MODUL 11
Persisting Data

THEME DESCRIPTION

In this module, students go in depth about data persistence in Android

WEEKLY LEARNING OUTCOME (SUB-LEARNING OUTCOME)

Students will know multiple ways to store (persist) data directly on a device and the frameworks accessible to do this. When dealing with a filesystem, they will know how it's partitioned and how they can read and write files in different locations and use different frameworks.

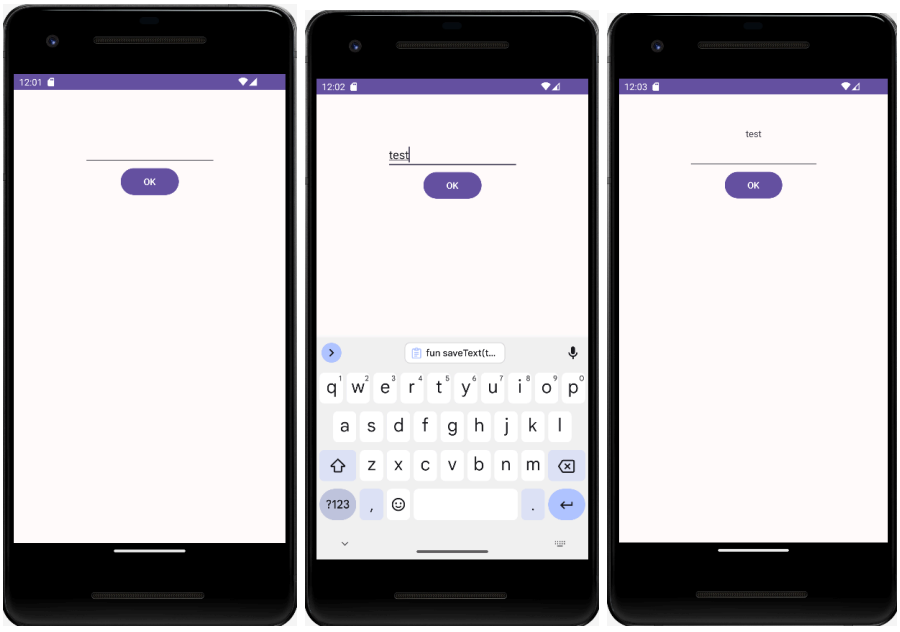
TOOLS/SOFTWARE USED

- Android Studio

PRACTICAL STEPS

Part 1 - Building App with SharedPreferences

1. Open Android Studio and click **New Project.**
2. Choose the **Empty Views Activity** to start with.
3. Name your project **"LAB_WEEK_11_A"**.
4. Set the minimum SDK to **"API 24: Android 7.0 (Nougat)"**.
5. Click **Finish**, and let your android application build itself.
6. Here's what you will be building in this part. An activity that consists of a **TextView**. **EditText** and a **Button**. The **TextView** displays a text, the **EditText** inputs a text and the **Button** saves the text into **SharedPreference**. Saving the text to **SharedPreference** keeps the value for the next time you open the app again. The **TextView** will then show the previously kept value from the **SharedPreference**. You will also be using **ViewModel** and **LiveData** just like in the previous module.



7. First, import the necessary **Dependencies** to your **build.gradle.kts (Module :app)** and don't forget to **Gradle Sync**.

```
implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.6.2")
implementation("androidx.lifecycle:lifecycle-livedata-ktx:2.6.2")
```

8. Now let's make the activity layout. Update your **activity_main.xml** to the code below.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/activity_main_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="50dp"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <EditText
```

```
        android:id="@+id/activity_main_edit_text"
        android:layout_width="200dp"
        android:layout_height="wrap_content"
        android:inputType="none"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@id/activity_main_text_view" />
    <Button
        android:id="@+id/activity_main_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:inputType="none"
        android:text="@android:string/ok"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@id/activity_main_edit_text" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

9. The layout is done, now let's make the **Preference Files**. You will be making 3 **Preference Files**:
    - PreferenceApplication.kt - Declare the SharedPreference instance
    - PreferenceWrapper.kt - Declare what to do with the SharedPreference instance
    - PreferenceViewModel.kt - Declare the ViewModel for data storing with SharedPreference

