Android Development with Kotlin

Data Storage in Android

Introduction:

So, in this lecture we will be focusing on the following data storage options:

a) Shared Preferences
b) Files

Storage:

Under Android the on-disk storage is split into two areas: **internal storage and external storage.**

Internal storage:

By default, any files that your app writes to the internal storage of an Android device are only accessible by your app. This protection is enforced by the Linux kernel through the app sandbox, and includes any files created indirectly by SQLite databases etc. We can use a **device's internal storage to store our data as well. Here internal storage doesn't mean the complete inbuilt storage. Internal storage is small memory,** which is used by Android for saving app data. Along with any SD Card, rest of the inbuilt storage is considered as external storage as well.

External storage:

Files created on external storage are world readable and writeable, i.e. any app can read or write to them. Indeed, since external storage can (often) be removed from the device and connected to any other computer, it is not possible to enforce access control for files on external storage. This includes any SD Card installed by the user as well as all user data like albums, ringtones, music falls into external storage as well.

This means any app that uses external storage should:

1. Encrypt any sensitive data that it writes to external storage.
2. Perform **input validation** on any data that it reads from external storage.

    a. Permissions: In our file explorer you will see that we have a folder called sdcard. Clicking on this folder doesn't show us any data. Try adding android.permission.READ_EXTERNAL_STORAGE permission in your manifest and install the app again. You will be able to see all the missing data in this folder now.

    b. Reading: In case you want to read a specific directory from the external storage, you should first check if the external storage is available or not. You can use Environment.getExternalStorageState() function to get the current state and you can read from external storage if the state is equal to Environment.MEDIA_MOUNTED or ENVIRONMENT.MEDIA_MOUNTED_READ_ONLY. After which you can use Environemnt.getExternalStoragePublicDirectory function to get various libraries. Checkout the documentation here for more details.

c. Writing: You can save files like any other files as long you have the right permissions to do so. You would need android.permission.WRITE_EXTERNAL_STORAGE to write to external storage.

From a security perspective therefore, the best policy is for your app to not use external storage unless it really needs to.

So, which data storage you should use? The solution you choose depends on your specific need:

How much space does your data require?
Internal storage has limited space for app-specific data. Use external storage if you need to save a substantial amount of data.

How reliable does data access need to be?
If your app's basic functionality requires certain data, such as when your app is starting up, place the data within internal storage directory or a database. App-specific files that are stored in external storage aren't always accessible because some devices allow users to remove a physical device that corresponds to external storage.

What kind of data do you need to store?
If you have data that's only meaningful for your app, use app-specific storage. For shareable media content, use shared storage so that other apps can access the content. For structured data, use either preferences (for key-value data) or a database (for data that contains more than 2 columns).

Should the data be private to your app?
When storing sensitive data—data that shouldn't be accessible from any other app—use internal storage, preferences, or a database. Internal storage has the added benefit of the data being hidden from users.

In next section, we will learn how to store/access media files **in external storage**. But before we jump into that let's **learn about content resolvers** which enable us to achieve this.

Content Resolver:

The Content Resolver is the single, global instance in your application that provides access to your (and other applications') content providers. The Content Resolver behaves exactly as its name implies: it accepts requests from clients and resolves these requests by directing them to the content provider with a distinct authority. To do this, the Content Resolver stores a mapping from authorities to Content Providers. This design is important, as it allows a simple and secure means of accessing other applications' Content Providers.
The Content Resolver includes the CRUD (create, read, update, delete) methods corresponding to the abstract methods (insert, query, update, delete) in the Content Provider class. The Content Resolver does not know the implementation of the Content Providers it is interacting with (nor does it need to know); each method is passed an URI that specifies the Content Provider to interact with.

In conclusion**, ContentResolver** is used to select **the right ContentProvider** based on the ContentUris. A **ContentUri** may look like

content://com.android.contacts/contacts/3

- **content://** is called scheme and indicates that it is a ContentUri.
- **com.android.contacts** is called Content authority and ContentResolver uses it to resolve to a unique provider (in this case, ContactProvider).
- **contacts** is the path that identify some subset of the provider's data (for example, Table name).
- **3** is the id used to uniquely identify a row within the subset of data.

