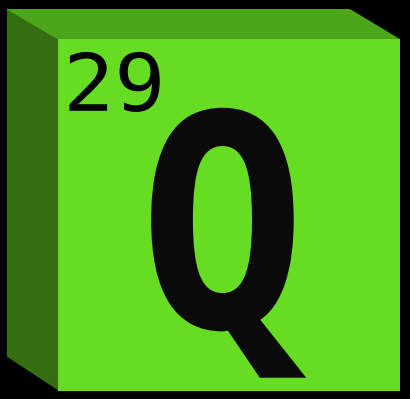
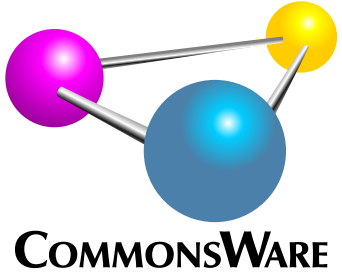
FINAL Version

Elements of Android Q



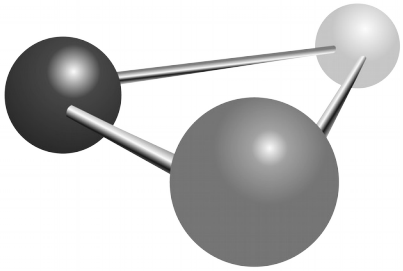
Mark L. Murphy



CommonsWare

Elements of Android Q

by Mark L. Murphy



Elements of Android Q
by Mark L. Murphy

Printed in the United States of America.

Printing History:
November 2019: FINAL

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 2019's Android 10 (Q) 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 10 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 10 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 update replaces many of the "Q" references with "10", and makes other changes to reflect the fact that Android 10 is now shipping. This update also fixes various bugs and adds a few bits of late-breaking news.

Note that this book will not receive further updates, given that Android 10's SDK is final and that Android 10 is shipping.

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 2023*. 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 Q.

The author would also like to thank:

- John De Lancie
- The late Desmond Llewelyn
- The occasionally-late John Cleese

The Death of External Storage

When Q Beta 1 was released, the biggest change for developers in Android 10 — by far — was what Google calls "scoped storage". In a nutshell, your ability to work with files and the filesystem was substantially curtailed. As a result, you had to adapt your app within a few months, to be ready by the time Android 10 shipped.

Everything will be affected in Android R, but there are steps that you can take to opt out of the changes for Android 10, at least until you are ready. And apps with a `targetSdkVersion` of 28 or lower will be unaffected... but eventually you are going to need to raise that level, at least if you plan on shipping your app through the Play Store.

Hence, if your app requests the `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` permissions, you are going to want to start adapting your app to the changes. One of the problems from the Q Beta 1 announcement was the short timeframe that we had for adapting; you do not want to be stuck with a similar short timeframe in 2020.

Introducing the Filter

In Android 1.0 through 9.0, external storage was relatively simple. All apps could access it with permission, and starting with Android 4.4 apps could access parts of it without permission (e.g., `getExternalFilesDir()` on `Context`). What the user saw and what all the apps saw were the same.

Scoped storage — when your app has to start working with it — changes this completely.

What Your App Sees

Your app can work with the external and removable storage location roots supplied by `Context`, just as it did in previous releases. So, `getExternalFilesDir()`, `getExternalCacheDir()`, and others work as they have.

Everything else, such as `Environment.getExternalStorageDirectory()` and `Environment.getExternalStoragePublicDirectory()`, is inaccessible. You can neither read nor write. In fact, those `Environment` methods are now deprecated — even though they will still return the correct values, those values are useless, as you cannot use those locations.

A side effect of this is that you cannot see, let alone modify, the files created by other apps on external storage.

What Other Apps See

Other apps are limited in the same way yours is. You cannot see those apps' files on external storage, and they cannot see yours, when using normal filesystem APIs.

Pre-installed apps from the device manufacturer represent a notable exception. Usually device manufacturers have ways of granting pre-installed apps more rights.

What the User Sees

Technically, there is no impact on the user. External storage can be seen using a desktop OS and a USB cable as before.

From a practical standpoint, the user will see fewer files in traditional locations, like `Documents/`, as fewer apps can write there.

And the user's on-device ability to see what is on external storage will be limited by the app that is used.

Controlling the Behavior

Fortunately, for Android 10 at least, your app has control over whether it has traditional ("legacy") external storage access or has "filtered" access.

Opting Out… For Now

To stick with legacy external storage, even on Android 10 devices, add `android:requestLegacyExternalStorage="true"` to your `<application>` element in your manifest.

With that in place, everything will work as it did in Android 4.4 through 9.0.

Opting In

Conversely, `android:requestLegacyExternalStorage="false"` opts into the "filtered" behavior. This works regardless of `targetSdkVersion`, so even if your `targetSdkVersion` is 28 or older, you can see how your app behaves when it is using scoped storage.

If you want, you can have `android:requestLegacyExternalStorage` be controlled by a `bool` resource value. The `StorageExplorer` sample module in the book's sample project does this:

```
<application
  android:name=".KoinApp"
  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"
  tools:ignore="GoogleAppIndexingWarning"
  android:requestLegacyExternalStorage="@bool/useLegacy">
```

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

The module has three flavor dimensions. One is called `legacy`, and it has two flavors: `legacy` and `normal`. Those drive the configuration of the `useLegacy` resource, via `resValue`:

```
        normal {
            dimension "legacy"
            applicationIdSuffix ".normal"
            resValue "bool", "useLegacy", "false"
        }

        legacy {
            dimension "legacy"
```

```
            applicationIdSuffix ".legacy"
            resValue "bool", "useLegacy", "true"
        }
```

(from StorageExplorer/build.gradle)

The result: a `legacy` build opts into the legacy external storage behavior, while a `normal` opts into "the new normal" filtered external storage.

Default Conditions

If your `targetSdkVersion` is 28 or lower, you will have legacy external storage behavior by default, as if you have opted out via `android:requestLegacyExternalStorage="true"`.

However, once you set your `targetSdkVersion` to `29`, **you will have filtered external storage by default**.

It is best if you add `android:requestLegacyExternalStorage` yourself and declare, positively, what scoped storage behavior you want to have for when your app runs on Android 10 devices.

To check whether you have scoped storage or not, you can call `isExternalStorageLegacy()` on `Environment`:

```
    val msg =
      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q && Environment.isExternalStorageLegacy())
        "This app has legacy external storage"
      else "This app has Q-normal external storage"
```

(from StorageExplorer/src/main/java/com/commonsware/android/storage/MainActivity.kt)

Note, though, that this will only return a valid value if you have a `<uses-permission>` element for `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` in the manifest. Otherwise, it always returns `false`, even if you have opted into legacy storage.

What Will Happen in Android R?

The author of this book is a time traveler, but only in the forward direction at a rate of one millisecond per millisecond.

(in other words, the author of this book is not really a time traveler)

We have no guaranteed way of knowing what 2020 and Android R will bring... but we can make some guesses.

Official Warning

In April 2019, Google announced that the then-forthcoming Q Beta 3 would allow apps to opt out of scoped storage. In that announcement, though, they wrote:

> Scoped Storage will be required in next year's major platform release for all apps, independent of target SDK level, so we recommend you add support to your app well in advance.

Hoped-For Outcome

With luck, Google will come to its senses, and only enable scoped storage on Android R for apps with a `targetSdkVersion` of `29` (or perhaps `'R'`) or higher. This would be in line with how `targetSdkVersion` normally works. And, given that the Play Store will require a `targetSdkVersion` of `29` in the second half of 2020, this policy will affect all actively-maintained apps. Yet, it would not break legacy apps that will not be updated to avoid the `Environment` methods or other ways of attempting to access inaccessible areas of external storage.

At the 2019 Android Developer Summit, Google made some statements that suggest that this indeed is their plan.

Adapting to Scoped Storage

One way or another, it seems likely that apps will be given filtered external storage eventually. While opting out is a good tactical decision, you need to plan out your work to discontinue or minimize your use of external storage. This is particularly true if you want to have files that remain on the device after your app is uninstalled.

Consuming Content

Perhaps there is content that already exists that you want to be able to read in. You might even want to modify that content yourself, if appropriate.

`ACTION_OPEN_DOCUMENT`

The best general-purpose solution is the Storage Access Framework. Specifically, for existing content, `ACTION_OPEN_DOCUMENT` is the Android equivalent of the "file open" dialogs that you might see on other platforms. The biggest difference is that you are really opening content, not files.

Offering the user the opportunity to pick a piece of content is merely a matter of a single `startActivityForResult()` call:

```
startActivityForResult(
  Intent(Intent.ACTION_OPEN_DOCUMENT).apply {
    addCategory(Intent.CATEGORY_OPENABLE); type = "*/*"
  },
  REQUEST_DOC
)
```

If there is a particular MIME type associated with your desired content, use that in place of `*/*`.

In `onActivityResult()`, if the result is for your request code (`RESULT_DOC` here) and the result code is `RESULT_OK`, then the `Intent` should have a `Uri` that points to the piece of content. You can use that directly with `ContentResolver` and methods like `openInputStream()`/`openOutputStream()`. And you can use `DocumentFile.fromSingleUri()` to create a `DocumentFile` to help you get at things like a display name or the content's length in bytes.

Inbound Actions

You might also consider setting up a suitable activity to support `ACTION_VIEW` and/or `ACTION_SEND`. Then, in `onCreate()` and `onNewIntent()`, you can get the `Uri` that the user wished your app to view or send. At that point, you can use `ContentResolver`, though perhaps not `DocumentFile`, to work with the content.

Other Options

If you need access to images, audio files, or video files, the next chapter will give you some additional options.

There are also a variety of miscellaneous ways to get `Uri` values from the user, by way of other apps, such as the clipboard and drag-and-drop.

Creating Content

Perhaps you want to create new content from scratch, rather than working with something that already exists.

Filesystem… Maybe

You are welcome to still use `getExternalFilesDir()` and similar methods on `Context`. This requires the user to navigate into `Android/data/.../files/` — where `...` is your application ID — in order to get access to those files. That is not particularly user-friendly.

`ACTION_CREATE_DOCUMENT`

`ACTION_CREATE_DOCUMENT` works much like `ACTION_OPEN_DOCUMENT`, except that you will get a `Uri` where you can create a new piece of content, rather than it pointing to an existing piece of content. This offers the user the most flexibility and is not that difficult to use for most cases.

Media

If you wish to save images, audio files, or video files for the user, `MediaStore` and related classes may be relevant options.

Sharing Content

If you want to get your content to another app directly, such as via `ACTION_VIEW` or `ACTION_SEND`, your only options in a world of scoped storage are:

- `FileProvider` (or equivalent `ContentProvider` implementations)
- `MediaStore`

But I *Need* a `File`!!!

Not everything can work with a `Uri`:

- Some libraries insist on files, possibly for the ability to randomly read (or perhaps write) to locations in the file, or to be able to start over reading the file from the beginning

- Some framework classes, like those for SQLite, can only work with files
- The NDK has no direct ability to work with `Uri` values
- And so on

Unfortunately, with the external storage restrictions placed on external storage, your options are very limited here.

Option #1: See if the API Supports File-Like Stuff

If the API you are using supports `InputStream` or `FileDescriptor`, you can use those with a `Uri` pointing to content... probably. Not all content `Uri` values necessarily support `FileDescriptor`. You can get an `InputStream` or `FileDescriptor` on your content via `ContentResolver`.

Similarly, some NDK code can work with file descriptors.

Option #2: Ask User to Put in App-Specific Location

You can ask the user to place the file in your app-specific directories on external or removable storage. If you are using methods like `getExternalFilesDir()` on `Context`, you would ask the users to put the files in locations inside `Android/data/.../` (where `...` is your application ID). Note, though, that these directories may not exist initially — be sure to create the directory first before expecting the user to use it.

As noted above, this will be aggravating for the user. Partly, that is because the directory structure is not very user-friendly, particularly given the long list of application IDs on many devices. It also may make it more difficult for the user to also use this file with other apps.

Option #3: Copy Stream to Local File

Otherwise, if you get a `Uri` from something like the Storage Access Framework, you are left with the unappetizing option of copying that content to some file that you control (e.g., on internal storage), then using that file.

On the plus side, you control your copy of the content and can manipulate it however you wish.

However, there are costs:

- The copy may take a while, for larger files
- You use additional storage space for the copy
- If you modify the copy, the only way that the user gets those modifications is through your app or if you copy the data back out to some `Uri`-identified content
- Depending on your use case for file, you might not know when a good time is to remove your copy (e.g., you use `FileProvider` to give access to some other app)

Other Problems To Consider

We will be running into all sorts of problems as a result of scoped storage. Some we can identify now, while others will become apparent over the coming months as we start to grapple with the changes.

Here are some possible problems that you will need to consider with your app.

Advertising Support for Files in the Manifest

With scoped storage, there will be files on the external storage filesystem that another app can access that your app cannot. The other app might try using a `file:Uri` with some implicit `Intent`, such as `ACTION_VIEW`. While the `file:` scheme is banned on Android 7.0+, that is a soft ban implemented by `StrictMode`, and there are ways for apps to get around that. Or, the app may be rather old, pre-dating the ban.

However, it is *very* unlikely that your app will be able to work with such a `file: Uri`, as you have virtually no access to external storage that might be accessible to another app.

As such, if you have an `<intent-filter>` with a `<data>` element for `android:scheme="file"`, you may receive `Uri` values that you cannot use.

Consider moving that `<intent-filter>` to an `<activity-alias>`, where you use a `boolean` version-dependent resource to conditionally enable that `<activity-alias>` on Android 9.0 and older. Then, you will not accept `file: Uri` values on Android 10 and newer devices. The `ConditionalFile` sample module in the book's sample project demonstrates this technique.

We have a boolean resource named `supportFileScheme`. This is set to `true` in `res/`

`values/bools.xml` and `false` in `res/values-v29/bools.xml`. So, `supportFileScheme` will be `false` for Android 10 and higher, `true` otherwise.

In our manifest, we have one `<activity>` element, for the typical `MainActivity` class. It has two `<intent-filter>` elements: the standard launcher, and one advertising support for `ACTION_VIEW` for `text/plain` content:

```
    <activity android:name=".MainActivity">
      <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" />
        <category android:name="android.intent.category.BROWSABLE" />

        <data android:scheme="https" />
        <data android:scheme="content" />
        <data android:mimeType="text/plain" />
      </intent-filter>
    </activity>
```

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

Notably, this advertises support for the `https` and `content` schemes, both of which are supported on Android 10 as well as older versions of Android. On older versions of Android, you could have `file` in here as well, but we want to avoid that on Android 10. Unfortunately, neither `<data>` nor `<intent-filter>` have an `android:enabled` option that we can use.

So, we split the `file` support out to an `<activity-alias>`:

```
    <activity-alias
      android:name=".FileAlias"
      android:enabled="@bool/supportFileScheme"
      android:targetActivity=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.VIEW" />

        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />

        <data android:scheme="file" />
```

```
        <data android:mimeType="text/plain" />
      </intent-filter>

    </activity-alias>
```

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

This alias points to `MainActivity`, so the effect is that it adds another `<intent-filter>` to `MainActivity`. That `<intent-filter>` is a clone of the `ACTION_VIEW` one from `MainActivity`, except that the scheme list is now just `file`. And, on the `<activity-alias>` itself, we have `android:enabled="@bool/supportFileScheme"`, so this alias will only be enabled on Android 9.0 and older.

