Sharing Files

Apps often have a need to offer one or more of their files to another app. For example, an image gallery may want to offer files to image editors, or a file management app may want to allow users to copy and paste files between areas in external storage. One way a sending app can share a file is to respond to a request from the receiving app.

In all cases, the only secure way to offer a file from your app to another app is to send the receiving app the file's content URI and grant temporary access permissions to that URI. Content URIs with temporary URI access permissions are secure because they apply only to the app that receives the URI, and they expire automatically. The Android [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) component provides the method[getUriForFile()](file:///G:\Studio\SDK\docs\reference\android\support\v4\content\FileProvider.html#getUriForFile(android.content.Context, java.lang.String, java.io.File)) for generating a file's content URI.

If you want to share small amounts of text or numeric data between apps, you should send an [Intent](file:///G:\\Studio\\SDK\\docs\\reference\\android\\content\\Intent.html)that contains the data. To learn how to send simple data with an [Intent](file:///G:\Studio\SDK\docs\reference\android\content\Intent.html), see the training class[Sharing Simple Data](file:///G:\Studio\SDK\docs\training\sharing\index.html).

This class explains how to securely share files from your app to another app using content URIs generated by the Android [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) component and temporary permissions that you grant to the receiving app for the content URI.

# Setting Up File Sharing

To securely offer a file from your app to another app, you need to configure your app to offer a secure handle to the file, in the form of a content URI. The Android [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) component generates content URIs for files, based on specifications you provide in XML. This lesson shows you how to add the default implementation of[FileProvider](file:///G:\Studio\SDK\docs\reference\android\support\v4\content\FileProvider.html) to your app, and how to specify the files you want to offer to other apps.

**Note:** The [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) class is part of the [v4 Support Library](file:///G:\Studio\SDK\docs\tools\support-library\features.html#v4). For information about including this library in your application, see [Support Library Setup](file:///G:\Studio\SDK\docs\tools\support-library\setup.html).

## Specify the FileProvider

Defining a [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) for your app requires an entry in your manifest. This entry specifies the authority to use in generating content URIs, as well as the name of an XML file that specifies the directories your app can share.

The following snippet shows you how to add to your manifest the [<provider>](file:///G:\Studio\SDK\docs\guide\topics\manifest\provider-element.html) element that specifies the [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) class, the authority, and the XML file name:

<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.example.myapp">  
    <application  
        ...>  
        <provider  
            android:name="android.support.v4.content.FileProvider"  
            android:authorities="com.example.myapp.fileprovider"  
            android:grantUriPermissions="true"  
            android:exported="false">  
            <meta-data  
                android:name="android.support.FILE\_PROVIDER\_PATHS"  
                android:resource="@xml/filepaths" />  
        </provider>  
        ...  
    </application>  
</manifest>

In this example, the [android:authorities](file:///G:\\Studio\\SDK\\docs\\guide\\topics\\manifest\\provider-element.html" \l "auth) attribute specifies the URI authority that you want to use for content URIs generated by the [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html). In the example, the authority iscom.example.myapp.fileprovider. For your own app, specify an authority consisting of the app's[android:package](file:///G:\Studio\SDK\docs\guide\topics\manifest\manifest-element.html#package) value with the string "fileprovider" appended to it. To learn more about the authority value, **see the topic**[**Content URIs**](file:///G:\Studio\SDK\docs\guide\topics\providers\content-provider-basics.html#ContentURIs)**and the documentation for the [android:authorities](file:///G:\\Studio\\SDK\\docs\\guide\\topics\\manifest\\provider-element.html" \l "auth) attribute.**

The [<meta-data>](file:///G:\\Studio\\SDK\\docs\\guide\\topics\\manifest\\meta-data-element.html) child element of the [<provider>](file:///G:\Studio\SDK\docs\guide\topics\manifest\provider-element.html) points to an XML file that specifies the directories you want to share. The android:resource attribute is the path and name of the file, without the .xml extension.The contents of this file are described in the next section.

## Specify Sharable Directories

Once you have added the [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) to your app manifest, you need to specify the directories that contain the files you want to share. To specify the directories, start by creating the filefilepaths.xml in the res/xml/ subdirectory of your project. In this file, specify the directories by adding an XML element for each directory. The following snippet shows you an example of the contents of res/xml/filepaths.xml. The snippet also demonstrates how to share a subdirectory of the files/ directory in your internal storage area:

<paths>  
    <files-path path="images/" name="myimages" />  
</paths>

In this example, the <files-path> tag shares directories within the files/ directory of your app's internal storage. The path attribute shares the images/ subdirectory of files/. Thename attribute tells the [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) to **add the path segment myimages to content URIs for files in the files/images/ subdirectory**.

