Settings screen for the user to grant you the permission:

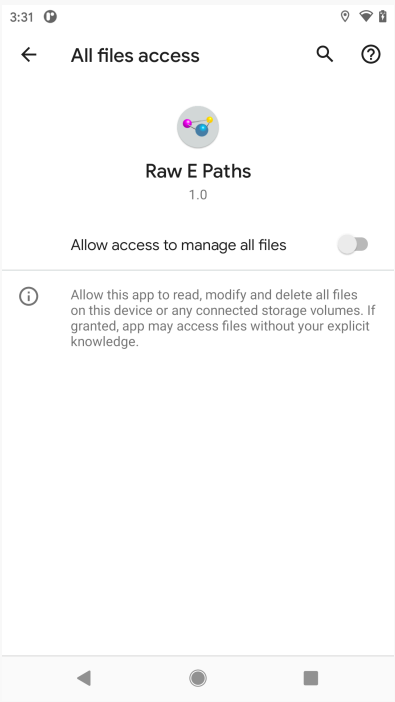


Figure 8: All Files Access Settings Screen

If the user grants you that permission, you now have write access to external storage, akin to what you would have with `WRITE_EXTERNAL_STORAGE` on Android 9 and below. In the `RawPaths` app, if you have write permission, tapping on a file will make a copy of that file in the same directory, reloading the list to show you the copy. Without `MANAGE_EXTERNAL_STORAGE`, you cannot write into many directories (though, as noted earlier, you do have write access to some locations). With `MANAGE_EXTERNAL_STORAGE`, you can work with a lot more.

So, on the plus side, `MANAGE_EXTERNAL_STORAGE` means it is conceivable that you could have write access, at least to external storage, on Android 11. However, that does not help for Android 10. So, you will still need some other solution for your Android 10 users, as there will be more of them than Android 11 users for a few years.

Detecting This Permission

In Android 11, `Environment` now has a pair of `isExternalStorageManager()` methods that will tell you if you hold `MANAGE_EXTERNAL_STORAGE`.

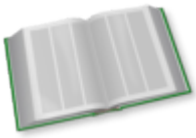
In theory, the zero-parameter `isExternalStorageManager()` would tell you if you can manage external storage, and the one-parameter `isExternalStorageManager()` would tell you if you can manage the storage volume containing the supplied `File`. In practice, the user seems to only be able to grant `MANAGE_EXTERNAL_STORAGE` device-wide, so it is unclear if there is a practical difference between the two.

`MediaStore` Modifications

In addition to the Storage Access Framework, Google has been pushing developers towards `MediaStore` more. In Android 10, that became fairly important, due to the restrictions placed on external and removable storage. Android 11 changes things again, mostly improving Android's behavior in some key areas related to `MediaStore`.

Recapping What We Got in Android 10

As we did with storage, let's first review what changed in Android 10, as some developers are still coming to grips with those changes.



You can learn more about `MediaStore` in Android 10 in the "Using MediaStore" chapter of *Elements of Android Q*!

Limited Access

The big thing is that, by default, you have limited access to the contents of `MediaStore`. Specifically, by default, you can only see the content that *your app* has added to the `MediaStore`.

To be able to query and retrieve content created by other apps, you need to hold `READ_EXTERNAL_STORAGE`. Even then, you lost access to some things:

- Location metadata for images is redacted, with workarounds to get that metadata if the user permits.

- The oft-reviled `DATA` column is deprecated. It was never reliable, and it is even less reliable now.

`RecoverableSecurityException`

If you want to modify content from other apps, that can be done, albeit with a somewhat cumbersome process... one that becomes *seriously* cumbersome if you are trying to modify lots of content at once.

Your calls on `ContentResolver` that would require write access — such as `delete()` or `openOutputStream()` — may now throw a `RecoverableSecurityException`. This indicates that you do not have write access to that content, but you could get it via the exception. Specifically, that exception has a `getUserAction()` method, one that returns a `RemoteAction`. That has a `getActionIntent()` method that returns a `PendingIntent`. You can use that `PendingIntent` to display a system dialog that asks the user if it is OK for you to have write access to this piece of content. If the user agrees, you can re-try your `ContentResolver` call, and it should succeed.

However, this only works on an individual basis. If you try to `delete()` a piece of content using its `Uri`, that may give you the `RecoverableSecurityException` that you need. If, instead, you use `delete()` with a collection `Uri` and a `WHERE` clause, that will simply fail. In Android 10, there is no bulk option, where you can request write access to a list of `Uri` values or an entire collection.

Also, not everything is writable, even with this per-content permission. For example, you can update the `TAGS` column for an image, but not the `DESCRIPTION`.

See this blog post from the author of this book for more about `RecoverableSecurityException`.

New Collections

Two new `MediaStore` collections appeared... though only one was documented.

The documented one was `MediaStore.Downloads`, to access the content of the `Download/` directory. The undocumented one was one for the `Documents/` directory, which requires a convoluted means to access.

Both of these are restricted more than the other collections. In particular, you can only see your own app's content, even *with* `READ_EXTERNAL_STORAGE`.

Getting the Right `Uri`

A `MediaStore` `Uri`, pointing to an individual piece of content, is the combination of a collection `Uri` and the ID of the content. The classic way to accomplish that was via `ContentUris.withAppendedId()`, which assembles the `MediaStore` `Uri` given those two pieces.

We now have an alternative: a two-parameter `getContentUri()` method on the various `MediaStore` collection classes (e.g., `MediaStore.Video.Media`). Whereas `ContentUris.withAppendedId()` takes a collection `Uri` and an ID, `getContentUri()` takes a "volume name" and an ID. `MediaStore.VOLUME_EXTERNAL` works for classic external storage as the volume name, giving us:

```
val contentUri = MediaStore.Video.Media.getContentUri(MediaStore.VOLUME_EXTERNAL, id)
```

where `id` is the ID of some piece of content (in this case, a video).

Batched Access

Earlier, we saw how in Android 10 we could catch a `RecoverableSecurityException` and use that to get permission to modify a single piece of content. For many apps, this is sufficient, because they only need to modify one piece of content at a time. However, for apps that manipulate multiple pieces of content, this per-piece-permission approach is awful. The user winds up having to accept a dialog for each piece, and that gets tedious quickly.

Android 11 offers a batched way to get permission from the user, though the API for it is rather odd.

The `VideoTagger` sample module in the book's sample project requests `READ_EXTERNAL_STORAGE` permission on startup and uses that to query the `MediaStore` for all the videos on external storage. Those are then presented in a checklist:

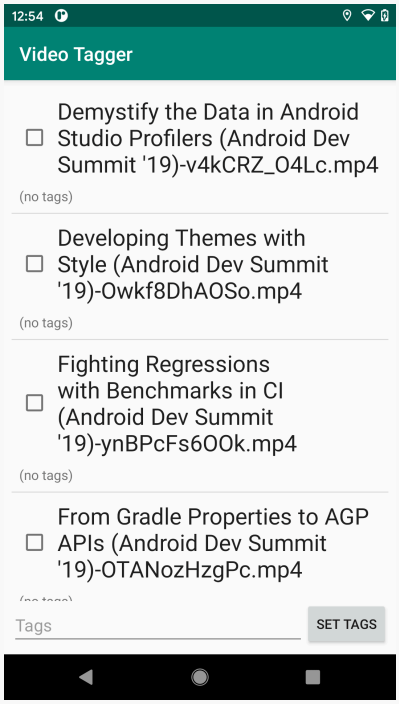


Figure 9: VideoTagger, As Initially Launched

The user can check one or more of the videos, then fill in "tags" in the field and click "Set Tags". This will attempt to update the `TAGS` property of the content... and that will fail initially. The app then requests write permission for all of those videos at once:

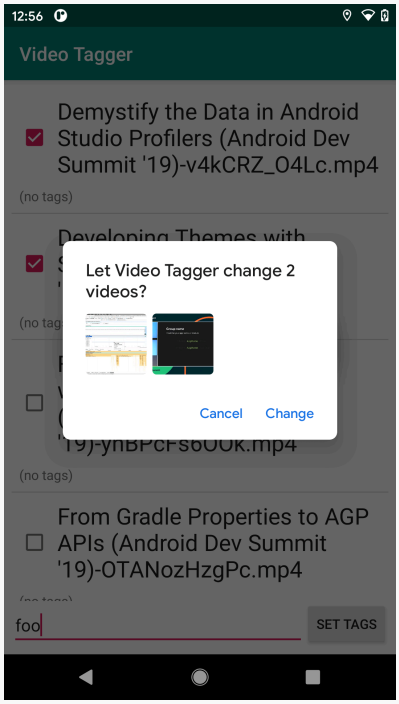


Figure 10: VideoTagger, with Permission Request Dialog

If the user grants the permission, the app updates the `TAGS` and reloads the list, showing those tags below the video title. If the user leaves the field blank and clicks "Set Tags", the same thing happens, except that any existing `TAGS` value is simply cleared.

Obtaining the Image `Uri` Values

There are many ways we could have gotten `Uri` values for the user's chosen set of videos. For example, we could have used `ACTION_OPEN_DOCUMENT`, `ACTION_GET_CONTENT`, or `ACTION_PICK`. In particular, `ACTION_OPEN_DOCUMENT` and `ACTION_GET_CONTENT` should support `EXTRA_ALLOW_MULTIPLE`, so we can request a UI that allows the user to pick several items at once.

However, the `Uri` values that we get back from those do *not* work with this batched permission request feature. If we try, we get:

```
java.lang.IllegalArgumentException: Missing volume name:
content://com.android.providers.media.documents/document/video%3A23
```

(for whatever `Uri` you happened to try)

For `ACTION_OPEN_DOCUMENT` and `ACTION_GET_CONTENT`, we get "document `Uri`" values. You might think that the solution is to convert those to "media `Uri`" values, using `getMediaUri()` on `MediaStore`. While we can call that method and convert the `Uri` values, the converted values *also* do not work, as we get:

```
java.lang.IllegalStateException: java.io.FileNotFoundException: No root
for video
```

(at least for a video `Uri` — the error probably varies based on media type)

The way that works is the one described earlier in this chapter: use `getContentUri()`.

That is what the `VideoTagger` app uses. We have a `VideoModel` that represents the data that we need for a video:

```
package com.commonsware.android.r.videotagger

import android.net.Uri

data class VideoModel(
  val uri: Uri,
  val title: String,
  val tags: String?,
  val description: String?,
  var isChecked: Boolean = false
)
```

(from VideoTagger/src/main/java/com/commonsware/android/r/videotagger/VideoModel.kt)

When our `VideoRepository` queries the `MediaStore` and converts the `Cursor` to a `List` of `VideoModel`, it uses `getContentUri()` to fill in the `uri` property:

```
  private val resolver = context.contentResolver

  suspend fun loadVideos(): List<VideoModel> =
    withContext(Dispatchers.IO) {
      val collection =
        MediaStore.Video.Media.getContentUri(MediaStore.VOLUME_EXTERNAL)
```

```
    resolver.query(collection, PROJECTION, null, null, SORT_ORDER)
      ?.use { cursor ->
        cursor.mapToList {
          VideoModel(
            uri = MediaStore.Video.Media.getContentUri(
              MediaStore.VOLUME_EXTERNAL,
              it.getLong(0)
            ),
            title = it.getString(1),
            tags = it.getString(2),
            description = it.getString(3)
          )
        }
      } ?: emptyList()
  }
```

(from VideoTagger/src/main/java/com/commonsware/android/r/videotagger/VideoRepository.kt)

Seeing If We Have Permission

Once the user has selected some videos, if the user clicks the "Set Tags" button, we need to:

- See if we have write permission for all of those videos
- Request write permission for those that we lack
- Update the tags once we get write permissions for the videos

All of this is a bit clunky.

To see if we have permission to modify the content identified by a `Uri`, we need to call `checkUriPermission()` on a `Context`. This takes four parameters:

- The `Uri` to check
- Your process ID (`Process.myPid()`)
- Your app's user ID (`Process.myUid()`)
- The permission to check (e.g., `Intent.FLAG_GRANT_WRITE_URI_PERMISSION`)

This will return `PackageManager.PERMISSION_GRANTED` if your app holds that particular permission.

`MainActivity` in `VideoTagger` has a `neededPermissions()` function that takes the list of videos and returns the subset for which we still need to obtain permission from the user:

```
private fun neededPermissions(selections: List<Uri>) =
  selections.filter {
    checkUriPermission(
      it,
      Process.myPid(),
      Process.myUid(),
      Intent.FLAG_GRANT_WRITE_URI_PERMISSION
    ) != PackageManager.PERMISSION_GRANTED
  }
```

(from VideoTagger/src/main/java/com/commonsware/android/r/videotagger/MainActivity.kt)

Requesting the Permission

To get write permission for those `Uri` values where we lack it, we need to do two things.

First, we need to call `MediaStore.createWriteRequest()`, supplying the list of `Uri` values and a `ContentResolver`. This returns a `PendingIntent` that represents this request for write access.

Then, we need to call `startIntentSenderForResult()`. If the `PendingIntent` is one for an activity (`PendingIntent.getActivity()`), then `startIntentSenderForResult()` will start that activity and send any result back to our `onActivityResult()` function. This works just as it would if we called `startActivityForResult()` on a regular `Intent`.

When the user clicks "Set Tags", that triggers a call to an `applyTags()` function. This uses `neededPermissions()` to find the `Uri` values for which we lack write access. If we have write access to all of them, we go ahead and ask `MainMotor` to update the tags. Otherwise, we call `MediaStore.createWriteRequest()` and `startIntentSenderForResult()` to request permission from the user:

```
private fun applyTags(models: List<VideoModel>) {
  val needed = neededPermissions(models.map { it.uri })

  if (needed.isEmpty()) {
    motor.applyTags(models, binding.tags.text.toString())
  } else {
    val pi = MediaStore.createWriteRequest(contentResolver, needed)

    startIntentSenderForResult(pi.intentSender, REQUEST_PERMS, null, 0, 0, 0)
  }
}
```

(from VideoTagger/src/main/java/com/commonsware/android/r/videotagger/MainActivity.kt)

This is what triggers the dialog shown earlier. In our `onActivityResult()` function, if our `REQUEST_PERMS` request succeeded, we can go ahead and try `applyTags()` again.

Permission Permutations

There have been some tweaks to how runtime permissions work in Android 11. In theory, most of these should not cause any harm to your app. Unfortunately, not everything works the way that we think it should, and so it is possible that you will need to make some tweaks to your app to accommodate these changes.

And, if you are using location permissions, and adjusted your app to deal with background locations for Android 10... you have more work to do here in Android 11.

One-Time Permissions

The biggest user-visible change is what Google calls "one-time permissions". For a certain set of permission groups, the user will be given an "Only this time" option in the runtime permission dialog:

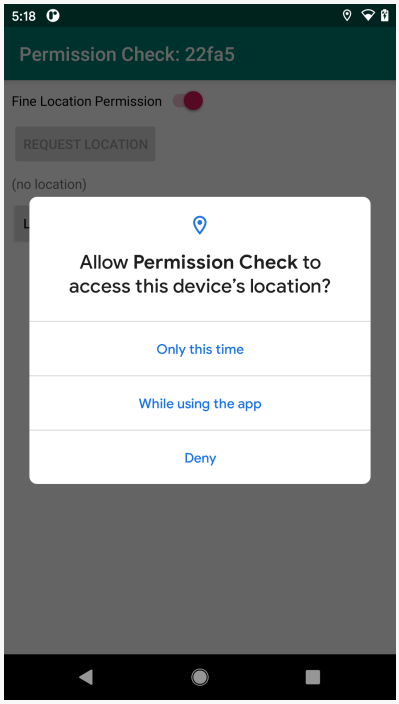


Figure 11: Runtime Permission Dialog with "Only This Time" Option

The documentation is a bit unclear over exactly which permissions get this treatment. Based on what is written, it is likely that the permissions in the `CAMERA`, `LOCATION`, and `MICROPHONE` permission groups will be affected.

The documentation is also a bit vague on how long the permission grant remains in effect — in others words, what is the scope of "Only this time"? Based on experiments, it appears that the answer is "for the current process", where a fresh process will need to bring up the permission dialog again... if that process is launched a bit after the old process ended.

Trying It Out

You can test this yourself using the `PermissionCheck` sample module in the book's

sample project.

When you run this module, you get an activity with a `Switch` to request the `ACCESS_FINE_LOCATION` runtime permission:

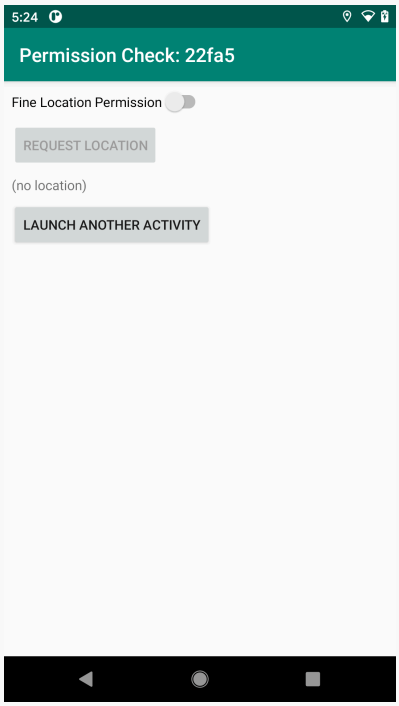


Figure 12: PermissionCheck App, As Initially Launched

That switch will be off and enabled if you do not hold the permission at the time of displaying the activity, and it will be on and disabled if you do hold the permission. Also, the "Request Location" button beneath it will be enabled if you hold the permission — clicking it will find your location and display it where you see "(no location)" in the screenshot. Finally, the "Launch Another Activity" button launches another instance of this activity, where the code in the toolbar will show you which instance is which.

If you click the switch and grant the permission "Only this time", you will find that:

- You can request the location in this activity
- You can launch another activity and request the location there without getting the permission dialog again
- You can navigate BACK to the first activity and request the location
- You can click BACK from the first activity to exit the app, then launch the

activity again from your launcher, and you can *still* request the location (since it is all the same process)
- If you swipe the app off the overview screen, then launch it again right away, you can *still* request the location
- If you swipe the app off the overview screen, then launch it again only after a delay, you will be presented with the runtime permission dialog again

Ramifications For You

In theory, your app should already handle this. After all, the user could revoke your runtime permission from the Settings app at any point. Your process is then terminated (if it was running), and the next time your app runs, you will need to request permission again. All "Only this time" does is automate that work.

The documentation *implies* that the scope of an "Only this time" permission is a single activity, or perhaps the combination of an activity and a foreground service. Most likely, this is just a documentation bug. However, you may wish to pay closer attention to this change as Android 11 evolves, in case they revise the behavior to match the documentation.

Multiple Rejections = Denial

By now, probably you are used to the triad of permission options in the runtime permission dialog:

- Allow
- Deny
- Deny (and "don't ask again")

If the user chooses "don't ask again", you will not be able to ask for the permission again from within your app. The system will refuse to display the runtime permission dialog and will simply report that the permission was denied (via `onRequestPermissionResult()`).

However, "don't ask again" may not appear in the runtime permission dialog all the time. For example, in the screenshot shown earlier in this chapter, the options are:

- Only this time
- While using the app
- Deny

However, the user still can get the “don’t ask again” effect.

If you display the dialog, and the user clicks “Deny”, then later you display the dialog again and the user clicks “Deny”, then your app is treated as though the user chose “Don’t ask again”.

The two “Deny” actions do not need to be sequential, so long as the user uses the BACK button to close the dialog other times in between. So, for example, the flow could be:

- Deny
- BACK
- BACK
- BACK
- Deny

Each of those five times, the dialog would be shown, but after that second “Deny”, it is treated as “Don’t ask again”.

Trying It Out

You can see this effect in action with the `PermissionCheck` sample. Uninstall the app (if you had it installed already). Re-run the app form the IDE, then tap the switch and click “Deny” on the resulting permission dialog. Do that two times, and you will find that future clicks on the switch have no practical effect, other than incrementally wearing out your smartphone screen.

Ramifications For You

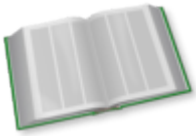
Once again, this should not really cause a problem in terms of app operation. You have had to deal with “don’t ask again” in the past. This simply provides that option with a different UI.

However, if you are trying to maintain decent end-user documentation, including showing the various permission flows... well, your job just got harder.

(sorry!)

Background Location Changes

Android 10 introduced `ACCESS_BACKGROUND_LOCATION` as a new permission. If you want your app to be able to access location data from the background, you need to hold this permission. This is a `dangerous` permission, one that you will need to request at runtime in addition to having it in the manifest.