The activity itself just shows a `Toast` with the string representation of the `Intent` used to start the activity.

This will give us what we want: `ACTION_VIEW` support for `file` only for Android 9.0 and older, with support for `https` and `content` for all Android versions.

Since the app does not actually use the `Uri` values supplied to it, you can test this behavior using simple `adb` commands. This one will pop up the `Toast` on all Android versions, as it uses a `Uri` with a `content` scheme:

```
adb shell am start -t text/plain -d content://respect.mah.authoritah/whatever
```

This one, though, will not match on Android 10, since it uses the `file` scheme:

```
adb shell am start -t text/plain -d file:///storage/emulated/0/whatever.txt
```

Assuming Valid `Uri` Values from `ACTION_SEND`

Unfortunately, the way that `ACTION_SEND` works does not allow you to filter incoming requests by scheme. As a result, you may get `EXTRA_STREAM` values with a `file:` `Uri` that you cannot access.

Ordinarily, you might just allow those errors to bring up some generic “oops” dialog, snackbar, etc. In this case, you should consider adding a more specific error message, indicating that the app that was used to send content to you is old and needs to be updated.

Assuming Valid `Uri` Values from the Clipboard

Similarly, you can get a `Uri` from the clipboard or via drag-and-drop, where you cannot filter by scheme. Add custom error messages here as well, indicating that the source of the `Uri` is old and needs to be updated.

Assuming Content is Seekable

Methods like `mark()` and `reset()` on `InputStream` may or may not work on a stream obtained for content identified by a `Uri`. Those methods usually work if the stream is backed directly by a file on the filesystem. They usually will not work for a stream that requires the `ContentProvider` to process the data, such as decrypting an encrypted file.

As a result, a `Uri` from things like the Storage Access Framework may or may not work with code that relies on the ability to rewind the stream, such as some media libraries.

If you in your own code rely on `mark()` and `reset()`, try to switch to some sort of buffering strategy, so your "rewind" operations work on data that you already read and do not assume that you can rewind the stream itself.

Related Deprecations That Might Affect You

`StorageVolume.createAccessIntent()` is deprecated. More importantly, it will fail fast. This was used to request access to one of the `Environment` public directories, as an alternative to needing `READ_EXTERNAL_STORAGE` and therefore getting access to the entire external storage area. It would appear that the scoped storage feature not only replaces this but is incompatible with this. So, if you are using `createAccessIntent()`, you will need to add code to take a new approach on Android 10 and higher devices.

Using MediaStore

One of the Google-recommended alternatives to working directly with external storage is to use MediaStore. MediaStore, unfortunately, has never had particularly good documentation. And, some aspects of using MediaStore changed in Android 10.

For general-purpose apps, the Storage Access Framework is a better solution for storing content. However, if your app has a particular focus on audio, video, or image media, then MediaStore is well worth consideration.

What Not To Do

Due to the shortage of documentation, proper use of MediaStore has always been a mess, rife with anti-patterns.

The biggest anti-pattern involves the use of the DATA column. Too many developers try using query() on a ContentResolver, given a MediaStore Uri, to get the DATA column. Those developers then treat the result as a filesystem path to access the media.

However:

- There is no requirement that the DATA column have a value
- There is no requirement that the DATA column have a filesystem path
- There is no requirement that the filesystem path in the DATA column be a file that you can access, even with READ_EXTERNAL_STORAGE

And, on Android 10, the MediaStore specifically will redact the DATA column from any query results.

To get a `Uri` that you can use with `ContentResolver` for things like `openInputStream()`:

- Query for the `_ID` column
- Use `ContentUris.withAppendedId()` to assemble a `MediaStore` `Uri` from the base `Uri` that you used in the `query()` and that ID value returned by the `query()`

`MediaStore` and Permissions

No permissions are required to work with the `MediaStore`, in terms of writing your own content. According to the documentation, `READ_EXTERNAL_STORAGE` is required, though, for consuming the content added to the `MediaStore` by others.

And, the documentation suggests that in the future, if you try to *modify* another app's content, the exception that will be raised will be a subclass of `RecoverableSecurityException`. This contains, among other things, a `RemoteAction` that can be used to present an option to the user for recovering from the problem. In this case, presumably it will display some sort of system dialog to have the user grant rights for your app to modify that content.

How to Consume Media

First, let's look at how you can get access to media that has been indexed by the `MediaStore`.

In particular, we will examine the `ConferenceVideos` sample module in the book's sample project. This app has a list of videos of presentations delivered by the book's author at various conferences. The app will see if these videos are already downloaded and indexed by `MediaStore`. For those that are not, the user will be able to download them, with the app handing the videos over to `MediaStore`. For the downloaded videos, the user can request to play the video using some existing video player app on the device.

Querying `MediaStore`

Consuming media centers around the media's `Uri`. You have two main approaches for getting such a `Uri`:

1. Ask some other app to help the user choose a piece of media, via `ACTION_PICK`, `ACTION_GET_CONTENT`, or `ACTION_OPEN_DOCUMENT`
2. Query the `MediaStore` yourself to find options and, where relevant, derive media `Uri` values for them

In the case of the `ConferenceVideos` app, we know what the videos are supposed to be, but we do not know if they have been downloaded or not. So, the second option is the right choice, as we can query to see which videos of our set are already known to `MediaStore` or not.

Getting the Root `Uri`

Classically, we would use some fixed values for querying `MediaStore`, typed by the sort of media we wanted:

- `MediaStore.Audio.Media.EXTERNAL_CONTENT_URI`
- `MediaStore.Image.Media.EXTERNAL_CONTENT_URI`
- `MediaStore.Video.Media.EXTERNAL_CONTENT_URI`

In principle, those should still work.

Another, albeit ill-used option, is `getContentUri()`. This is a method on classes like `MediaStore.Video.Media`. Given the name of some "volume", it returns a `Uri` for querying that volume. The Android 10 documentation steers you in the direction of `getContentUri()`, in part because there is a `getExternalVolumeNames()` method on `MediaStore` that returns a list of values that you could supply to `getContentUri()`. `Mediastore.VOLUME_EXTERNAL` should give you a representation including both external and removable storage.

`ConferenceVideos` is a fairly unsophisticated app, so it just uses `MediaStore.VOLUME_EXTERNAL` on Android 10 devices, falling back to `MediaStore.Video.Media.EXTERNAL_CONTENT_URI` on older devices:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) MediaStore.Video.Media.getContentUri(
  MediaStore.VOLUME_EXTERNAL
) else MediaStore.Video.Media.EXTERNAL_CONTENT_URI
```

(from ConferenceVideos/src/main/java/com/commonsware/android/conferencevideos/VideoRepository.kt)

`MediaStore` also has some "decorator" methods that can be used to augment a `Uri` with some parameters to opt into certain behavior. These methods take a `Uri` and return a new `Uri` based on the one that you supplied, "decorated" with additional

information. For example, `setIncludePending()` will decorate a `Uri` to indicate that you want to see pending and final results in your query results, where a "pending" result means that the content may not yet be ready (e.g., it is being downloaded).

Deriving a Media Content `Uri`

Given a root `Uri`, `query()` on `ContentResolver` is unchanged in Android 10. You provide the root `Uri`, a "projection" of columns to return, your query, any arguments to that query, and something to serve as the equivalent of a SQL `ORDER BY` clause. It returns a `Cursor` with the results.

The `ConferenceVideos` sample module has a `VideoRepository` that is responsible for communications with `MediaStore` and the CommonsWare Web server (where the videos are available for download). It has a `getLocalUri()` function that tries to derive the `Uri` for a video, given the video's filename:

```
suspend fun getLocalUri(filename: String): Uri? =
  withContext(Dispatchers.IO) {
    val resolver = context.contentResolver

    resolver.query(collection, PROJECTION, QUERY, arrayOf(filename), null)
      ?.use { cursor ->
        if (cursor.count > 0) {
          cursor.moveToFirst()
          return@withContext ContentUris.withAppendedId(
            collection,
            cursor.getLong(0)
          )
        }
      }

    null
  }
```

(from ConferenceVideos/src/main/java/com/commonsware/android/conferencevideos/VideoRepository.kt)

Here, `PROJECTION` and `QUERY` are defined as constants:

```
private val PROJECTION = arrayOf(MediaStore.Video.Media._ID)
private const val QUERY = MediaStore.Video.Media.DISPLAY_NAME + " = ?"
```

(from ConferenceVideos/src/main/java/com/commonsware/android/conferencevideos/VideoRepository.kt)

If the query returns a `Cursor` with a row, we move to that row and use `ContentUris.withAppendedId()` to assemble the `Uri` from the `collection` root `Uri`

and the `_ID` value returned from the query.

Using a Media Content `Uri`

Given the `Uri` to a piece of media, you have lots of options. The sample module just wraps it in an `ACTION_VIEW` `Intent` when the user taps on the video (represented in a row in a `RecyclerView`), to play back that video.

If you want to consume the content directly in your app, you have lots of options on `ContentResolver`, including:

- `openInputStream()`
- `openOutputStream()`
- `openFileDescriptor()`

Android 10 adds a `loadThumbnail()` method on `ContentResolver`. This will attempt to give you a `Bitmap` representation of the content identified by the `Uri`. The exact source of the data is undocumented but probably amounts to:

- Images: the image itself
- Audio: album art, if any
- Video: a frame of the video itself

How to Create Media

It used to be that creating media was a matter of writing the media to some file on external storage, then getting the `MediaStore` to index it (e.g., via `MediaScannerConnection`). However, now that writing to external storage is less of an option, we need to switch techniques. The technique that the Android 10 documentation cites is to use `insert()` on `ContentResolver`... though that approach *only* works on Android 10, as you will see.

Getting the Root `Uri`

Once again, you will need a `Uri` identifying the collection (audio, image, video) of the content that you want to create, and possibly the storage volume on which to create it. This is the same as for querying, where you can use `getContentUri()` or try one of the legacy constants (e.g., `MediaStore.Video.Media.EXTERNAL_CONTENT_URI`).

The `ConferenceVideos` sample uses the same `collection` value that we used for

querying:

```
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) MediaStore.Video.Media.getContentUri(
      MediaStore.VOLUME_EXTERNAL
    ) else MediaStore.Video.Media.EXTERNAL_CONTENT_URI
```

(from ConferenceVideos/src/main/java/com/commonsware/android/conferencevideos/VideoRepository.kt)

Crafting the Metadata

`insert()` takes a `ContentValues` that describes the content that you wish to add to the media collection identified by the `Uri` that you supply. At minimum, your `ContentValues` needs to provide `DISPLAY_NAME` (for a human-readable identifier for this content) and `MIME_TYPE` (to indicate what sort of content this is).

Android 10 adds a few additional options that you can consider.

`IS_PENDING`

One is `IS_PENDING`. As the name suggests, this indicates whether or not the content is "pending". "Pending" implies that the content is not yet ready for use — for example, the `MediaStore` database entry is created, but we are still downloading the content. `IS_PENDING` controls whether other apps, querying the `MediaStore`, will see this entry:

- `1` means that the content is pending, and other apps will not see this content by default (unless they use `setIncludePending()`)
- `0` means that the content is ready for use

Hence, the recipe is to set `IS_PENDING` to `1` initially, then flip it to `0` via an `update()` call on a `ContentResolver` when the content is ready for consumption by other apps.

Directory Hints

Android 10 also offers `RELATIVE_PATH`. This addresses a key problem with `MediaStore`: controlling where the media goes on the device.

Suppose you are downloading MP3 files representing songs. Typically, those would go in `Music/[artist]/[album]/` as a directory, substituting in suitable values for `<artist>` and `<album>`. However, if the content itself lacks the metadata (e.g., MP3 tags), `MediaStore` will not realize that the MP3 should go in this location.

`RELATIVE_PATH` allows your code to assemble that relative path and suggest it to `MediaStore`. It should be a relative path from a storage root (e.g., the root of external storage) that identifies a directory which `MediaStore` should create (if needed) and use for the content.

This is a hint, and there is no requirement for `MediaStore` to honor the request. In particular, if the top-level path segment of the relative path makes no sense (e.g., `Stuff/Goes/Here`), `MediaStore` may elect to ignore the request.

Using the Media Content `Uri`

`insert()` returns a `Uri` that represents where you can write your content. You can use that with `openOutputStream()` or `openFileDescriptor()` to write your content to the designated location. Afterwards, if you set `IS_PENDING` to `1`, you can use `update()` and that `Uri` to reset it to `0` and allow other apps to see your content.

The `VideoRepository` has a `downloadQ()` function that:

- Assembles a URL to the video
- Uses OkHttp to request that video
- Uses `insert()` to get the `Uri` to where the video should be saved, with `RELATIVE_PATH` suggesting to put the video in a `ConferenceVideos/` directory off of the stock `Movies/` directory
- Writes the video content to that location
- Uses `update()` to set `IS_PENDING` to `0`

```
private suspend fun downloadQ(filename: String): Uri =
  withContext(Dispatchers.IO) {
    val url = URL_BASE + filename
    val response = ok.newCall(Request.Builder().url(url).build()).execute()

    if (response.isSuccessful) {
      val values = ContentValues().apply {
        put(MediaStore.Video.Media.DISPLAY_NAME, filename)
        put(MediaStore.Video.Media.RELATIVE_PATH, "Movies/ConferenceVideos")
        put(MediaStore.Video.Media.MIME_TYPE, "video/mp4")
        put(MediaStore.Video.Media.IS_PENDING, 1)
      }

      val resolver = context.contentResolver
      val uri = resolver.insert(collection, values)

      uri?.let {
```

```
        resolver.openOutputStream(uri)?.use { outputStream ->
          val sink = Okio.buffer(Okio.sink(outputStream))

          response.body()?.source()?.let { sink.writeAll(it) }
          sink.close()
        }

        values.clear()
        values.put(MediaStore.Video.Media.IS_PENDING, 0)
        resolver.update(uri, values, null, null)
      } ?: throw RuntimeException("MediaStore failed for some reason")

      uri
    } else {
      throw RuntimeException("OkHttp failed for some reason")
    }
  }
```

(from ConferenceVideos/src/main/java/com/commonsware/android/conferencevideos/VideoRepository.kt)

Backwards-Compatibility Woe

Unfortunately, this does not work well on prior versions of Android:

- While what you write to the `Uri` location is saved to disk, it is not obvious where that location actually is
- The content metadata, such as the filename, does not seem to be saved in `MediaStore`

So, in the sample, `downloadQ()` is wrapped by a `download()` function that only uses `downloadQ()` on Android 10 devices:

```
suspend fun download(filename: String): Uri =
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) downloadQ(filename)
  else downloadLegacy(filename)
```

(from ConferenceVideos/src/main/java/com/commonsware/android/conferencevideos/VideoRepository.kt)

`download()` delegates to `downloadLegacy()` on older devices, using the classic approach of writing the content to external storage, then indexing the result:

```
private suspend fun downloadLegacy(filename: String): Uri =
  withContext(Dispatchers.IO) {
    val file = File(
      Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_MOVIES),
      filename
    )
```

```
    val url = URL_BASE + filename
    val response = ok.newCall(Request.Builder().url(url).build()).execute()

    if (response.isSuccessful) {
      val sink = Okio.buffer(Okio.sink(file))

      response.body()?.source()?.let { sink.writeAll(it) }
      sink.close()

