60. Accessing Cloud Storage using the Android Storage Access Framework

Recent years have seen the wide adoption of remote storage services (otherwise known as "cloud storage") to store user files and data. Driving this growth are two key factors. One is that most mobile devices now provide continuous, high speed internet connectivity, thereby making the transfer of data fast and affordable. The second factor is that, relative to traditional computer systems (such as desktops and laptops) these mobile devices are constrained in terms of internal storage resources. A high specification Android tablet today, for example, typically comes with 128Gb of storage capacity. When compared with a mid-range laptop system with a 750Gb disk drive, the need for the seamless remote storage of files is a key requirement for many mobile applications today.

In recognition of this fact, Google introduced the Storage Access Framework as part of the Android 4.4 SDK. This chapter will provide a high level overview of the storage access framework in preparation for the more detail oriented tutorial contained in the next chapter, entitled *"An Android Storage Access Framework Example"*.

60.1 The Storage Access Framework

From the perspective of the user, the Storage Access Framework provides an intuitive user interface that allows the user to browse, select, delete and create files hosted by storage services (also referred to as *document providers*) from within Android applications. Using this browsing interface (also referred to as the *picker*), users can, for example, browse through the files (such as documents, audio, images and videos) hosted by their chosen document providers. Figure 60-1, for example, shows the picker user interface displaying a collection of files hosted by a document provider service:

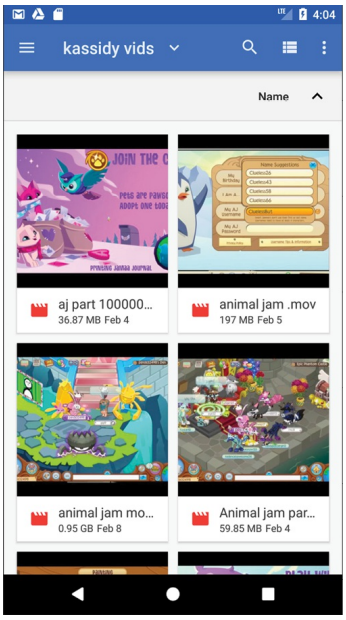

Figure 60-1

Document providers can range from cloud-based services to local document providers running on the same device as the client application. At the time of writing, the most prominent document providers compatible with the Storage Access Framework are Box and, unsurprisingly, Google Drive. It is highly likely that other cloud storage providers and application developers will soon also provide services that conform to the Android Storage Access Framework.

In addition to cloud based document providers the picker also provides access to internal storage on the device, providing a range of file storage options to the application user.

Through a set of Intents included with Android 4.4, Android application developers can incorporate these storage capabilities into applications with just a few lines of code. A particularly compelling aspect of the Storage Access Framework from the point of view of the developer is that the underlying document provider selected by the user is completely transparent to the application. Once the storage functionality has been implemented using the framework within an application, it will work with all document providers without any code modifications.

60.2 Working with the Storage Access Framework

Android 4.4 introduced a new set of Intents designed to integrate the features of the Storage Access Framework into Android applications. These intents

display the Storage Access Framework picker user interface to the user and return the results of the interaction to the application via a call to the *onActivityResult()* method of the activity that launched the intent. When the *onActivityResult()* method is called, it is passed the Uri of the selected file together with a value indicating the success or otherwise of the operation.

The Storage Access Framework intents can be summarized as follows:

- **ACTION_OPEN_DOCUMENT** – Provides the user with access to the picker user interface so that files may be selected from the document providers configured on the device. Selected files are passed back to the application in the form of Uri objects.
- **ACTION_CREATE_DOCUMENT** – Allows the user to select a document provider, a location on that provider's storage and a file name for a new file. Once selected, the file is created by the Storage Access Framework and the Uri of that file returned to the application for further processing.

60.3 Filtering Picker File Listings

The files listed within the picker user interface when an intent is started may be filtered using a variety of options. Consider, for example, the following code to start an ACTION_OPEN_DOCUMENT intent:

```
private static final int OPEN_REQUEST_CODE = 41;

Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
startActivityForResult(intent, OPEN_REQUEST_CODE);
```

When executed, the above code will cause the picker user interface to be displayed, allowing the user to browse and select any files hosted by available document providers. Once a file has been selected by the user, a reference to that file will be provided to the application in the form of a Uri object. The application can then open the file using the *openFileDescriptor(Uri, String)* method. There is some risk, however, that not all files listed by a document provider can be opened in this way. The exclusion of such files within the picker can be achieved by modifying the intent using the *CATEGORY_OPENABLE* option. For example:

```
private static final int OPEN_REQUEST_CODE = 41;
```

```
Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
intent.addCategory(Intent.CATEGORY_OPENABLE);
startActivityForResult(intent, OPEN_REQUEST_CODE);
```

When the picker is now displayed, files which cannot be opened using the *openFileDescriptor()* method will be listed but not selectable by the user.