10. Create a new kotlin file called **PreferenceWrapper.kt**, and update it to the code below.

```
class PreferenceWrapper(private val sharedPreferences: SharedPreferences) {
    // The text live data is used to notify the view model when the text
changes
    private val textLiveData = MutableLiveData<String>()

    init {
        // Register a listener to the shared preferences
        // The listener is called when the shared preferences change
        sharedPreferences.registerOnSharedPreferenceChangeListener {
                _, key ->
            when (key) {
                KEY_TEXT -> {
                    // Notify the view model that the text has changed
                    // The view model will then notify the activity
                    textLiveData.postValue(
```

```
                        sharedPreferences
                            .getString(KEY_TEXT, "")
                    )
                }
            }
        }
    }

    // Save the text to the shared preferences
    fun saveText(text: String) {
        sharedPreferences.edit()
            .putString(KEY_TEXT, text)
            .apply()
    }

    // Get the text from the shared preferences
    fun getText(): LiveData<String> {
        textLiveData.postValue(sharedPreferences.getString(KEY_TEXT, ""))
        return textLiveData
    }

    // The key used to store the text in the shared preferences
    companion object{
        const val KEY_TEXT = "keyText"
    }
}
```

11. Create a new kotlin file called **PreferenceApplication.kt**, and update it to the code below.

```
class PreferenceApplication : Application() {
    lateinit var preferenceWrapper: PreferenceWrapper
    override fun onCreate() {
        super.onCreate()

        // Initialize the preference wrapper
        // The preference wrapper is used to access the shared preferences
        preferenceWrapper = PreferenceWrapper(
            // Get the shared preferences
            // The shared preferences are stored in the file
            // /data/data/com.example.lab_week_11_a/shared_prefs/prefs.xml
            getSharedPreferences(
```

```
                // The name of the file
                "prefs",
                // The mode of the file
                // MODE_PRIVATE means that only this application can access
the file
                Context.MODE_PRIVATE
            )
        )
    }
}
```

12. Create a new kotlin file called **PreferenceViewModel.kt**, and update it to the code below.

```
class PreferenceViewModel(private val preferenceWrapper: PreferenceWrapper)
: ViewModel() {
    // Save the text to the shared preferences
    fun saveText(text: String) {
        preferenceWrapper.saveText(text)
    }
    // Get the text from the shared preferences
    fun getText(): LiveData<String> {
        return preferenceWrapper.getText()
    }
}
```

13. After all the **Preference Files** are ready, now update your **MainActivity.kt** to the code below.

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

        // Get the preference wrapper from the application
        val preferenceWrapper = (application as
PreferenceApplication).preferenceWrapper
        // Create the view model instance with the preference wrapper as the
constructor parameter
        // To pass the preference wrapper to the view model, we need to use
```

```
a view model factory
        val preferenceViewModel = ViewModelProvider(this, object :
ViewModelProvider.Factory {
            override fun <T : ViewModel> create(modelClass: Class<T>): T {
                return PreferenceViewModel(preferenceWrapper) as T
            }
        })[PreferenceViewModel::class.java]

        // Observe the text live data
        preferenceViewModel.getText().observe(this
        ) {
            // Update the text view when the text live data changes
            findViewById<TextView>(
                R.id.activity_main_text_view
            ).text = it
        }
        findViewById<Button>(R.id.activity_main_button).setOnClickListener {
            // Save the text when the button is clicked
            preferenceViewModel.saveText(
                findViewById<EditText>(R.id.activity_main_edit_text)
                    .text.toString()
            )
        }
    }
}
```

14. Lastly, you need to bind your **PreferenceApplication** to your actual application. Add the code below as a new attribute in your **application** element.