You can learn more about `ACCESS_BACKGROUND_LOCATION` in Android 10 in the "Location Access Restrictions" chapter of *Elements of Android Q*!

Android 11 does not change any of that.

However, what Android 11 *does* change is when you can ask for it. On Android 10, you could request `ACCESS_BACKGROUND_LOCATION` at the same time as you requested `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`. In Android 11, you have to request it *separately* and *after* requesting `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`.

The `BackgroundLocation` sample module in the book's sample project requests `ACCESS_FINE_LOCATION`, then asks for `ACCESS_BACKGROUND_LOCATION` after `ACCESS_FINE_LOCATION` is granted (and the user clicks a button). This module also has two product flavors, to demonstrate two different types of builds:

Product Flavor	targetSdkVersion
quince	29
rutabaga	R

(remember: Android versions are not "tasty treats" anymore!)

The following sections illustrate what you get when you request `ACCESS_FINE_LOCATION`, followed by `ACCESS_BACKGROUND_LOCATION`.

Android 10

We start off with a dialog offering to grant permission only while using the app:

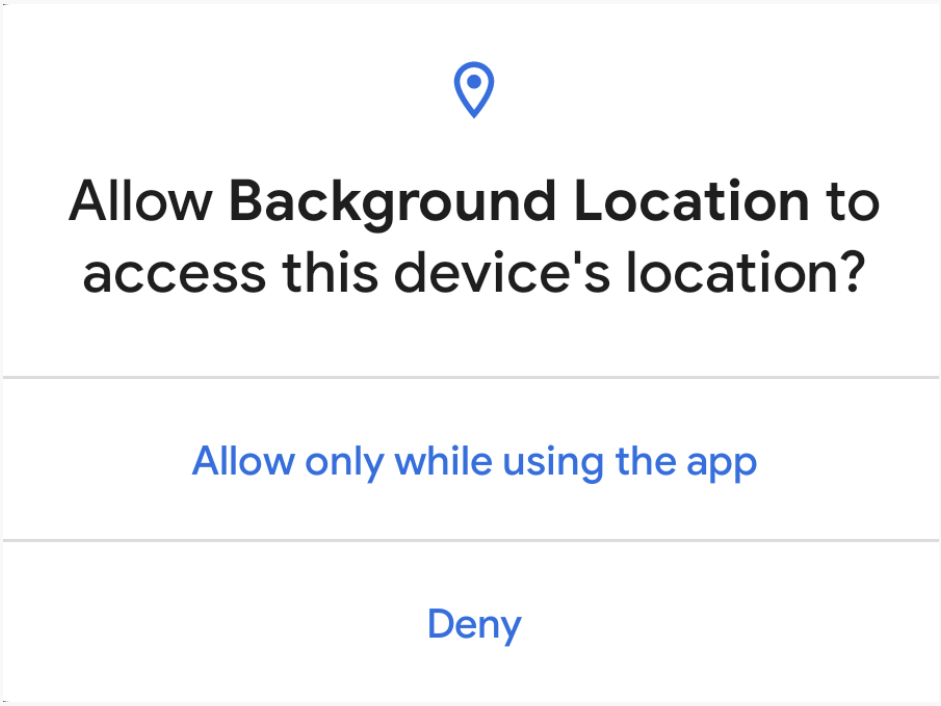


Figure 13: `BackgroundLocation`, As Initially Launched, on Android 10

Later, if you request `ACCESS_BACKGROUND_LOCATION`, the user gets a dialog offering to upgrade your access to "all the time":

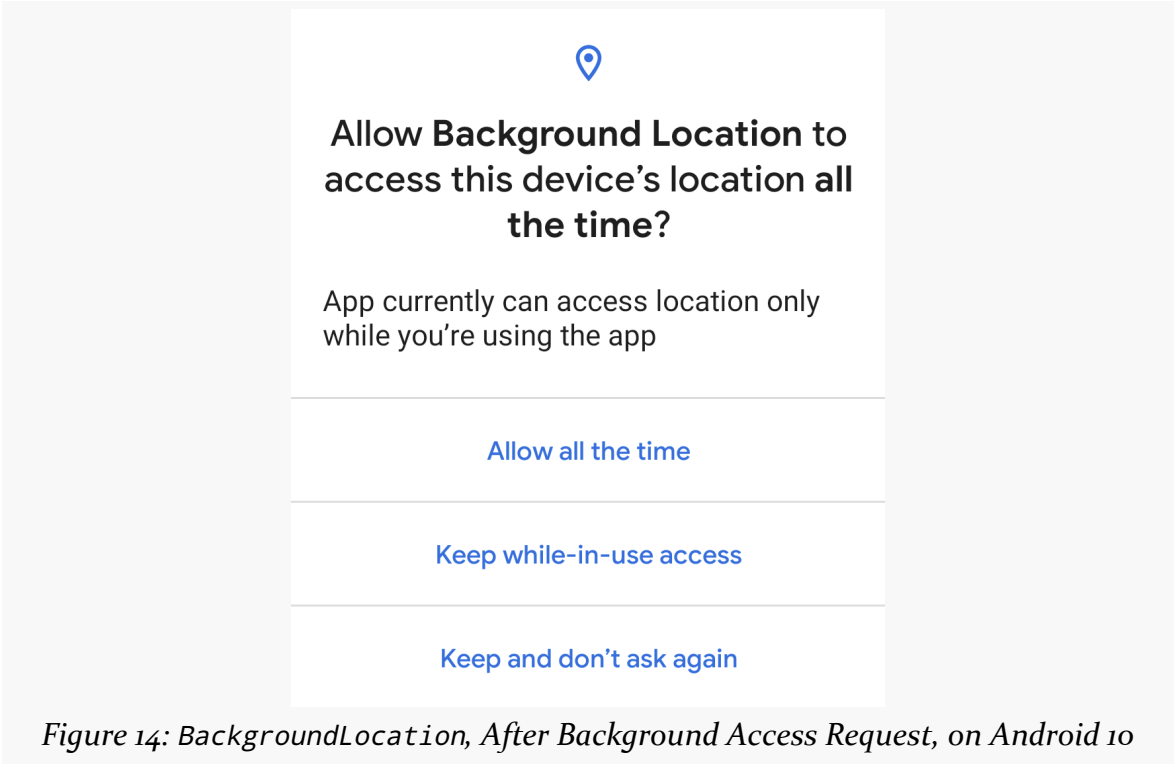


Figure 14: `BackgroundLocation`, After Background Access Request, on Android 10

Android 11, `targetSdkVersion` 29

While your `targetSdkVersion` remains at 29, the first difference is that the first dialog offers the "Only this time" option, discussed earlier in this chapter:

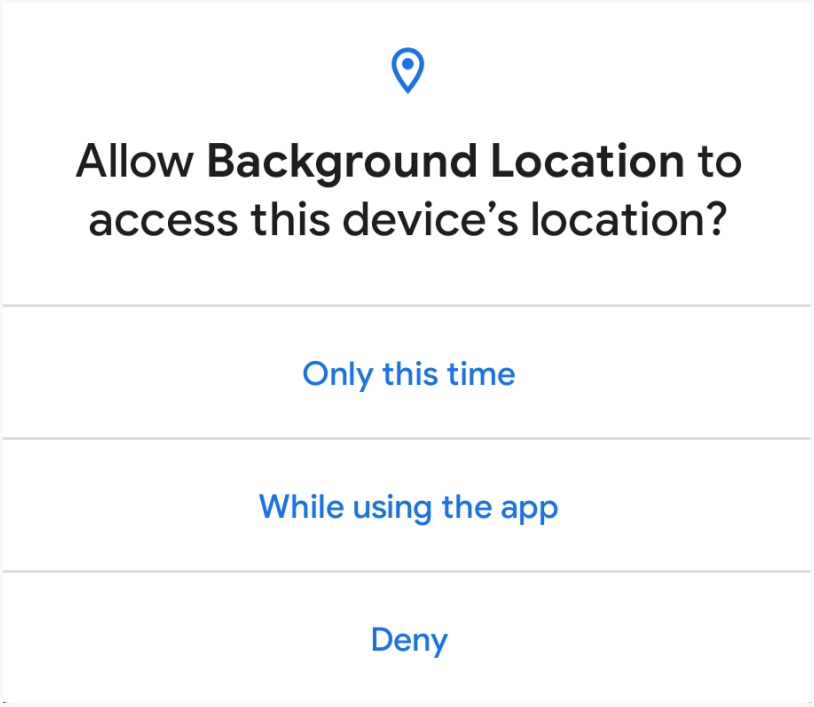


Figure 15: `BackgroundLocation`, As Initially Launched, on Android 11 with `targetSdkVersion` 29

The “upgrade” dialog that you get when you request `ACCESS_BACKGROUND_LOCATION` is also a bit different. The only direct option that the user can choose is to keep their current value:

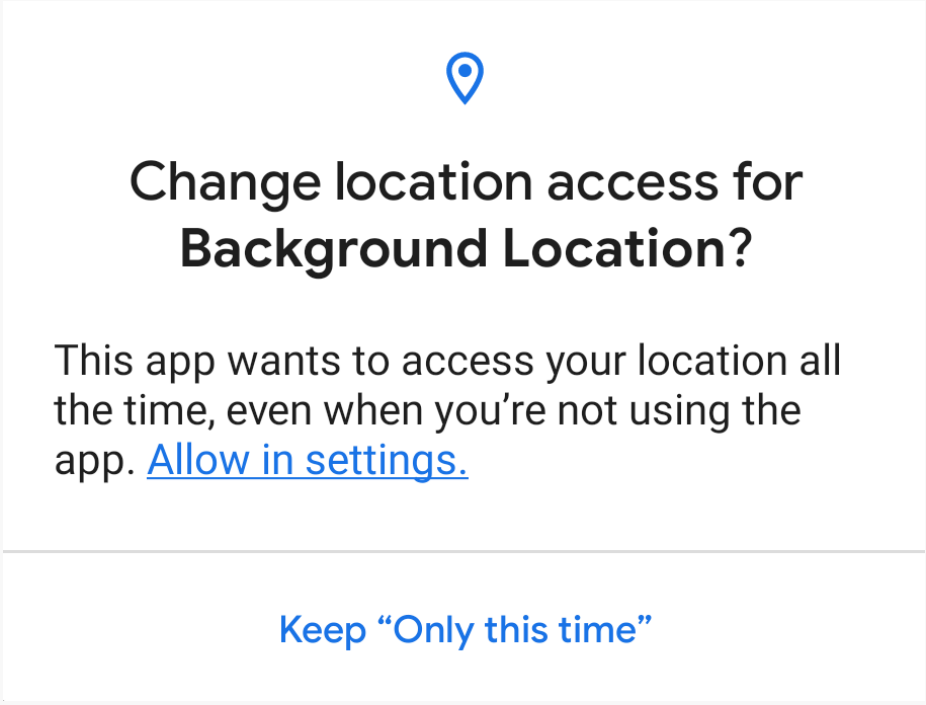


Figure 16: `BackgroundLocation`, After Background Access Request, on Android 11 with `targetSdkVersion` 29

In order to actually grant you `ACCESS_BACKGROUND_LOCATION`, the user has to click that "Allow in settings" link, which will bring up the permission screen for this permission group for your app in Settings:

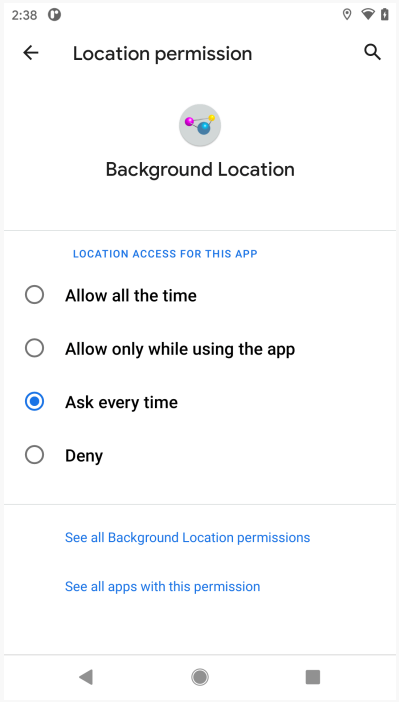


Figure 17: Android 11 Location Permission Settings Screen

Android 11, `targetSdkVersion` 30

Once you upgrade to `targetSdkVersion` of 30, the initial dialog remains unchanged.

Later, when you request `ACCESS_BACKGROUND_LOCATION`, the user is taken straight to the Settings app, bypassing that intermediate dialog and its "Allow in settings" link. The idea is that you would provide your own UI explaining what is about to happen, before you request the `ACCESS_BACKGROUND_LOCATION` permission.

Automatic Permission Removal

Android 11 adds a user option to have Android automatically revoke permissions if your app is unused for an extended period of time (months):

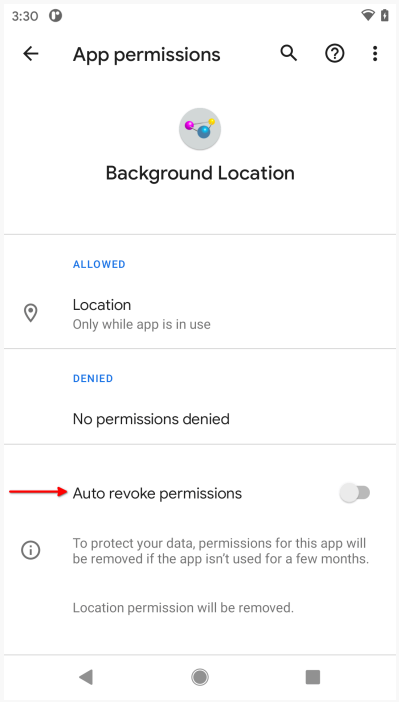


Figure 18: `BackgroundLocation` Permissions Screen in Settings, Showing Auto Revoke Permissions Option

Once again, this should not be a major problem for well-written apps. Partly, that is because this is not significantly different than the user revoking permissions manually, which you have needed to handle. Partly, that is because a well-written app is likely to be used frequently enough to avoid the automatic permission removal.

In theory, you can add `android:autoRevokePermissions="disallowed"` to the `<application>` element to say that the switch would be off by default. The documentation for this setting states "This declaration may cause an additional review when publishing your app". In addition, it does not work.

Another option is `android:autoRevokePermissions="discouraged"`. This is supposed to allow you to use `ACTION_AUTO_REVOKE_PERMISSIONS` to lead the user to a

screen where they can toggle the switch to the off position. However, this does not work, insofar as it only leads the user to the main page for your app in Settings, not to "UI to manage auto-revoke state" as is stated in the documentation.

Auditing Alternatives

"Audit" as a term sometimes has negative connotations ("you have been cordially invited to attend your upcoming tax audit..."). Really, though, an audit is simply a form of testing, confirming that everything is working as you might expect. It's just that testing usually occurs in development, while auditing is something that you apply in production.

Android has had some auditing options in the past, such as using `TrafficStats` or `NetworkStatsManager` to get a sense of how much bandwidth your app is using. Android 11 adds two more auditing options, for determining what sorts of protected services you might be accessing, and why your application's process is terminated.

Data Access Auditing

If your app uses `dangerous` permissions — the ones we need to request at runtime — Android 11 lets you find out when and where your app uses those permissions. That includes both direct uses in your own code and uses by any libraries that you add as dependencies.

If you collect this data and get it back to your organization, you can determine if your app is using these permissions in an expected fashion. Or, conversely, you might find that some third-party library that you are using is siphoning off user data in ways that your users (and your qualified legal counsel) might not appreciate.

Collecting the Data

Android has had an `AppOpsManager` system service for a few releases. In Android 11, it now has a `setNotedAppOpsCollector()` method. This takes an instance of an `AppOpsManager.AppOpsCollector` abstract class, which serves as your callback for

the various events. Since there is one collector per process, this is the sort of thing that you might configure in your custom `Application` class.

We looked at the `PermissionCheck` sample module in the book's sample project in the chapter on permission changes. This module also demonstrates the data access auditing code. Specifically, in `MainApp`, along with setting up Koin for dependency inversion, we also register an `AppOpsManager.OnOpNotedCallback`:

```
package com.commonsware.android.r.permcheck

import android.app.AppOpsManager
import android.app.Application
import android.app.AsyncNotedAppOp
import android.app.SyncNotedAppOp
import android.util.Log
import org.koin.android.ext.koin.androidContext
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.context.startKoin
import org.koin.dsl.module
import java.util.concurrent.Executors

private const val TAG = "PermissionCheck"
private const val FEATURE_ID = "awesome-stuff"

class MainApp : Application() {
  private val module = module {
    viewModel { MainMotor(androidContext().createAttributionContext(FEATURE_ID)) }
  }
  private val executor = Executors.newSingleThreadExecutor()

  override fun onCreate() {
    super.onCreate()

    startKoin {
      androidLogger()
      androidContext(this@MainApp)
      modules(module)
    }

    getSystemService(AppOpsManager::class.java)
      ?.setOnOpNotedCallback(executor, object : AppOpsManager.OnOpNotedCallback() {
        override fun onNoted(op: SyncNotedAppOp) {
          Log.d(TAG, "onNoted: ${op.toDebugString()}")
          RuntimeException().printStackTrace(System.out)
        }

        override fun onSelfNoted(op: SyncNotedAppOp) {
          Log.d(TAG, "onSelfNoted: ${op.toDebugString()}")
          RuntimeException().printStackTrace(System.out)
        }

        override fun onAsyncNoted(op: AsyncNotedAppOp) {
          Log.d(TAG, "onAsyncNoted: ${op.toDebugString()}")
          RuntimeException().printStackTrace(System.out)
        }
```

```
      })
  }

  private fun SyncNotedAppOp.toDebugString() =
    "SyncNotedAppOp[attributionTag = $attributionTag, op = $op"

  private fun AsyncNotedAppOp.toDebugString() =
    "AsyncNotedAppOp[attributionTag = $attributionTag, op = $op, time = $time, uid = $notingUid, message =
$message"
}
```

(from PermissionCheck/src/main/java/com/commonsware/android/r/permcheck/MainApp.kt)

There are three methods that you need to implement on your `OnOpNotedCallback`:

- `onNoted()` will be called when the data access occurred synchronously with a call from your app
- `onAsyncNoted()` will be called when the data access occurred in some callback
- `onSelfNoted()`, which is not very well documented

In all three cases, `MainApp` dumps a "debug string" to Logcat, along with the current stack trace (culled from a `RuntimeException` instance).

If you run the app and request a location, you will get data access information in Logcat (with some portions replaced with `...` for the sake of brevity):

```
D/PermissionCheck: onNoted: SyncNotedAppOp[attributionTag = awesome-
stuff, op = android:fine_location
I/System.out: java.lang.RuntimeException
I/System.out:     at ...MainApp$onCreate$2.onNoted(MainApp.kt:53)
I/System.out:     at
android.app.AppOpsManager.readAndLogNotedAppops(AppOpsManager.java:8154)
...
I/System.out:     at
...MainMotor$fetchLocationAsync$2.invokeSuspend(MainMotor.kt:75)
...
D/PermissionCheck: onAsyncNoted: AsyncNotedAppOp[attributionTag =
awesome-stuff, op = android:fine_location, time = 1588186761021, uid =
1000, message = Location sent to
...MainMotor$fetchLocationAsync$2$invokeSuspend$$...
I/System.out: java.lang.RuntimeException
I/System.out:     at ...MainApp$onCreate$2.onAsyncNoted(MainApp.kt:63)
...
```

We have one `onNoted()` call. As part of the `SyncNotedAppOp` that we receive, we know that the `op` was `android:fine_location`, meaning that we did something that required `ACCESS_FINE_LOCATION` permission. The stack trace shows that this came from the `getCurrentLocation()` call on a `LocationManager` inside of `MainMotor`, our viewmodel:

```
package com.commonsware.android.r.permcheck

import android.Manifest
import android.content.Context
import android.content.pm.PackageManager
import android.location.Location
import android.location.LocationManager
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.*
import java.util.concurrent.Executors
import java.util.function.Consumer
import kotlin.coroutines.resume
import kotlin.coroutines.suspendCoroutine

sealed class MainViewState {
  data class Content(
    val hasPermission: Boolean,
    val location: Location? = null
  ) :
    MainViewState()

  object Error : MainViewState()
}

class MainMotor(private val context: Context) : ViewModel() {
  private val _states = MutableLiveData<MainViewState>()
  val states: LiveData<MainViewState> = _states

  fun checkPermission() {
    _states.value = MainViewState.Content(hasLocationPermission())
  }

  fun fetchLocation() {
    viewModelScope.launch(Dispatchers.Main) {
      _states.value =
        MainViewState.Content(hasLocationPermission(), fetchLocationAsync())
    }
  }
```

```
  private fun hasLocationPermission() =
    context.checkSelfPermission(Manifest.permission.ACCESS_FINE_LOCATION) ==
        PackageManager.PERMISSION_GRANTED

  private suspend fun fetchLocationAsync(): Location {
    val locationManager =
      context.getSystemService(LocationManager::class.java)!!
    val executor = Executors.newSingleThreadExecutor()

    return withContext(executor.asCoroutineDispatcher()) {
      suspendCoroutine<Location> { continuation ->
        val consumer =
          Consumer<Location?> { location ->
            if (isActive && location != null) continuation.resume(location)
          }

        locationManager.getCurrentLocation(
          LocationManager.GPS_PROVIDER,
          null,
          executor,
          consumer
        )
      }
    }
  }
}
```

(from PermissionCheck/src/main/java/com/commonsware/android/r/permcheck/MainMotor.kt)

We also have one `onAsyncNoted()` call. Here, the `op` is also `android:fine_location`. The `message` property shows that the operation came from something inside `MainMotor`, but it is buried in the `suspendCoroutine()` lambda expression. *Probably*, this is being triggered when our `Consumer` receives a location, but that is just a guess, given that we have no line number to use.