Media Store API:

To provide a more enriched user experience, many apps allow users to contribute and access media that's available on an external storage volume. The framework provides an optimized index into media collections, called the media store, that allows for retrieving and updating these media files more easily. Even after your app is uninstalled, these files remain on the user's device.
To interact with the media store abstraction, use a **ContentResolver** object that you retrieve from your app's context:

Generic Code:

```
val projection = arrayOf(media-database-columns-to-retrieve)
val selection = sql-where-clause-with-placeholder-variables
val selectionArgs = values-of-placeholder-variables
val sortOrder = sql-order-by-clause

applicationContext.contentResolver.query(
    MediaStore.media-type.Media.EXTERNAL_CONTENT_URI,
    projection,
    selection,
    selectionArgs,
    sortOrder
)?.use { cursor ->
    while (cursor.moveToNext()) {
        // Use an ID column from the projection to get
        // a URI representing the media item itself.
    }
}
```

For example, to interact with the **images table** you would do something like the following:

```
val projection = arrayOf(
        //you only want to retrieve _ID and DISPLAY_NAME columns
        MediaStore.Images.Media._ID,
        MediaStore.Images.Media.DISPLAY_NAME)
    context.contentResolver.query(
        uri, projection, null, null, null, null)?.use { cursor ->

      //cache column indices
      val idColumn = cursor.getColumnIndex(MediaStore.Images.Media._ID)
      val nameColumn =
cursor.getColumnIndex(MediaStore.Images.Media.DISPLAY_NAME)

        //iterating over all of the found images
      while (cursor.moveToNext()) {
        val imageId = cursor.getString(idColumn)
        val imageName = cursor.getString(nameColumn)
      }
    }
```

The system automatically scans an external storage volume and adds media files to the following well-defined collections:

- Images, including photographs and screenshots, which are stored in the DCIM/ and Pictures/ directories. The system adds these files to the MediaStore.Images table.
- Videos, which are stored in the DCIM/, Movies/, and Pictures/ directories. The system adds these files to the MediaStore.Video table.
- Audio files, which are stored in the Alarms/, Audiobooks/, Music/, Notifications/, Podcasts/, and Ringtones/ directories, as well as audio playlists that are in the Music/ or Movies/ directories. The system adds these files to the MediaStore.Audio table.
- Downloaded files, which are stored in the Download/ directory. On devices that run Android 10 (API level 29) and higher, these files are stored in the MediaStore.Downloads table. This table isn't available on Android 9 (API level 28) and lower.

Taking photos and accessing the camera app:

Add the necessary permissions for camera:

If an essential function of your application is taking pictures, then restrict its visibility on Google Play to devices that have a camera. To advertise that your application depends on having a camera, put a **<uses-feature>** tag in your manifest file:

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

Take a photo with camera app:

The Android way of delegating actions to other applications is to invoke an **Intent** that describes what you want done. This process involves three pieces: The **Intent** itself, a call to start the external **Activity,** and some code to handle the image data when focus returns to your activity.
Here's a function that invokes an intent to capture a photo.

```
val REQUEST_IMAGE_CAPTURE = 1

private fun dispatchTakePictureIntent() {
    Intent(MediaStore.ACTION_IMAGE_CAPTURE).also { takePictureIntent ->
        takePictureIntent.resolveActivity(packageManager)?.also {
            startActivityForResult(takePictureIntent,
REQUEST_IMAGE_CAPTURE)
        }
    }
}
```

Saving the full size photo:

The Android Camera application saves a full-size photo if you give it a file to save into. You must provide a fully qualified file name where the camera app should save the photo.

```
lateinit var currentPhotoPath: String

@Throws(IOException::class)
private fun createImageFile(): File {
    // Create an image file name
    val timeStamp: String =
SimpleDateFormat("yyyyMMdd_HHmmss").format(Date())
    val storageDir: File =
getExternalFilesDir(Environment.DIRECTORY_PICTURES)
    return File.createTempFile(
            "JPEG_${timeStamp}_", /* prefix */
            ".jpg", /* suffix */
            storageDir /* directory */
    ).apply {
        // Save a file: path for use with ACTION_VIEW intents
        currentPhotoPath = absolutePath
    }
}
```

Picking Image from gallery:

After adding permissions you can call the following method to pick an image from Gallery.

```
private fun pickImageFromGallery() {
        //Intent to pick image
        val intent = Intent(Intent.ACTION_PICK)
        intent.type = "image/*"
        startActivityForResult(intent, IMAGE_PICK_CODE)
    }

    companion object {
        //image pick code
        private val IMAGE_PICK_CODE = 1000;
        //Permission code
        private val PERMISSION_CODE = 1001;
    }

    //handle requested permission result
    override fun onRequestPermissionsResult(requestCode: Int, permissions: 
Array<out String>, grantResults: IntArray) {
        when(requestCode){
            PERMISSION_CODE -> {
                if (grantResults.size >0 && grantResults[0] ==
                        PackageManager.PERMISSION_GRANTED){
                    //permission from popup granted
                    pickImageFromGallery()
                }
                else{
                    //permission from popup denied
                    Toast.makeText(this, "Permission denied", 
Toast.LENGTH_SHORT).show()
                }
            }
        }
    }
```