```
android:name=".PreferenceApplication"
```

15. Now **Run** your application. After you input a text and press the button, the next time you open the app again, that previous value will be displayed.

Part 2 - Building App with DataStore

1. You've built an app that uses **SharedPreference** to store its data, though there's a drawback. **SharedPreference** is not as efficient as you think. A more modern approach is to use **DataStore**. Here's some differences between them:

	SharedPreference	DataStore
Storage Type	XML (great for simple dataset)	JSON (great for large dataset)
Thread Safety	No	Yes
Asynchronous	No	Yes
Data Type	Primitive Types	Complex Types

2. You are now going to build the same app as before but now using **DataStore** instead of **SharedPreference**.
3. First, import the necessary **Dependencies** to your **build.gradle.kts (Module :app)** and don't forget to **Gradle Sync**.

```
implementation "androidx.datastore: datastore-preferences:1.0.0"
```

4. Same as before, there are **3 Files** for **DataStore**:

    - SettingsApplication.kt - Declare the DataStore instance
    - SettingsStore.kt - Declare what to do with the DataStore instance
    - SettingsViewModel.kt - Declare the ViewModel for data storing with DataStore

5. Create a new kotlin file called **SettingsStore.kt** and update it to the code below.

```
// Create the data store
// The data store is stored in the file
// /data/data/com.example.lab_week_11_a/files/settingsStore
val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name
= "settingsStore")

class SettingsStore(private val context: Context) {
   // The text flow is used to notify the view model when the text changes
   val text: Flow<String> = context.dataStore.data
       .map { preferences ->
           preferences[KEY_TEXT] ?: ""
       }
   // Save the text to the data store
   suspend fun saveText(text: String) {
       context.dataStore.edit { preferences ->
           preferences[KEY_TEXT] = text
```

```
        }
    }

    // The key used to access the data in the data store
    companion object {
        val KEY_TEXT = stringPreferencesKey("key_text")
    }
}
```

6. Create a new kotlin file called **SettingsApplication.kt** and update it to the code below.

```
class SettingsApplication : Application() {
    lateinit var settingsStore: SettingsStore
    override fun onCreate() {
        super.onCreate()

        // Initialize the settings store
        // The settings store is used to access the data store
        settingsStore = SettingsStore(this)
    }
}
```

7. Create a new kotlin file called **SettingsViewModel.kt** and update it to the code below.

```
class SettingsViewModel(private val settingsStore: SettingsStore) :
ViewModel() {
    // The text live data is used to notify the view model when the text
changes
    private val _textLiveData = MutableLiveData<String>()
    val textLiveData: LiveData<String> = _textLiveData

    init {
        // Launch a coroutine to get the text from the data store
asynchronously
        viewModelScope.launch {
            settingsStore.text.collect {
                _textLiveData.value = it
            }
        }
    }
```

```
    // Save the text to the data store
    fun saveText(text: String) {
        // launch a coroutine to save the text to the data store
asynchronously
        viewModelScope.launch {
            settingsStore.saveText(text)
        }
    }
}
```

8. After all the **DataStore Files** are ready, now update your **MainActivity.kt** to the code below.

```
val preferenceWrapper = (application as SettingsApplication).settingsStore
val preferenceViewModel = ViewModelProvider(this, object :
ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return SettingsViewModel(preferenceWrapper) as T
    }
})[SettingsViewModel::class.java]