      MediaScannerConnection.scanFile(
        context,
        arrayOf(file.absolutePath),
        arrayOf("video/mp4"),
        null
      )

      FileProvider.getUriForFile(context, AUTHORITY, file)
    } else {
      throw RuntimeException("OkHttp failed for some reason")
    }
  }
```

(from ConferenceVideos/src/main/java/com/commonsware/android/conferencevideos/VideoRepository.kt)

However, we no longer have a `Uri` representing the video this way, and the indexing operation is asynchronous. So, we settle for `FileProvider` to give us access in the short term. Future runs of the app should see the content in the `MediaStore` and be able to use a `MediaStore`-supplied `Uri` to work with it.

Also note that we use `MediaScannerConnection.scanFile()` in this scenario. That is not needed with `downloadQ()`, as we are directly putting the image into the `MediaStore`.

Other `MediaStore` Changes

There have been a few other `MediaStore` changes of note in Android 10.

Removed Fields

A bunch of fields from `MediaStore.MediaColumns` were removed, including `ORIENTATION`, `DURATION`, and `DATE_TAKEN`. If your app has been querying on those columns, they may no longer be available to you.

`MediaStore.Downloads`

There is a new collection, called `MediaStore.Downloads`. This appears to map to the `Downloads/` directory on external storage. However, it is largely undocumented.

Location Access Restrictions

One of the bigger privacy issues in Android is the availability of location data. Users like certain types of apps, such as navigation assistants, knowing where the user is. Users do not like arbitrary use of location data, though, and various ad networks and malicious apps have been harvesting location data inappropriately.

So, in Android 10, location access gets locked down even further than before.

Background Location Access

The change that will get the most attention is that there are new limitations on getting location data in the background. There are two main scenarios for this:

1. The app had been in the foreground, but the user switches to another app. For example, the user might be using a navigation app but then receive a phone call, at which point the device UI switches to an in-call screen. Ideally, the navigation app will continue to receive location data, despite being (temporarily) in the background... but in Android 10, this requires a bit of additional work.
2. The app is operating purely in the background (e.g., `JobScheduler` jobs) and wants to get the user's location. This requires an additional permission on Android 10.

You can learn more about `LocationManager` in the "Accessing Location-Based Services" chapter of *The Busy Coder's Guide to Android Development*!

Started from Foreground

Your app might mostly need locations in the foreground, but its UI might be moved to the background based on user interactions. You might want to keep getting the location updates while your UI is not in the foreground, so when you *do* return to the foreground, you have up-to-date location data.

The recommended pattern to make this work is to start a foreground service when your app moves to the background, where that service has `android:foregroundServiceType="location"` on its `<service>` manifest element. Then, the service can continue receiving notification updates, even though the UI is not in the foreground.

The `LocationForeground` sample module in the book's sample project illustrates this process.

The app has an activity that displays the latitude, longitude, and fix time of the latest GPS fix. It gets those from a `LocationRepository` that exposes the location updates via `LiveData`:

```
package com.commonsware.android.q.loc.fg

import android.Manifest
import android.content.Context
import android.content.pm.PackageManager
import android.location.Location
import android.location.LocationListener
import android.location.LocationManager
import android.os.Bundle
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData

class LocationRepository(private val context: Context) {
  private val _locations = MutableLiveData<Location>()
  val locations: LiveData<Location> = _locations
  private var locationsRequested = false

  init {
    initRequest()
  }

  fun initRequest() {
    if (!locationsRequested) {
      val mgr = context.getSystemService(LocationManager::class.java)

      if (context.checkSelfPermission(Manifest.permission.ACCESS_FINE_LOCATION) ==
        PackageManager.PERMISSION_GRANTED
      ) {
        locationsRequested = true
        mgr.requestLocationUpdates(
          LocationManager.GPS_PROVIDER,
```

```
          0,
          0.0f,
          object : LocationListener {
            override fun onLocationChanged(location: Location) {
              _locations.postValue(location)
            }

            override fun onStatusChanged(p0: String?, p1: Int, p2: Bundle?) {
              // unused
            }

            override fun onProviderEnabled(p0: String?) {
              // unused
            }

            override fun onProviderDisabled(p0: String?) {
              // unused
            }
          })
      }
    }
  }
}
```

(from LocationForeground/src/main/java/com/commonsware/android/q/loc/fg/LocationRepository.kt)

The activity has a viewmodel that gets the `LocationRepository` (via Koin-supplied dependency injection), and the activity gets the data to provide to data binding from that viewmodel.

This works great when the UI is in the foreground. However, we also want to ensure that `LocationRepository` can continue getting location data when the UI moves to the background.

For that, we have a `ForegroundService`. Not surprisingly, `ForegroundService` is a foreground service. It too gets the `LocationRepository` and dumps the latitude and longitude to Logcat:

```
package com.commonsware.android.q.loc.fg

import android.app.Notification
import android.app.NotificationChannel
import android.app.NotificationManager
import android.app.PendingIntent
import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.os.Build
import android.util.Log
import androidx.core.app.NotificationCompat
import androidx.lifecycle.LifecycleService
```

```
import androidx.lifecycle.Observer
import org.koin.android.ext.android.inject

private const val CHANNEL_WHATEVER = "channel_whatever"
private const val FOREGROUND_ID = 1338

class ForegroundService : LifecycleService() {
  private val repo: LocationRepository by inject()

  override fun onCreate() {
    super.onCreate()

    val mgr = getSystemService(NotificationManager::class.java)!!

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O &&
      mgr.getNotificationChannel(CHANNEL_WHATEVER) == null
    ) {
      mgr.createNotificationChannel(
        NotificationChannel(
          CHANNEL_WHATEVER,
          "Whatever",
          NotificationManager.IMPORTANCE_DEFAULT
        )
      )
    }

    startForeground(FOREGROUND_ID, buildForegroundNotification())

    repo.locations.observe(this, Observer {
      Log.d(
        "LocationForeground",
        "Latitude: ${it.latitude} Longitude: ${it.longitude}"
      )
    })
  }

  private fun buildForegroundNotification(): Notification {
    val pi = PendingIntent.getBroadcast(
      this,
      1337,
      Intent(this, StopServiceReceiver::class.java),
      0
    )
    val b = NotificationCompat.Builder(this, CHANNEL_WHATEVER)

    b.setOngoing(true)
      .setContentTitle(getString(R.string.app_name))
      .setContentText(getString(R.string.notif_text))
```

```
      .setSmallIcon(R.drawable.ic_notification)
      .setContentIntent(pi)

    return b.build()
  }
}

class StopServiceReceiver : BroadcastReceiver() {
  override fun onReceive(context: Context, intent: Intent) {
    context.stopService(Intent(context, ForegroundService::class.java))
  }
}
```

(from LocationForeground/src/main/java/com/commonsware/android/q/loc/fg/ForegroundService.kt)

Its manifest entry has the new `android:foregroundServiceType` attribute:

```
    <service
      android:name=".ForegroundService"
      android:foregroundServiceType="location" />
```

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

This attribute is used to tell Android that the service:

- Will be a foreground service, and
- Will want to continue using some class of API (in this case, `location`)

Then, to start and stop the service, we leverage `ProcessLifecycleOwner` from the Architecture Components. This lets us know when our UI comes to the foreground or moves to the background overall. In this case, we have only one activity, but most apps have more than one activity, so `ProcessLifecycleOwner` will be a better choice, as it reports the overall foreground/background status, not just for a single activity. So, our custom `Application` subclass (`KoinApp`), in addition to setting up Koin dependency injection, also registers a `DefaultLifecycleObserver` to find out about the UI state changes:

```
package com.commonsware.android.q.loc.fg

import android.app.Application
import android.content.Intent
import androidx.lifecycle.DefaultLifecycleObserver
import androidx.lifecycle.LifecycleOwner
import androidx.lifecycle.ProcessLifecycleOwner
import org.koin.android.ext.android.startKoin
import org.koin.android.ext.koin.androidContext
import org.koin.androidx.viewmodel.ext.koin.viewModel
import org.koin.dsl.module.module
```

```
class KoinApp : Application() {
  private val koinModule = module {
    single { LocationRepository(androidContext()) }
    viewModel { MainMotor(get()) }
  }

  override fun onCreate() {
    super.onCreate()

    startKoin(this, listOf(koinModule))

    ProcessLifecycleOwner.get()
      .lifecycle
      .addObserver(object : DefaultLifecycleObserver {
        override fun onStart(owner: LifecycleOwner) {
          stopService(Intent(this@KoinApp, ForegroundService::class.java))
        }

        override fun onStop(owner: LifecycleOwner) {
          startForegroundService(Intent(this@KoinApp, ForegroundService::class.java))
        }
      })
  }
}
```

(from LocationForeground/src/main/java/com/commonsware/android/q/loc/fg/KoinApp.kt)

When the UI moves to the background, we start the `ForegroundService`. When the UI moves to the foreground, we stop the `ForegroundService` (even if the service was not necessarily started, as `stopService()` does not crash or anything if you do that).

This "run the service while the UI is in the background" approach works reasonably well... except that it always starts this service, which may include some times when the user does not really want it. For the purposes of the book sample, the notification itself will stop the service if the user clicks on it. A production-grade app may need greater sophistication here.

However, for the purposes of the Android 10 problem, we are able to continue receiving location updates, even when our UI is no longer in the foreground.

Requested from Background

Doing work purely in the background is difficult on modern versions of Android, owing to all the changes related to Doze mode and similar features. Getting location data purely in the background already was a pain, as some of the preferred background options — such as `WorkManager` — do not integrate well with asynchronous APIs like we have with the location APIs.

In light of that, Android 10's changes are not a big deal.

There is a new permission, `ACCESS_BACKGROUND_LOCATION`, that you will need to request. This is a `dangerous` permission, so you not only need the `<uses-permission>` element for it in the manifest, but you need to request it at runtime. Since you are already requesting `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`, this additional permission just adds a bit of incremental code.

The user winds up with three basic options in terms of granting rights to your app:

- Unfettered access to location
- "Only while the app is in use", which translates to "only while the app has UI in the foreground"
- No access at all

The middle option will mean that your app will hold `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` (whichever you requested) but *not* `ACCESS_BACKGROUND_LOCATION`. And, if you do not hold `ACCESS_BACKGROUND_LOCATION`, you cannot obtain location data from the background, unless you originally were getting the locations in the foreground, as we saw in the preceding section.

However, this just means that your background code will need to check for this new permission and handle it the same as if the user revoked your `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` rights. Once again, this adds a bit of additional code, and it adds a new scenario (foreground location access but not background location access), but it should not cause much significant harm to your app's functionality.

EXIF Metadata Redaction

JPEG images can have EXIF metadata "tags". For example, one important one is orientation, indicating how the device was being held at the time the picture was taken. This allows image-viewing code to rotate the image as needed to properly orient it for viewing.

"Geotagged" photos represent another set of EXIF tags. A camera app can elect to include location information in photos as tags. This enables a lot of interesting features and services, such as allowing a user to browse a photo gallery via a map

instead of only chronologically.

However, geotagged photos represent semi-intentional leaks of location information. For example, if a photo was created five minutes ago and has GPS coordinates, it is reasonable to think that the device is still in the general vicinity. This is less true of a photo created five months ago... but if there are a lot of photos in a similar area, there is a decent chance that the user lives in that area and is taking photos of local events.

As a result, in Android 10, access to this information is much more restricted than it had been.

At least, in theory it is.

Individual Files

According to the documentation, EXIF data is supposed to be redacted when reading in images.

However, that does not seem to be working, at least for:

- Files that you can access on the filesystem
- Content that you can access via the Storage Access Framework

However, it *does* work, by default, for a `Uri` from the `MediaStore`. For example, consider this function:

```
fun gimmeTehTags(image: Uri) {
  context.contentResolver.openInputStream(image)?.use { src ->
    val exif = ExifInterface(src)
    val location = exif.latLong
  }
}
```

We get a valid `location` for images that have those EXIF tags if the `Uri`:

- Has the `file` scheme (e.g., a file on the filesystem that you can access)
- Is from the Storage Access Framework

By default, we get `null` if the `Uri` came from the `MediaStore`.

However, we still can get the location information, even from the `MediaStore`. This

requires two things, in theory:

1. Your app needs to hold the `ACCESS_MEDIA_LOCATION` permission (which is `dangerous` and needs to go through runtime permissions)
2. Your app needs to call `MediaStore.setRequireOriginal()`, supplying the `Uri` for which you would like the location — this method then returns a decorated `Uri` that can be used with `openInputStream()`

Then, if you use `openInputStream()` for the `setRequireOriginal()`-supplied `Uri`, you will get a stream that includes the location EXIF tags.

`MediaStore`

As part of indexing the images available on external storage, the `MediaStore` used to examine the EXIF headers and save location data. That could then be accessed by querying the `MediaStore` for `MediaStore.Images.Media.LATITUDE` and `MediaStore.Images.Media.LONGITUDE` values.

However, that is not done on Android 10, and therefore you will be unable to obtain this data. This is not dependent upon `targetSdkVersion` — `MediaStore` simply does not seem to aggregate this data.

As a result, your only option appears to be to get the `Uri` for each individual image and use `ExifInterface`, as shown above. This is far slower than obtaining the data directly from `MediaStore`, so ideally you are not attempting to get this data in bulk.

Share Targets

Android 10 is changing the "share sheet" that is displayed when an `ACTION_SEND` `Intent` is used. A new system of "share targets" is available, to help the user not only send content to your app, but send it to some specific context. To do this, Android 10 hacks in a change to dynamic shortcuts, reusing those for `ACTION_SEND` scenarios.

What Came Before

Android 10, in effect, implements a mash-up of the old "direct share" system with the old "dynamic shortcuts" system to create the new "share targets" system.

Direct Share

Many apps that support responding to `ACTION_SEND` just do that, nothing more. Android 6.0, though, added a "direct share" API that allows apps to offer not only the app as a place to share, but something specific within the app:

- A messaging client might offer sharing to a particular contact
- A note-taking app might offer "sharing" to a particular category, filing the shared content under that category for later retrieval
- A file manager might offer "sharing" to a particular directory, saving the content as a file in that specified location
- And so on

This involved:

- Creating a subclass of `ChooserTargetService`
- Registering it in the manifest with an `<intent-filter>` having `<action`