Identifying Uses by Attribution

Both logs show an `attributionTag` of `awesome-stuff`. By default, you will not have an `attributionTag` value. And for many apps, that's fine. However, if you want to be able to tag your data access, you can do so by setting the value to appear in that `attributionTag` to some string.

To do this, you need to use `createAttributionContext()`, a method available on `Context`. This gives you another `Context`, one with your specified `attributionTag`

value. You then use that augmented `Context` when doing things involving the data access, such as requesting the `LocationManager` system service.

In this app, we do that as part of our Koin setup in `MainApp`:

```
private val module = module {
  viewModel { MainMotor(androidContext().createAttributionContext(FEATURE_ID)) }
}
```

(from PermissionCheck/src/main/java/com/commonsware/android/r/permcheck/MainApp.kt)

When we inject a `Context` into `MainMotor`, we get the Koin `androidContext()`, then call `createAttributionContext()` on that `Context`. The `MainMotor` gets the `Context` from `createAttributionContext()` and uses that to get the `LocationManager`. And, since we set the `attributionTag` to `awesome-stuff` in the `createAttributionContext()` call, that is why we get `awesome-stuff` as the `attributionTag` in our output.

What To Do With the Results?

You might consider collecting the data and sending it to your backend, to get a sense of what is using protected data and how frequently. You could send all of the data, or use heuristics to determine expected-vs.-unexpected scenarios and report them differently.

One imagines that future versions of crash logging or analytics libraries will "bake in" the ability to gather this data as part of their normal operation.

Application Exits

Your process does not live forever. In fact, your process gets terminated a lot, particularly depending on the sort of work that you do. For example, if you are using `WorkManager` to get control periodically in the background to do some work, your process will be terminated sometime after each piece of work.

This is nothing new.

What *is* new is the ability to find out *why* your process got terminated. You are not told this in real-time as your process is being terminated, but you can find out past reasons for process termination.

Collecting the Data

The `ActivityManager` system service now has a `getHistoricalProcessExitReasons()` method. This will return a list of `ApplicationExitInfo` objects, representing past process termination reasons.

The `ForensicPathologist` sample module in the book's sample project gets those `ApplicationExitInfo` objects in its `MainMotor`:

```
class MainMotor(private val context: Context) : ViewModel() {
  private val _content = MutableLiveData<List<ExitInfo>>()
  val content: LiveData<List<ExitInfo>> = _content

  init {
    _content.value =
      context.getSystemService(ActivityManager::class.java)
        ?.getHistoricalProcessExitReasons(null, 0, 0).orEmpty()
        .map { convert(it) }
  }
```

(from ForensicPathologist/src/main/java/com/commonsware/android/r/forensics/MainMotor.kt)

We get the `ActivityManager`, call `getHistoricalProcessExitReasons()`, coerce null results into an empty list, and use that to populate a `MutableLiveData`.

`getHistoricalProcessExitReasons()` takes three parameters:

- The package name of app whose processes you wish to collect, or `null` for your own process
- A particular process ID to examine, or `0` to not filter by process ID
- The maximum number of items to return, or `0` to return all that are available

Note that if you specify a package name from some other app, you will need to hold the `DUMP` permission. This is not available to ordinary third-party apps. And, usually, we will not know any particular process ID to filter upon. So, typically, the call to `getHistoricalProcessExitReasons()` will be `getHistoricalProcessExitReasons(null, 0, 0)`, to collect all known process exit reasons.

The `ApplicationExitInfo` contains several fields that you can use. The big one is the `reason` field, which says why the process was terminated. This is an `Int` that maps to various `REASON_` constants on `ApplicationExitInfo`. As part of converting an `ApplicationExitInfo` into data to fill into our UI, `MainMotor` converts a `reason`

value into a string resource ID:

```
@StringRes
private fun convertReason(reason: Int): Int = when (reason) {
  ApplicationExitInfo.REASON_ANR -> R.string.reason_anr
  ApplicationExitInfo.REASON_CRASH -> R.string.reason_crash
  ApplicationExitInfo.REASON_CRASH_NATIVE -> R.string.reason_crash_native
  ApplicationExitInfo.REASON_DEPENDENCY_DIED -> R.string.reason_dependency_died
  ApplicationExitInfo.REASON_EXCESSIVE_RESOURCE_USAGE -> R.string.reason_excessive_resource_usage
  ApplicationExitInfo.REASON_EXIT_SELF -> R.string.reason_exit_self
  ApplicationExitInfo.REASON_INITIALIZATION_FAILURE -> R.string.reason_init_failure
  ApplicationExitInfo.REASON_LOW_MEMORY -> R.string.reason_low_memory
  ApplicationExitInfo.REASON_OTHER -> R.string.reason_other
  ApplicationExitInfo.REASON_PERMISSION_CHANGE -> R.string.reason_permission_change
  ApplicationExitInfo.REASON_SIGNALED -> R.string.reason_signaled
  ApplicationExitInfo.REASON_USER_REQUESTED -> R.string.reason_user_requested
  ApplicationExitInfo.REASON_USER_STOPPED -> R.string.reason_user_stopped
  else -> R.string.shrug
}
```

(from ForensicPathologist/src/main/java/com/commonsware/android/r/forensics/MainMotor.kt)

The `description` field may contain additional details about the reason for the process to be terminated, depending on what the `reason` value is. Similarly, `status` may contain an additional number related to the process termination (e.g., if the OS signaled for process termination, `status` will contain the signal number)

We also have:

- `pss` and `rss`, to tell us about the memory consumption of the process at the time that it was terminated
- `timestamp`, indicating when the process was terminated
- `importance`, indicating the importance of the process at the time that it was terminated

`MainMotor` converts all of that stuff into `ExitInfo` model objects:

```
private fun convert(appExitInfo: ApplicationExitInfo): ExitInfo {
  return ExitInfo(
    description = appExitInfo.description.orEmpty(),
    importance = convertImportance(appExitInfo.importance),
    pss = appExitInfo.pss,
    rss = appExitInfo.rss,
    reason = convertReason(appExitInfo.reason),
    status = appExitInfo.status,
    timestamp = DateUtils.getRelativeTimeSpanString(
      context,
      appExitInfo.timestamp
    )
  )
}
```

```
  @StringRes
  private fun convertImportance(importance: Int): Int = when (importance) {
    ActivityManager.RunningAppProcessInfo.IMPORTANCE_CACHED -> R.string.importance_cached
    ActivityManager.RunningAppProcessInfo.IMPORTANCE_CANT_SAVE_STATE -> R.string.importance_cant_save_state
    ActivityManager.RunningAppProcessInfo.IMPORTANCE_FOREGROUND -> R.string.importance_foreground
    ActivityManager.RunningAppProcessInfo.IMPORTANCE_FOREGROUND_SERVICE ->
R.string.importance_foreground_service
    ActivityManager.RunningAppProcessInfo.IMPORTANCE_GONE -> R.string.importance_gone
    ActivityManager.RunningAppProcessInfo.IMPORTANCE_PERCEPTIBLE -> R.string.importance_perceptible
    ActivityManager.RunningAppProcessInfo.IMPORTANCE_SERVICE -> R.string.importance_service
    ActivityManager.RunningAppProcessInfo.IMPORTANCE_TOP_SLEEPING -> R.string.importance_top_sleeping
    ActivityManager.RunningAppProcessInfo.IMPORTANCE_VISIBLE -> R.string.importance_visible
    else -> R.string.shrug
  }
```

(from ForensicPathologist/src/main/java/com/commonsware/android/r/forensics/MainMotor.kt)

The UI then renders the results in a `RecyclerView`... once `ForensicPathologist` has been run a time or two and actually *has* results:

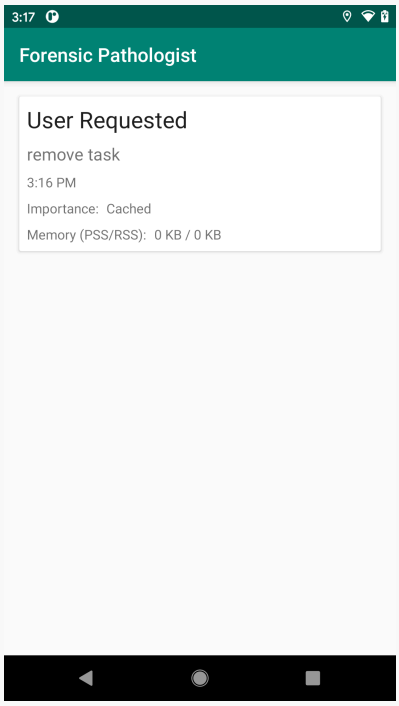


Figure 19: `ForensicPathologist`, Showing Results After Previous Process Termination

What To Do With the Results?

Similar to the data access auditing, the idea is that you might:

- Send aggregated statistics back to your server, such as the total number of process terminations and the count of each `reason` type
- Identify unusual cases and report more details on those (e.g., `REASON_ANR`, `REASON_PERMISSION_CHANGE`)

One limiting factor is that you cannot readily identify which `ApplicationExitInfo` objects you have seen previously, because there is no unique ID on them. You can attempt to work around this by checking for the last process termination reason immediately upon startup of a fresh process. Otherwise, you might count the same `ApplicationExitInfo` results multiple times.

Tracking Application State

You might find it useful to know some details about the nature of your app as part of the application exit reasons.

For example, you might be using "feature flags" to conditionally enable certain features. Depending on how you implemented those, you may not know, for any given situation, which feature flags are enabled and which are disabled. Perhaps they are random for A|B testing, or perhaps you are just worried that the flags might change between when an application exited and when somebody gets a chance to look at a report that you generate using these new APIs.

Apps can add a bit of information to the application exit data. If you call `setProcessStateSummary()` on `ActivityManager`, you can provide a byte array of data. When the application exits, if you called `setProcessStateSummary()` during the life of that process, the byte array gets recorded. Later on, when you retrieve the `ApplicationExitInfo` for this exit, you can call `getProcessStateSummary()` to retrieve the byte array, to include its contents in whatever your analysis does.

So, going back to the earlier example, once you find out what feature flags are enabled and disabled, you can record those as the process state summary. Later on, if you are trying to diagnose why your app is behaving as it is with respect to exits, you can see what feature flags are being used and perhaps determine if one of those flags is having a particular impact.

However:

- The documentation warns against calling `setProcessStateSummary()` too often
- There is no documentation on the size limit for the byte array, but you

should not assume it can hold large blobs of data
- This is not a replacement for existing ways of getting data between process invocations of your app (saved instance state, files, preferences, etc.)

ANRs and Traces

Perhaps no single error message caused developers more angst in the early years of Android than did "application not responding". A dialog with that message would appear if the app tied up the main application thread for a ridiculous amount of time (around 10 seconds). It became so well-known that developers started referring to it by the abbreviation ANR.

The problem with ANRs is that you do not know exactly where you are spending your time. We have had access to ANR trace files showing the state of our threads when an ANR occurs. However, on modern versions of Android, this data is only accessible on certain emulators or non-production builds, where we can achieve root access.

In Android 11, if your app exits with a reason of `REASON_ANR`, you can call `getTraceInputStream()` on the `ApplicationExitInfo` to access the trace data. You can read that in and do something with it (e.g., send it to your server), if desired.

Package Visibility

In a somewhat surprising move, Android 11 introduces a new privacy change: app isolation. Apps can no longer directly communicate with other apps, or even know that other apps exist, except in fairly limited ways.

In this chapter, we will explore more about what this means for you and your app.

The Way Things Were

Once upon a time — otherwise known as "before Android 11" — all apps were visible to all other apps. The *data* of those apps might be private, but the apps themselves were not. And one app could communicate with another app, via any exposed IPC interface, so long as permissions were met.

The quintessential example of this is a launcher. To provide the list (or grid or whatever) of icons for launchable activities, a launcher can call `queryIntentActivities()` on `PackageManager`, asking for all activities that have a `MAIN/LAUNCHER` `<intent-filter>`. `PackageManager` would return a list of matching activities, and the launcher would use that to populate its UI. And, when the user chooses an activity to start, the launcher would be able to call `startActivity()` on an `Intent` identifying that activity.

Even without `PackageManager`, you might still have your app communicate with other apps. For example, an upcoming chapter has a sample with two apps that communicate via a bound service and a pair of `Messenger` objects. The client side of the pair can call `bindService()` to bind to the service offered by the other app. So long as any permissions are met (e.g., `android:permission` attribute on the `<service>` element), that binding would be allowed.

Social Distancing for Apps

With Android 11, this is no longer the case.

Suppose, as in the latter example above, you have written two apps. The user assures you that both apps are installed. With Android 11, by default, you have no means of verifying from one app that the other app is installed. And, with Android 11, by default you will not be able to communicate between those apps. In the case from above, the client app cannot bind to the service app, even with a properly-constructed `Intent`.

(How does the author know this? The author beat his head against a wall for an hour trying to figure out why the sample wouldn't run, before remembering this new restriction...)

There are ways to whitelist certain things that will allow your app to see other apps. And there is an option to eliminate this restriction entirely. However, as you will see, neither are complete solutions — you should assume that the original forms of interoperability available to Android apps before will not be available to you going forward.

Also note that there are a number of other built-in limits on this restriction:

- If you start an activity using an implicit `Intent` (e.g., `ACTION_VIEW` on some `Uri`), that is allowed
- If you provide a fine-grained permission grant, such as `FLAG_GRANT_READ_URI_PERMISSION` on a `Uri` from `FileProvider`, the recipient can use that grant (e.g., open the content identified by the `Uri`)
- If you can talk to an app, the app can respond along that same IPC channel (e.g., `setResult()` in response to a `startActivityForResult()` call)
- You can talk among your own app's processes without restrictions

Whitelisting

The preferred way to relax this restriction is to "whitelist" certain things. Basically, you tell Android, via a `<queries>` element in the manifest, what sorts of other components you want to be able to see.

NOTE: Android Studio 4.0.1 — the current stable release of Android Studio — does not recognize this `<queries>` element, even if you put it as a child of the `<manifest>`

element (the correct location). Just ignore the warning, and hope that, in the future, newer versions of Android Studio will ship with knowledge of this `<queries>` element.

By Package

If you wish to have your app integrate with specific other apps, you can whitelist the package of that other app, by having a `<package>` element inside of the `<queries>` element, listing the package that you want to use:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.r.embed.client">
  <queries>
    <package android:name="com.commonsware.android.r.embed.server" />
  </queries>

  <application
    android:name=".MainApp"
    android:allowBackup="false"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from EmbedClient/src/main/AndroidManifest.xml)

Here, we have a fairly generic manifest, except for the `<queries>` element towards the top. Here, we say that this app wants to be able to communicate with the `com.commonsware.android.r.embed.server` app. This allows the client app to bind to a service exposed by `com.commonsware.android.r.embed.server`, and it allows that service to return data (e.g., send messages back via a supplied `Messenger`).

It appears that you can have as many `<package>` elements as you want. However,

there is no sign of support for wildcard pattern matching — you need to know, at compile time, what packages you need.

By Intent Signature

In our manifests, we are used to having `<intent-filter>` elements on components to say that they are available to other apps via matching `Intent` objects.

Now, in `<queries>`, we can have `<intent>` elements — with the same basic structure as `<intent-filter>` elements — advertising what other components we want to talk to via IPC.

So, for example, a launcher might have:

```
<manifest package="com.example.game">
    <queries>
        <intent>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent>
    </queries>
    <!-- rest of manifest goes here -->
</manifest>
```

This, in principle, should allow the launcher app to query for `MAIN/LAUNCHER` activities as before.

In general, everything allowed in an `<intent-filter>` is allowed in an `<intent>` element... with some restrictions:

- There must be exactly one `<action>` element, though you can use wildcards
- You are limited to `mimeType`, `scheme`, and `host` attributes on your `<data>` element, where wildcards are also supported

We will see an example of this later in the chapter.

`<queries>` and Gradle

The Android Gradle Plugin needs to know about new manifest elements, particularly for the manifest merger process. The plugin has a tendency to get confused if it sees elements in the manifest merger that it does not recognize, tossing out build errors like: “unexpected element <queries> found in <manifest>”.

And, as you might guess from that error, the Android Gradle Plugin was not happy about the introduction of `<queries>`.

The fact that this occurs from manifest merger means that simply upgrading a dependency might bring about this error. For example, if you upgrade to the latest version of `com.awesome:awesome-library`, and it contained a `<queries>` element in its manifest, you would crash with the aforementioned error in your builds, even without any actual app changes in your code.

Google released a series of patch versions of the Android Gradle Plugin to address this:

- `3.3.3`
- `3.4.3`
- `3.5.4`
- `3.6.4`
- `4.0.1`

If you are using an existing plugin in the `3.3.*` through `4.0.*` series, upgrade to the associated patch version (or higher) from that list, and you should no longer run into that error.

If you are using Android Studio 4.1 or higher, with a matching Android Gradle Plugin (e.g., in the `4.1.*` series), you should be fine without any changes. Those plugin versions were already aware of `<queries>`.

Escaping the Sandbox

There is a `QUERY_ALL_PACKAGES` permission. If your app holds it, all of these restrictions should be lifted.

This permission appears to be one with a `normal` value for `protectionLevel`. You do not need to request it at runtime — simply having it in the manifest is sufficient.

However it is documented as being restricted by the Play Store:

> In upcoming versions of the Developer Preview, look for Google Play to provide guidelines for apps that need this permission.

Based on similar restrictions, be prepared to fill out a form to explain why your app needs this permission. And, based on the experience of far too many developers, be

prepared to be banned from the Play Store by a bot if you try using it.

The next section will examine the impacts of holding this permission.

Effects and Ramifications

Certain types of apps will be able to cope using the whitelist mechanism. Launchers, for example, will be able to request, via the whitelist, to be able to see all `MAIN`/`LAUNCHER` activities, and be able to function more or less as before. Similarly, integration between a known pair of apps — such as the client binding to the service described above — can be whitelisted, since those apps are known in advance.

Apps that need flexibility across both whitelist axes — needing to know about arbitrary components in arbitrary packages — are in deep trouble. While `QUERY_ALL_PACKAGES` offers an escape hatch, it is a risky one for apps distributed via the Play Store. So, for example, various types of anti-malware app need that sort of flexibility. In theory, these apps should be eligible for `QUERY_ALL_PACKAGES` usage. In practice, unless you have a deep relationship with Google, you need to assume that your app will be targeted for removal.

To see all of this in action, the `QueryPackages` sample module in the book's sample project has a UI that lists the outcomes of the following sorts of calls on `PackageManager`:

- `getInstalledApplications()`
- `getInstalledPackages()`
- `getPackagesHoldingPermissions()` for the `INTERNET` permission
- `queryBroadcastReceivers()` for `ACTION_BOOT_COMPLETED` broadcasts
- `queryContentProviders()`
- `queryIntentActivities()` for the launcher activities

These are then presented in a long scrolling list with section headers, courtesy of `RecyclerView` and `MergeAdapter`.