// Observe the text live data
preferenceViewModel.textLiveData.observe(this
) {
    // Update the text view when the text live data changes
    findViewById<TextView>(
        R.id.activity_main_text_view
    ).text = it
}
```

9. Lastly, bind your **SettingsApplication** to your actual application. Update the attribute below in your **application** element.

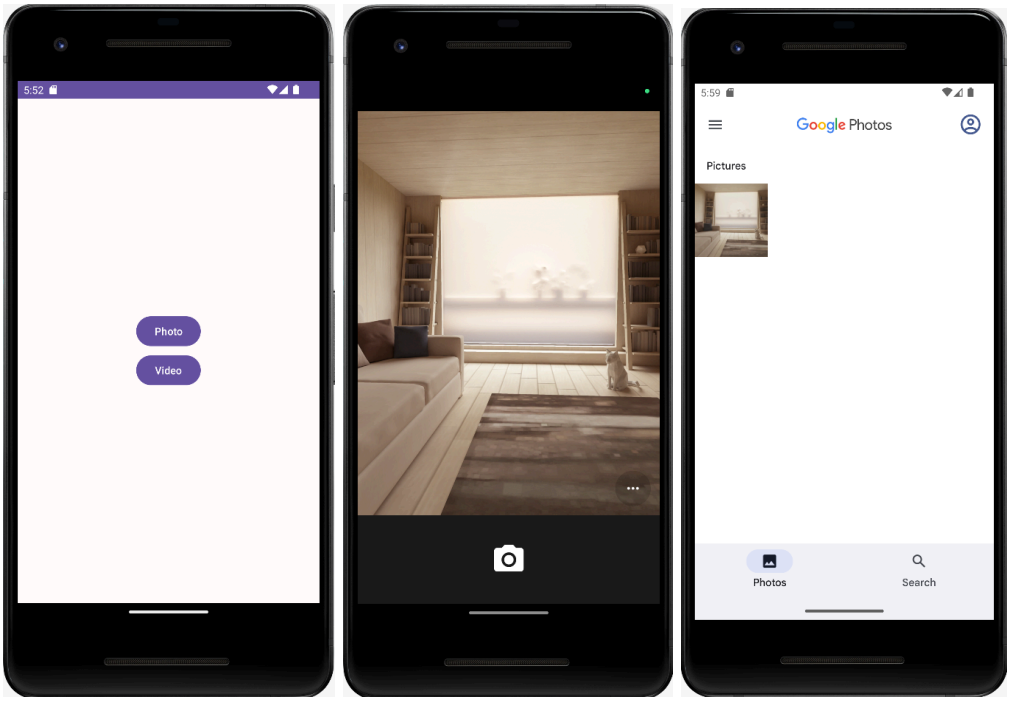
```
android:name=".SettingsApplication"
```

10. **Run** your application, it should work the same as before, but now it's a lot more efficient. The only thing that changes is that the text automatically updates when you enter an input, this is because **DataStore** uses **Flow** for their **DataStream**.

COMMIT to GITHUB at this point. Commit Message: Commit No. 1: Building App with SharedPreferences and DataStore

Part 3 - Photo and Video using FileProvider and MediaStore

1. Open Android Studio and click **New Project.**
2. Choose the **Empty Views Activity** to start with.
3. Name your project **“LAB_WEEK_11_B”**.
4. Set the minimum SDK to **“API 24: Android 7.0 (Nougat)”**.
5. Click **Finish**, and let your android application build itself.
6. Here’s what you will be building in this part. An activity that consists of **2 Buttons**. The **First Button** lets you take a picture and the **Second Button** lets you record a video. Both files will be saved into **External Storage** using **FileProvider** and **MediaStore**. **External Storage** is just like **Internal Storage** (Room, SharedPreference, DataStore), the difference is that other apps can also access that stored files. This is particularly useful when you try to access the image/video from your **Gallery** or other similar app. On the other hand, this isn’t possible in **Internal Storage**.



7. First, import the necessary **Dependencies** to your **build.gradle.kts (Module :app)** and don’t forget to **Gradle Sync**.

```
implementation("commons-io:commons-io:2.6")
```

8. Because you’re now playing with **File** and **Storage**, a **Path** is definitely needed to define where the **File** is located. To define the paths, create a new **XML File** called

file_provider_paths.xml inside the **res/xml Folder.** Then, add the code below to the file.

```
<?xml version="1.0" encoding="utf-8"?>
<paths>
   <external-path
       name="photos"
       path="Android/data/com.example.lab_week_11_b/files/Pictures"
/>
   <external-path
       name="videos"
       path="Android/data/com.example.lab_week_11_b/files/Movies" />
</paths>
```

9. Here you have defined 2 external paths, one for **Images** and the other for **Videos (Movies)**.
10. Now let's bind the defined paths in the **AndroidManifest.xml** file. Add the code below inside the **Application** element.

```
<provider
   android:name="androidx.core.content.FileProvider"
   android:authorities="com.example.lab_week_11_b.camera"
   android:exported="false"
   android:grantUriPermissions="true">
   <meta-data
       android:name="android.support.FILE_PROVIDER_PATHS"
       android:resource="@xml/file_provider_paths" />
</provider>
```

11. Don't forget to also add the permission to access external storage for **API 28 (Android 9)** and below.