```
android:name="android.service.chooser.ChooserTargetService"/>
```

- Having your `ACTION_SEND` activity have a `<meta-data>` element pointing to the service
- Implementing `onGetChooserTargets()` to supply a list of `ChooserTarget` options based on the `IntentFilter` matched for your `ACTION_SEND` activity

The `ChooserTarget` objects would contain a title, an icon, and a `Bundle` of extras. The `Bundle` would be merged into the rest of the `ACTION_SEND` `Intent` if the user chose that particular `ChooserTarget`, where the icon and title would be shown as an additional option on the "share sheet" UI that appeared for an `ACTION_SEND` request

This worked, but it had performance implications. This approach was a "pull" mechanism, where Android would need to collect the `ChooserTarget` objects before the "share sheet" could be fully displayed. For any candidate apps that lacked running processes, this would involve forking fresh processes for those apps, so their `ChooserTargetService` subclasses could do their work. This added a lot of overhead, particularly for users who installed a bunch of `ACTION_SEND`-capable apps that advertised support for a wide range of content (e.g., `*/*` as a MIME type).

Part of the reason for Android 10's changes is to switch to a "push" mechanism, whereby apps can register possible share targets in advance. That way, the data is available immediately when an `ACTION_SEND` is requested, and the "share sheet" can be displayed more rapidly.

Dynamic Shortcuts

Android 7.1 added app shortcuts. This is a way for an app to advertise additional entry points into the app, without having multiple launcher icons. Instead, the app can teach Android additional shortcuts, which launcher apps (or other apps) could then present to the user. For example, long-pressing on a launcher icon might pop up a list of these shortcuts for a user to choose from.

Static shortcuts are the easiest ones to set up, as they just require a resource and a manifest entry. However, they are fixed options. Dynamic shortcuts, by comparison, allow your app to offer shortcuts based on user data and behavior (e.g., have a shortcut to compose an email to a particular contact). However, they are more complex to set up, requiring you to work with a `ShortcutManager` system service and define the available shortcuts from your Java or Kotlin code.

Implementing the New Approach

In effect, Android 10 allows dynamic shortcuts to serve as share targets, in lieu of using a `ChooserTargetService`.

However, it will be years before Android 10 and higher devices become dominant. Google is making available an AndroidX library that uses the new Android 10 solution on compatible devices and falls back to `ChooserTargetService` for older ones.

The `ShareTargets` sample module in the book's sample project demonstrates how this works.

Add the Dependency

The AndroidX dependency that lets Android 10 share targets work on older devices is `androidx.sharetarget`:

```
implementation "androidx.sharetarget:sharetarget:$sharetarget_version"
```

(from ShareTargets/build.gradle)

```
sharetarget_version = "1.0.0-beta02"
```

(from build.gradle)

This library provides a `ChooserTargetService` implementation called `androidx.sharetarget.ChooserTargetServiceCompat`. We get its `<service>` manifest element automatically through the manifest merger process. However, we have to add an `android.service.chooser.chooser_target_service` `<meta-data>` element to the `<activity>` element that represents our `ACTION_SEND` implementation, where the `<meta-data>` points to this supplied service:

```
    <activity android:name=".ShareActivity">
      <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="*/*" />
      </intent-filter>
      <meta-data
        android:name="android.service.chooser.chooser_target_service"
        android:value="androidx.sharetarget.ChooserTargetServiceCompat" />
    </activity>
```

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

Declare Share Targets

We then need to have an XML resource (`res/xml/`) that contains our share targets. For projects that already use static shortcuts, you can add a `<share-target>` element to your existing static shortcuts resource. Otherwise, you will need to create one.

So, our sample module has a `res/xml/share_targets.xml` resource to fulfill this requirement:

```
<?xml version="1.0" encoding="utf-8"?>
<shortcuts xmlns:android="http://schemas.android.com/apk/res/android">
  <share-target android:targetClass="com.commonsware.android.q.sharetargets.ShareActivity">
    <data android:mimeType="*/*" />
    <category android:name="com.commonsware.android.q.sharetargets.CUSTOM_SHARE_TARGET" />
  </share-target>
</shortcuts>
```

(from ShareTargets/src/main/res/xml/share_targets.xml)

Your `<share-target>` element will need an `android:targetClass` attribute, containing the fully-qualified class name of the `ACTION_SEND` activity. You also need:

- A `<data>` element identifying the MIME type pattern that you wish to receive
- A `<category>` element with a unique "category" name

In this case, the category is not an `<intent-filter>` category. Rather, it is simply an identifier that we will use to connect this `<share-target>` element with some corresponding dynamic shortcuts.

You can have as many `<share-target>` elements as needed, though a typical app will not need more than one `ACTION_SEND` activity. An individual `<share-target>` element can have as many `<data>` and `<category>` elements as needed, and in this case, more than one `<data>` element may be needed to match the desired roster of MIME types.

As with regular static shortcuts, this XML resource needs to be identified by a `android.app.shortcuts` `<meta-data>` element on the `LAUNCHER` `<activity>` element:

```
    <activity
      android:name=".MainActivity"
      android:theme="@android:style/Theme.Translucent.NoTitleBar">
      <intent-filter>
```

```
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
      <intent-filter>
        <action android:name="${applicationId}.ACTION_WHATEVER" />

        <category android:name="android.intent.category.DEFAULT" />
      </intent-filter>

      <meta-data
        android:name="android.app.shortcuts"
        android:resource="@xml/share_targets" />
    </activity>
```

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

(we will discuss that odd second `<intent-filter>` shortly)

Register Dynamic Shortcuts

We then need to register dynamic shortcuts using `ShortcutManager` (or the AndroidX `ShortcutManagerCompat`). These represent the actual entries that should show up in the "share sheet" UI. And, in particular, they need to have a category that matches a category used in a `<share-target>` element from the shortcuts XML resource.

The app's `MainActivity` will create those dynamic shortcuts in `onCreate()`, if they have not already been registered:

```
package com.commonsware.android.q.sharetargets

import android.content.Intent
import android.os.Bundle
import android.widget.Toast
import androidx.annotation.DrawableRes
import androidx.annotation.StringRes
import androidx.appcompat.app.AppCompatActivity
import androidx.core.content.pm.ShortcutInfoCompat
import androidx.core.content.pm.ShortcutManagerCompat
import androidx.core.graphics.drawable.IconCompat

data class ShareTarget(
  val id: String,
  @StringRes val shortLabelRes: Int,
  @DrawableRes val iconRes: Int
)

private val SHARE_CATEGORIES =
  setOf("com.commonsware.android.q.sharetargets.CUSTOM_SHARE_TARGET")
```

```
private val TARGETS = listOf(
  ShareTarget("one", R.string.tag_one, R.drawable.ic_looks_one_black_24dp),
  ShareTarget("two", R.string.tag_two, R.drawable.ic_looks_two_black_24dp),
  ShareTarget("five", R.string.tag_five, R.drawable.ic_looks_5_black_24dp)
)

class MainActivity : AppCompatActivity() {

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

    if (ShortcutManagerCompat.getDynamicShortcuts(this).size == 0) {
      val intent = Intent("$packageName.ACTION_WHATEVER")

      ShortcutManagerCompat.addDynamicShortcuts(this, TARGETS.map { tag ->
        ShortcutInfoCompat.Builder(this, tag.id)
          .setShortLabel(getString(tag.shortLabelRes))
          .setIcon(IconCompat.createWithResource(this, tag.iconRes))
          .setIntent(intent)
          .setLongLived(true)
          .setCategories(SHARE_CATEGORIES)
          .build()
      })

      Toast.makeText(this, "Share targets ready!", Toast.LENGTH_LONG).show()
    } else {
      Toast.makeText(this, "${intent.action} received!", Toast.LENGTH_LONG).show()
    }

    finish()
  }
}
```

(from ShareTargets/src/main/java/com/commonsware/android/q/sharetargets/MainActivity.kt)

Here, we iterate over a `TARGETS` list of `ShareTarget` objects. Those simply aggregate some metadata that we need for the dynamic shortcuts: an ID, an icon, and a label. We convert those to `ShortcutInfoCompat` objects via `ShortcutInfoCompat.Builder`. Each of those `ShortcutInfoCompat` objects also gets:

- A set of categories (`CATEGORIES`) that includes the category that we used in the `<share-target>` element
- An `Intent` for a custom action (`ACTION_WHATEVER` in the namespace of the application), and that `Intent` will be used if the user actually uses this dynamic shortcut

The User Experience

From a sharing standpoint, the user experience is about what you would expect... albeit with a side-effect that may be less than ideal for some apps.

Sharing

If an app requests an `ACTION_SEND` that matches your desired MIME type(s), your dynamic shortcuts will appear in the "share sheet":

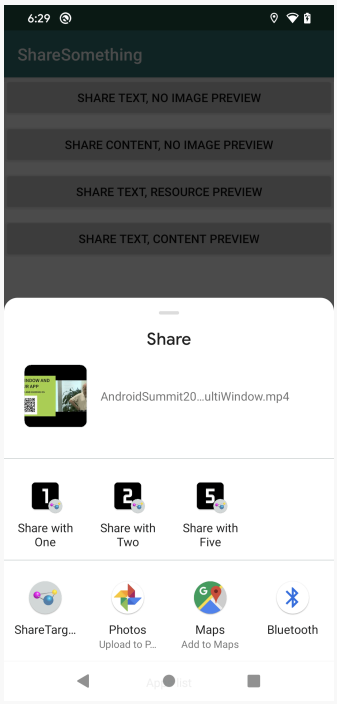


Figure 1: Android 10 "Share Sheet", Showing Custom Share Targets

It is unclear how Android 10 will decide which of your share targets to show, if you have a lot of them.

If the user chooses one of those, your `ACTION_SEND` activity will be started. Among the extras will be an `Intent.EXTRA_SHORTCUT_ID` value that is the ID that you supplied to the `ShortcutInfoCompat.Builder` for the dynamic shortcut. You can use this to look up relevant information to determine the context for the user's choice of share targets.

Shortcuts

However, your share targets *also* show up as dynamic shortcuts:

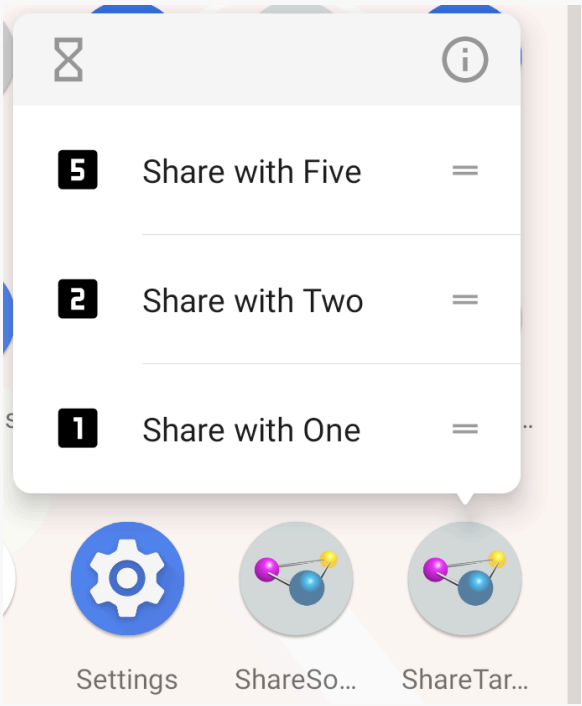


Figure 2: Pixel Launcher, Showing Share Targets as Shortcuts

This all but eliminates your app's ability to use shortcuts for anything else.

It also means that your share targets have to make sense in cases where the user is not sharing anything, because they clicked a dynamic shortcut.

Apparently, this is working as intended.

If the user taps one of the dynamic shortcuts, whatever `Intent` you gave to the `ShortcutInfoCompat.Builder` will be invoked. This `Intent` can be for whatever activity you want, not necessarily the `ACTION_SEND` activity. In the sample module, we use `ACTION_WHATEVER` for the `Intent` and have a corresponding `<intent-filter>` for it on `MainActivity`, not on `ShareActivity`.

Pre-10

If you named your shortcut XML resource `shortcuts.xml`, then on Android 6.0-9.0

devices, your share targets will show up in the "share sheet", courtesy of `androidx.sharetarget`:

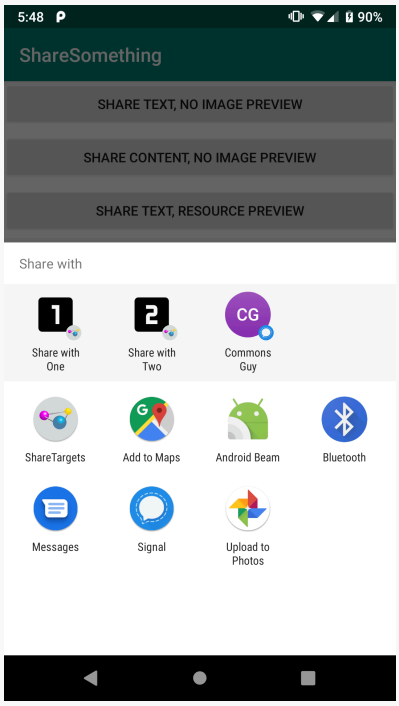


Figure 3: Android 9.0 "Share Sheet", Showing Two Custom Share Targets

And, on Android 7.1-9.0 devices, your share targets will also show up as dynamic shortcuts.

Dark Mode

Android 10 offers a system-level option to enable "dark mode". In dark mode, light UI backgrounds get flipped to dark ones. This primarily affects system UI, but apps can elect to react to this change as well, or otherwise support a dark theme for their apps.

In this chapter, we will explore Android 10's dark mode options and see how our apps might adopt a dark theme.

Turning to the Dark Side

In the early days of Android, dark themes were typical. Then, starting with Android 4.x and increasing afterwards, Google started encouraging light themes. Now, Google is back to endorsing dark themes.

Reasons

Partly, this is for the user experience. People using their devices at night can do so more easily if the UI is darker and therefore offers less glare. This is why navigation apps often switch into a dark mode at different points (e.g., when ambient light seems to be low), so drivers do not have this bright light shining at them constantly. Also, some users may have visual impairments or other conditions where such glare is a bigger problem than for other people.

Also, with some types of modern displays, black pixels consume less power.

User Actions

Users can switch to dark mode via the Settings app and the "Dark theme" option in

the Display screen:

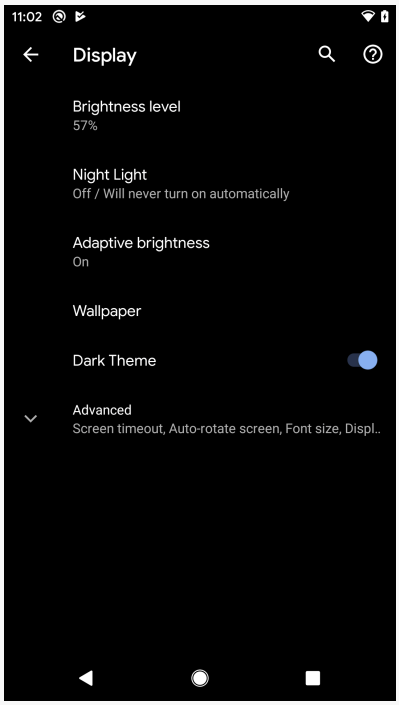


Figure 4: Dark Theme in Settings

The user can also add a tile to the notification shade to be able to rapidly toggle between normal and dark modes:

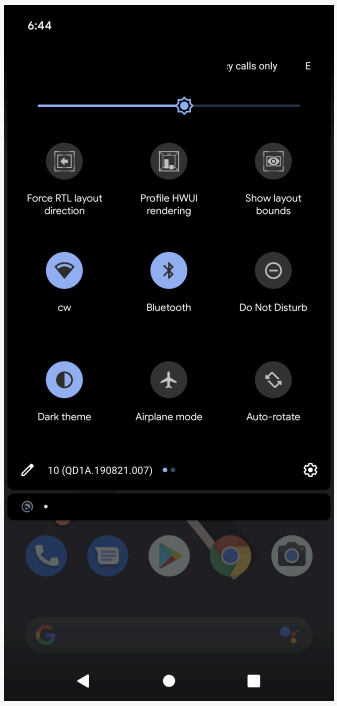


Figure 5: Dark Theme Tile

Also, according to the documentation:

> On Pixel devices, the Battery Saver mode also enables Dark theme at the same time. Other OEMs may or may not support this behavior.

And, if you use AppCompat with its `DayNight` support, you could offer an in-app toggle between light and dark themes, as we will explore later in the chapter.

The Dark-All-The-Time Solution

The simplest solution for supporting dark mode is simply to always have a dark theme. This means you have just one theme with one set of colors and artwork, to minimize the work of graphic designers. The user gets the benefits all the time, and the dark theme benefits users across Android versions (not just Android 10 users).

However, if you already have a light theme, this will require some amount of work to

revise the design.

The System Override Solution

You could try to cheat a bit and have the system create a dark theme for you on the fly. For that, add this entry to your theme resource:

```
<item name="android:forceDarkAllowed">true</item>
```

Then, on Android 10 and higher devices, the system will examine your UI and swap colors to try to make the app appear dark. It even has the smarts to determine whether an `ImageView` appears to be containing an icon (that might be converted) or a photo (that should not be converted).

So, in the default mode, you might have:

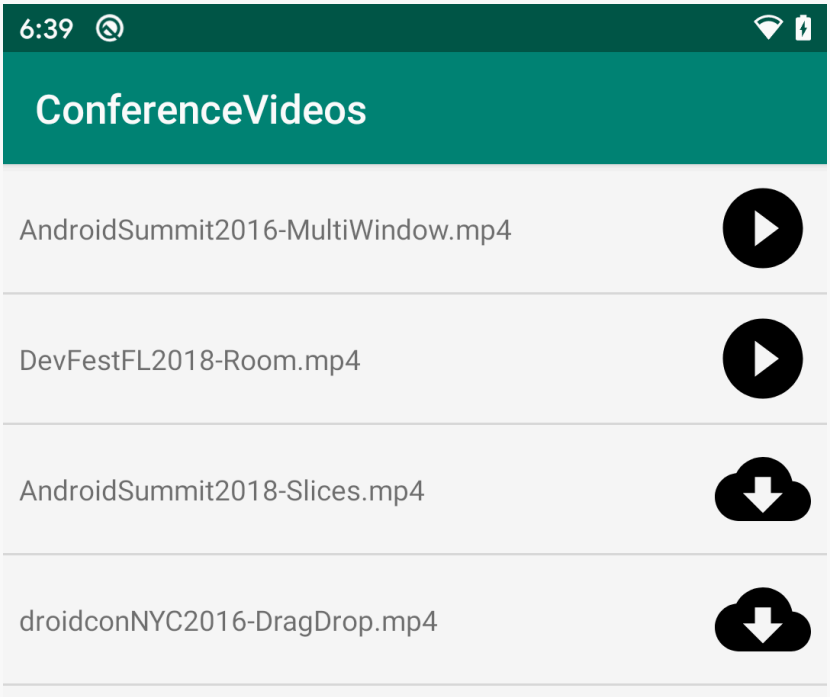


Figure 6: Sample App, in Normal Mode

...while if the user opts into the dark mode, `android:forceDarkAllowed="true"` will give the user:

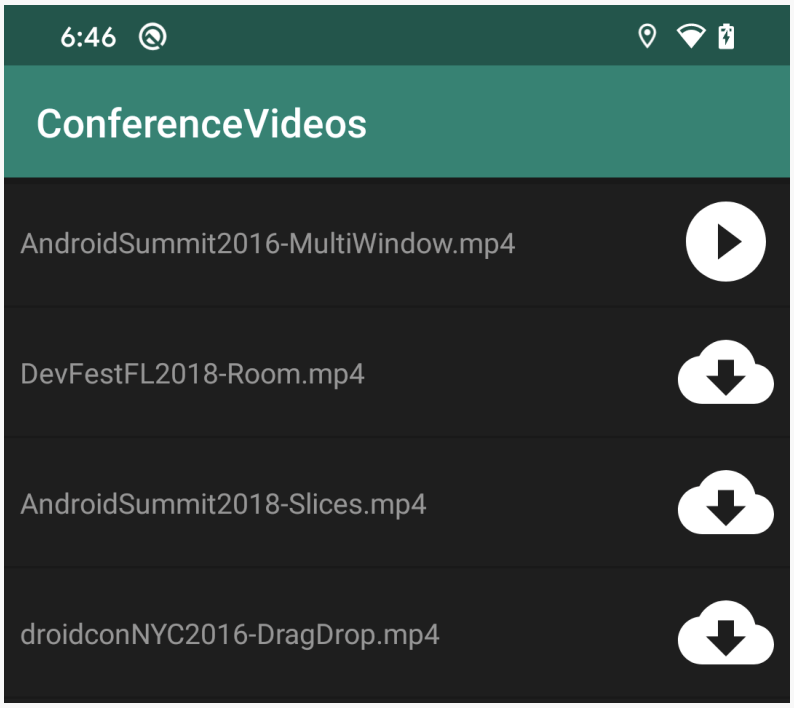


Figure 7: Sample App, in Force-Dark Mode

This is quick and easy. However:

- You do not have any control over the color substitutions, which may make your designers unhappy
- Some things may get converted by accident, requiring you to add `android:forceDarkAllowed="false"` to individual widgets to get them to be left alone
- This only works on Android 10 and higher, so you will have different behavior by OS version

The `DayNight` Solution

Google's preferred solution is for you to use a theme that adapts based upon whether the device is in dark mode or not. That way, you can have a light theme "normally" while having a dark theme in dark mode.

In particular, AppCompat supports this via its `DayNight` theme family, though for best results on Android 10 you should use `1.1.0-beta01` or newer.

The `TypeInfo` sample module uses a `DayNight` theme. This sample presents information about a bunch of MIME types, as will be discussed in an upcoming chapter.

Use a `DayNight` Theme

Switching to a `DayNight` theme, in many cases, only requires you to change the parent theme to `Theme.AppCompat.DayNight` (or to another theme that extends from `Theme.AppCompat.DayNight`):

```
<resources>

  <!-- Base application theme. -->
  <style name="AppTheme" parent="Theme.AppCompat.DayNight">
    <!-- Customize your theme here. -->
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
  </style>

</resources>
```

(from TypeInfo/src/main/res/values/styles.xml)

In dark mode, `Theme.AppCompat.DayNight` inherits from `Theme.AppCompat`, and so it has a dark base to the theme. Otherwise, `Theme.AppCompat.DayNight` inherits from `Theme.AppCompat.Light`, and so it has a light base to the theme. And, there are sub-themes, such as `Theme.AppCompat.DayNight.DarkActionBar`, that might fit your needs better.

Define `-night` Resources

Then, you can create alternative versions of colors, drawables, etc. that will be used in dark mode. These should go in resource sets with the `-night` qualifier.

For example, you might have one set of colors for normal mode in `res/values/`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="colorPrimary">#ffc107</color>
```

```
  <color name="colorPrimaryDark">#ffaa00</color>
  <color name="colorAccent">#536dfe</color>
</resources>
```

(from TypeInfo/src/main/res/values/colors.xml)

...and replacements for some of those colors in `res/values-night/`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="colorPrimary">#3F51B5</color>
  <color name="colorPrimaryDark">#303F9F</color>
  <color name="colorAccent">#FFC107</color>
</resources>
```

(from TypeInfo/src/main/res/values-night/colors.xml)

The combination of the `DayNight` theme and your custom `-night` resources will allow your app to adapt automatically as the user switches between normal and dark mode:

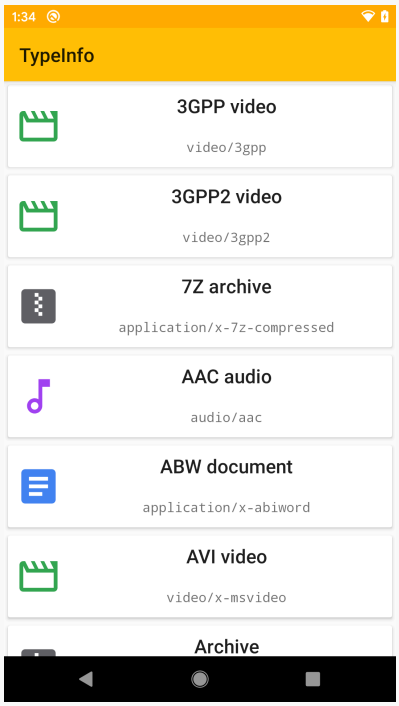


Figure 8: Sample App, in Normal Mode

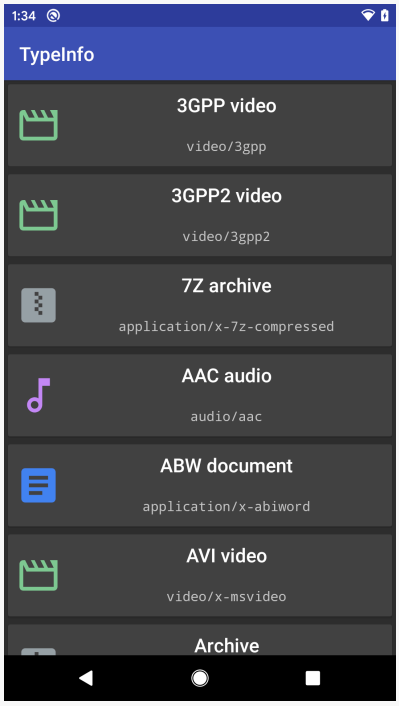


Figure 9: Sample App, in Dark Mode

Setting the Dark Mode Policy

You can teach AppCompat — specifically `AppCompatDelegate` — how you want your `DayNight` theme to behave, such as forcing it to always use dark mode.

Policy Options

There are four main options, identified by constants on `AppCompatDelegate`:

Constant	Meaning
MODE_NIGHT_NO	Use a light theme
MODE_NIGHT_YES	Use a dark theme
MODE_NIGHT_FOLLOW_SYSTEM	Use a light or dark theme based on Android 10's system status
MODE_NIGHT_AUTO_BATTERY	Use a dark theme when the device battery level is low, otherwise use a light theme

The overall default is MODE_NIGHT_FOLLOW_SYSTEM, even though this only really works on Android 10. The effect of this mode on older devices is undocumented.

Policy Locations

If you pass one of those constants to AppCompat.setDefaultNightMode(), this will update all current activities for that new policy, plus that policy will be used for future activities in your running process.

If you want to affect only a single activity, you can call getDelegate() to retrieve the AppCompatDelegate instance for your AppCompatActivity, then call setLocalNightMode() on it.

The TypeInfo sample app uses AppCompat.setDefaultNightMode(), even though there is only one activity.

Policy Persistence

setLocalNightMode() affects only that one activity instance, while AppCompat.setDefaultNightMode() affects your entire process. However, neither is persisted. You will want to establish your policy on each process invocation, such as in a custom Application subclass.

The TypeInfo sample app, since it contains only one activity, applies the policy in that activity (MainActivity). MainActivity has a checkable overflow menu to allow the user to toggle the policy. MainMotor persists that in SharedPreferences, plus loads the last-saved value on startup.

Policy Policies

If you are bothering with `DayNight`, presumably there are cases where you want light themes and cases where you want dark themes. As a result, there are three main patterns for using these policies:

1. You elect to use `MODE_NIGHT_FOLLOW_SYSTEM` or `MODE_NIGHT_AUTO_BATTERY`, putting control over the light/dark decision into the system and AppCompat implementation
2. You elect to allow the user to choose between these modes, perhaps via a preference screen (or, as in the case of the `TypeInfo` sample, a menu)
3. You elect to toggle modes yourself based on other criteria (ambient light sensor, particular times of the day, etc.)

The Design Problem

Technically, supporting `DayNight` is easy.

From a design standpoint, now you need two designs, with two color schemes, two sets of artwork, etc. Careful creation of those designs can minimize the differences, to help both the designers and the developers maintain these things over time. However, the designs are needed, to confirm that both the light and dark themes are usable (e.g., text is readable in both themes, despite changing text and background colors).

Dark Mode and Configuration Changes

When the device changes between normal and dark mode, any visible activities immediately will undergo a configuration change, for the `uiMode` configuration. Even manual changes to `DayNight` (e.g., `setDefaultNightMode()`) will recreate your activity, as if it underwent a configuration change.

If your activities already handle configuration changes (e.g., screen rotation), you should be fine. However:

- If you have activities where you skipped supporting configuration changes, you will need to fix that soon
- If you have activities where you are handling configuration changes manually, via `android:configChanges` in the manifest, consider whether you want to handle `uiMode` manually as well

Gesture Navigation

Android 10 offers users yet another option for system navigation, such as "home" and "back" actions. The particular implementation — edge swipe gestures — may cause problems for users with some Android apps.

A Tale of Three (or More) Nav Patterns

Way back in the beginning, navigation actions were handled by hardware buttons. Android 3.0 introduced the notion of a "navigation bar" for handling "home", "back", and "overview" navigation actions, leading to the classic three-button bar:

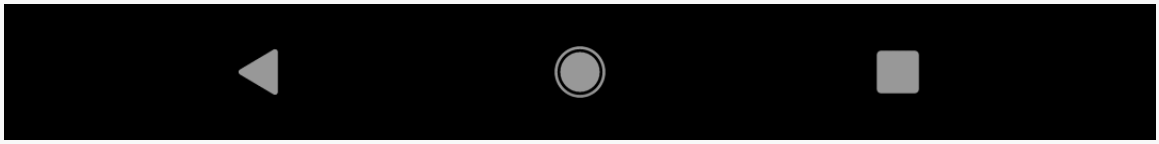


Figure 10: Three-Button Android Nav Bar

Android 9.0 added another option for users: a two-button nav, where "home" and "overview" actions were handled by gestures on a central pill affordance:

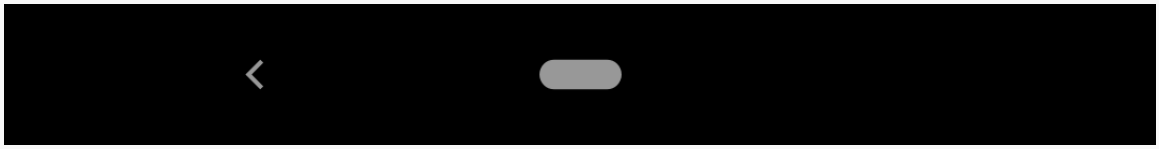


Figure 11: Button-and-Pill Android 9.0 Nav Bar

Android 10 adds a new nav option that is based on gestures:

Action	Associated Gesture
Home	swipe up from bottom screen edge
Back	swipe inward from the screen edge on left or right
Overview	swipe up from the bottom screen edge and hold

Users can choose among those three by visiting Settings > System > Gestures > "System navigation":

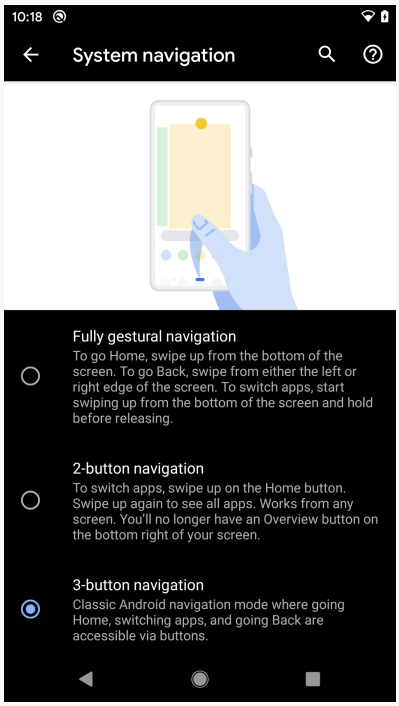


Figure 12: System Navigation Settings in Android 10

The user can choose between gesture-based nav, the Android 9.0 button-and-pill option, or the classic three-button nav option. Note, though, that not all users will have access to all of those options. Pixel 4 users, for example, cannot choose the two-button nav option.

On top of this, some device manufacturers have created their own gesture-based nav options. Device manufacturers will be allowed to continue coming up with their own schemes for this, meaning that a user might have three or four navigation

options on Android 10 devices.

Impacts on Apps

The system "steals" touch events from apps to handle these navigation gestures. If your app relies upon touch events near the edges, you may run into some problems. In particular, the user may get confused when trying to use your app, trying to apply *your* gestures and winding up with system responses. While simple taps will be passed through to your app from these system edge areas, anything else is indeterminate.

For example, suppose that you have a `ViewPager` that spans the width of the screen. Based on a subtle and invisible line of demarcation, the same gesture might either switch pages in your pager or invoke a "back" action (probably navigating out of this screen).

You may need to consider redesigning your UI to:

- Avoid expecting swipe gestures near screen edges, and
- Provide a visual distinction of where swipe gestures are valid, to help the user learn where to swipe to control your UI

Technically, there is a way that you can tell the system to ignore "back" gestures and pass those along to your app. However, from a practical standpoint, this has problems:

- The user may not know how to exit this screen and may get frustrated as a result
- This approach may not be honored by manufacturer-specific nav schemes

Avoiding the edges is a safer approach.

The OS informs your app about "window insets", to indicate areas where the system will steal your touch events. This library helps you leverage that information to adjust your UI based upon the particular device's window insets, based on device model and whether the user has enabled gesture-based nav or not.

Installing Apps Using PackageInstaller

In the beginning, to install an APK, you would use an `ACTION_VIEW` `Intent`, with a `file` `Uri` pointing to the APK. Pass that to `startActivity()`, and Android would take over from there.

This process evolved over the years, such as adding `ACTION_INSTALL_PACKAGE` in Android 4.0 and adding `content` `Uri` support in Android 7.0. A `PackageInstaller` class was added in Android 5.0, but it seemed complicated, so a lot of developers stuck with the earlier `Intent`-based solutions.

However, `ACTION_INSTALL_PACKAGE` was deprecated in API Level 29, with a request that we use `PackageInstaller` instead. While not specifically deprecated, one imagines that `ACTION_VIEW` is also frowned upon for installing apps.

`PackageInstaller` is designed for more complex scenarios, including dealing with split APKs, where a single app might require more than one APK to completely install. As a result, it has a convoluted API, to go along with the typical skimpy documentation.

So, in this chapter, we will examine how to use `PackageInstaller` to install a simple APK, for a functional equivalent to the deprecated `ACTION_INSTALL_PACKAGE`.

Note that `ACTION_UNINSTALL_PACKAGE` was also deprecated in Android 10. However, while `PackageInstaller` has a pair of `uninstall()` methods, these cannot be used by ordinary apps.

Applying `PackageInstaller`

The `AppInstaller` sample module in the book's sample project has a stub activity with an "open" action bar item. Clicking that will open the standard `ACTION_OPEN_DOCUMENT` content picker UI, for you to find an APK to install. If you select an APK, the app then uses `PackageInstaller` to install that APK, with a bit of an assist from you as the user.

Permissions

Android 6.0 debuted the `REQUEST_INSTALL_PACKAGES` permission, and Android 8.0 started enforcing it for apps using `ACTION_INSTALL_PACKAGE`. Not surprisingly, you need it for `PackageInstaller` as well. This is a `normal` permission, so you do not need to request it at runtime — just have the `<uses-permission>` element in the manifest:

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

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

Creating and Using a Session

The `AppInstaller` app uses the same sort of architecture pattern as seen in several of the other samples, where we have a `ViewModel` implementation called `MainMotor` that our UI layer uses. In this case, `MainActivity` calls an `install()` function on `MainMotor`, handing over the `Uri` that it received from the `ACTION_OPEN_DOCUMENT` request.

`MainMotor` actually is an `AndroidViewModel`, as we need two things tied to a `Context`:

- A `PackageInstaller` instance, obtained by requesting one from `PackageManager`
- A `ContentResolver` instance