Handling Result of picked Image:

The above method will open the phone’s default Gallery App. After choosing an image, Gallery will automatically be closed and your Activity’s **onActivityResult** method will be called. The code is the following.

```
  override fun onActivityResult(requestCode: Int, resultCode: Int, data: 
Intent?) {
        if (resultCode == Activity.RESULT_OK && requestCode == 
IMAGE_PICK_CODE){
            image_view.setImageURI(data?.data)
        }
    }
```

Importing files:

You can use the below code snippet as an example to import app specific data to other apps. In this case we are importing the employees data.

```
   private fun importEmployees(){
        Intent(Intent.ACTION_GET_CONTENT).also { readFileIntent ->
            readFileIntent.addCategory(Intent.CATEGORY_OPENABLE)
            readFileIntent.type = "text/*"

readFileIntent.resolveActivity(requireActivity().packageManager)?.also {
                startActivityForResult(readFileIntent, READ_FILE_REQUEST)
            }
        }
    }
```

Exporting files:

You can use the below function to export the employees data with a uri giving the file provider with a buildconfig.Application_ID.

```
private suspend fun exportEmployees(){
        var csvFile: File? = null
        withContext(Dispatchers.IO) {
            csvFile = try {
                createFile(requireActivity(),"Documents", "csv")
            } catch (ex: IOException) {
                Toast.makeText(requireActivity(),
getString(R.string.file_create_error, ex.message),
                    Toast.LENGTH_SHORT). show()
                null
            }

            csvFile?.printWriter()?.use { out ->
                val employees = viewModel.getEmployeeList()
                if(employees.isNotEmpty()){
                    employees.forEach{
                        out.println(it.name + "," + it.role + "," + it.age
+ "," + it.gender)
                    }
                }
            }
        }
        withContext(Dispatchers.Main){
            csvFile?.let{
                val uri = FileProvider.getUriForFile(
                    requireActivity(), BuildConfig.APPLICATION_ID +
".fileprovider",
                    it)
                launchFile(uri, "csv")
            }
        }
```

```
    }
```

Storage Access Framework:

Android 4.4 (API level 19) introduces the Storage Access Framework (SAF). The SAF makes it simple for users to browse and open documents, images, and other files across all of their preferred document storage providers. A standard, easy-to-use UI lets users browse files and access recents in a consistent way across apps and providers.

The SAF includes the following:

- **Document provider**—A content provider that allows a storage service (such as Google Drive) to reveal the files it manages. A document provider is implemented as a subclass of the DocumentsProvider class. The document-provider schema is based on a traditional file hierarchy, though how your document provider physically stores data is up to you. The Android platform includes several built-in document providers, such as Downloads, Images, and Videos.

- **Client app**— A custom app that invokes the **ACTION_CREATE_DOCUMENT, ACTION_OPEN_DOCUMENT**, and **ACTION_OPEN_DOCUMENT_TREE** intent actions and receives the files returned by document providers.

- **Picker**— A system UI that lets users access documents from all document providers that satisfy the client app's search criteria.

Sharing data:

Text data:

The most straightforward and common use of the Android Sharesheet is to send text content from one activity to another. For example, most browsers can share the URL of the currently-displayed page as text with another app. This is useful for sharing an article or website with friends via email or social networking. Here's an example of how to do this:

```
val sendIntent: Intent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, "This is my text to send.")
    type = "text/plain"
}

val shareIntent = Intent.createChooser(sendIntent, null)
startActivity(shareIntent)
```

Binary Content:

Share binary data using the ACTION_SEND action. Set the appropriate MIME type and place a URI to the data in the extra EXTRA_STREAM. This is commonly used to share an image but can be used to share any type of binary content:

```
val shareIntent: Intent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_STREAM, uriToImage)
    type = "image/jpeg"
}
startActivity(Intent.createChooser(shareIntent,
resources.getText(R.string.send_to)))
```

All the concepts in data storage options have been covered extensively in the lectures. To explore more about the above topics, feel free to visit these links:

Media store api: https://developer.android.com/training/data-storage/shared/media

Sharing data: https://developer.android.com/training/sharing/send

Storing data using Storage Access Framework: https://github.com/android/storage-samples

Data Storage and File overview: https://developer.android.com/training/data-storage (Important)