```
<uses-permission
   android:name="android.permission.WRITE_EXTERNAL_STORAGE"
   android:maxSdkVersion="28" />
```

12. You've defined your paths, now let's make the layout. Update your **activity_main.xml** to the code below.

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:gravity="center"
   android:orientation="vertical"
   tools:context=".MainActivity">
   <Button
       android:id="@+id/photo_button"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="@string/photo" />
   <Button
       android:id="@+id/video_button"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginTop="5dp"
       android:text="@string/video" />
</LinearLayout>
```

13. Also add these string resources into your **Strings.xml**.

```
<string name="photo">Photo</string>
<string name="video">Video</string>
```

14. In this app, you will be creating **4** new files to utilize **FileProvider** and **MediaStore**:

    - FileInfo.kt - Provide the **Data Model** for the file.
    - FileHelper.kt - Provide a way to get the **URI** and **Relative Path** for the file.
    - MediaContentHelper.kt - Provide a way to generate **URI** for files to be stored in **MediaStore**.
    - ProviderFileManager.kt - Provide a way to store the files in **MediaStore** using the given **URI**.

15. Create a new **Kotlin File** called **FileInfo.kt** and update it to the code below.

```
data class FileInfo(
   val uri: Uri,
   val file: File,
   val name: String,
```

```
    val relativePath:String,
    val mimeType:String
)
```

16. Create a new **Kotlin File** called **FileHelper.kt** and update it to the code below.

```
class FileHelper(private val context: Context) {
    // Generate a URI to access the file
    // The URI will be temporary to limit access from other apps
    fun getUriFromFile(file: File): Uri {
        return FileProvider.getUriForFile(
            context, "com.example.lab_week_11_b.camera", file
        )
    }

    // Get the folder name for pictures
    // The name is defined in file_provider_paths.xml
    fun getPicturesFolder(): String =
        Environment.DIRECTORY_PICTURES
    // Get the folder name for videos
    // The name is defined in file_provider_paths.xml
    fun getVideosFolder(): String =
        Environment.DIRECTORY_MOVIES
}
```

17. Create a new **Kotlin File** called **MediaContentHelper.kt** and update it to the code below.

```
// Helper class to generate the URI and ContentValues for MediaStore
class MediaContentHelper {
    // Get the URI to store images and videos in MediaStore
    // The URI will be different for Android 10 and above
    fun getImageContentUri(): Uri =
        if (android.os.Build.VERSION.SDK_INT >=
android.os.Build.VERSION_CODES.Q) {

MediaStore.Images.Media.getContentUri(MediaStore.VOLUME_EXTERNAL_PRIMARY)
        } else {
            MediaStore.Images.Media.EXTERNAL_CONTENT_URI
        }
    fun getVideoContentUri(): Uri =
```

```
        if (android.os.Build.VERSION.SDK_INT >=
android.os.Build.VERSION_CODES.Q) {

MediaStore.Video.Media.getContentUri(MediaStore.VOLUME_EXTERNAL_PRIMARY)
        } else {
            MediaStore.Video.Media.EXTERNAL_CONTENT_URI
        }

    // Generate the ContentValues to store images and videos in MediaStore
    // It contains the name, relative path, and MIME type of the file
    fun generateImageContentValues(fileInfo: FileInfo) =
ContentValues().apply {
        this.put(MediaStore.Images.Media.DISPLAY_NAME, fileInfo.name)
        if (android.os.Build.VERSION.SDK_INT >=
android.os.Build.VERSION_CODES.Q) {
            this.put(MediaStore.Images.Media.RELATIVE_PATH,
fileInfo.relativePath)
        }
        this.put(MediaStore.Images.Media.MIME_TYPE, fileInfo.mimeType)
    }
    fun generateVideoContentValues(fileInfo: FileInfo) =
ContentValues().apply {
        this.put(MediaStore.Video.Media.DISPLAY_NAME, fileInfo.name)
        if (android.os.Build.VERSION.SDK_INT >=
android.os.Build.VERSION_CODES.Q) {
            this.put(MediaStore.Video.Media.RELATIVE_PATH,
fileInfo.relativePath)
        }
        this.put(MediaStore.Video.Media.MIME_TYPE, fileInfo.mimeType)
    }
}
```