```
private val installer = app.packageManager.packageInstaller
private val resolver = app.contentResolver
```

(from AppInstaller/src/main/java/com/commonsware/q/appinstaller/MainMotor.kt)

Unlike `ACTION_INSTALL_PACKAGE`, we need to do our own I/O to install APKs using `PackageInstaller`. So, `install()` in `MainMotor` turns around and calls an `installCoroutine()` function, launched from `viewModelScope`:

```
fun install(apkUri: Uri) {
  viewModelScope.launch(Dispatchers.Main) {
    installCoroutine(apkUri)
  }
}
```

(from AppInstaller/src/main/java/com/commonsware/q/appinstaller/MainMotor.kt)

`installCoroutine()`, in turn, is a `suspend` function that wraps its work in a `withContext(Dispatchers.IO)` block, to have our I/O be performed on a background thread:

```
private suspend fun installCoroutine(apkUri: Uri) =
  withContext(Dispatchers.IO) {
    resolver.openInputStream(apkUri)?.use { apkStream ->
      val length =
        DocumentFile.fromSingleUri(getApplication(), apkUri)?.length() ?: -1
      val params =
        PackageInstaller.SessionParams(PackageInstaller.SessionParams.MODE_FULL_INSTALL)
      val sessionId = installer.createSession(params)
      val session = installer.openSession(sessionId)

      session.openWrite(NAME, 0, length).use { sessionStream ->
        apkStream.copyTo(sessionStream)
        session.fsync(sessionStream)
      }

      val intent = Intent(getApplication(), InstallReceiver::class.java)
      val pi = PendingIntent.getBroadcast(
        getApplication(),
        PI_INSTALL,
        intent,
        PendingIntent.FLAG_UPDATE_CURRENT
      )

      session.commit(pi.intentSender)
      session.close()
    }
  }
```

(from AppInstaller/src/main/java/com/commonsware/q/appinstaller/MainMotor.kt)

First, we get an `InputStream` on the content identified by the `Uri` and use `DocumentFile` to find out the length of that content.

Then, we create and open a `PackageInstaller.Session`. To create a session, we call `createSession()` on `PackageManager`, providing a `PackageInstaller.SessionParams` object as a parameter. Most of the time, you will use `MODE_FULL_INSTALL` as the type of session that we want, to install an app from scratch — there is also a `MODE_INHERIT_EXISTING` to add new split APKs to an already-installed app. `createSession()` does not give us the `Session` object, though

— we get an `Int` identifier instead, and we need to call `openSession()` to get the actual `Session`.

With `ACTION_INSTALL_PACKAGE`, we provided a `Uri` that pointed to the APK to install. With `PackageInstaller`, instead, we need to provide the bytes of that APK manually. And, instead of us just passing an `InputStream` to `PackageInstaller.Session`, we have a more complex API:

- Call `openWrite()` on the `Session` to get an `OutputStream`
- Copy the bytes from our `InputStream` to that `OutputStream`
- Call `fsync()` on the `Session` to say "we're done, please ensure everything is written to disk"

The three parameters to `openWrite()` are:

- Some seemingly arbitrary "name" string
- The offset into the bytes that the `Session` should start using (typically pass `0`)
- The number of bytes that will need to be read in, or `-1` if you do not know the length

We then call `commit()` and `close()` on the `Session` to request the actual install to occur. `commit()` takes an `IntentSender` object — typically you get one of these by calling `getIntentSender()` on some `PendingIntent` that you create.

Getting the Results

Roughly speaking, there are three possible outcomes of our request:

- It succeeds
- It fails for some reason (e.g., duplicate `ContentProvider` authority conflict)
- The user needs to approve the installation

For an ordinary app, that third outcome will always happen, en route to some final success or failure state.

We find out about all of this via the `PendingIntent` that we set up. In this case, that pointed to an `InstallReceiver`, a manifest-registered `BroadcastReceiver` that will be invoked when needed:

```
package com.commonsware.q.appinstaller
```

```
import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.content.pm.PackageInstaller
import android.media.AudioManager
import android.media.ToneGenerator
import android.util.Log

private const val TAG = "AppInstaller"

class InstallReceiver : BroadcastReceiver() {
  override fun onReceive(context: Context, intent: Intent) {

    when (val status = intent.getIntExtra(PackageInstaller.EXTRA_STATUS, -1)) {
      PackageInstaller.STATUS_PENDING_USER_ACTION -> {
        val activityIntent =
          intent.getParcelableExtra<Intent>(Intent.EXTRA_INTENT)

        context.startActivity(activityIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK))
      }
      PackageInstaller.STATUS_SUCCESS ->
        ToneGenerator(AudioManager.STREAM_NOTIFICATION, 100)
          .startTone(ToneGenerator.TONE_PROP_ACK)
      else -> {
        val msg = intent.getStringExtra(PackageInstaller.EXTRA_STATUS_MESSAGE)

        Log.e(TAG, "received $status and $msg")
      }
    }
  }
}
```

(from AppInstaller/src/main/java/com/commonsware/q/appinstaller/InstallReceiver.kt)

We find out which of those scenarios occurs via the `PackageInstaller.EXTRA_STATUS` extra on the `Intent` delivered to the component. This is an `Int` value that will correspond to one of a set of `STATUS_` constants on `PackageInstaller`.

When we get the `PackageInstaller.STATUS_PENDING_USER_ACTION` status, we can get a pre-populated `Intent` from the `Intent.EXTRA_INTENT` extra on the `Intent` that we received. We can then use that with `startActivity()` to bring up system dialogs for the user to confirm that they want us to be able to install apps and they want this particular app to be installed. Note, though, that since we are calling `startActivity()` from `onReceive()` of a `BroadcastReceiver`, we need to add `FLAG_ACTIVITY_NEW_TASK` to be able to start the activity.

If we get `PackageInstaller.STATUS_SUCCESS`, then the APK was successfully installed. This app simply plays an acknowledgment tone via `ToneGenerator`, but a more sophisticated app would update its UI, display a `Notification`, or something.

Any other status code indicates some type of error condition, such as "this app is already installed with the same or higher version" (`STATUS_FAILURE_CONFLICT`) or "this app is incompatible with the device" (`STATUS_FAILURE_INCOMPATIBLE`). A human-readable status message should be in the `PackageInstaller.EXTRA_STATUS_MESSAGE` extra. This sample app just logs that information to Logcat, but a more sophisticated app might have some sort of error state in the UI, such as an error dialog.

System Notification, Maybe

In theory, the system is supposed to display a notification on Android 10 devices after the app is installed. In practice, that is not working. If someday it starts working, though, there are two `<meta-data>` elements that you can add to tailor the icon for that notification:

- `com.android.packageinstaller.notification.smallIcon`, pointing to a drawable resource representing your desired icon
- `com.android.packageinstaller.notification.color`, pointing to a color resource for your desired tint on that icon (presumably)

The `AppInstaller` app customizes the icon but leaves the color alone:

```
    <meta-data
      android:name="com.android.packageinstaller.notification.smallIcon"
      android:resource="@drawable/ic_install_notification" />
```

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

However, in the short term, you can ignore those `<meta-data>` elements, as they have no effect.

Other Changes of Note

There are lots of other changes in Android 10, 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 scoped storage certainly qualify for that.

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

Background Activity Starts Banned

Starting an activity from the background is banned on Android 10. This change affects **all apps**, not just those with a `targetSdkVersion` of `29` or higher.

Definition of "Background Start"

Your app will be considered to be starting an activity from the background if it calls `startActivity()` (or equivalent methods) when it does not have a foreground activity.

The *system* can start one of your activities from the background — after all, that is how you get to the foreground in the first place. And select system-triggered events that execute a `PendingIntent` can start an activity from the background, notably a `PendingIntent` tied to a `Notification`.

However, a foreground service is not "foreground enough" to be considered starting an activity from the foreground.

What Happens

If you try to start an activity from the background... nothing visible happens. A system app will file a warning message in Logcat:

```
W/ActivityTaskManager: Background activity start [callingPackage:
com.commonsware.android.q.attention; ...]
```

But, otherwise, that's it.

In the Q beta releases, a `Toast` would appear, but this was removed in the final shipping version.

The Full-Screen Notification Alternative

What Google wants you to do is to switch to a `Notification` that uses the "full-screen `Intent`" feature. A high-importance, high-priority `Notification` that uses this feature will get a heads-up presentation, where if the user taps on the bubble, the `PendingIntent` associated with the full-screen feature will be executed. That same `PendingIntent` is executed if the `Notification` is raised while the screen is off, so the user can be taken to your activity immediately upon getting through the keyguard.

The idea is that this "full-screen" option still allows rapid access to your activity, while not interfering with the user while the user is using their device.

The `PayAttention` sample module in the book's sample project serves as a playground for doing both background activity starts and background full-screen notifications.

The UI consists of two really big buttons, one to start an activity, and the other to show a notification. When the user clicks one of those buttons, the `MainActivity` uses `WorkManager` to do some work in 10 seconds, then calls `finish()` to destroy the activity and ensure that we are in the background:

```
package com.commonsware.android.q.attention

import android.os.Bundle
```

```
import androidx.appcompat.app.AppCompatActivity
import androidx.work.OneTimeWorkRequestBuilder
import androidx.work.WorkManager
import kotlinx.android.synthetic.main.activity_main.*
import java.util.concurrent.TimeUnit

class MainActivity : AppCompatActivity() {
  private val workManager by lazy { WorkManager.getInstance() }

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    toolbar.title = getString(R.string.app_name)

    activity.setOnClickListener {
      workManager.enqueue(OneTimeWorkRequestBuilder<StartActivityWorker>()
        .setInitialDelay(10, TimeUnit.SECONDS)
        .build())
      finish()
    }

    notification.setOnClickListener {
      workManager.enqueue(OneTimeWorkRequestBuilder<ShowNotificationWorker>()
        .setInitialDelay(10, TimeUnit.SECONDS)
        .build())
      finish()
    }
  }
}
```

(from PayAttention/src/main/java/com/commonsware/android/q/attention/MainActivity.kt)

`StartActivityWorker` just starts an activity, though it needs to use `FLAG_ACTIVITY_NEW_TASK` since we are starting the activity using the `Application` context:

```
class StartActivityWorker(
  private val appContext: Context,
  workerParams: WorkerParameters
) : Worker(appContext, workerParams) {
  override fun doWork(): Result {
    appContext.startActivity(
      Intent(
        appContext,
        MainActivity::class.java
      ).addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
```

```
    )

    return Result.success()
  }
}
```

(from PayAttention/src/main/java/com/commonsware/android/q/attention/Work.kt)

ShowNotificationWorker, by contrast, sets up a Notification that uses setFullScreenIntent() to make it a full-screen notification:

```
class ShowNotificationWorker(
  private val appContext: Context,
  workerParams: WorkerParameters
) : Worker(appContext, workerParams) {
  override fun doWork(): Result {
    val pi = PendingIntent.getActivity(
      appContext,
      0,
      Intent(appContext, MainActivity::class.java),
      PendingIntent.FLAG_UPDATE_CURRENT
    )

    val builder = NotificationCompat.Builder(appContext, CHANNEL_WHATEVER)
      .setSmallIcon(R.drawable.ic_notification)
      .setContentTitle("Um, hi!")
      .setContentText("remove me")
      .setAutoCancel(true)
      .setPriority(NotificationCompat.PRIORITY_HIGH)
      .setFullScreenIntent(pi, true)

    val mgr = appContext.getSystemService(NotificationManager::class.java)

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O
      && mgr.getNotificationChannel(CHANNEL_WHATEVER) == null
    ) {
      mgr.createNotificationChannel(
        NotificationChannel(
          CHANNEL_WHATEVER,
          "Whatever",
          NotificationManager.IMPORTANCE_HIGH
        )
      )
    }

    mgr.notify(NOTIF_ID, builder.build())

    return Result.success()
```

```
  }
```

(from PayAttention/src/main/java/com/commonsware/android/q/attention/Work.kt)

Note that the `Notification` needs to be `PRIORITY_HIGH` and the channel needs to be `IMPORTANCE_HIGH` for this to work.

The module has two product flavors:

- `legacy` has `targetSdkVersion` set to 28
- `q` has `targetSdkVersion` set to `29`

Partly, this is so you can see the behavior of both existing apps and Android 10-ready apps on Android 10.

Partly, though, it is to point out another requirement of using full-screen notifications on Android 10. If your app has `targetSdkVersion 29`, you need to request the `USE_FULL_SCREEN_INTENT` permission. This is a `normal` permission, so all you need is the `<uses-permission>` element for it in your manifest. This module handles that through a manifest in the `q` source set, so the `<uses-permission>` element only gets merged in for `q` builds:

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

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

</manifest>
```

(from PayAttention/src/q/AndroidManifest.xml)

If you skip this permission, your notification will still be displayed, but the full-screen feature will not be enabled.