The <paths> element can have multiple children, each specifying a different directory to share. In addition to the <files-path> element, you can use the <external-path> element to share directories in external storage, and the <cache-path> element to share directories in your internal cache directory. To learn more about the child elements that specify shared directories, see the [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) reference documentation.

**Note:** The XML file is the only way you can specify the directories you want to share; you can't programmatically add a directory.

You now have a complete specification of a [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) that generates content URIs for files in the files/ directory of your app's internal storage or for files in subdirectories of files/. When your app generates a content URI for a file, it contains the authority specified in the[<provider>](file:///G:\Studio\SDK\docs\guide\topics\manifest\provider-element.html) element (com.example.myapp.fileprovider), the path myimages/, and the name of the file.

For example, if you define a [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) according to the snippets in this lesson, and you request a content URI for the file **default\_image.jpg,** [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) returns the following URI:

**content://com.example.myapp.fileprovider/myimages/default\_image.jpg**

# Sharing a File

Once you have set up your app to share files using content URIs, you can respond to other apps' requests for those files. One way to respond to these requests is to provide a file selection interface from the server app that other applications can invoke. This approach allows a client application to let users select a file from the server app and then receive the selected file's content URI.

This lesson shows you how to create a file selection [Activity](file:///G:\Studio\SDK\docs\reference\android\app\Activity.html) in your app that responds to requests for files.

## Receive File Requests

To receive requests for files from client apps and respond with a content URI, your app should provide a file selection [Activity](file:///G:\Studio\SDK\docs\reference\android\app\Activity.html). Client apps start this[Activity](file:///G:\Studio\SDK\docs\reference\android\app\Activity.html) by calling [startActivityForResult()](file:///G:\\Studio\\SDK\\docs\\reference\\android\\app\\Activity.html" \l "startActivityForResult(android.content.Intent, int)) with an [Intent](file:///G:\Studio\SDK\docs\reference\android\content\Intent.html) containing the action[ACTION\_PICK](file:///G:\Studio\SDK\docs\reference\android\content\Intent.html#ACTION_PICK). When the client app calls [startActivityForResult()](file:///G:\\Studio\\SDK\\docs\\reference\\android\\app\\Activity.html" \l "startActivityForResult(android.content.Intent, int)), your app can return a result to the client app, in the form of a content URI for the file the user selected.

To learn how to implement a request for a file in a client app, see the lesson [Requesting a Shared File](file:///G:\Studio\SDK\docs\training\secure-file-sharing\request-file.html).

## Create a File Selection Activity

To set up the file selection [Activity](file:///G:\Studio\SDK\docs\reference\android\app\Activity.html), start by specifying the [Activity](file:///G:\Studio\SDK\docs\reference\android\app\Activity.html) in your manifest, along with an intent filter that matches the action [ACTION\_PICK](file:///G:\Studio\SDK\docs\reference\android\content\Intent.html#ACTION_PICK) and the categories [CATEGORY\_DEFAULT](file:///G:\\Studio\\SDK\\docs\\reference\\android\\content\\Intent.html" \l "CATEGORY_DEFAULT)and [CATEGORY\_OPENABLE](file:///G:\Studio\SDK\docs\reference\android\content\Intent.html#CATEGORY_OPENABLE). Also add MIME type filters for the files your app serves to other apps. The following snippet shows you how to specify the new [Activity](file:///G:\Studio\SDK\docs\reference\android\app\Activity.html) and intent filter:

<manifest xmlns:android="http://schemas.android.com/apk/res/android">  
    ...  
        <application>  
        ...  
            <activity  
                android:name=".FileSelectActivity"  
                android:label="@"File Selector" >  
                <intent-filter>  
                    <action  
                        android:name="android.intent.action.PICK"/>  
                    <category  
                        android:name="android.intent.category.DEFAULT"/>  
                    <category  
                        android:name="android.intent.category.OPENABLE"/>  
                    <data android:mimeType="text/plain"/>  
                    <data android:mimeType="image/\*"/>  
                </intent-filter>  
            </activity>