18. Create a new **Kotlin File** called **ProviderFileManager.kt** and update it to the code below.

```
// Helper class to manage files in MediaStore
class ProviderFileManager(
    private val context: Context,
    private val fileHelper: FileHelper,
    private val contentResolver: ContentResolver,
    private val executor: Executor,
    private val mediaContentHelper: MediaContentHelper
```

```
) {
    // Generate the data model (FileInfo) for the file
    // The data model contains the URI, file, name, relative path,
and MIME type of the file
    fun generatePhotoUri(time: Long): FileInfo {
        val name = "img_$time.jpg"
        // Get the file object
        // The file will be stored in the folder defined in
file_provider_paths.xml
        val file = File(
            context.getExternalFilesDir(fileHelper.getPicturesFolder()),
            name
        )
        return FileInfo(
            fileHelper.getUriFromFile(file),
            file,
            name,
            fileHelper.getPicturesFolder(),
            "image/jpeg"
        )
    }
    fun generateVideoUri(time: Long): FileInfo {
        val name = "video_$time.mp4"
        // Get the file object
        // The file will be stored in the folder defined in
file_provider_paths.xml
        val file = File(
            context.getExternalFilesDir(fileHelper.getVideosFolder()),
            name
        )
        return FileInfo(
            fileHelper.getUriFromFile(file),
            file,
            name,
            fileHelper.getVideosFolder(),
            "video/mp4"
        )
    }

    // Insert the image/video to MediaStore
    fun insertImageToStore(fileInfo: FileInfo?) {
        fileInfo?.let {
            insertToStore(
```

```
                fileInfo,
                mediaContentHelper.getImageContentUri(),
                mediaContentHelper.generateImageContentValues(it)
            )
        }
    }
    fun insertVideoToStore(fileInfo: FileInfo?) {
        fileInfo?.let {
            insertToStore(
                fileInfo,
                mediaContentHelper.getVideoContentUri(),
                mediaContentHelper.generateVideoContentValues(it)
            )
        }
    }

    // Insert the file to MediaStore
    // The file will be copied to the given relative path
    // Input Stream is used to read the file
    // Output Stream is used to write the file
    private fun insertToStore(fileInfo: FileInfo, contentUri: Uri,
contentValues: ContentValues) {
        executor.execute {
            val insertedUri = contentResolver.insert(contentUri,
contentValues)
            insertedUri?.let {
                val inputStream =
contentResolver.openInputStream(fileInfo.uri)
                val outputStream =
contentResolver.openOutputStream(insertedUri)
                IOUtils.copy(inputStream, outputStream)
            }
        }
    }
}
```