The project also has five product flavors, for different scenarios:

Flavor	targetSdkVersion	<queries> Setup	Requests QUERY_ALL_PACKAGES?
alfa	29	none	no
bravo	30	none	no
charlie	30	<package>	no
delta	30	<intent>	no
echo	30	none	yes

The impacts of the package visibility changes only kicks in once your `targetSdkVersion` rises to `30` or higher. So, if you run `alfa` on an Android 11 device, you will see the full range of results, but if you run `bravo`, you only see pre-installed applications and their components. It is unclear if this is the long-term subset that you will be able to see by default, and it is also unclear to what extent device manufacturers can tweak this behavior.

The other three flavors opt into seeing more things.

The `charlie` flavor wants to be able to see the `ForensicPathlogist` module's package, from a sample profiled in another chapter:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.r.query">

  <queries>
    <package android:name="com.commonsware.android.r.forensics" />
  </queries>

</manifest>
```

(from QueryPackages/src/charlie/AndroidManifest.xml)

If you have that app installed, it will appear in the list of installed apps, installed packages, and launcher activities.

The `delta` flavor wants to be able to see apps with launcher activities:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.r.query">

  <queries>
    <intent>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
    </intent>

  </queries>

</manifest>
```

(from QueryPackages/src/delta/AndroidManifest.xml)

And, indeed, if you run that flavor, you will see those activities show up in the list of launcher activities. However, in Android 11, those apps *also* show up in all the other lists, as appropriate. Since most apps have a launcher activity, this particular `<queries>` setup largely reverses the restrictions placed here by Android 11.

The `echo` flavor requests `QUERY_ALL_PACKAGES`, just as a regular permission:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.r.query">

  <uses-permission android:name="android.permission.QUERY_ALL_PACKAGES" />

</manifest>
```

(from QueryPackages/src/echo/AndroidManifest.xml)

Running that flavor appears to give you the same results as does the `alfa` flavor, where our `targetSdkVersion` is still `29`.

So, if Google allows you to hold `QUERY_ALL_PACKAGES` (for apps distributed on the Play Store), you will be able to have the same behavior on Android 11 as you would on older devices. But, if you can live with just being able to opt into seeing user-installed apps with launcher activities, the `<queries>` structure seen in the `delta` flavor grants that, without `QUERY_ALL_PACKAGES`... assuming that Google does not change anything in future Android 11 updates.

So… Why Bother?

You might wonder why Google is bothering with this, given that the whitelists allow you to bypass the restrictions, and that's ignoring the official `QUERY_ALL_PACKAGES` opt-out.

In general, it appears as though this is simply a tightening of the security rules ("principle of least access"), not tied to anything specific.

And, while Google expressly hints about possible restrictions on Play Store distribution for apps using `QUERY_ALL_PACKAGES`, do not assume that the whitelists are some form of escape hatch:

- Future versions of Android might present information about your whitelists to the user (on the Play Store or at install time). If you ask for too much, the user might elect to abandon your app.
- Google might put Play Store restrictions on certain whitelist options. For example, they might allow actual launchers to whitelist apps with `MAIN`/`LAUNCHER` activities, but they might ban non-launchers from doing the same.

Logging What Was Filtered

Google added Logcat messages related to filtering. They will appear with the `AppFilter` tag and will be of the form:

```
? I/AppsFilter: interaction: PackageSetting{...} -> PackageSetting{...}
BLOCKED
```

...where the first `...` will contain your application ID and the second `...` will contain the application ID of the app that was filtered out.

This logging is enabled automatically for `debug` builds. If you need to test your production app on Android 11 for this sort of filtering behavior, you can use `adb` to enable it:

```
adb shell pm log-visibility --enable ...
```

...where the `...` is your application ID.

Note that the Logcat output may be rather extensive, as it lists *everything* that was blocked by your query. If you have a narrow whitelist, the list of stuff outside the whitelist may be rather long.

Sharing UIs

Over Android's history, many developers have wanted to embed the UI of one app in another app. However, support for this pattern was lacking, in part due to the performance limitations of early-generation Android devices.

Android 11 seems to be unwrapping a new approach to this problem, one that may make cross-app UI embedding much more practical and powerful. In this chapter, we will explore how this mechanism works and what its current limitations are.

UI Embedding: The Classic Approaches

If you have ever created an app widget or a "custom view" for a `Notification`, you have worked with `RemoteViews`. This is the primary way for one app to provide a hunk of UI to be embedded into another app, such as the UI for an app widget to be embedded in a home screen of a launcher. `RemoteViews`, though, have not been significantly improved since API Level 11. You can use relatively few widgets with them, and the only real event that you can respond to is a click.

Technically speaking, `RemoteViews` is simply a data structure describing a UI. It is entirely possible to create your own replacement for `RemoteViews` that uses a different data structure, in an attempt to get past the limitations of `RemoteViews`. Android 9's slices, for example, take this approach, even allowing it to be available via a Jetpack library for use with older devices. However, slices is more aimed at describing data to display, where some client code decides how to render that data. This adds flexibility at the cost of control — graphic designers, for example, cannot get "pixel perfect" UIs if the client can elect to render a slice differently than does another client.

As a result, most cross-app UI work has been handled simply between activities,

with App A starting an activity of App B when needed. This certainly works, but it is very coarse-grained, usually with each app taking over the screen of the phone or tablet.

What Android 11 Offers

Android 11 adds `SurfaceControlViewHost`. The name is awkward, but the capability it enables is enticing. Simply put, a `SurfaceControlViewHost` allows one app to have a view hierarchy (e.g., an inflated layout) be displayed in another app.

The approach taken by `RemoteViews` and slices is to send a description of a UI between processes, with the recipient being responsible for rendering the UI. By contrast, `SurfaceControlViewHost` works by sharing a `Surface` between the two apps, by means of the `SurfaceControl` class added in Android 10. The app with the view hierarchy renders to the `Surface` through the `SurfaceControlViewHost`, while the recipient renders the `Surface` itself through a `SurfaceView`.

Now the app with the view hierarchy has much greater control over what the result looks like. Any desired widgets, styles, and themes are available, because there is no requirement that the recipient have access to any of those things — all the recipient is doing is displaying the shared `Surface`.

The biggest limitation is that `SurfaceControlViewHost` is new to Android 11, as is the ability to connect the shared `Surface` to a `SurfaceView`. It *may* be possible to backport this to Android 10, when `SurfaceControl` was introduced, but that is far from certain. Since it takes years for users to get new versions of Android (usually via replacing their devices), this means that this capability, while interesting, will have limited real-world applicability for a while.

How to Share

The `EmbedClient` sample module in the book's sample project, along with the `EmbedServer` module, demonstrate how to work with `SurfaceControlViewHost`.

In the terminology being used in this chapter:

- The "server" is the app with the view hierarchy
- The "client" is the app that is displaying the UI from that view hierarchy

Client Setup

The first thing that the client needs is a `SurfaceView`, which will be where the embedded UI will be rendered. `EmbedClient` has an `activity_main` layout resource that consists of a `SurfaceView` and a `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:padding="8dp"
  tools:context=".MainActivity">

  <SurfaceView
    android:id="@+id/surface"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintBottom_toTopOf="@id/connect"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <Button
    android:id="@+id/connect"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="8dp"
    android:text="@string/connect"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from EmbedClient/src/main/res/layout/activity_main.xml)

Figure 20: EmbedClient, As Initially Launched

Android 11 adds a `getHostToken()` method to `SurfaceView`, returning an `IBinder` that represents the `SurfaceView`. The client needs to get that "host token", along with the ID of the `Display` used for that `SurfaceView`, and the dimensions of that `SurfaceView` over to the server app. `MainActivity` delegates this to a `MainMotor` viewmodel, by calling a `bind()` function when the user clicks that "Connect to Server" button:

```
package com.commonsware.android.r.embed.client

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.lifecycle.observe
import com.commonsware.android.r.embed.client.databinding.ActivityMainBinding
import org.koin.androidx.viewmodel.ext.android.viewModel

class MainActivity : AppCompatActivity() {
  private val motor: MainMotor by viewModel()

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
```

```
    val binding = ActivityMainBinding.inflate(layoutInflater)

    setContentView(binding.root)

    binding.surface.setZOrderOnTop(true)

    motor.surfacePackage.observe(this) {
      binding.surface.setChildSurfacePackage(it)
    }

    binding.connect.setOnClickListener {
      motor.bind(
        binding.surface.hostToken,
        binding.surface.display.displayId,
        binding.surface.width,
        binding.surface.height
      )
    }
  }
}
```

(from EmbedClient/src/main/java/com/commonsware/android/r/embed/client/MainActivity.kt)

`IBinder` can go into a `Bundle`, and the `Int` values for the display ID, width, and height can all be transferred easily between processes. We want to ensure that the server process runs as long as our client needs it, so `EmbedClient` and `EmbedServer` use the bound service pattern, with `EmbedServer` hosting the service. In particular, `EmbedServer` will expose a `Messenger` as its `Binder`, so `EmbedClient` can send a `Message` to it with the `IBinder` and `Int` values. So, `MainMotor` has a `bindToService()` function that uses `suspendCoroutine` to make the asynchronous act of binding to a service and getting the `Messenger` appear synchronous:

```
  private suspend fun bindToService(): MessengerConnection {
    return withContext(Dispatchers.Default) {
      suspendCoroutine<MessengerConnection> { continuation ->
        context.bindService(
          Intent().setClassName(
            "com.commonsware.android.r.embed.server",
            "com.commonsware.android.r.embed.server.ViewService"
          ),
          MessengerConnection { if (isActive) continuation.resume(it) },
          Context.BIND_AUTO_CREATE
        )
      }
    }
  }
}

private class MessengerConnection(private val onConnected: (MessengerConnection) -> Unit) :
```

```
  ServiceConnection {
  var messenger: Messenger? = null

  override fun onServiceConnected(name: ComponentName?, binder: IBinder?) {
    messenger = Messenger(binder)
    onConnected(this)
  }

  override fun onServiceDisconnected(name: ComponentName?) {
    messenger = null
  }
}
```

(from EmbedClient/src/main/java/com/commonsware/android/r/embed/client/MainMotor.kt)

The `bind()` function that `MainActivity` calls then binds to the service and sends a `Message` with our four pieces of data:

```
  fun bind(
    hostToken: IBinder?,
    displayId: Int,
    width: Int,
    height: Int
  ) {
    viewModelScope.launch {
      conn = bindToService()

      conn?.messenger?.send(Message.obtain().apply {
        data = bundleOf(
          KEY_HOST_TOKEN to hostToken,
          KEY_DISPLAY_ID to displayId,
          KEY_WIDTH to width,
          KEY_HEIGHT to height
        )
        replyTo = messenger
      })
    }
  }
```

(from EmbedClient/src/main/java/com/commonsware/android/r/embed/client/MainMotor.kt)

Setting up the `Bundle` is a bit clunky, because the `bundleOf()` implementation in `androidx.core:core-ktx:1.2.0` does not support `IBinder`, the data type of our "host token". So, we have to add that via a separate call to `putBinder()`.

Also note that our `Message` includes another `Messenger` in the `replyTo` property. This `Messenger` will be used by the server to send data back to the client. We will look more at that part of the process later in this chapter.

Also, as was seen in an earlier chapter, we need to whitelist the server app in order to be able to bind to it:

```
  <queries>
    <package android:name="com.commonsware.android.r.embed.server" />
  </queries>
```

(from EmbedClient/src/main/AndroidManifest.xml)

Otherwise, any `bindService()` call will fail, even with a valid `Intent`.

Server Setup

The job of the server is to set up the `SurfaceControlViewHost` and the UI to be displayed in the client.

All of that is handled by the `ViewService` being bound to by the client. While the app has an activity, that is simply for convenience when launching this sample from the IDE — the activity plays no role in the UI being served up.

The UI in question consists of a really big `Button`, plus a `TextView`:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="@android:color/white"
  android:padding="8dp">

  <Button
    android:id="@+id/button"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintBottom_toTopOf="@id/time"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <TextView
    android:id="@+id/time"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="8dp"
```

```
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from EmbedServer/src/main/res/layout/embedded.xml)

In `onCreate()` of `ViewService`, we use view binding to `inflate()` that layout and configure the widgets:

```
class ViewService : Service() {
  private lateinit var messenger: Messenger
  private val handlerThread = HandlerThread("ViewService")
  private lateinit var binding: EmbeddedBinding

  override fun onCreate() {
    super.onCreate()

    handlerThread.start()

    binding = EmbeddedBinding.inflate(LayoutInflater.from(this))
    var count = 0

    binding.button.text = getString(R.string.caption, count)
    binding.button.setOnClickListener {
      Log.d("ViewService", "button clicked")
      count += 1
      binding.button.text = getString(R.string.caption, count)
    }
    binding.time.text = Date().toString()

    messenger = Messenger(ViewHandler(this, binding, handlerThread.looper))

    Log.d("ViewService", "onCreate() finished")
  }

  override fun onBind(p0: Intent?): IBinder = messenger.binder
}
```

(from EmbedServer/src/main/java/com/commonsware/android/r/embed/server/ViewService.kt)

The `Message` from `EmbedClient` will be received by `handleMessage()` on the `ViewHandler` implementation of `Handler`. Our job is to process that message and, for the first message, set up the `SurfaceControlViewHost`:

```
private class ViewHandler(
  private val context: Context,
  private val binding: EmbeddedBinding,
  looper: Looper
) : Handler(looper) {
  private var host: SurfaceControlViewHost? = null

  override fun handleMessage(msg: Message) {
    Log.d("ViewService", "handleMessage() called")

    msg.data.apply {
      if (host == null) {
        val hostToken = getBinder(KEY_HOST_TOKEN)
        val displayId = getInt(KEY_DISPLAY_ID)
        val width = getInt(KEY_WIDTH)
        val height = getInt(KEY_HEIGHT)
        val display = context.getSystemService(DisplayManager::class.java)
          .getDisplay(displayId)

        host = SurfaceControlViewHost(context, display, hostToken).apply {
          setView(binding.root, width, height)

          val pkg = surfacePackage

          msg.replyTo.send(Message.obtain().apply {
            data = bundleOf(KEY_SURFACE_PACKAGE to pkg)
          })
        }
      } else {
        binding.time.text = Date().toString()
      }
    }
  }
}
```

(from EmbedServer/src/main/java/com/commonsware/android/r/embed/server/ViewService.kt)

If we have not set up the `host` previously, we grab the values out of the `Message` and obtain the `Display` object for our display ID. We then:

- Create the `SurfaceControlViewHost`, passing the "host token" and `SurfaceView` dimensions to the constructor
- Call `setView()` to attach our inflated layout to the host
- Call `getSurfacePackage()` on the `host` and send that back to the client via the `replyTo` `Messenger`

If, on the other hand, we already have the `host` set up from before, we just update the `TextView` to show the now-current `Date`.

Client Completion

The `replyTo` `Messenger` that we attached to the outbound message is set up in the `init` block of `MainMotor`:

```
class MainMotor(private val context: Context) : ViewModel() {
  private var conn: MessengerConnection? = null
  private val _surfacePackage =
    MutableLiveData<SurfaceControlViewHost.SurfacePackage>()
  val surfacePackage: LiveData<SurfaceControlViewHost.SurfacePackage> =
    _surfacePackage
  private val handlerThread = HandlerThread("EmbedClient")
  private val handler: Handler
  private val messenger: Messenger

  init {
    handlerThread.start()
    handler = PackageHandler(handlerThread.looper) {
      _surfacePackage.postValue(it)
    }
    messenger = Messenger(handler)
  }
```

(from EmbedClient/src/main/java/com/commonsware/android/r/embed/client/MainMotor.kt)

The `PackageHandler` simply calls the supplied callback upon receipt of a `Message`, extracting out the `SurfacePackage` sent by the server:

```
private class PackageHandler(
  looper: Looper,
  private val onPackageReceipt: (SurfaceControlViewHost.SurfacePackage) -> Unit
) : Handler(looper) {
  override fun handleMessage(msg: Message) {
    val pkg = msg.data.getParcelable<SurfaceControlViewHost.SurfacePackage>(
      KEY_SURFACE_PACKAGE
    )

    pkg?.let { onPackageReceipt(it) }
  }
}
```

(from EmbedClient/src/main/java/com/commonsware/android/r/embed/client/MainMotor.kt)

The SurfacePackage is supplied to MainActivity via LiveData, and MainActivity calls setChildSurfacePackage() on the SurfaceView to attach it:

```
    motor.surfacePackage.observe(this) {
      binding.surface.setChildSurfacePackage(it)
    }
```

(from EmbedClient/src/main/java/com/commonsware/android/r/embed/client/MainActivity.kt)

The Results

If you install both apps, then launch EmbedClient and click the "Connect to Server" button, you will see the EmbedServer-supplied UI in what had been the big open area of the SurfaceView:

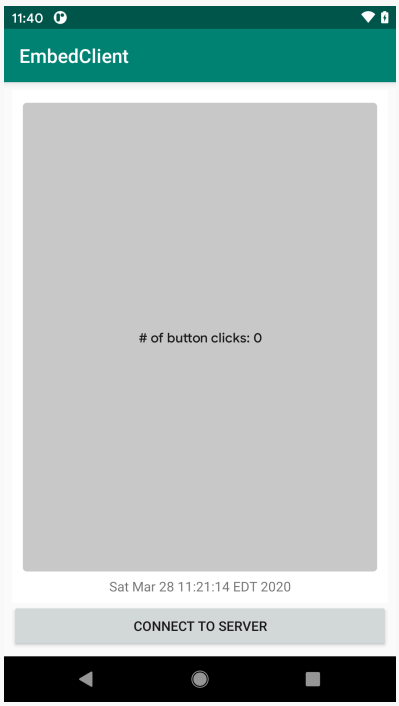


Figure 21: EmbedClient, After Connecting to Server

If you click the "Connect to Server" button again, the TextView text will show the now-current date, illustrating that the connection between client and server is live. All the server is doing on these subsequent button clicks is updating the text in the TextView — it is not doing anything else to "push" a new rendition of the UI to the client. That is handled by SurfaceControlViewHost and the underlying shared Surface.

Enabling Input

If you set the `SurfaceView` to be on top from a Z axis standpoint, then input events delivered to the `SurfaceView` will be routed to the corresponding widgets and their listeners.

In these examples, `ViewService` added an `OnClickListener` to the `Button`. If you click the `Button` as viewed in `EmbedClient`, you see the button caption being updated by that listener. You also see the standard ripple effect, though fine-grained animations like that do not seem to work well, even on relatively good hardware (e.g., a Pixel 2).

The trick to making this work is setting that Z axis order, which we do in the `EmbedClient` edition of `MainActivity` via `setZOrderOnTop(true)`:

```
    binding.surface.setZOrderOnTop(true)
```

(from EmbedClient/src/main/java/com/commonsware/android/r/embed/client/MainActivity.kt)

Conversations and Bubbles

There are very few user-facing features in Android 11. One is the re-introduction of "bubbles" as an extension of the notification system:

Figure 22: Bubble, As Initially Launched

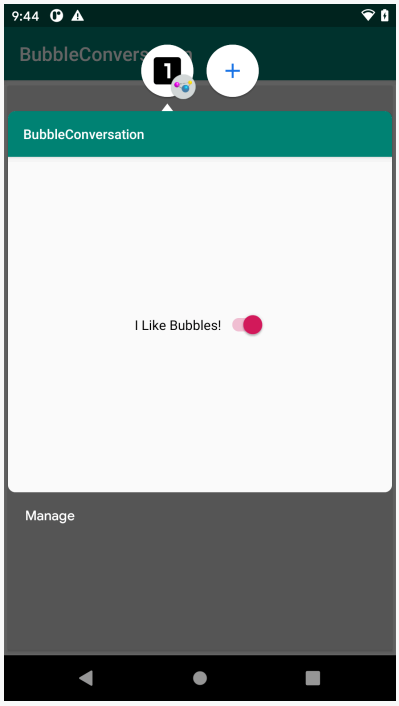


Figure 23: Bubble, Showing Expanded State

This is tied to a new "conversations" system for notifications, with an eye towards messaging apps and similar sorts of situations where you are interacting with another person.

From "Chat Heads" to Bubbles

In 2013, Facebook debuted the "chat heads" UI for their Android app. These allowed the user to participate in Facebook chats while being (mostly) in other apps, by having a floating avatar of your chat partner appear over the UI of whatever app you were in.

Technically, this was somewhat of an abuse of the `SYSTEM_ALERT_WINDOW` permission and related system-level windows. Facebook's "leadership" in this area led many other developers to apply the same technique. However, allowing arbitrary apps to interpose arbitrary UI in front of other UI has security risks, and Google is starting to restrict the use of `SYSTEM_ALERT_WINDOW` as a result.

However, Google recognizes the utility of this sort of system, which is why they are adding bubbles as a framework-supported, user-controllable option for the same

sort of effect... albeit one that is tied to a "conversation".

The Basics of Conversations

Google has portrayed "conversations" as being a major thing in Android 11. And, in truth, it is one of the few user-facing features of Android 11. However, from a programming standpoint, "conversations" may be just a minor extension to what you are already doing with notifications.

Conversation Presentation

Conversation notifications are placed above regular notifications in the notification shade:

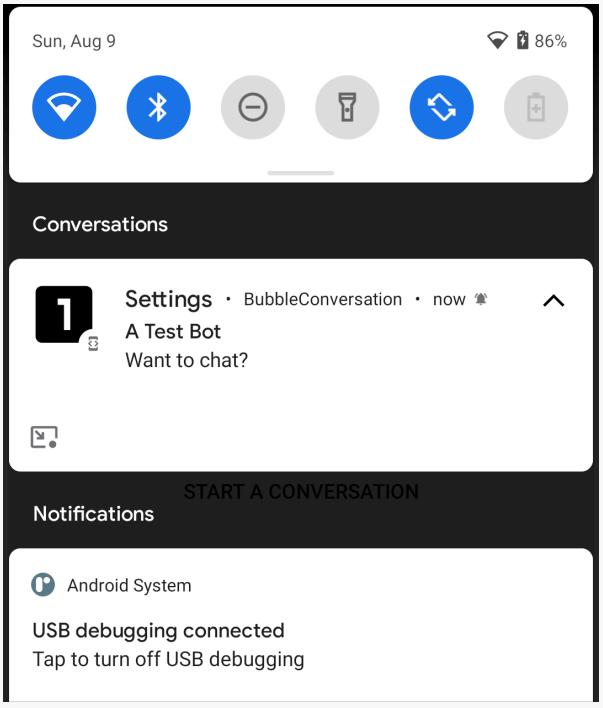


Figure 24: A Conversation Notification in the Shade

They have slightly different presentation, with a greater emphasis on a developer-supplied icon, typically representing the person (or bot or other non-corporeal entity) with which the "conversation" is being held. Tapping the caret toggles between expanded and collapsed perspectives, and long-pressing the caret can bring up options, such as making the conversation be priority or silent:

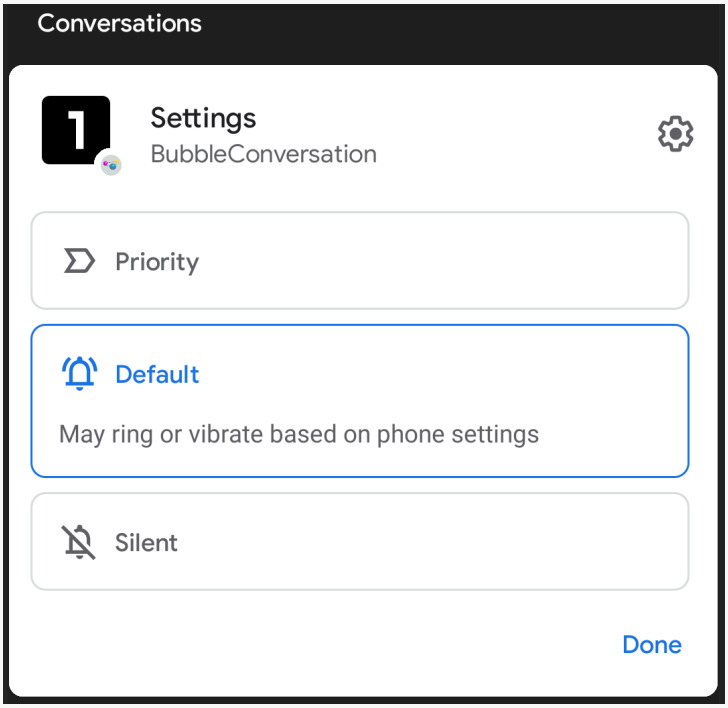


Figure 25: Conversation Options from Caret Long-Press

But, mostly, this is just a `MessagingStyle` notification.

However, you may notice an icon in the lower-left of the notification. If this is available, the user can turn this conversation into a bubble by tapping on it. This implies that you have set up this notification with the metadata needed for bubbles, and we will explore that and the rest of the bubble setup and presentation later in the chapter.

Constructing a Conversation

A conversation notification is a `MessagingStyle` notification with a "long-lived" shortcut associated with it. The documentation states that the shortcut must be associated with `Person` objects in the conversation, though this does not appear to be required. The shortcut that gets created not only helps set up your conversation,

but it appears as a real shortcut, such as on a long-press of your launcher icon. As such, the shortcut needs to be real and to work, even though the `Intent` associated with that shortcut is not used with the notification itself.

The `BubbleConversation` sample module in the book's sample project shows a basic recipe for getting a bubble to display and work, in the context of a "conversation"-style notification. It has an activity with a really big button that, when clicked, raises a conversation-style notification.

This code sets up a `NotificationCompat.Builder` for a conversation:

```
    val shortcutInfo = ShortcutInfoCompat.Builder(this, SHORTCUT_ID)
      .setLongLived(true)
      .setShortLabel("Settings")
      .setIntent(Intent(Settings.ACTION_SETTINGS))
      .setIcon(IconCompat.createWithResource(this, R.drawable.ic_one))
      .build()

    ShortcutManagerCompat.pushDynamicShortcut(this, shortcutInfo)

    val builder = NotificationCompat.Builder(
      appContext,
      CHANNEL_WHATEVER
    )
      .setSmallIcon(R.drawable.ic_notification)
      .setContentTitle("Um, hi!")
      .setBubbleMetadata(bubble)
      .setShortcutInfo(shortcutInfo)

    val person = Person.Builder()
      .setBot(true)
      .setName("A Test Bot")
      .setImportant(true)
      .build()

    val style = NotificationCompat.MessagingStyle(person)
      .setConversationTitle("A Fake Chat")

    style.addMessage("Want to chat?", System.currentTimeMillis(), person)
    builder.setStyle(style)
```

(from BubbleConversation/src/main/java/com/commonsware/android/r/bubble/MainActivity.kt)

Most of this code is mostly there to set up the `MessagingStyle` notification. And the `setBubbleMetadata()` call is tied to bubbles, as we will see later in the chapter.

To make this be categorized as a conversation, we:

- Create a `ShortcutInfoCompat` using `ShortcutInfoCompat.Builder`, providing enough information to create valid shortcut, plus `setLongLived(true)`
- Register that shortcut with `ShortcutManagerCompat`, in this case via `pushDynamicShortcut()`
- Attach that shortcut to the notification using `setShortcutInfo()` on `NotificationCompat.Builder`

In this case, the shortcut itself just launches the device's Settings app. A more typical solution would be to have it launch something related to the conversation or its participants.

Note that the icon used as the primary visual indicator in the conversation is the icon associated with the shortcut (in this case, `R.drawable.ic_one`). Hence, you will want to set that icon to be something that represents the conversation or its participants, further emphasizing the need for the shortcut itself to be tied to the same things.

The Basics of Bubbles

A bubble is an option for a conversation-style notification. It basically detaches the notification from the notification shade and has it be represented by a free-floating icon that, when tapped, will display a designated activity in a floating window. As a result, there are two main steps in enabling bubbles for a conversation:

- Setting up that activity for the floating window
- Teaching the notification that you want a bubble as an option

Crafting the Activity

There are two elements to a bubble:

- The actual bubble dot itself
- The content that is shown when the user taps on the bubble

That content is in the form of an Android activity.

The UI

Since this is an activity, you have access to the full range of Android UI options. There are no known technical limitations, so if you want to use a `SurfaceView` or `WebView` or whatever, you should be fine.

However, do bear in mind that your activity is likely to be smaller than the full screen height. Also, while it is possible for that activity to start other activities, those by default will remain in the bubble's window, forming its own back stack.

You are not *technically* restricted to simple UIs like those of app widgets, slices, and wearables. However, a bubble activity still should err on the side of simplicity, particularly while users are getting used to how bubbles look and operate.

The Manifest Entry

As with any activity, the bubble content activity will have an `<activity>` element in the manifest.

The documentation states that the `<activity>` must have three key attributes:

- `android:allowEmbedded="true"`, to say that this activity can be embedded in some other UI
- `android:documentLaunchMode="always"`, to say that if there is a `Uri` associated with the activity, different `Uri` values will result in different tasks and separate activity instances
- `android:resizeableActivity="true"`, to say that the activity window can be resized at will by the user

The documentation claims that if these requirements are not met, then your requested bubble will not be created and you wind up with a plain `Notification` instead. In reality, none of these are required for the bubble to appear.

In the `BubbleConversation` module, we have a `BubbleActivity` that we want to display as the bubble content. As a result, we put those three attribute values on the `BasicBubble` `<activity>` element:

```
    <activity
      android:name=".BubbleActivity"
      android:allowEmbedded="true"
      android:documentLaunchMode="always"
      android:resizeableActivity="true" />
```

(from BubbleConversation/src/main/AndroidManifest.xml)

`BubbleActivity` simply loads a layout with a `Switch` widget to indicate whether or not you like bubbles:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent" android:layout_height="match_parent">

  <Switch
    android:id="@+id/switch1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:checked="true"
    android:switchTextAppearance="@style/TextAppearance.AppCompat.Large"
    android:text="I Like Bubbles!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from BubbleConversation/src/main/res/layout/activity_bubble.xml)

Note that the `Switch` does not actually do anything. But, feel free to toggle it, if you like!

Requesting the Bubble

To show the bubble, you need to display a conversation notification that has `BubbleMetadata` attached to it.

`NotificationCompat.Builder` has a `setBubbleMetadata()` method that we can use

to request a bubble for our `Notification`. There is a corresponding `NotificationCompat.BubbleMetadata` class, with a `Builder`, that we can use to create that metadata to supply to `setBubbleMetadata()`.

This code snippet illustrates setting up the metadata, as part of setting up the conversation notification as seen earlier in this chapter:

```
private fun buildBubbleNotification(appContext: Context, showExpanded: Boolean = false): Notification {
  val pi = PendingIntent.getActivity(
    appContext,
    0,
    Intent(appContext, BubbleActivity::class.java),
    PendingIntent.FLAG_UPDATE_CURRENT
  )

  val bubble = NotificationCompat.BubbleMetadata.Builder()
    .setDesiredHeight(400)
    .setIcon(IconCompat.createWithResource(appContext, R.drawable.ic_two))
    .setIntent(pi)
    .apply { if (showExpanded) setAutoExpandBubble(true); setSuppressNotification(true) }
    .build()

  val shortcutInfo = ShortcutInfoCompat.Builder(this, SHORTCUT_ID)
    .setLongLived(true)
    .setShortLabel("Settings")
    .setIntent(Intent(Settings.ACTION_SETTINGS))
    .setIcon(IconCompat.createWithResource(this, R.drawable.ic_one))
    .build()

  ShortcutManagerCompat.pushDynamicShortcut(this, shortcutInfo)

  val builder = NotificationCompat.Builder(
    appContext,
    CHANNEL_WHATEVER
  )
    .setSmallIcon(R.drawable.ic_notification)
    .setContentTitle("Um, hi!")
    .setBubbleMetadata(bubble)
    .setShortcutInfo(shortcutInfo)

  val person = Person.Builder()
    .setBot(true)
    .setName("A Test Bot")
    .setImportant(true)
    .build()

  val style = NotificationCompat.MessagingStyle(person)
    .setConversationTitle("A Fake Chat")

  style.addMessage("Want to chat?", System.currentTimeMillis(), person)
  builder.setStyle(style)

  return builder.build()
}
```

(from BubbleConversation/src/main/java/com/commonsware/android/r/bubble/MainActivity.kt)

There are three key methods on `NotificationCompat.BubbleMetadata.Builder`:

- `setDesiredHeight()` indicates how much vertical space you want for your bubble content, measured in `dp`. Note that this is a request, and your actual height may be larger or smaller than what you request.
- `setIcon()` eventually will let you specify an icon for the bubble itself. Note that your launcher icon will appear superimposed on the bubble icon. As a result, using the launcher icon *as* the bubble icon is atypical, but we are doing that here anyway.
- `setIntent()` provides the `PendingIntent` to start the activity that is your bubble content. In theory, this `PendingIntent` could be one for a service or receiver instead of an activity, though this may give you undesired results.

If `showExpanded` is `true`, we also call two additional builder methods:

- `setAutoExpandBubble()`, to indicate that we want the bubble to show up in full, not just as a simple bubble
- `setSuppressNotification()`, to indicate that we do not want the actual `Notification` to appear if the bubble is displayed — the bubble alone is all that we need

The resulting `Notification` can be displayed as normal:

```
private fun showBubble(appContext: Context, showExpanded: Boolean = false) {
  NotificationManagerCompat.from(appContext).let { mgr ->
    mgr.createNotificationChannel(
      NotificationChannel(
        CHANNEL_WHATEVER,
        "Whatever",
        NotificationManager.IMPORTANCE_DEFAULT
      )
    )

    mgr.notify(NOTIF_ID, buildBubbleNotification(appContext, showExpanded))
  }
}
```

(from BubbleConversation/src/main/java/com/commonsware/android/r/bubble/MainActivity.kt)

The UX

As was noted earlier, there is an icon in the lower-left of a conversation notification that, when tapped, will bring up the bubble. Future uses of bubbles by your

notifications for the same channel, by default, will bring up the bubble icon (or the expanded bubble) immediately.

In the collapsed form, the bubble consists of the shortcut icon with your app's launcher icon superimposed upon it:

Figure 26: Bubble, As Initially Launched

The bubble itself can be dragged around the screen, though it will always gravitate toward one of the sides. This allows the user to reposition it to not obscure something of importance.

Tapping the bubble itself expands it:

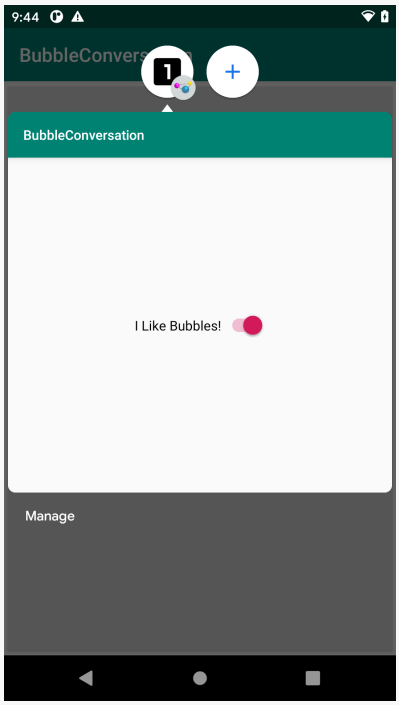


Figure 27: Bubble, Showing Expanded State

You go directly to this expanded state if you use `setAutoExpandBubble(true)` in your `BubbleMetadata.Builder`.

Despite the `android:resizeableActivity="true"` attribute in the manifest, the system UI does not seem to allow the user to resize or move the bubble content.

The user can:

- Tap outside of the expanded view to collapse it back into a bubble
- Get rid of the bubble by dragging it to the bottom-center of the screen, over an X icon that appears while dragging
- Click the "Manage" button on the lower edge, to bring up a small menu of options for the user to configure the behavior of bubbles from your app
- Click the plus icon in a circle that appears next to your bubble above your activity, to do... something apparently with recent bubbles (this has little documentation)

Security Stuff

Aspects of the changes to scoped storage and the `MediaStore` pertain to security, as does the introduction of package filtering and the tweaks to permissions. In this chapter, we will explore other security changes introduced in Android 11.

New Foreground Service Types

Android 10 introduced the `android:foregroundServiceType` attribute for the `<service>` element in the manifest. Apps using certain capabilities need to declare those intentions using this attribute. For example, if your foreground service needs to use location APIs, you would need `android:foregroundServiceType="location"`. Failing to include this attribute may mean that your app will be unable to use those capabilities when the app only has a foreground service and no foreground UI.

Despite Google's efforts to pretend that nothing changed here, two new foreground service types were added in Android 11:

- `camera`
- `microphone`

In addition, these two new foreground service types have an interesting phrase in the JavaDoc comments for their corresponding `ServiceInfo` constants: `FOREGROUND_SERVICE_TYPE_CAMERA` and `FOREGROUND_SERVICE_TYPE_MICROPHONE`:

> For apps with targetSdkVersion Build.VERSION_CODES.R and above, a foreground service will not be able to access the [camera|microphone] if this type is not specified in the manifest and in `Service.startForeground(int, android.app.Notification, int)`.

Normally, we can use the two-parameter `startForeground()` on `Service` to establish our service as a foreground service. That method implies that we want to use all of the foreground service type flags specified in `android:foregroundServiceType` in the manifest. The documentation's use of "and", though, suggests that for `camera` and `microphone` that we not only need them in the manifest but also need to pass them specifically to the three-parameter `startForeground()` method, which takes a bitmask of foreground service types as the third parameter.

However, *elsewhere* in the documentation, we have:

> If your app starts a foreground service while running in the background, the foreground service cannot access the microphone or camera.

So, if you use the camera or microphone APIs from a foreground service, you should test your app early and often on Android 11.

`BiometricPrompt` and Weak Biometrics

`BiometricPrompt` is the framework class responsible for authenticating the user, in-app, based on how the user has secured their device. Through `BiometricPrompt`, your app can re-confirm that the person using the app is authorized to use the device, in case somebody else took the device from its normal user.

There is some history behind `BiometricPrompt`:

- The original focus was on `KeyguardManager` and the entry of PINs or passwords ("device credentials")
- `FingerprintManager` focused on fingerprints; part of the reason for the change to `BiometricPrompt` was to support other forms of biometrics besides fingerprints

Android 11 now divides the biometric options into "strong" and "weak". Fingerprints are strong; face recognition is weak. When you set up your `BiometricPrompt`, you can indicate what authenticators are considered to be acceptable, so you can opt into supporting weak biometrics if you choose to.

The SecureCheq sample module in the book's sample project is an updated version of a sample from *Elements of Android Q* that re-authenticates the user when they tap a large fingerprint icon:

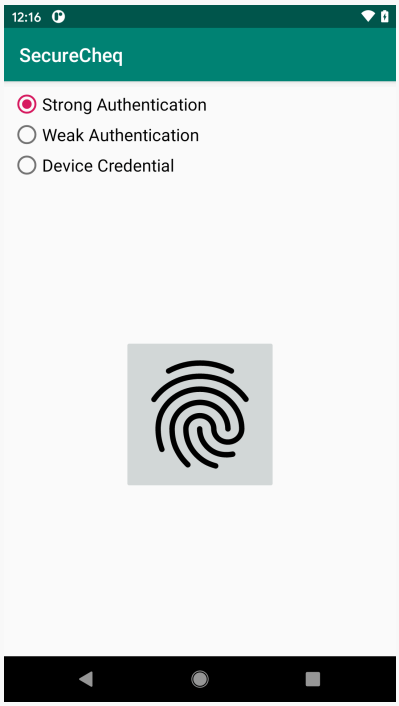


Figure 28: SecureCheq Sample App, As Initially Launched

New to the `RSampler` project's edition of `SecureCheq` is the `RadioGroup` at the top, allowing the user to choose an authenticator. This, in turn, maps to a `BiometricManager.Authenticators` constant:

- `BIOMETRIC_STRONG`
- `BIOMETRIC_WEAK`
- `DEVICE_CREDENTIAL`

`setAllowedAuthenticators()`, called on a `BiometricPrompt.Builder`, lets you specify which of those three options you support. The method accepts a `vararg`, so you can pass as many of these authenticator options as you wish. This sample only passes in one, based on the selected radio button:

```
val prompt = BiometricPrompt.Builder(this)
  .setTitle("This is the title")
  .setDescription("This is the description")
  .setSubtitle("This is the subtitle")
  .apply {
```