APIs Newly Requiring `ACCESS_FINE_LOCATION`

A long list of APIs have been added to those that require your app to hold the `ACCESS_FINE_LOCATION` permission. While none of these methods give you GPS coordinates, they do provide information that can be used to derive the user's location.

`ACCESS_FINE_LOCATION` is a `dangerous` permission, so you will need to both have the

<uses-permission> element for it and request it at runtime.

Alternatively, see if you can find some way to avoid needing to call those methods.

Continued Fight Against "Non-SDK Interfaces"

Starting with Android 9.0, Google began blocking your ability to access classes and members marked with the @hide pseudo-annotation, plus private and package-private members. Simply put, if it is not documented in the Android SDK, you may not be able to refer to it at compile time by using Java reflection or similar techniques.

Android 10 adds yet more methods that are restricted.

The previous "dark greylist"/"light greylist" distinction has been replaced by a new system. Restricted methods may have an indication of whether they are allowed for certain targetSdkVersion values or not. Look for @UnsupportedAppUsage annotations in the AOSP source code, as those represent the banned methods. If they have a maxTargetSdk value, that indicates the highest targetSdkVersion for which those methods are supported — otherwise, they are banned. If maxTargetSdk is 0, that method is banned for all targetSdkVersion values.

Developers should examine their code bases for any signs of using reflection to access hidden members: Class.forName(), getConstructor(), getField(), and so on. This is particularly important for library authors, as issues with a library get amplified by the number of library users.

Remember that StrictMode offers callbacks to find out when violations occur, and treats these violations as part of the VM policy. So, you could arrange to have StrictMode report these violations to you, where you then log the stack trace somewhere. For example, you might modify your test suites to enable StrictMode to collect this information, then write the output somewhere that your tests can pick up and incorporate into their results.

If you would like to be more proactive about dealing with this, the warning system for non-SDK usage is integrated with StrictMode. detectNonSdkApiUsage() is an option for StrictMode.VmPolicy.Builder, so you can tie it into your overall StrictMode reporting approach (e.g., crash app in debug builds, but in release builds just use the listener option on Android 9+ to integrate with your crash reporting engine).

SYSTEM_ALERT_WINDOW Restrictions

SYSTEM_ALERT_WINDOW is a special permission that allows an app to draw over other apps. First made prominent by Facebook's "chat heads" feature, a dizzying array of apps now request SYSTEM_ALERT_WINDOW. However, the ability to draw over other apps raises serious security issues.

At minimum, Android 10 is set to block SYSTEM_ALERT_WINDOW on low-end Android Go devices, citing performance concerns.

Beyond that, Android Police claims that there are more significant limitations applied across the board:

- Apps that were installed via the Play Store and received SYSTEM_ALERT_WINDOW automatically lose it upon a device reboot
- Apps that were "side-loaded" (i.e., installed from somewhere other than the Play Store) lose the permission 30 seconds after installation

If your app relies upon SYSTEM_ALERT_WINDOW, you will need to investigate these items further and perhaps come up with some other solution for your problem. That might involve bubbles, at least starting in 2020.

ContactsContract Field Deprecations

A handful of data elements that you can request from ContactsContract are "obsolete". Reading between the lines of the note in the JavaDocs about this, it appears that Google is proactively flushing this data.

And, if that is correct, it would appear that this data is not reliable on *any* relevant version of Android, not just Android 10.

In particular, they mention that these four values are affected:

- ContactsContract.ContactOptionsColumns.LAST_TIME_CONTACTED
- ContactsContract.ContactOptionsColumns.TIMES_CONTACTED
- ContactsContract.DataUsageStatColumns.LAST_TIME_USED
- ContactsContract.DataUsageStatColumns.TIMES_USED

These have privacy implications beyond merely knowing who the contact is, as they give insight into the contact's behavior.

If you use `ContactsContract`, please review the note and determine if you are affected.

Background Clipboard Access Banned

Apps no longer have access to the clipboard, unless:

- They are the default input method editor, or
- They are "the app that currently has focus"

In a typical single-window environment, this would mean that your activity is in the foreground. In split-screen or multi-window environments, this phrasing suggests that even if your activity is visible, it may not have access to the clipboard, if it lacks the focus. In other words, if your activity has been called with `onResume()` but not `onPause()`, you should have access to the clipboard.

`android:sharedUserId` Deprecated

`android:sharedUserId` is an attribute that you can have in the manifests of multiple apps to try to have them share a common Linux-style user account when installed on Android. This would allow them to read and write each other's *internal* storage directly.

This feature was added largely for the benefit of pre-installed apps. It always represented a fair amount of risk for ordinary app developers. In particular, if you changed the `android:sharedUserId` value — including adding one when one did not exist — now your own app would be unable to access its own internal storage from any previously installed version of the app. Those files were owned by some other Linux-style user account, not the new account requested by the new `android:sharedUserId` value.

In Android 10, `android:sharedUserId` is deprecated, and it is slated for outright removal in some future release.

If you are using `android:sharedUserId`, start switching to IPC-based means of app-to-app communication, rather than having one app modify another app's files "behind its back".

`DownloadManager` Deprecations

`DownloadManager` deprecated some things, presuambly as a side effect of scoped

storage:

- `addCompletedDownload()`
- `allowScanningByMediaScanner()` on `DownloadManager.Request`
- `setVisibleInDownloadsUi()` on `DownloadManager.Request`

Items listed in the new `MediaStore.Downloads` collection will be what appears in the Downloads UI now. So, if you have content that was not downloaded to the `Downloads/` directory by `DownloadManager`, and you want it to appear in the Downloads UI, you need to write it to a `Uri` supplied by `MediaStore.Downloads`.

Moar Densities!

There are four new `DisplayMetrics` screen densities: 140, 180, 200, 220. Those are the first new low-end densities we have had in years, and it is unclear what hardware would have such screens. Most developers will not need to worry about these, as Android will scale `mdpi` or `hdpi` drawables for you. But, if you have other code that cares about these `DENSITY_` constants on `DisplayMetrics`, you have four more to deal with.

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 10 devices.

`Preference` Deprecation

The framework set of `Preference` classes are marked as deprecated on Android 10.

This makes some sense. `Preference` and its subclasses (e.g., `EditTextPreference`, `ListPreference`) are wrappers around widgets. However, those wrappers wrap framework widgets and therefore may not adapt to AppCompat themes.

Google steers you in the direction of the AndroidX preference library (`androidx.preference:preference` and `androidx.preference:preference-ktx`). This appears to be their direction, at least for the next few years.

Note, though, that the AndroidX preference library does not have a `RingtonePreference`, nor does Google plan to add one. On the other hand, the AndroidX preference library does offer `DropDownPreference` and

SeekBarPreference, which were lacking in the native preference system.

ACTION_SEND Previews

Android 10 offers a new content preview feature when using ACTION_SEND, where the user can see a customized preview of what it is that they are sharing.

To implement this, you can:

- Add EXTRA_TITLE to the ACTION_SEND Intent, with some text to appear as the title of the preview
- Add a ClipData to the ACTION_SEND Intent, via setClipData(), that represents the image to show as part of the preview

On the plus side, this requires no new APIs, so this code will work on older versions of Android.

However, this feature has issues:

- This does not work with an android.resource Uri for the preview image — the Uri that you use must be a content Uri (e.g., from FileProvider)
- If you provide both a title and a preview image, you only get the preview image

Settings Panels

We have long had a series of Intent actions defined on the Settings class to be able to launch into a particular screen of the Settings app, using startActivity().

Android 10 extends this with "panel" actions. These launch a screen in the Settings app that is styled as a bottom-sheet dialog:

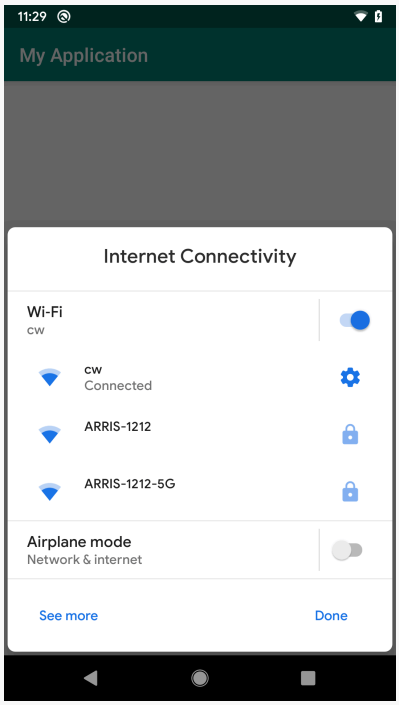


Figure 13: Internet Connectivity Settings Panel

There are four such panel actions:

- `Settings.Panel.ACTION_INTERNET_CONNECTIVITY`
- `Settings.Panel.ACTION_NFC`
- `Settings.Panel.ACTION_VOLUME`
- `Settings.Panel.ACTION_WIFI`

Changes in Device Authentication

If you have been using `KeyguardManager` and its `createConfirmDeviceCredentialIntent()` method to authenticate the user, it has been deprecated on Android 10. Like most deprecations, this method still works. However, Google is steering you in a different direction: using `setDeviceCredentialAllowed()` with `BiometricPrompt`.

A Quick Device Authentication Recap

Sometimes, we want to confirm that the person holding the device is the authorized user of that device, before proceeding with something sensitive.

For devices with a `minSdkVersion` of 21 or higher, the simple solution was to use `createConfirmDeviceCredentialIntent()` on `KeyguardManager`. This would return an `Intent` that you could pass to `startActivityForResult()`. It would bring up the system PIN/password screen to authenticate the user, letting you know of success or failure via `onActivityResult()`.

However, that approach only offers the PIN/password option. It does not allow the user to use biometrics, such as a fingerprint, to authenticate. For that, we have `BiometricPrompt` on Android 9.0+ (or `FingerprintManager` and `FingerprintDialog` for API Level 23-27). Those allow the user to authenticate using biometrics... but *only* via biometrics. That does not work for users who have not set up a biometric authentication option, or for devices that lack biometric capability.

Compounding the complexity is Android 10's deprecation of `createConfirmDeviceCredentialIntent()`.

`setDeviceCredentialAllowed()`

The replacement is `setDeviceCredentialAllowed()` on `BiometricPrompt`. On Android 10 devices, this brings up the PIN/password screen if either:

- The user does not have a registered fingerprint or other biometric authentication option, or
- The user elects to skip the biometric check and use the PIN/password instead

Presumably, it will also use the PIN/password screen for devices that lack any biometric hardware, though this has not been tested.

The `SecureCheq` sample module in the book's sample project demonstrates the use of `setDeviceCredentialAllowed()`. This sample app is based on a sample that originally appeared in *The Busy Coder's Guide to Android Development* for showing how to use `BiometricPrompt`. Here, it is converted to Kotlin and tweaked to employ `setDeviceCredentialAllowed()`, along with another new Android 10 method, `setConfirmationRequired()`.

As with that earlier sample, here use a `BiometricPrompt.Builder` to create an instance of `BiometricPrompt`:

```
    val prompt = BiometricPrompt.Builder(this)
      .setTitle("This is the title")
      .setDescription("This is the description")
      .setSubtitle("This is the subtitle")
      .setConfirmationRequired(true)
      .setDeviceCredentialAllowed(true)
      .build()
```

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

`setConfirmationRequired()` is primarily for non-fingerprint sorts of biometrics, such as face recognition. Those can be fairly automatic: if the user happens to be looking at the screen at the time of authentication, no actual user input is required. This may be *too* easy, and it is why Android allows the user to disable "passive" authentication. You too can disable passive authentication, via `setConfirmationRequired(true)`. This indicates that for biometrics like face recognition, that you want the user both to authenticate *and* tap an on-screen button to confirm that they wish to proceed. This is new to Android 10.

Also new to Android 10 is `setDeviceCredentialAllowed()`. As noted, this will fall back to PIN/password authentication, if biometrics are unavailable or opted-out by the user, though there appears to be a bug here. In the sample code, we call `setDeviceCredentialAllowed(true)` to request this behavior. If you do not call `setDeviceCredentialAllowed(true)` — either by passing `false` or skipping the call entirely — you *must* call `setNegativeButton()`, to provide details of what to do if the user elects to skip the authentication process entirely. We will see an example of that shortly.

Nothing changes with our `authenticate()` call or the `AuthenticationCallback` object. However, there should be a slight behavior change with the `onAuthenticationError()` function on the `AuthenticationCallback`. `onAuthenticationError()` can be called with an error code of `BiometricPrompt.BIOMETRIC_ERROR_NO_BIOMETRICS` to indicate that the user has not enrolled any fingerprints or other biometric identifiers. In the case of `setAllowDeviceCredential(true)`, though, this should not occur. Instead, if the user has no registered identifiers, the user will be sent to the PIN/password screen to authenticate that way.

Note that the `androidx.biometric` library offers a backwards-compatible

implementation of `BiometricPrompt` that includes an implementation of `setDeviceCredentialAllowed()`. This does not give prior versions of Android this capability, but it will "gracefully degrade" on older devices.