### Define the file selection Activity in code

Next, define an [Activity](file:///G:\Studio\SDK\docs\reference\android\app\Activity.html) subclass that displays the files available from your app'sfiles/images/ directory in internal storage and allows the user to pick the desired file. The following snippet demonstrates how to define this [Activity](file:///G:\Studio\SDK\docs\reference\android\app\Activity.html) and respond to the user's selection:

public class MainActivity extends Activity {  
    // The path to the root of this app's internal storage  
    private File mPrivateRootDir;  
    // The path to the "images" subdirectory  
    private File mImagesDir;  
    // Array of files in the images subdirectory  
    File[] mImageFiles;  
    // Array of filenames corresponding to mImageFiles  
    String[] mImageFilenames;  
    // Initialize the Activity  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        ...  
        // Set up an Intent to send back to apps that request a file  
        mResultIntent =  
                new Intent("com.example.myapp.ACTION\_RETURN\_FILE");  
        // Get the files/ subdirectory of internal storage  
        mPrivateRootDir = getFilesDir();  
        // Get the files/images subdirectory;  
        mImagesDir = new File(mPrivateRootDir, "images");  
        // Get the files in the images subdirectory  
        mImageFiles = mImagesDir.listFiles();  
        // Set the Activity's result to null to begin with  
        setResult(Activity.RESULT\_CANCELED, null);  
        /\*  
         \* Display the file names in the ListView mFileListView.  
         \* Back the ListView with the array mImageFilenames, which  
         \* you can create by iterating through mImageFiles and  
         \* calling File.getAbsolutePath() for each File  
         \*/  
         ...  
    }  
    ...  
}

## Respond to a File Selection

Once a user selects a shared file, your application must determine what file was selected and then generate a content URI for the file. Since the [Activity](file:///G:\Studio\SDK\docs\reference\android\app\Activity.html) displays the list of available files in a[ListView](file:///G:\Studio\SDK\docs\reference\android\widget\ListView.html), when the user clicks a file name the system calls the method [onItemClick()](file:///G:\\Studio\\SDK\\docs\\reference\\android\\widget\\AdapterView.OnItemClickListener.html" \l "onItemClick(android.widget.AdapterView<?>, android.view.View, int, long)), in which you can get the selected file.

In [onItemClick()](file:///G:\\Studio\\SDK\\docs\\reference\\android\\widget\\AdapterView.OnItemClickListener.html" \l "onItemClick(android.widget.AdapterView<?>, android.view.View, int, long)), get a [File](file:///G:\Studio\SDK\docs\reference\java\io\File.html) object for the file name of the selected file and pass it as an argument to [getUriForFile()](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html" \l "getUriForFile(android.content.Context, java.lang.String, java.io.File)), along with（连同。。一起） the authority that you specified in the [<provider>](file:///G:\Studio\SDK\docs\guide\topics\manifest\provider-element.html)element for the [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html). The resulting content URI contains the authority, a path segment corresponding to the file's directory (as specified in the XML meta-data), and the name of the file including its extension. How [FileProvider](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html) maps directories to path segments based on XML meta-data is described in the section [Specify Sharable Directories](file:///G:\Studio\SDK\docs\training\secure-file-sharing\setup-sharing.html#DefineMetaData).

The following snippet shows you how to detect the selected file and get a content URI for it:

    protected void onCreate(Bundle savedInstanceState) {  
        ...  
        // Define a listener that responds to clicks on a file in the ListView  
        mFileListView.setOnItemClickListener(  
                new AdapterView.OnItemClickListener() {  
            @Override  
            /\*  
             \* When a filename in the ListView is clicked, get its  
             \* content URI and send it to the requesting app  
             \*/  
            public void onItemClick(AdapterView<?> adapterView,  
                    View view,  
                    int position,  
                    long rowId) {  
                /\*  
                 \* Get a File for the selected file name.  
                 \* Assume that the file names are in the  
                 \* mImageFilename array.  
                 \*/  
                File requestFile = new File(mImageFilename[position]);  
                /\*  
                 \* Most file-related method calls need to be in  
                 \* try-catch blocks.  
                 \*/  
                // Use the FileProvider to get a content URI  
                try {  
                    fileUri = FileProvider.getUriForFile(  
                            MainActivity.this,  
                            "com.example.myapp.fileprovider",  
                            requestFile);  
                } catch (IllegalArgumentException e) {  
                    Log.e("File Selector",  
                          "The selected file can't be shared: " +  
                          clickedFilename);  
                }  
                ...  
            }  
        });  
        ...  
    }

Remember that you can only generate content URIs for files that reside in a directory you've specified in the meta-data file that contains the <paths> element, as described in the section[Specify Sharable Directories](file:///G:\Studio\SDK\docs\training\secure-file-sharing\setup-sharing.html#DefineMetaData). If you call [getUriForFile()](file:///G:\\Studio\\SDK\\docs\\reference\\android\\support\\v4\\content\\FileProvider.html" \l "getUriForFile(android.content.Context, java.lang.String, java.io.File)) for a [File](file:///G:\Studio\SDK\docs\reference\java\io\File.html) in a path that you haven't specified, you receive an [IllegalArgumentException](file:///G:\\Studio\\SDK\\docs\\reference\\java\\lang\\IllegalArgumentException.html).