```
      when {
        Build.VERSION.SDK_INT > 29 && strong.isChecked ->
          setAllowedAuthenticators(BiometricManager.Authenticators.BIOMETRIC_STRONG)
        Build.VERSION.SDK_INT > 29 && weak.isChecked ->
          setAllowedAuthenticators(BiometricManager.Authenticators.BIOMETRIC_WEAK)
        Build.VERSION.SDK_INT > 29 ->
          setAllowedAuthenticators(BiometricManager.Authenticators.DEVICE_CREDENTIAL)
        Build.VERSION.SDK_INT > 28 -> {
          setDeviceCredentialAllowed(true)
        }
        else -> {
          setNegativeButton(
            getString(R.string.btn_negative),
            mainExecutor,
            DialogInterface.OnClickListener { _, _ ->
              fingerprint.setImageDrawable(off)
              Toast.makeText(
                this@MainActivity,
                R.string.msg_negative,
                Toast.LENGTH_LONG
              ).show()
            })
        }
      }
    }
    .build()
```

(from SecureCheq/src/main/java/com/commonsware/android/r/auth/MainActivity.kt)

However, since this app supports older versions, we have a few possible patterns:

- On Android 11, we call `setAllowedAuthenticators()` with the user-selected authenticator
- On Android 10, we instead call `setDeviceCredentialAllowed(true)`, which roughly equates to passing `BiometricManager.Authenticators.DEVICE_CREDENTIAL` to `setAllowedAuthenticators()`
- On Android 9.0, we settle for configuring a "negative" button that the user can click to exit the biometric prompt

What you get depends on the authenticator and the user's device. So, for example, if the device is set up for fingerprints, and the user chooses "Strong Authentication", the fingerprint dialog appears. That dialog is now protected using `FLAG_SECURE`, blocking people (such as book authors) from taking screenshots.

If, on the other hand, the device is *not* set up for fingerprints, even `BIOMETRIC_STRONG` falls back to `DEVICE_CREDENTIAL`, and the user is prompted for their PIN or password.

Right now, the author of this book does not have an Android 11-equipped device that

offers weak authentication options (e.g., face recognition). This section will be updated later in 2020 with more details on how that works with the new `BiometricPrompt`.

`Toast` Restrictions

There are a number of new limitations on `Toast` that you will need to take into account.

The big one is displaying a `Toast` with a custom `View` (via `setView()`). This is still allowed, but only from the foreground. If you try this from the background, the `Toast` is not displayed and a warning about the violation is recorded in Logcat.

For apps with a `targetSdkVersion` of 30 or higher, you also:

- Cannot call `getView()` to get the `View` displayed by the `Toast` (it just returns `null`)
- Cannot modify the margins or gravity
- Cannot retrieve the margins or gravity (return values from methods like `getGravity()` "don't reflect the actual values, so you shouldn't rely on them in your app")

The rationale for all of these is to prevent the user from using a `Toast` to occlude critical aspects of the UI. For example, in principle, an app could display a custom `Toast` over top of a system permissions dialog. The `Toast` could show a message that implies that the permission being requested is more benign than it really is. While this attack is unreliable, given the short and transient nature of a `Toast`, it could still work.

Further CA Certificate Restrictions

In general, having custom certificate authority (CA) certificates in a device opens up security problems. CAs are used to verify SSL/TLS certificate chains, and a fraudulent CA certificate makes it possible for malicious parties to pretend to be Web sites and services that they are not. Google has been slowly tightening the screws on where these certificates can come from for years.

However, there are plenty of legitimate uses for them, including enterprises (who sometimes use custom certificate authorities to help secure their own internal Web sites) and debugging tools (HTTP Toolkit, Charles Proxy, etc.).

Android 11 adds a lightly-documented new restriction: apps cannot ask users to install a certificate via `KeyChain.createInstallIntent()`. Added back in Android 4.1, this method would build an `Intent` where you could supply a CA certificate via an `Intent` extra, and a `startActivityForResult()` call would ask the user if she wanted that certificate to be installed. For CA certificates, on Android 11 and higher, this no longer works.

Instead, users now need to do this manually:

- Go to the Security screen in the Settings app
- In there, navigate to "Encryption & credentials" > "Install a certificate" > "CA certificate"
- Agree to proceed, despite a security warning
- Use the Storage Access Framework UI to find the CA certificate and choose to open it
- Confirm the installation

And, of course, these instructions will vary by manufacturer, as manufacturers have a habit of changing how the Settings app looks and works.

See this blog post for more on the subject.

Device Controls

As was noted earlier, few significant user-facing new features were added in Android 11. One is "device controls". Akin to how Android 8.0 allowed developers to offer items to show in the notification shade through a `TileService`, Android 11 allows developers `ControlsProviderService`.

In this chapter, we will explore what our options are for device controls.

The High-Level View

Android has long supported a "power menu" that is triggered by a long-press on the device's POWER button. Traditionally, this had an option for a complete device shutdown, where a short press of the POWER button just turns off the screen. Later versions of Android started offering other items here, such as a device reboot or capturing of a screenshot. Android 11 restyles this "menu" once again, now taking up the entire screen... and part of that screen represents stuff that we as developers can control.

What the User Gets

If you long-press the POWER button on an Android 11 device, a screen akin to this one will appear:

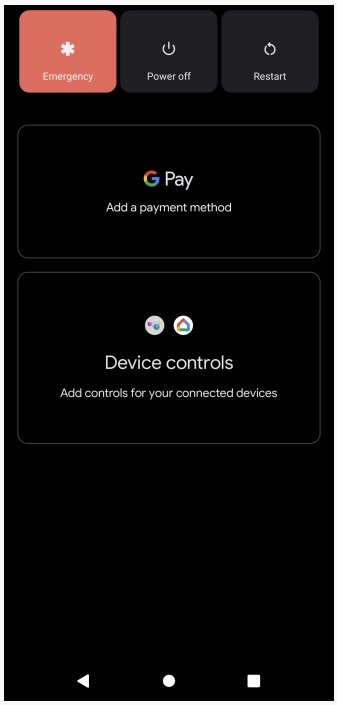


Figure 29: Android 11 Power Menu, As Initially Launched

In the middle of the screen is a “Device controls” card. Tapping that will display various sources of device controls:

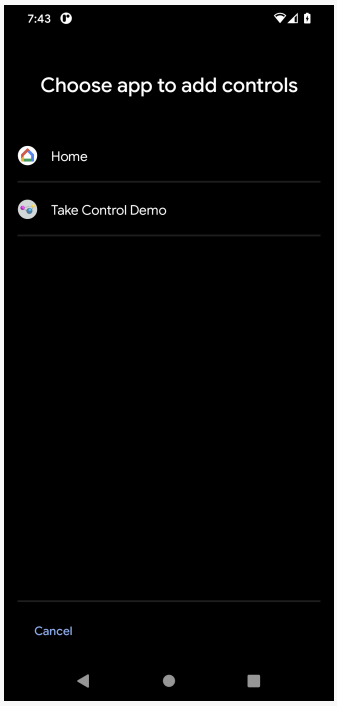


Figure 30: Device Controls Sources List

Tapping one of those will give you a lineup of available devices to control, perhaps based on other configuration performed in the main app:

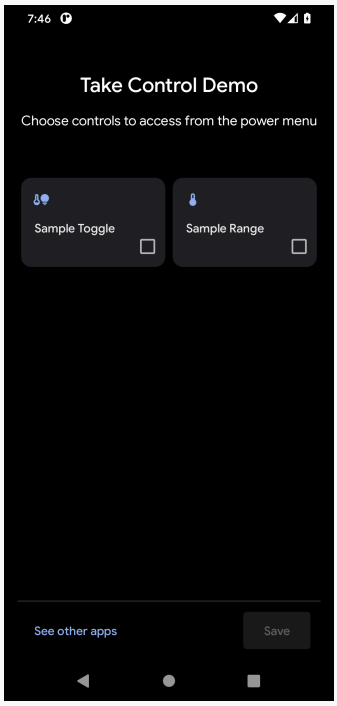


Figure 31: Device Controls From Sample App

You can check and uncheck the checkboxes in the lower right corner of each of those tiles to determine which of these you want to appear on your "power menu". They will then show up on the main "power menu" screen, and you can interact with them, as we will explore more later in this chapter.

How We Build It

Your graphic designers might look at this and envision all sorts of possibilities for what they might do in this space.

Your graphic designers will be very disappointed.

We do not directly control the look or interactivity of these tiles. Instead, we describe what we want in fairly generic terms ("we want the user to control a value in a range from 1 to 10"), and Android renders that how Android sees fit.

In this respect, this is reminiscent of Android 9.0's flagship feature: slices. There, we

did not provide the direct UI of a slice, but instead described the general structure of what we want, and the slice host would decide how to render that structure.

(and, if you forgot about slices, or never heard of them in the first place, you did not miss much...)

A subclass of `ControlsProviderService` provides the API that you will expose to Android to supply the contents of these tiles. So, to offer device control tiles to your users, you will create a `ControlsProviderService` subclass and implement its API to publish the roster of possible tiles and support the user interacting with a chosen subset of those tiles.

Elements of a Control Tile

Each tile is represented by a `Control`. When we create instances of a `Control`, we get to define a `Template` and a device type for that `Control`, plus we get to react to an `Action` when one is raised.

Template

The element that determines what a tile looks and works like is the template. As the name suggests, it provides a general structure for a tile, where you get to fill in the details, such as the text that appears in the tile.

Templates are defined via subclasses of `ControlTemplate`, where the specific subclass determines the type of interactivity:

- `ToggleTemplate` allows the user to toggle between two states (e.g., on and off)
- `RangeTemplate` allows the user to choose a value within a range, reminiscent of a `SeekBar`
- `ToggleRangeTemplate` allows the user to do both of those: turn something on/off and, if on, control the value within a range
- `TemperatureControlTemplate` supports toggling between multiple distinct modes (e.g., heat, cool, "eco"), in addition to wrapping one of the aforementioned template types (e.g., allow the user to toggle between heating and cooling, plus specify a value within a temperature range)

All of those have an associated state that the user is modifying via the tile. The assumption is that your app has the ability to determine the current state of

whatever the tile controls and pass along the user's requested changes to that state.

There is also a `StatelessTemplate`. While you can find out about taps on this type of tile (via actions, covered in the next section), there is no state that your app — or the device controls framework — needs to track.

Action

Your choice of template in turn drives how the user can interact with it. Your app will find out about the user's choices via an action. Actions are represented by subclasses of `ControlAction`. The subclasses represent the type of data that is being tracked as state for this tile; your app, upon receiving the action, is supposed to change the state of the associated device to that new value:

Action Class	State Data Type	Associated Templates
`BooleanAction`	boolean	`ToggleTemplate`, `ToggleRangeTemplate`, `TemperatureControlTemplate`
`FloatAction`	float	`RangeTemplate`, `ToggleRangeTemplate`, `TemperatureControlTemplate`
`ModeAction`	int	`TemperatureControlTemplate`

There is also a `CommandAction`, emitted by a `StatelessTemplate`, that just tells you that the user clicked on the tile. The idea is that you might use this to send some command to the device that is not necessarily tied to any state.

Device Type

Our choice of "device type" for a tile primarily controls an icon that goes in the upper-start corner of the tile. Secondarily, it might have other visual effects, such as changing the default color that gets used.

A device type is represented by an `Int` value. `DeviceTypes` contains a list of `Int` constants for the supported device types. This is a long list, containing both obvious device types (lights, thermostats), esoteric ones (hood, mop, drawer), and generic ones (on/off, lock/unlock, temperature).

In reality, not every one of these gets a distinct icon, at least in stock versions of Android 11.

Control

All of the above get wrapped up into a `Control`. You create instances of a `Control` via builders. Principally, you will use `Control.StatefulBuilder` to not only provide the details of the control but also the current state value associated with the device (e.g., is it on or off for a `ToggleTemplate`). There is also a `Control.StatelessBuilder` that you will use for a specific initial-loading scenario (but, despite the name, this is only somewhat related to `StatelessControl`).

`Flow`... But Not That `Flow`

The point behind device controls is to give the user an accessible UI for manipulating some device via your app — a thermostat, an overhead door, a lamp, etc. There may be some delays involved in your app finding out the current state of the device and in changing that state via the user's interaction with these tiles. Plus, the state often might change via other inputs, such as somebody pushing a button or flipping a switch, so you might need to deliver state changes over time.

This calls for an asynchronous API, where you can have some time to deliver responses, use background threads, and so on.

You might think that Google would use `LiveData` here. However, `LiveData` is specifically designed for dealing with lifecycles, and there isn't really a "lifecycle" involved with this work. Plus, `LiveData` is a Jetpack library, and framework classes like `ControlsProviderService` cannot depend upon Jetpack libraries.

You might think that given Google's strong interest in Kotlin that they would use coroutines, such as `Flow`. However, while Google overall is Kotlin-friendly, the framework is not. Google's coroutine-centric APIs are all in Jetpack libraries. The device controls API needs to be more readily usable by Java apps.

You might think that Google would go "retro" and use callbacks or listeners, since they are still all over the place in the Android SDK. Or, perhaps they would use some sort of `Future` or one of the seemingly-endless number of classes and interfaces named `Observable`.

Instead, they chose to use the Reactive Streams API... somewhat.

The Reactive Streams API is a cross-platform initiative for defining reactive APIs. Android 11 includes the JDK 9 edition of an API based on Reactive Streams, in the form of interfaces like `Flow.Publisher`, used to provide a stream of results to some subscriber.

However, Android 11 does not include an *implementation* of Reactive Streams, just the API embodied in those interfaces. The expectation is that you will use a third-party library that can provide implementations of those interfaces, where the leading contender for this is RxJava.

If your project already uses RxJava, great! However, the Reactive Streams classes in RxJava may be different than the ones that you are used to. In RxJava, typically we use `Observable` and `Subject`, but the Reactive Streams RxJava equivalents are `Flowable` and `Processor`. Still, the concepts are fairly similar.

If your project does not use RxJava right now... you get to start! Isn't that fun!

(narrator: it will not be fun, but many Android developers use RxJava successfully, so you will be able to do so as well)

Taking Control of the Situation

The `TakeControl` sample module in the book's sample project contains a `ControlsProviderService` implementation that offers two tiles to the user: one based on a `ToggleTemplate` and one based on a `RangeTemplate`. The screenshots shown earlier in this chapter show "Take Control Demo" and those two tiles.

The Dependencies

While `ControlsProviderService`, `ControlTemplate`, and such are all part of the Android SDK, your Reactive Streams implementation is not. So, you will need to add in dependencies for that implementation.

This sample app uses RxJava 2, and so we have dependencies for it and the `reactive-streams` library that helps adapt RxJava 2 to the JDK's `Flow...` set of interfaces:

```
  implementation "org.reactivestreams:reactive-streams:1.0.3"
  implementation "io.reactivex.rxjava2:rxjava:2.2.9"
```

(from TakeControl/build.gradle)

The <service> Element

TakeControlService is our ControlsProviderService implementation. Like any Service, TakeControlService appears in the manifest with a <service> element. However, the <service> element has a few important pieces, beyond the android:name attribute that identifies the service class:

```
<service
  android:name=".TakeControlService"
  android:label="@string/serviceLabel"
  android:permission="android.permission.BIND_CONTROLS">
  <intent-filter>
    <action android:name="android.service.controls.ControlsProviderService" />
  </intent-filter>
</service>
```

(from TakeControl/src/main/AndroidManifest.xml)

The two that are documented are:

- You need the <intent-filter> to advertise that your service is a ControlsProviderService
- You need the android:permission attribute to ensure that only the OS will be able to bind to your service

The requirement that is undocumented is `android:label`. This forms the display name of your `ControlsProviderService`. In the screenshots, where you see "Take Control Demo" is where your service's `android:label` value appears, such as on the list of available providers:

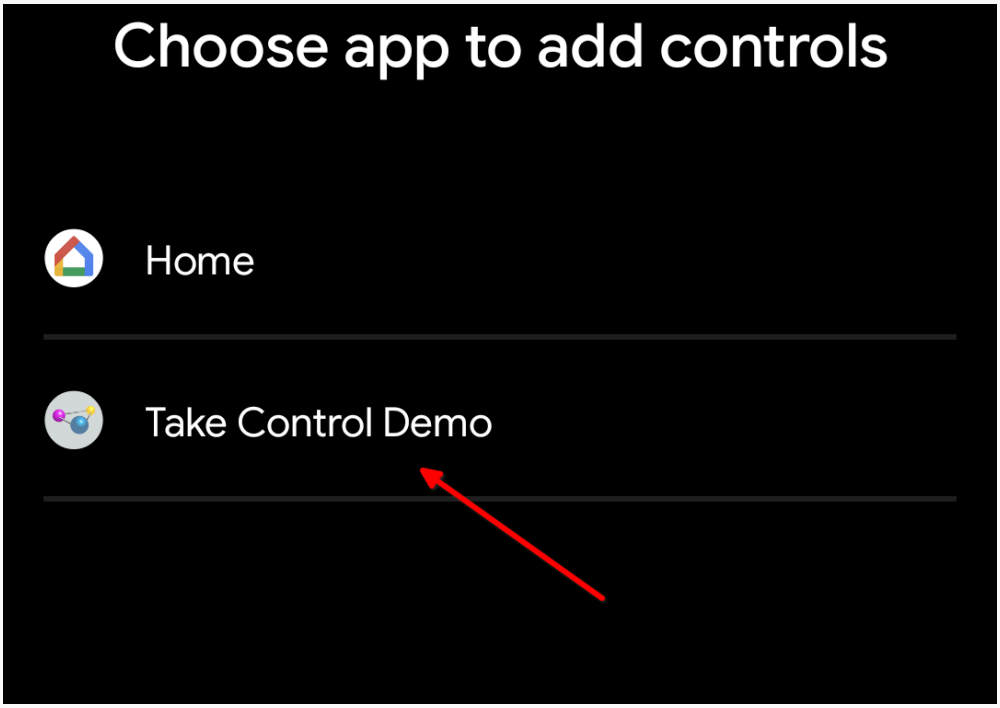


Figure 32: Device Controls Provider Chooser, with Demo Provider Display Name Highlighted

Publishing All Available Controls

There are three `abstract` functions that we need to override in a `ControlsProviderService`. The first is `createPublisherForAllAvailable()`. Here, "all available" is referring to the roster of controls: we need to tell Android what are all the possible controls that can be offered to the user. This is what populates the control picker screen that we saw earlier:

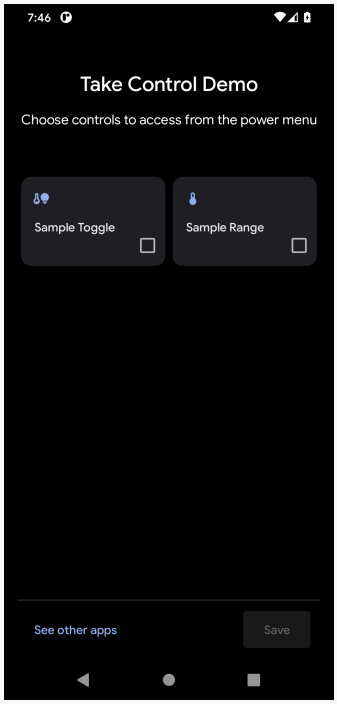


Figure 33: Device Controls From Sample App

This function needs to return a `Flow.Publisher` that will emit the `Control` objects as they become available... or all at once, as is the case in the sample:

```
override fun createPublisherForAllAvailable(): Flow.Publisher<Control> =
  FlowAdapters.toFlowPublisher(
    Flowable.fromIterable(
      listOf(
        buildStatelessControl(TOGGLE_ID, TOGGLE_TITLE, TOGGLE_TYPE),
        buildStatelessControl(RANGE_ID, RANGE_TITLE, RANGE_TYPE)
      )
    )
  )
```

(from TakeControl/src/main/java/com/commonsware/android/r/control/TakeControlService.kt)

With the RxJava 2 and Reactive Streams libraries, the easiest way to create this `Flow.Publisher` is to create an RxJava `Flowable`, then convert it to a `Flow.Publisher` via `FlowAdapters.toFlowPublisher()`. And, the easiest possible `Flowable` (other than an empty one) is to create one from a `List` of objects using `Flowable.fromIterable()`.

The `Control` objects that we need to emit on our `Flow.Publisher` must be made using `Control.StatelessBuilder`. At least in part, that is because the control picker screen is showing tiles representing controls, but not anything regarding the current state of the device that those controls control.

To that end, `TakeControlService` has a `buildStatelessControl()` function that uses `Control.StatelessBuilder` to build a `Control`. We pass in a unique `Int` ID, a string resource representing a title, and a `DeviceType` value, defined as constants:

```
private const val TOGGLE_ID = 1337
private const val TOGGLE_TITLE = R.string.toggleTitle
private const val TOGGLE_TYPE = DeviceTypes.TYPE_GENERIC_ON_OFF
private const val RANGE_ID = 1338
private const val RANGE_TITLE = R.string.rangeTitle
private const val RANGE_TYPE = DeviceTypes.TYPE_THERMOSTAT
```

(from TakeControl/src/main/java/com/commonsware/android/r/control/TakeControlService.kt)

`buildStatelessControl()`, in turn, builds the stateless `Control`:

```
  private fun buildStatelessControl(
    id: Int,
    @StringRes titleRes: Int,
    type: Int
  ): Control {
    val title = getString(titleRes)
    val intent = MainActivity.buildIntent(this, title)
      .addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
    val actionPI = PendingIntent.getActivity(
      this,
      id,
      intent,
      PendingIntent.FLAG_UPDATE_CURRENT
    )

    return Control.StatelessBuilder(id.toString(), actionPI)
      .setTitle(title)
```

```
      .setDeviceType(type)
      .build()
  }
```

(from TakeControl/src/main/java/com/commonsware/android/r/control/TakeControlService.kt)

The `Control.StatelessBuilder` constructor takes a unique ID (as a `String`) for this control, along with a `PendingIntent`. That `PendingIntent` will be invoked if the user long-taps on the tile for a control. It needs to be an activity `PendingIntent`, and it will be displayed in a system-supplied bottom sheet. That "embed the activity in a bottom sheet" hack has a side effect: the `Intent` used to build the `PendingIntent` must have `FLAG_ACTIVITY_NEW_TASK` on it, or else that bottom sheet will crash when it goes to display the activity. Alas, that requirement is undocumented.

As the name suggests, `Control.StatelessBuilder` is a class with a builder-style API. There are two configuration methods that we need to call, before we call `build()` to build the actual `Control`:

- `setTitle()` sets the title that you see at the top of the tile
- `setDeviceType()` sets the `DeviceType` `Int` value that controls the icon associated with the tile

In a real app, you would be examining what devices the user has configured in your app, determining what controls you can offer for those, building stateless `Control` objects for those, and then emitting them via your `Flow.Publisher`. This sample hard-codes the available controls for simplicity.

If you have a *lot* of controls, you might also consider overriding `createPublisherForSuggested()`. This allows you to supply a separate `Flow.Publisher` for a subset of your controls, indicating ones that you feel are most likely to be of use to the user.

Updating Specific Controls

The second required method is `createPublisherFor()`. This will be called with a list of the string IDs of the controls that the user selected from the control picker. Your job is emit *stateful* `Control` objects on a `Flow.Publisher` for those controls, both initially and if the device state represented by the control changes. So, for example, if you have a control representing the on/off state of a light switch, you will need to emit a stateful `Control` to indicate the current state of that switch as of the call to `createPublisherFor()` *and* if the state of that switch changes.

This time, since (in theory) we are delivering results over time, we cannot just create a `Flowable` from a `List`. Instead, we use RxJava's `ReplayProcessor`:

```
override fun createPublisherFor(controlIds: List<String>): Flow.Publisher<Control> {
  val flow: ReplayProcessor<Control> = ReplayProcessor.create(controlIds.size)

  controlIds.forEach { controlFlows[it] = flow }

  executor.execute {
    // TODO real work to figure out the state, simulated by a one-second delay
    SystemClock.sleep(1000)

    flow.onNext(buildToggleStatefulControl())

    // TODO real work to figure out the state, simulated by a one-second delay
    SystemClock.sleep(1000)

    flow.onNext(buildRangeStatefulControl())
  }

  return FlowAdapters.toFlowPublisher(flow)
}
```

(from TakeControl/src/main/java/com/commonsware/android/r/control/TakeControlService.kt)

We are going to need to use that `Flow.Publisher` over time, and the API for `ControlsProviderService` does not hand it back to us. In theory, we might be called with `createPublisherFor()` several times for several lists of controls — this is not well-documented. So, the sample holds onto the `ReplayProcessor` in a `MutableMap`, keyed by the string ID value:

```
private val controlFlows =
  mutableMapOf<String, ReplayProcessor<Control>>()
```

(from TakeControl/src/main/java/com/commonsware/android/r/control/TakeControlService.kt)

And, at the end of our `createPublisherFor()` function, use use `FlowAdapters.toFlowPublisher()` to convert that `ReplayProcessor` into a `Flow.Publisher` for Android to use.

We also need to arrange to emit the stateful controls for those IDs. However, this may take time — you might need to talk to some hardware over a slow BLE connection, for example. To simulate this, the sample uses a single-thread `Executor` and a couple of `sleep()` calls to pretend to do work. Then, we use a `buildToggleStatefulControl()` and `buildRangeStatefulControl()` to emit `Control` objects representing the now-current state.

Those two functions mostly delegate to a `buildStatefulControl()` function, just passing in a bunch of values:

```
private fun buildToggleStatefulControl() = buildStatefulControl(
  TOGGLE_ID,
  TOGGLE_TITLE,
  TOGGLE_TYPE,
  toggleState,
  ToggleTemplate(
    TOGGLE_ID.toString(),
    ControlButton(
      toggleState,
      toggleState.toString().toUpperCase(Locale.getDefault())
    )
  )
)

private fun buildRangeStatefulControl() = buildStatefulControl(
  RANGE_ID,
  RANGE_TITLE,
  RANGE_TYPE,
  rangeState,
  RangeTemplate(
    RANGE_ID.toString(),
    1f,
    10f,
    rangeState,
    0.1f,
    "%1.1f"
  )
)
```

(from TakeControl/src/main/java/com/commonsware/android/r/control/TakeControlService.kt)

We need to supply the same ID, title, and type as we did with the stateless controls. We also need to supply the value of the current state, which for a toggle is a `Boolean` and for a range is a `Float`. Those are simply held onto as properties in the service in this trivial sample:

```
private var toggleState = false
private var rangeState = 5f
```

(from TakeControl/src/main/java/com/commonsware/android/r/control/TakeControlService.kt)

A real app would be getting them from the actual device being controlled by this service. And, a real app would not make any assumptions about how long the `ControlsProviderService` instance might be running — the framework could destroy and recreate the service as it sees fit. But, for a sample, this will suffice.

`buildStatefulControl()` also takes the `ControlTemplate` for the control that we are trying to build. In the case of the toggle control, that is a `ToggleTemplate`, and in the case of the range control that is a `RangeTemplate`. A template also gets a unique ID as a string, though reusing the same ID as is used for the `Control` the template goes into seems to work, at least for simple templates like these. The rest of the template configuration is based on the type of the template:

- A `ToggleTemplate` just takes a `ControlButton`, with a boolean value to indicate if it is checked and a seemingly-pointless `String` parameter
- A `RangeTemplate` takes the minimum and maximum values of the range (e.g., `1f` to `10f`), the current value, how granular the changes should be (e.g., `0.1f`), and a "format string" that will be used to format the state for display

Here, "format string" refers to the sort of template that you use with `String.format()` or string resources. Here, we use `%1.1f` to show the current value to one decimal point.

`buildStatefulControl()` then uses all of that stuff and assembles our `Control` using `Control.StatefulBuilder`:

```
private fun <T> buildStatefulControl(
  id: Int,
  @StringRes titleRes: Int,
  type: Int,
  state: T,
  template: ControlTemplate
): Control {
  val title = getString(titleRes)
  val intent = MainActivity.buildIntent(this, "$title $state")
    .addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
  val actionPI = PendingIntent.getActivity(
    this,
    id,
    intent,
    PendingIntent.FLAG_UPDATE_CURRENT
  )

  return Control.StatefulBuilder(id.toString(), actionPI)
    .setTitle(title)
    .setDeviceType(type)
    .setStatus(Control.STATUS_OK)
    .setControlTemplate(template)
    .build()
}
```

(from TakeControl/src/main/java/com/commonsware/android/r/control/TakeControlService.kt)

`Control.StatefulBuilder` has a builder-style API with the same `setTitle()` and `setDeviceType()` methods as does `Control.StatelessBuilder`. You also need to call:

- `setStatus()`, typically with `STATUS_OK`, to indicate that you are able to determine the status of the control
- `setControlTemplate()`, with your configured `ControlTemplate` for this control

There are other methods that you can call, such as `setStatusText()`, which provides the `String` to show after the icon (and, in the case of `RangeTemplate`, before the formatted value of the current selection).

Once you deliver those to Android — by calling `onNext()` on your `ReplayProcessor` — Android will update the UI of the tile to show the state, if the user happens to have the power menu open at the time. You will be called with `createPublisherFor()` each time the user opens the power menu.

Responding to Actions

The third method that you need to override is `performControlAction()`. This will be called when the user interacts with the control, other than via a long-click (which invokes your `PendingIntent`). Your job is to update the device based on that action, then emit a fresh stateful `Control` with the updated state.

```
override fun performControlAction(
  controlId: String,
  action: ControlAction,
  consumer: Consumer<Int>
) {
  controlFlows[controlId]?.let { flow ->
    when (controlId) {
      TOGGLE_ID.toString() -> {
        consumer.accept(ControlAction.RESPONSE_OK)
        if (action is BooleanAction) toggleState = action.newState
        flow.onNext(buildToggleStatefulControl())
      }
      RANGE_ID.toString() -> {
        consumer.accept(ControlAction.RESPONSE_OK)
        if (action is FloatAction) rangeState = action.newValue
        flow.onNext(buildRangeStatefulControl())
      }
```

```
        else -> consumer.accept(ControlAction.RESPONSE_FAIL)
      }
    } ?: consumer.accept(ControlAction.RESPONSE_FAIL)
  }
```

(from TakeControl/src/main/java/com/commonsware/android/r/control/TakeControlService.kt)

The first parameter to `performControlAction()` is the `String` ID of the control that the user used. We use that both to look up the cached `ReplayProcessor` for that control and to branch in a `when()` to process the action.

The second parameter is a `ControlAction` object, representing the actual action that the user performed. For a `ToggleTemplate` control, it should be a `BooleanAction`, and for a `RangeTemplate` control, it should be a `FloatAction`.

The third parameter is a `Consumer`, which we use to tell Android whether we understood the request. Call `accept()` on the `Consumer` with `RESPONSE_OK` if you are able to process the action or `RESPONSE_FAIL` if you cannot for some reason.

In our case, we:

- Look up the `ReplayProcessor` for the supplied ID
- If we recognize the ID, call `accept()` with `RESPONSE_OK`
- Update the state property for that control based on the value contained in the action
- Use `buildToggleStatefulControl()` or `buildRangeStatefulControl()` to build a fresh stateful `Control` representing the updated state, then emit that using `onNext()` on our `ReplayProcessor`

Here, the sample does all of that immediately. In a real app, updating the device with the new state may take time, and so you would have some background thread do that work and emit the updated `Control` when the device has been modified.

The Results

If the user chooses “Take Control Demo” from the available control providers, they will see the stateless editions of our tiles as samples:

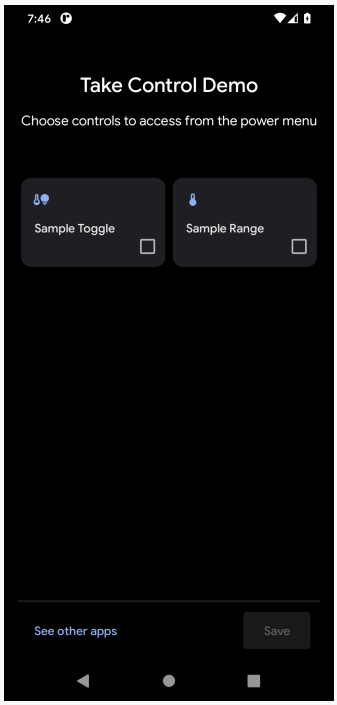


Figure 34: Device Controls From Sample App, In Control Picker

If they check both of those, those tiles will appear in the power menu, initially as stateless editions:

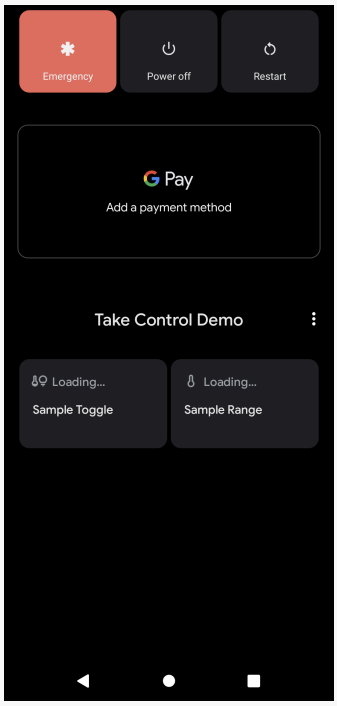


Figure 35: Chosen Device Controls From Sample App, Shortly After POWER Long Press

Eventually, the stateful editions of our tiles are displayed. In particular, the "Sample Range" tile shows our current value (via that format string) and has a shaded fill to highlight how far along the range the current value is:

Figure 36: Chosen Device Controls, in Stateful Form

The user can tap on the "Sample Toggle" to toggle it on, which shows up with a highlight when on:

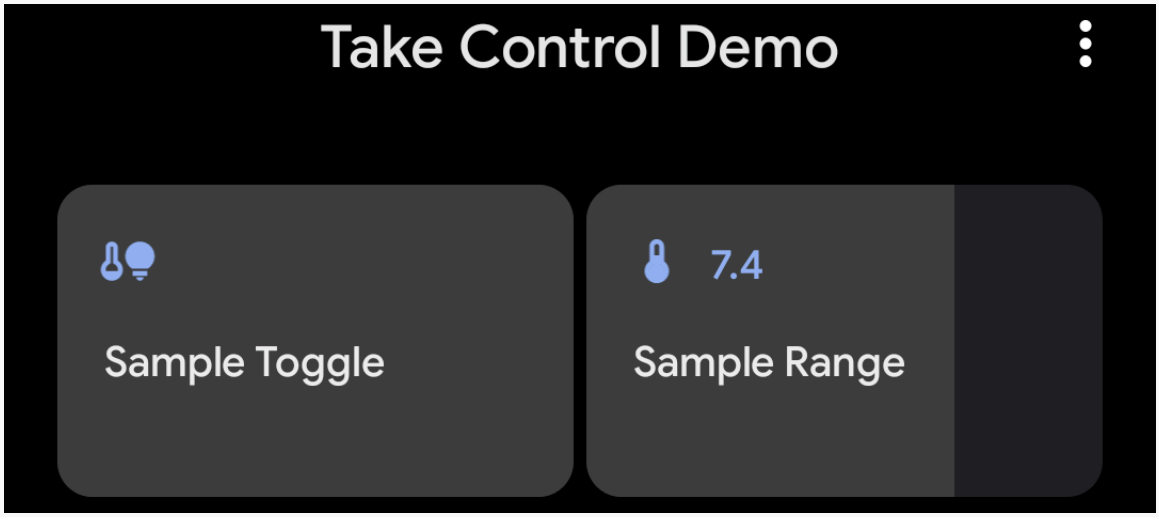


Figure 37: Chosen Device Controls, With Toggle Switched On

Similarly, the user can slide their finger horizontally across the range tile to change its value.

As the user makes changes, actions get sent to our `performControlAction()` implementation, and it is our emitted stateful control in response that helps determine the end visual state.

If the user long-presses on a tile, the activity identified in our `PendingIntent` is created and shown... in a bottom sheet:

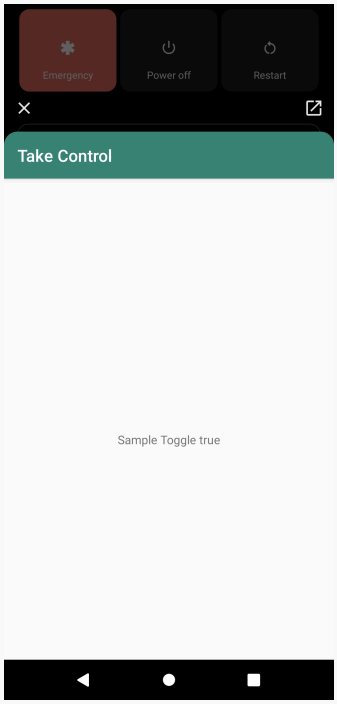


Figure 38: Activity Launched from Tile Long-Press

The icon in the upper-right allows the user to expand the activity into a traditional full-screen size.

Ideally, the activity will show something of relevance to the tile just long-pressed-upon.

Other APIs

You can call `ControlsProviderService.requestAddControl()` to ask the system to ask the user if a particular control should be added to the POWER screen. For example, you might do this in response to some user input or configuration of a device for which you offer controls.

In addition to the elements that we configured on controls in the sample, you can supply:

- A subtitle to appear below the title
- A "structure" and a "zone", to help the user identify the specific device managed by the tile (e.g., which light switch?)
- A custom color or icon to use in the tile

Also, stateful controls can have "status text", which is shown adjacent to the icon and gives you another way to textually represent the current status of whatever the tile manages.

Other Changes of Note

There are lots of other changes in Android 11, far more than can be presented in this book. This chapter covers a variety of additional changes that you may want to pay attention to.

Stuff That Might Break You

The scariest batch of changes in any Android release are the ones that may break existing app behavior. Things like package visibility might qualify for that.

Here are a few other smaller changes that may cause problems for reasonably-ordinary apps.

Dismissable Ongoing Notifications

For your foreground services, you may be used to raising "ongoing" notifications. These normally are not dismissable by the user.

However, in Android 11, they are.

On the plus side, this does not appear to affect your process importance. You are still registered as having a foreground service, even if the user gets rid of your notification.

However, if you were used to that notification always being there to give the user control over that background work... now the user might remove that notification, intentionally or accidentally. Make sure that your app can still function reasonably, from a UX standpoint, if the user dismisses your notification.

Phone Number Permissions

Some Android SDK methods let you attempt to get the phone number of the device, such as `getLine1Number()` on `TelephonyManager`. In practice, these are not very reliable, but you are welcome to try to use them.

For years, in order to call those methods, you needed the `READ_PHONE_STATE` permission. This is a `dangerous` permission, and you needed to use runtime permissions to further request it on Android 6.0+.

In Android 11, once your `targetSdkVersion` reaches 30, you need to request a *different* permission for those methods: `READ_PHONE_NUMBERS`. This too is a `dangerous` permission.

As a result, you need to decide which of those permissions you need based on API level, and include the correct permission in the array that you pass to `requestPermissions()`.

If the only reason you were requesting `READ_PHONE_STATE` was to use these methods, you may find that you no longer need it on Android 11+ devices. If so, you could elect to add `android:maxSdkVersion="29"` to your `<uses-permission>` element for `READ_PHONE_STATE` to drop it off for API Level 30 and higher devices:

```
<uses-permission android:name="READ_PHONE_STATE" android:maxSdkVersion="29" />
```

Conversely, if you are using `READ_PHONE_STATE` for other things, you will need to request *both* `READ_PHONE_STATE` and `READ_PHONE_NUMBERS` on API Level 30 devices, but only `READ_PHONE_STATE` on older devices.

Overlay Tweak

Hopefully, you are not using the `SYSTEM_ALERT_WINDOW` permission in your app. If you are, then you probably have noticed that Google is continuing to "tighten the screws" with each passing release. While it is possible that they will never outright ban `SYSTEM_ALERT_WINDOW`, they certainly seem intent upon making it more aggravating for developers and users.

In Android 11, the change is to `ACTION_MANAGE_OVERLAY_PERMISSION`.

`SYSTEM_ALERT_WINDOW` is one of those special permissions that does not go through the standard `dangerous` runtime permission system. Rather, the user needs to go

into the "Special app access" section of the Apps screen in Settings, and from there go into "Display over other apps". There, the user can tap on an app that is requesting `SYSTEM_ALERT_WINDOW` and elect to grant or reject that permission for that app.

`ACTION_MANAGE_OVERLAY_PERMISSION` is an `Intent` action that allows you to send the user to this place in Settings. In Android 6.0 through 10, there were two ways to use this `Intent`:

- On its own, to lead the user to the "Display over other apps" screen
- With a `package` `Uri` tied to your application ID, to lead the user straight to the screen where they can grant (or deny) that permission for your app

In Android 11, that latter option is no longer available. You can provide the `Uri`, but it will be ignored by the Settings app. The user is always taken to the "Display over other apps" screen, where the user will need to click on your app, then grant the permission.

No Third-Party Image Capture Support

`ACTION_IMAGE_CAPTURE` is a popular means for an app to take a picture, by asking a camera app to do the "heavy lifting". This means that the app can skip all of the headache of setting up a camera itself (e.g., via the CameraX library) and does not need the `CAMERA` permission. There are also `ACTION_IMAGE_CAPTURE_SECURE` and `ACTION_VIDEO_CAPTURE` `Intent` actions that perform similar operations.

The problem is that pre-installed camera apps often do not test these actions much, so they tend to have bugs. In lieu of dumping these actions and doing the camera work directly in the app, many apps simply guide the user to install a third-party camera app, one that is known to have a good implementation of things like `ACTION_IMAGE_CAPTURE`. Then, when the app invokes the `ACTION_IMAGE_CAPTURE` `Intent`, the user could choose the third-party camera app in the chooser.

That is no longer an option on Android 11, once your `targetSdkVersion` reaches 30.

Those three `Intent` actions will only start a pre-installed camera app. User-installed camera apps are ignored. Even if the user has disabled all pre-installed camera apps, user-installed camera apps are still ignored — Android throws an `ActivityNotFoundException` instead.

At minimum, if you are using these `Intent` actions, you will need to handle the

`ActivityNotFoundException` scenario. You really needed that anyway, as work policies or similar constraints might have caused that `Intent` to fail anyway.

Even if you try enabling package visibility for your desired `Intent` action, you will find that third-party apps are ignored.

An explicit `Intent` works for the `startActivity()`/`startActivityForResult()` call, if you happen to know of a camera app to try (e.g., `net.sourceforge.opencamera` for Open Camera). This also works with `queryIntentActivities()` on `PackageManager`, if you also enable package visibility, so you can determine whether the `Intent` would succeed or not before trying to start the activity.

The `CamChooser` sample module in the book's sample project demonstrates a related approach: adding candidate camera apps to a chooser:

```
package com.commonsware.android.r.camchooser

import android.content.Context
import android.content.Intent


private val CAMERA_CANDIDATES = listOf(
  "net.sourceforge.opencamera"
)

fun enhanceCameraIntent(
  context: Context,
  baseIntent: Intent,
  title: String
): Intent {
  val pm = context.packageManager

  val cameraIntents =
    CAMERA_CANDIDATES.map { Intent(baseIntent).setPackage(it) }
      .filter { pm.queryIntentActivities(it, 0).isNotEmpty() }
      .toTypedArray()

  return if (cameraIntents.isEmpty()) {
    baseIntent
  } else {
    Intent
      .createChooser(baseIntent, title)
      .putExtra(Intent.EXTRA_INITIAL_INTENTS, cameraIntents)
  }
}
```

(from CamChooser/src/main/java/com/commonsware/android/r/camchooser/CameraIntent.kt)

The `enhanceCameraIntent()` function will sift through the application IDs from `CAMERA_CANDIDATES` and see if any appear to exist and support some `Intent` action (e.g., `ACTION_IMAGE_CAPTURE`). If there are some, they are attached to `Intent.createChooser()` via `EXTRA_INITIAL_INTENTS`. You can then use the returned `Intent` with `startActivityForResult()`. The result is:

- If the user only has a pre-installed camera app available, that app is launched directly
- If the user only has a matching third-party camera app installed, that app is launched directly
- If the user has both, a chooser appears
- If the user has none, you get an `ActivityNotFoundException`

So, you still need to worry about `ActivityNotFoundException`, but that was always the case with these `Intent` actions. The user might be running in a restricted profile and lack access to any camera apps, for example.

This sample only shows one candidate camera app, that being Open Camera. Enterprising developers might create a broader list of candidate apps that could be detected and used. The overall `CamChooser` sample app implements a testbed, to allow you to see whether these media capture `Intent` actions appear to work properly for your selected camera app. The checks are rudimentary, mostly confirming that we did not crash and did get the expected result (e.g., `Bitmap` returned to us, photo/video stored in the `EXTRA_OUTPUT` location).

Maps V1 Removed

If your app is very old, it is possible that you are still trying to limp along with the *original* Google Maps on Android implementation, sometimes referred to as "Maps v1". If you are using classes like `com.google.android.maps.MapView` and have a `<uses-library android:name="com.google.android.maps" />` manifest entry, you are using Maps v1.

And you need to stop. Maps v1 has been deprecated for quite some time, it stopped working in Android 10, and in Android 11, it is simply gone.

If you have `<uses-library android:name="com.google.android.maps" android:required="false" />`, and you are checking at runtime for the existence of `com.google.android.maps.MapView` (e.g.,

`Class.forName("com.google.android.maps.MapView")`), your app should not find that class, and you should go through whatever sort of "graceful degradation" code path that you have set up for such devices.

Stuff That Might Interest You

Then, we have some items that will not break your app but represent features that you might want to opt into, for Android 11 devices.

Wireless Debugging

Android has offered `adb` access over a network connection for several years, though the support on devices has been somewhat "hit or miss", and it usually required short-term `adb` access via a USB cable. This feature has been overhauled in Android 11 and is now a first-class option, with improved security and (presumably) no USB cable requirement. However, it does take several steps to set up.

First, if you do not already have `adb` in your `PATH`, this would be a fine time to add it!

Next, you need to find the “Wireless Debugging” option in the Developer Options screen in the Settings app:

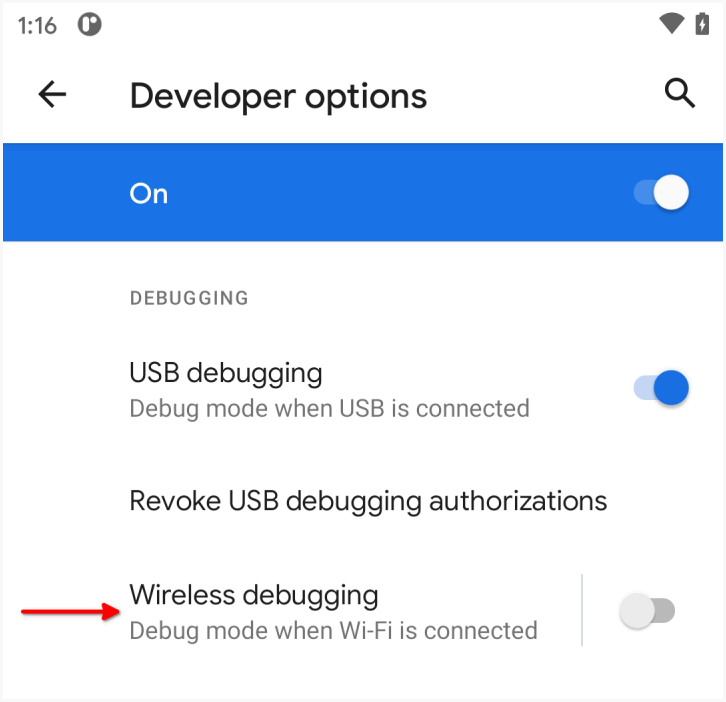


Figure 39: Developer Options, Wireless Debugging Option Highlighted

Tap on the switch to turn it on. This will bring up a dialog confirming that you want to enable this option:

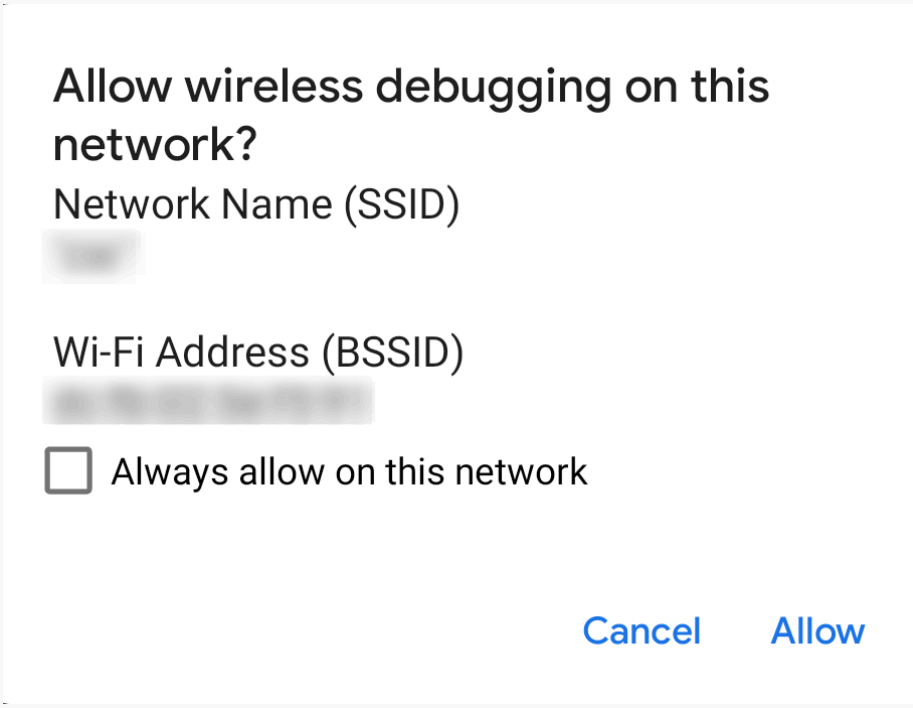


Figure 40: Wireless Debugging Network Confirmation Dialog

After accepting the dialog, tap on the “Wireless Debugging” option itself (not the switch). This brings up the dedicated wireless debugging screen:

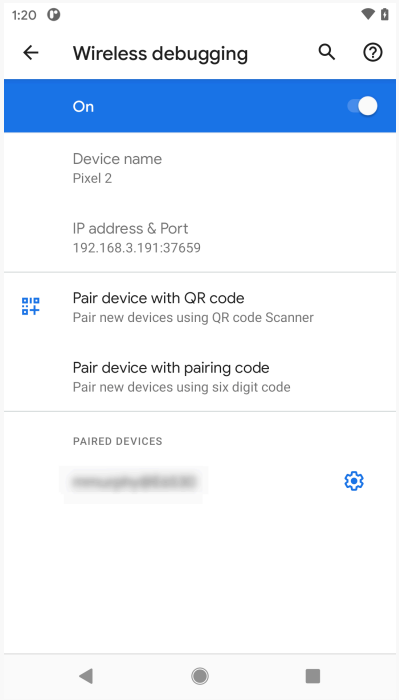


Figure 41: Developer Options, Wireless Debugging Screen

There, tap on "Pair device with pairing code", to bring up a dialog with a pairing code and an IP address and port number:

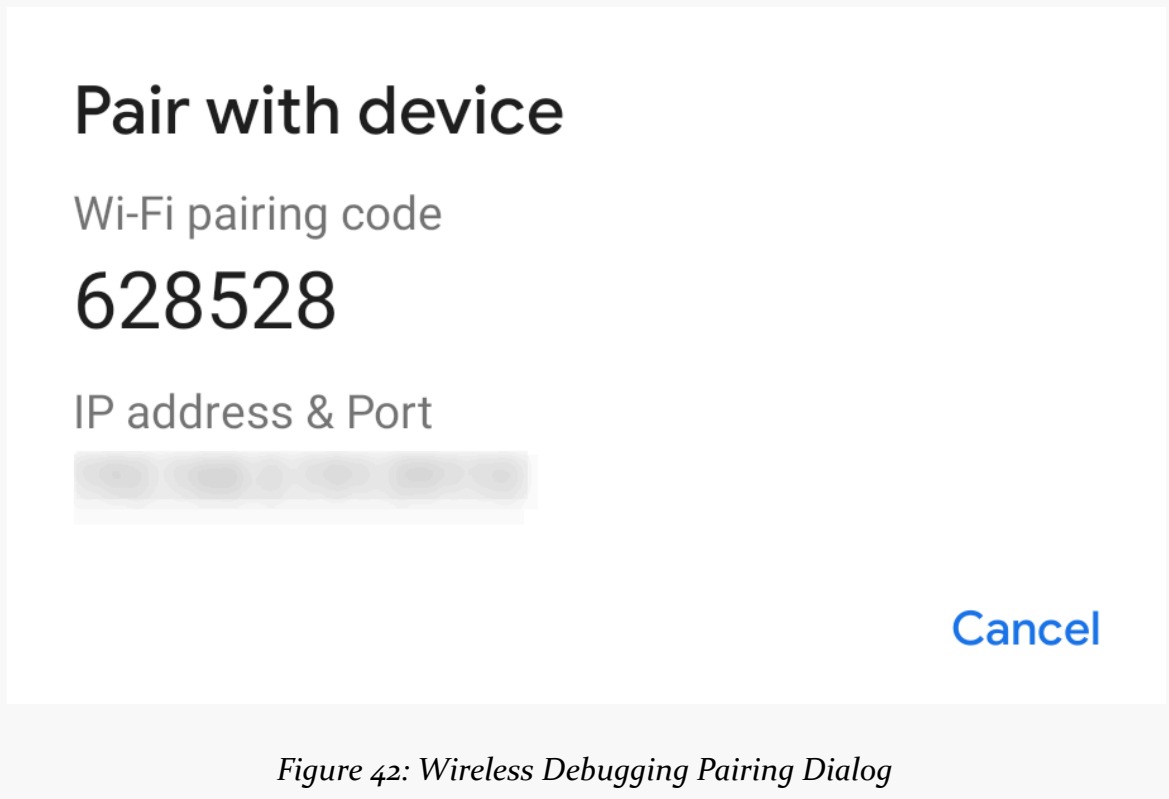


Figure 42: Wireless Debugging Pairing Dialog

At the command line on your development machine, run `adb pair ...:...`, where the `...:...` is the IP address and port combination shown *in the pairing dialog*. You will then be asked to enter the pairing code shown on the dialog. The `adb pair` command will tell you if this succeeds or fails.

If it succeeds, and you are running macOS, you should be done. If you are using Windows or Linux, you then need to run `adb connect ...:...`, where this `...:...` is the IP address and port combination shown *in the Wireless Debugging screen*. In particular, the port number will be different than the one that you used for the pairing operation.

If that command succeeds, then you will be set up for wireless debugging, both from `adb` (e.g., `adb logcat`) and from Android Studio. Use `adb disconnect ...:...`, using the same IP address/port combination that you used for `adb connect`, to disconnect from the device from a debugging standpoint.

Night Mode Tweaks

Android 11 allows the user to schedule when to enable and disable the "dark theme" mode:

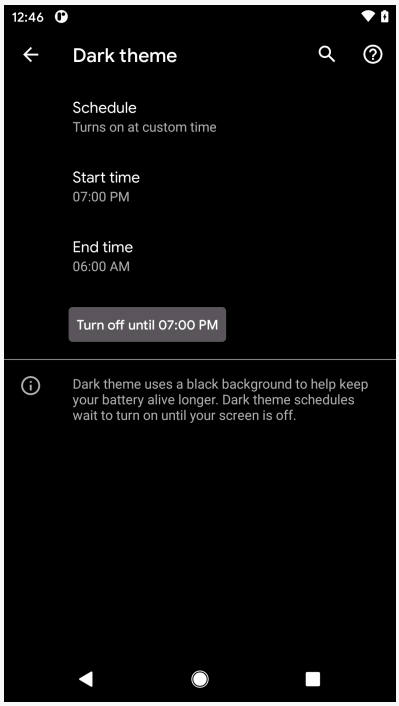


Figure 43: Dark Theme in Settings, Showing Scheduling Options

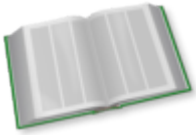
On its own, this does not affect apps. However, it does increase the likelihood that your users will be using the dark theme option, which puts increasing pressure on you to support it.

However, while Android has had support for what the SDK calls "night mode" for years, there was never a way to determine whether or not we were in night mode programmatically. If you needed that, you had to have your own boolean resources in `res/values/` and `res/values-night/` to distinguish between the cases.

Android 11, though, gives us `isNightModeActive()` on `Configuration`. You get a `Configuration` from `Resources`, which you can get from any `Context` (such as your `Activity`):

```
val mode: Boolean = resources.configuration.isNightModeActive
```

Since this is only available on Android 11 and higher, most likely you will wind up continuing to use the boolean-resource trick for the time being, until such time as you can raise your `minSdKVersion` past 29.



You can learn more about theme support for night mode in Android 10 in the "Dark Mode" chapter of *Elements of Android Q*!

Shared Datasets

In principle, Android 11 allows multiple apps to share large blobs of data. Cited examples include machine learning datasets and media for playback. Through `BlobStoreManager`, your app can contribute such blobs to the device and indicate the level of access — such as `allowSameSignatureAccess()` to allow a set of apps in a suite to all access the blobs.

In practice, this feature is very poorly documented.

Per-Connection SQL

When working with SQLite, you may want some common setup for your database connections. There may be certain `PRAGMA` statements that you want to execute, for example. However, `SQLiteDatabase` can elect to disconnect and reconnect to SQLite. So, even if you try executing those statements as soon as you open a database, you may find later that you have a connection on which you did not execute those statements.

To help with this, Android 11 offers `execPerConnectionSQL()`. The syntax is the same as `execSQL()`, taking the SQL statement and an optional array of positional parameter values. And, like `execSQL()`, your SQL is executed immediately. However, it is also cached by the `SQLiteDatabase` and re-executed on any new database connection that is created.

Note that this implies that your parameters to `execPerConnectionSQL()` will be held onto as long as the `SQLiteDatabase` object is around. If you use this, be certain not to introduce a memory leak!

Dynamic Intent Filters

One of the big limitations of the manifest is that it is declared at compile time. There are very few options for changing what is in the manifest at runtime, beyond `setComponentEnabledSetting()` on `PackageManager`.

Android 11 gives us another option: we can dynamically declare MIME types to be supported by an `<intent-filter>`. The stated reason is to support:

> virtualization apps (such as virtual machines and remote desktops) because they have no way of knowing exactly what software the user will install inside them

That is a rather esoteric scenario, and it is likely that developers will find other uses for this.

There are two steps for implementing this.

First, in one or more `<intent-filter>` elements, you can use `android:mimeGroup`, instead of `android:mimeType`, in a `<data>` element:

```
    <activity
      android:name=".MainActivity"
      android:launchMode="singleTop">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeGroup="testMimeGroup" />
        <data android:scheme="content" />
      </intent-filter>
    </activity>
```

(from DynFilter/src/main/AndroidManifest.xml)

Here, we have an `<activity>` with two `<intent-filter>` elements. The second one uses `android:mimeGroup="testMimeGroup"` in a `<data>` element as part of declaring an `ACTION_VIEW` handler.

The second step is to define at runtime what MIME types belong to the group, using

`setMimeGroup()` on `PackageManager`:

```
private fun updateMimeGroup(prefs: SharedPreferences) {
  val types = prefs.getStringSet(mimeTypesKey, emptySet()).orEmpty()

  packageManager.setMimeGroup("testMimeGroup", types)
}
```

(from DynFilter/src/main/java/com/commonsware/android/r/dynfilter/MainActivity.kt)

This code, pulled from the `DynFilter` sample module in the book's sample project, retrieves the value of a `MultiSelectListPreference` and uses that for the MIME types to pass to `setMimeGroup()`. `setMimeGroup()` simply takes the name of the MIME group (that you used in the `android:mimeGroup` attribute) and a `Set` of strings representing the MIME types.

When you call `setMimeGroup()`, Android will update its metadata for your app and cause your `<intent-filter>` to be valid for all of the requested MIME types. Note, though, that apps that look up matching activities via methods like `queryIntentActivities()` may not react to the change right away, if they cache results from before your `setMimeGroup()` call. However, standard system options, such as the `Intent` chooser, will handle the revised set of MIME types fairly quickly.

ACTION_CLEAR_APP_CACHE

There is a new `Intent` action, found on `StorageManager`: `ACTION_CLEAR_APP_CACHE`. Simply put, launching it with `startActivityForResult()` pops up a dialog to allow the user to clear "app external cache directories":

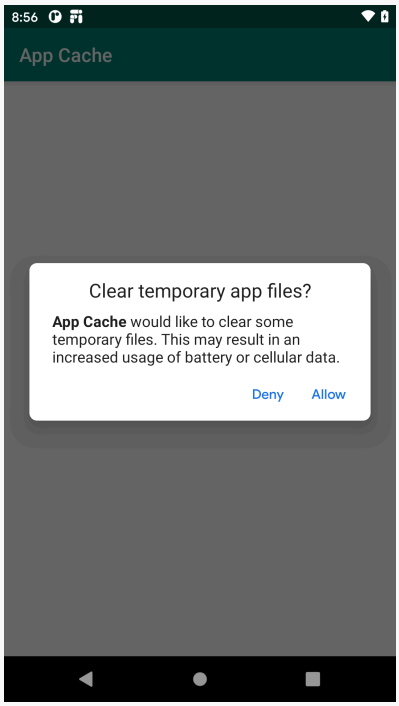


Figure 44: AppCache Sample App, Showing System Clear-Cache Dialog

Presumably, "app external cache directories" refers to `getExternalCacheDirs()` for all apps.

If you get `RESULT_OK` in `onActivityResult()`, then the user accepted the dialog and cleared the caches. If you get `RESULT_CANCELED`, the user declined your offer.

However, note that you need to hold `MANAGE_EXTERNAL_STORAGE`, which is the scary new permission covered earlier in the book.