Learning More About MIME Types

`ContentResolver` offers a `getTypeInfo()` method. Given a MIME type, it returns a `MimeTypeInfo` object. This offers:

- A label
- A content description, which might be the same as the label
- An icon

The `TypeInfo` sample module contains a small app that presents a list of 65 MIME types and the associated label and icon from Android 10's `MimeTypeInfo`:

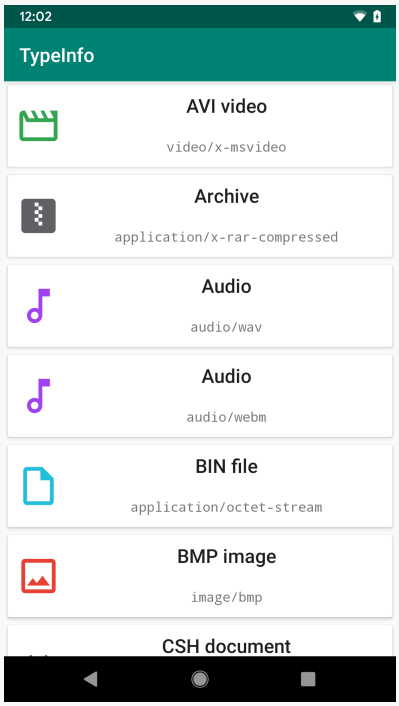


Figure 14: TypeInfo Sample App, Showing Some MIME Types

Given a long `listOf()` MIME types named `MIME_TYPES`, `MainMotor` loads them in the background using a `ContentResolver` and maps the `MimeTypeInfo` data into a `RowState` for use by the UI:

```
private suspend fun mapTypes(context: Context) =
  withContext(Dispatchers.Default) {
    val resolver = context.contentResolver

    MIME_TYPES
      .map { RowState(it, resolver.getTypeInfo(it)) }
      .sortedBy { it.description.toString() }
  }
```

(from TypeInfo/src/main/java/com/commonsware/android/q/typeinfo/MainMotor.kt)

This is useful for cases where you have an arbitrary `Uri` and you want to have more information about it for presentation, whether in a list like `TypeInfo` or for single-`Uri` "attachments". Given a `Uri` and `DocumentFile.fromSingleUri()`, you can get the MIME type and display name; `MimeTypeInfo` just gives you more information about the type.

However, the data returned in a `MimeTypeInfo` object is very generic in general. Mostly, they seem to have a category of types, which controls the icon and part of the label (e.g., "Archive", "Audio"). In many cases, if the MIME type is not recognized, the label is simply "File". For casual use, this may be acceptable, but serious apps should consider serious solutions (and ones that are not tied to Android 10).

Foreground Service Types

Depending on what your app does, you may start seeing crashes with the following sort of error message:

```
Caused by: java.lang.SecurityException: Media projections require a foreground
service of type ServiceInfo.FOREGROUND_SERVICE_TYPE_MEDIA_PROJECTION
```

Android 10 adds an `android:foregroundServiceType` attribute to the `<service>` element in the manifest. We saw this used for a `location` value in the chapter on location changes. While this is not documented on the `<service>` element itself, there is a bit of documentation in the `R.attr` JavaDocs.

Depending on what your app does in its service, you may have a total of three requirements now:

- Have it be a foreground service, to get past the runtime limitations imposed by Android 8.0
- Request the `FOREGROUND_SERVICE` permission

- Have a `android:foregroundServiceType` attribute that lists the protected operations that your foreground service intends to perform

Right now, there are six possible values:

Constant Value	Apparently Required If You...
`connectedDevice`	...use Bluetooth, Android Auto, or Android TV APIs
`dataSync`	...perform Internet operations
`location`	...work with `LocationManager` or things layered atop of it
`mediaPlayback`	...work with audio/video APIs (e.g., `MediaPlayer`)
`mediaProjection`	...capture screenshots or record screencasts with `MediaProjection`
`phoneCall`	...participate in an "ongoing phone call or video conference"

It is unclear what actual code might trigger some of these. For example, it is unclear if any Internet operations require `dataSync` or only certain things (e.g., `DownloadManager`).

If your service might perform more than one of these, you can combine these values with `|` operators (e.g., `android:foregroundServiceType="location|mediaProjection"`).

Audio Capture

Since Android 5.0, we have had an API for capturing screenshots and recording screencasts. However, any such screencast was a "silent film", consisting only of video, not audio.

Android 10 introduces an official solution for capturing audio from other apps, tied to the same "media projection" system used for screenshots and screencasts.

You can learn more about `MediaProjectionManager` in the "The Media Projection APIs" chapter of *The Busy Coder's Guide to Android Development*!

Capturing Audio

There are three major requirements for your app to be able to capture audio from other apps:

1. You will need to request the `RECORD_AUDIO` permission. This is a `dangerous` permission and therefore will require you to request it both in the manifest and at runtime via `ActivityCompat.requestPermissions()`.
2. If the audio capture will be performed by a service (the typical case), that service will need to be a foreground service and have its `android:foregroundServiceType` `<service>` attribute contain `mediaProjection` (along with any other values that you might need.
3. You will need to obtain a `MediaProjection` object.

You get a `MediaProjection` through a slightly-annoying process:

- Use `startActivityForResult()` to start the `createScreenCaptureIntent()` supplied by `MediaProjectionManager`:

```
val mgr =
  getSystemService(Context.MEDIA_PROJECTION_SERVICE) as MediaProjectionManager

startActivityForResult(mgr.createScreenCaptureIntent(), REQUEST_SCREENCAST)
```

- Hand the `resultCode` and `data` from the request to `getMediaProjection()` on the `MediaProjectionManager` to get the `MediaProjection` object:

```
override fun onActivityResult(
  requestCode: Int,
  resultCode: Int,
  data: Intent?
) {
  if (requestCode == REQUEST_SCREENCAST) {
    if (resultCode == RESULT_OK) {
      val projection = mgr.getMediaProjection(resultCode, data!!)

      // TODO something with this
```

```
    }
  }
}
```

Given all of that, in theory, you can:

- Create an `AudioPlaybackCaptureConfiguration` using that `MediaProjection` object:

```
val playbackConfig = AudioPlaybackCaptureConfiguration.Builder(projection).build()
```

- Include that in an `AudioRecord.Builder` using `setAutoPlaybackCaptureConfig()`:

```
val audioRecord = AudioRecord.Builder()
    .setAudioFormat(TODO())
    .setAudioPlaybackCaptureConfig(playbackConfig)
    .setAudioSource(TODO())
    .setBufferSizeInBytes(TODO())
    .build()
```

(with `TODO()` shown as placeholders for the rest of the `AudioRecord` configuration)

- Use that `AudioRecord` object the same way that you might use it to record off of the microphone

However, the documentation for `AudioRecord` is sketchy in general, and even worse with respect to audio capture. For example, it is unclear what the value should be for `setAudioSource()` and what the configuration should be for the `AudioFormat` passed to `setAudioFormat()`.

Note that the result of this will be an audio file (or some other collection of audio bytes). If you are also trying to capture the screen using `MediaProjection`, this approach may give you the audio, but it will not synchronize that audio with the video, let alone put it in the same file as the video.

Note that not all apps' audio can be captured, as we will explore in the next section.

Availability of Audio Capture

Only apps that allow audio capture can have their audio captured. In effect, apps can opt out of audio capture, just as they can use `FLAG_SECURE` to opt out of screen

capture.

One way to block audio capture is to generate audio that is of a type that is not designed for capture. Specifically, the only audio that can be captured is audio flagged as `USAGE_MEDIA`, `USAGE_GAME`, or `USAGE_UNKNOWN`, referring to constants on `AudioAttributes`. Apps using `AudioTrack` for audio playback get to specify this value; other APIs might set their own value (e.g., `MediaPlayer` uses `USAGE_MEDIA`). So, if you cannot capture the audio from some app, it may be that the app — intentionally or accidentally — has specified some other usage type, such as `USAGE_VOICE_COMMUNICATIONS`.

In addition, an app can have an overall capture policy. The default is:

- To block capture for apps with a `targetSdkVersion` of 28 or lower
- To allow capture for apps with a `targetSdkVersion` of 29 or higher

If you wish to control this more directly for your app, you can:

- Configure capture of specific audio by using `setAllowedCapturePolicy()` on an `AudioAttributes` (for use with `AudioTrack`) or `AAudioStreamBuilder` (for use with `AAudio`)
- Configure capture of all audio from your app dynamically by calling `setAllowedCapturePolicy()` on an instance of `AudioManager`
- Configure capture of all audio from your app statically by setting `android:allowAudioPlaybackCapture` on an undocumented element in the manifest (try `<application>`)

`setAllowedCapturePolicy()` has three possible values:

- `ALLOW_CAPTURE_BY_ALL` allows the system and third-party apps to capture the audio
- `ALLOW_CAPTURE_BY_SYSTEM` allows only system apps to capture the audio
- `ALLOW_CAPTURE_BY_NONE` blocks all audio capture, even by system apps

Deep Presses

Via methods like `onTouchEvent()`, you can get `MotionEvent` objects describing low-level interactions between the user and input devices, particularly touchscreens.

Android 10 adds a "classification" of touch event: a deep press. This is described as stemming from "the user intentionally pressing harder on the screen".

Android Police surmises that this may lead to iOS-like "3D Touch" behavior. At present, though, it is unclear whether all hardware will support these events or if they require a special digitizer.

`useEmbeddedDex`

Your APK contains your code and the code from the libraries that you add to your app. That compiled code is packaged as "DEX" files representing Dalvik bytecode. Smaller apps might have a single DEX file, larger apps might have more than one ("multidex").

On Android 4.3 and below, the Dalvik runtime would read the DEX content directly from the APK. APKs are digitally signed, so there is no way for an outside party to tamper with the code before Dalvik loads and runs it. This is great from a security standpoint but adds overhead.

On Android 4.4 and higher, the Dalvik and ART runtimes started pre-processing those DEX files, including the ahead-of-time (AOT) compilation added in Android 5.0. The output of that work is stored as ordinary files, and so processes with root privileges could tamper with them. The result is improved app performance at the cost of weakened security.

Android 10 allows you to set `android:useEmbeddedDex` to `true` in your `<application>` element. This tells ART to go back to Android 4.3-style approaches, avoiding the pre-processing and only using the DEX files packaged in the APK. This allows developers to opt into the tighter security, for cases where that security is worth the performance penalties (e.g., no ahead-of-time compilation).

`hasFragileUserData`

XDA-Developers pointed out that there is an `android:hasFragileUserData` flag in Android 10. This is lightly documented — we know the flag exists, but the documentation fails to indicate where the flag belongs.

According to the article, setting this to `true` will prompt the user whether to keep your app's data when the system uninstalls the app. In principle, this would allow the user to get back at your data if they later re-install the app. In practice, it remains to be seen how well this works.

Mystifying Things

A couple of items were introduced in Android 10 but were removed from being public-facing for the final release. It is possible that these will become more important in future versions of Android.

Roles

Q Beta 1 introduced roles and `RoleManager`.

Q Beta 2 removed the documentation and, um, primary role for `RoleManager`.

Q Beta 3 through the Android 10 release still have `RoleManager`, but there is no sign of whether it is being used.

Here is what we know about what roles were to be used for, in case they show up again.

What Is a Role?

A role is a bit like a runtime permission:

- You need to have stuff in the manifest to be eligible for it
- You need to have code to detect if you have the role — and if not, to ask the user to grant you the role
- The user can grant or revoke roles at any point (e.g., via the Settings app)
- The role helps determine what your app can do

However, a permission is a statement of a desired capability, such as "I want to be able to read files on external storage". A role, rather, is a statement of what job the app will fulfill for the user.

According to the `RoleManager` JavaDocs, there are eight available roles:

- `ROLE_ASSISTANT`
- `ROLE_BROWSER`
- `ROLE_CALL_REDIRECTION`
- `ROLE_CALL_SCREENING`
- `ROLE_DIALER`
- `ROLE_EMERGENCY`

- ROLE_HOME
- ROLE_SMS

In general, the names of the roles indicate their purpose. For example, ROLE_BROWSER represents a Web browser, while ROLE_SMS represents a messaging client.

If your app requests and is granted a role, you get certain capabilities. For example, an app with ROLE_GALLERY has access to more of the device content than does an ordinary app, as part of the new scoped storage system.

There Can Only Be One

One substantial difference between permissions and roles is that any number of apps can hold a permission, while only one app can hold a role. If an app requests a role and some other app holds that role, if that second app asks the user for the role and is granted it, the first app loses the role.

For some roles, other system limitations already imply that only one app could fill a role. For example, an Android device cannot readily handle more than one home screen. In other cases, the "there can only be one" rule for roles may introduce some user headaches, such as when switching between Web browsers.

Becoming Role-Eligible

As noted above, to become eligible for a role, there are a few things that you will need in your manifest.

The direct determining factor of whether you are eligible for a role comes from an <intent-filter>.

On your MAIN/LAUNCHER activity, you need to add two more <category> elements:

- <category android:name="android.intent.category.DEFAULT" />, to declare your wish to be the default app for the role
- A second <category> element tied to the specific role

Unfortunately, for the current set of roles, the categories are undocumented.

Obtaining a Role

Just because you request a permission in the manifest does not mean that your app holds it — on Android 6.0+, you need to request `dangerous` permissions at runtime, using functions like `ActivityCompat.requestPermissions()`.

Similarly, just because you have the right stuff in the manifest to be eligible for a role does not mean that your app holds that role. Instead, you need to ask the user for that role, using a `RoleManager` system service added to Android 10.

There are three key functions on `RoleManager`:

- `isRoleAvailable()` indicates if the version of Android and the device that you are on knows about a particular role that you want, identified by a `ROLE_`-prefixed name defined as constants on `RoleManager`. Google reserves the right to change the mix of roles over time, and it is conceivable that device manufacturers might make their own changes. So, you need to call `isRoleAvailable()` and be able to cope if it returns `false`.
- `isRoleHeld()` returns `true` if your app has been granted the role, `false` otherwise. This is reminiscent of calling `checkSelfPermission()` to see if you hold a runtime permission.
- `createRequestRoleIntent()` does pretty much what the name indicates: it creates an `Intent` for you to be able to ask the user to be able to hold a role. You pass that `Intent` to `startActivityForResult()`, and in `onActivityResult()` you can find out if you got the role (by looking for an `ACTIVITY_OK` response or calling `isRoleHeld()` again)

Losing a Role

The user is welcome to go into the Settings app and revoke your role.

Similar to losing a runtime permission, if you lose a role, Android will terminate your process. Hence, just as you should check to see if you hold a runtime permission on every start of your app, you should check to see if you hold any desired roles on every start of your app.

However, due to another bug, `RoleManager` will incorrectly return `true` from `isRoleHeld()` after a user revokes the role. As a result, there is no great way to determine whether you really hold the role or you held it previously and now no longer hold it.

Role Powers

Due to the very limited role documentation, it is unclear how to obtain roles and what apps holding those roles can do (and what apps *without* those roles) cannot do.

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 slowly starting to restrict the use of `SYSTEM_ALERT_WINDOW` as a result.

In Q Beta 1, Google announced that they were introducing "bubbles" and indicated that this would be a long-term replacement for `SYSTEM_ALERT_WINDOW`. In Q Beta 3, Google announced that bubbles would be limited to developers, with an opt-in toggle in the developer options to allow them to be seen. That remains the status with the shipping version of Android 10: bubbles are available for developers but not for ordinary users.

As a result, it is quite possible that bubbles never become available for users, just as Android 9.0' slices have been largely unused, at least as of May 2019. Most likely, you should just leave bubbles alone until Android R, then worry about implementing them at that point.