## Grant Permissions for the File

Now that you have a content URI for the file you want to share with another app, you need to allow the client app to access the file. To allow access, grant permissions to the client app by adding the content URI to an [Intent](file:///G:\Studio\SDK\docs\reference\android\content\Intent.html) and then setting permission flags on the [Intent](file:///G:\Studio\SDK\docs\reference\android\content\Intent.html). The permissions you grant are temporary and expire automatically when the receiving app's task stack is finished.

The following code snippet shows you how to set read permission for the file:

    protected void onCreate(Bundle savedInstanceState) {  
        ...  
        // Define a listener that responds to clicks in the ListView  
        mFileListView.setOnItemClickListener(  
                new AdapterView.OnItemClickListener() {  
            @Override  
            public void onItemClick(AdapterView<?> adapterView,  
                    View view,  
                    int position,  
                    long rowId) {  
                ...  
                if (fileUri != null) {  
                    // Grant temporary read permission to the content URI  
                    mResultIntent.addFlags(  
                        Intent.FLAG\_GRANT\_READ\_URI\_PERMISSION);  
                }  
                ...  
             }  
             ...  
        });  
    ...  
    }

**Caution: Calling [setFlags()](file:///G:\\Studio\\SDK\\docs\\reference\\android\\content\\Intent.html" \l "setFlags(int)) is the only way to securely grant access to your files using temporary access permissions**. Avoid calling [Context.grantUriPermission()](file:///G:\\Studio\\SDK\\docs\\reference\\android\\content\\Context.html" \l "grantUriPermission(java.lang.String, android.net.Uri, int)) method for a file's content URI, since this method grants access that you can only revoke by calling[Context.revokeUriPermission()](file:///G:\Studio\SDK\docs\reference\android\content\Context.html#revokeUriPermission(android.net.Uri, int)).

## Share the File with the Requesting App

To share the file with the app that requested it, pass the [Intent](file:///G:\Studio\SDK\docs\reference\android\content\Intent.html) containing the content URI and permissions to [setResult()](file:///G:\\Studio\\SDK\\docs\\reference\\android\\app\\Activity.html" \l "setResult(int)). When the [Activity](file:///G:\Studio\SDK\docs\reference\android\app\Activity.html) you have just defined is finished, the system sends the [Intent](file:///G:\Studio\SDK\docs\reference\android\content\Intent.html) containing the content URI to the client app. The following code snippet shows you how to do this:

    protected void onCreate(Bundle savedInstanceState) {  
        ...  
        // Define a listener that responds to clicks on a file in the ListView  
        mFileListView.setOnItemClickListener(  
                new AdapterView.OnItemClickListener() {  
            @Override  
            public void onItemClick(AdapterView<?> adapterView,  
                    View view,  
                    int position,  
                    long rowId) {  
                ...  
                if (fileUri != null) {  
                    ...  
                    // Put the Uri and MIME type in the result Intent  
                    mResultIntent.setDataAndType(  
                            fileUri,  
                            getContentResolver().getType(fileUri));  
                    // Set the result  
                    MainActivity.this.setResult(Activity.RESULT\_OK,  
                            mResultIntent);  
                    } else {  
                        mResultIntent.setDataAndType(null, "");  
                        MainActivity.this.setResult(RESULT\_CANCELED,  
                                mResultIntent);  
                    }  
                }  
        });

Provide users with an way to return immediately to the client app once they have chosen a file. One way to do this is to provide a checkmark or **Done** button. Associate a method with the button using the button's [android:onClick](file:///G:\\Studio\\SDK\\docs\\reference\\android\\view\\View.html" \l "attr_android:onClick) attribute. In the method, call [finish()](file:///G:\Studio\SDK\docs\reference\android\app\Activity.html#finish()). For example:

    public void onDoneClick(View v) {  
        // Associate a method with the Done button  
        finish();  
    }