Another useful approach to filtering allows the files available for selection to be restricted by file type. This involves specifying the types of the files the application is able to handle. An image editing application might, for example, only want to provide the user with the option of selecting image files from the document providers. This is achieved by configuring the intent object with the MIME types of the files that are to be selectable by the user. The following code, for example, specifies that only image files are suitable for selection in the picker:

```
Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);

intent.addCategory(Intent.CATEGORY_OPENABLE);
intent.setType("image/*");
startActivityForResult(intent, OPEN_REQUEST_CODE);
```

This could be further refined to limit selection to JPEG images:

```
intent.setType("image/jpeg");
```

Alternatively, an audio player app might only be able to handle audio files:

```
intent.setType("audio/*");
```

The audio app might be limited even further in only supporting the playback of MP4 based audio files:

```
intent.setType("audio/mp4");
```

A wide range of MIME type settings are available for use when working with the Storage Access Framework, the more common of which can be found listed online at:

http://en.wikipedia.org/wiki/Internet_media_type#List_of_common_media_typ

60.4 Handling Intent Results

When an intent returns control to the application, it does so by calling the *onActivityResult()* method of the activity which started the intent. This method is passed the request code that was handed to the intent at launch time, a result code indicating whether or not the intent was successful and a

result data object containing the Uri of the selected file. The following code, for example, might be used as the basis for handling the results from the ACTION_OPEN_DOCUMENT intent outlined in the previous section:

```
public void onActivityResult(int requestCode, int resultCode,
        Intent resultData) {

       Uri currentUri = null;

       if (resultCode == Activity.RESULT_OK)
       {
              if (requestCode == OPEN_REQUEST_CODE)
              {
               if (resultData != null) {
                            currentUri = resultData.getData();
                     readFileContent(currentUri);
                    }
       }
}
```

The above method verifies that the intent was successful, checks that the request code matches that for a file open request and then extracts the Uri from the intent data. The Uri can then be used to read the content of the file.

60.5 Reading the Content of a File

The exact steps required to read the content of a file hosted by a document provider will depend to a large extent on the type of the file. The steps to read lines from a text file, for example, differ from those for image or audio files.

An image file can be assigned to a Bitmap object by extracting the file descriptor from the Uri object and then decoding the image into a BitmapFactory instance. For example:

```
ParcelFileDescriptor pFileDescriptor =
            getContentResolver().openFileDescriptor(uri, "r");

FileDescriptor fileDescriptor =
        pFileDescriptor.getFileDescriptor();

Bitmap image = BitmapFactory.decodeFileDescriptor(fileDescriptor);

pFileDescriptor.close();

myImageView.setImageBitmap(image);
```

Note that the file descriptor is opened in “r” mode. This indicates that the file is to be opened for reading. Other options are “w” for write access and “rwt” for read and write access, where existing content in the file is truncated by the new content.

Reading the content of a text file requires slightly more work and the use of an InputStream object. The following code, for example, reads the lines from a text file:

```
InputStream inputStream = getContentResolver().openInputStream(uri);

BufferedReader reader = new BufferedReader(new InputStreamReader(
                inputStream));
String readline;

while ((readline = reader.readLine()) != null) {
       // Do something with each line in the file
}
inputStream.close();
```

60.6 Writing Content to a File

Writing to an open file hosted by a document provider is similar to reading with the exception that an output stream is used instead of an input stream. The following code, for example, writes text to the output stream of the storage based file referenced by the specified Uri:

```
try{
       ParcelFileDescriptor pFileDescriptor =
this.getContentResolver().
                openFileDescriptor(uri, "w");

       FileOutputStream fileOutputStream =
             new
FileOutputStream(pFileDescriptor.getFileDescriptor());

       String textContent = "Some sample text";
       fileOutputStream.write(textContent.getBytes());
       fileOutputStream.close();
       pFileDescriptor.close();
} catch (FileNotFoundException e) {
       e.printStackTrace();
} catch (IOException e) {
       e.printStackTrace();
```

```
}
```

First, the file descriptor is extracted from the Uri, this time requesting write permission to the target file. The file descriptor is subsequently used to obtain a reference to the file's output stream. The content (in this example, some text) is then written to the output stream before the file descriptor and output stream are closed.

60.7 Deleting a File

Whether a file can be deleted from storage depends on whether or not the file's document provider supports deletion of the file. Assuming deletion is permitted, it may be performed on a designated Uri as follows:

```
if (DocumentsContract.deleteDocument(getContentResolver(), uri))
       // Deletion was successful
else
       // Deletion failed
```

60.8 Gaining Persistent Access to a File

When an application gains access to a file via the Storage Access Framework, the access will remain valid until the Android device on which the application is running is restarted. Persistent access to a specific file can be obtained by "taking" the necessary permissions for the Uri. The following code, for example, persists read and write permissions for the file referenced by the *fileUri* Uri instance:

```
final int takeFlags = intent.getFlags()
            & (Intent.FLAG_GRANT_READ_URI_PERMISSION
            | Intent.FLAG_GRANT_WRITE_URI_PERMISSION);

getContentResolver().takePersistableUriPermission(fileUri,
takeFlags);
```

Once the permissions for the file have been taken by the application, and assuming the Uri has been saved by the application, the user should be able to continue accessing the file after a device restart without the user having to reselect the file from the picker interface.

If, at any time, the persistent permissions are no longer required, they can be released via a call to the *releasePersistableUriPermission()* method of the content resolver:

```
final int releaseFlags = intent.getFlags()
            & (Intent.FLAG_GRANT_READ_URI_PERMISSION
```

```
            | Intent.FLAG_GRANT_WRITE_URI_PERMISSION);

getContentResolver().releasePersistableUriPermission(fileUri,
                            releaseFlags);
```

60.9 Summary

It is interesting to consider how perceptions of storage have changed in recent years. Once synonymous with high capacity internal hard disk drives, the term "storage" is now just as likely to refer to storage space hosted remotely in the cloud and accessed over an internet connection. This is increasingly the case with the wide adoption of resource constrained, "always-connected" mobile devices with minimal internal storage capacity.

The Android Storage Access Framework provides a simple mechanism for both users and application developers to seamlessly gain access to files stored in the cloud. Through the use of a set of intents introduced into Android 4.4 and a built-in user interface for selecting document providers and files, comprehensive cloud based storage can now be integrated into Android applications with a minimal amount of coding.