19. Lastly, you need to call your **ProviderFileManager**, check the **External Storage Permission**, and launch an activity to open your **Camera**. Update your **MainActivity.kt** to the code below.

```
class MainActivity : AppCompatActivity() {
    // Request code for permission request to external storage
```

```
    companion object {
        private const val REQUEST_EXTERNAL_STORAGE = 3
    }

    // Helper class to manage files in MediaStore
    private lateinit var providerFileManager: ProviderFileManager
    // Data model for the file
    private var photoInfo: FileInfo? = null
    private var videoInfo: FileInfo? = null
    // Flag to indicate whether the user is capturing a photo or video
    private var isCapturingVideo = false
    // Activity result launcher to capture images and videos
    private lateinit var takePictureLauncher: ActivityResultLauncher<Uri>
    private lateinit var takeVideoLauncher: ActivityResultLauncher<Uri>

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

        // Initialize the ProviderFileManager
        providerFileManager =
            ProviderFileManager(
                applicationContext,
                FileHelper(applicationContext),
                contentResolver,
                Executors.newSingleThreadExecutor(),
                MediaContentHelper()
            )

        // Initialize the activity result launcher
        // .TakePicture() and .CaptureVideo() are the built-in contracts
        // They are used to capture images and videos
        // The result will be stored in the URI passed to the launcher
        takePictureLauncher =
registerForActivityResult(ActivityResultContracts.TakePicture()) {
            providerFileManager.insertImageToStore(photoInfo)
        }
        takeVideoLauncher =
registerForActivityResult(ActivityResultContracts.CaptureVideo()) {
            providerFileManager.insertVideoToStore(videoInfo)
        }

        findViewById<Button>(R.id.photo_button).setOnClickListener {
```

```
            // Set the flag to indicate that the user is capturing a
photo
            isCapturingVideo = false
            // Check the storage permission
            // If the permission is granted, open the camera
            // Otherwise, request the permission
            checkStoragePermission {
                openImageCapture()
            }
        }
        findViewById<Button>(R.id.video_button).setOnClickListener {
            // Set the flag to indicate that the user is capturing a video
            isCapturingVideo = true
            // Check the storage permission
            // If the permission is granted, open the camera
            // Otherwise, request the permission
            checkStoragePermission {
                openVideoCapture()
            }
        }
    }

    // Open the camera to capture an image
    private fun openImageCapture() {
        photoInfo =
providerFileManager.generatePhotoUri(System.currentTimeMillis())
        takePictureLauncher.launch(photoInfo?.uri)
    }

    // Open the camera to capture a video
    private fun openVideoCapture() {
        videoInfo =
providerFileManager.generateVideoUri(System.currentTimeMillis())
        takeVideoLauncher.launch(videoInfo?.uri)
    }

    // Check the storage permission
    // For Android 10 and above, the permission is not required
    // For Android 9 and below, the permission is required
    private fun checkStoragePermission(onPermissionGranted: () -> Unit) {
        if (android.os.Build.VERSION.SDK_INT <
android.os.Build.VERSION_CODES.Q) {
            //Check for the WRITE_EXTERNAL_STORAGE permission
```

```
            when (ContextCompat.checkSelfPermission(
                this,
                Manifest.permission.WRITE_EXTERNAL_STORAGE
            )) {
                // If the permission is granted
                PackageManager.PERMISSION_GRANTED -> {
                    onPermissionGranted()
                }
                // if the permission is not granted, request the permission
                else -> {
                    ActivityCompat.requestPermissions(
                        this,
                        arrayOf(Manifest.permission.WRITE_EXTERNAL_STORAGE),
                        REQUEST_EXTERNAL_STORAGE
                    )
                }
            }
        } else {
            onPermissionGranted()
        }
    }

    // For android 9 and below
    // Handle the permission request result
    override fun onRequestPermissionsResult(
        requestCode: Int,
        permissions: Array<String>,
        grantResults: IntArray
    ) {
        super.onRequestPermissionsResult(requestCode, permissions,
grantResults)
        when (requestCode) {
            //Check if requestCode is for the External Storage permission or
not
            REQUEST_EXTERNAL_STORAGE -> {
                // If granted, open the camera
                if ((grantResults.isNotEmpty() && grantResults[0] ==
PackageManager.PERMISSION_GRANTED)) {
                    if (isCapturingVideo) {
                        openVideoCapture()
                    } else {
                        openImageCapture()
                    }
```

```
                }
                return
            }
            // for other request code, do nothing
            else -> {
            }
        }
    }
}
```

20. **Run** your application and try pressing one of the buttons. Pressing one will launch the **Camera** as an **Activity**. The result of the camera will be stored inside your **MediaStore** which can be accessed with your **Gallery App**.

COMMIT to GITHUB at this point. Commit Message: Commit No. 2: Photo and Video using FileProvider and MediaStore

ASSIGNMENT

Answer these questions based on **Part 3** of the tutorial:

1. When a user takes a picture, that picture is stored in a path based on the given **URI**. In which part of the code handles this? (Copy that part of the code as the answer)
2. In your **FileInfo.kt**, there are 5 attributes. On the first attribute, what does the **URI** refer to? And on the fourth attribute, what does **relativePath** refer to?
3. **[Bonus]** Explain the chronological order from when a user takes a picture until the file is stored in the **MediaStore**.

Write your answer in a **Text File** and name it **LAB_WEEK_11.txt**.

COMMIT to GITHUB at this point. Commit Message: Commit No. 3: Assignment