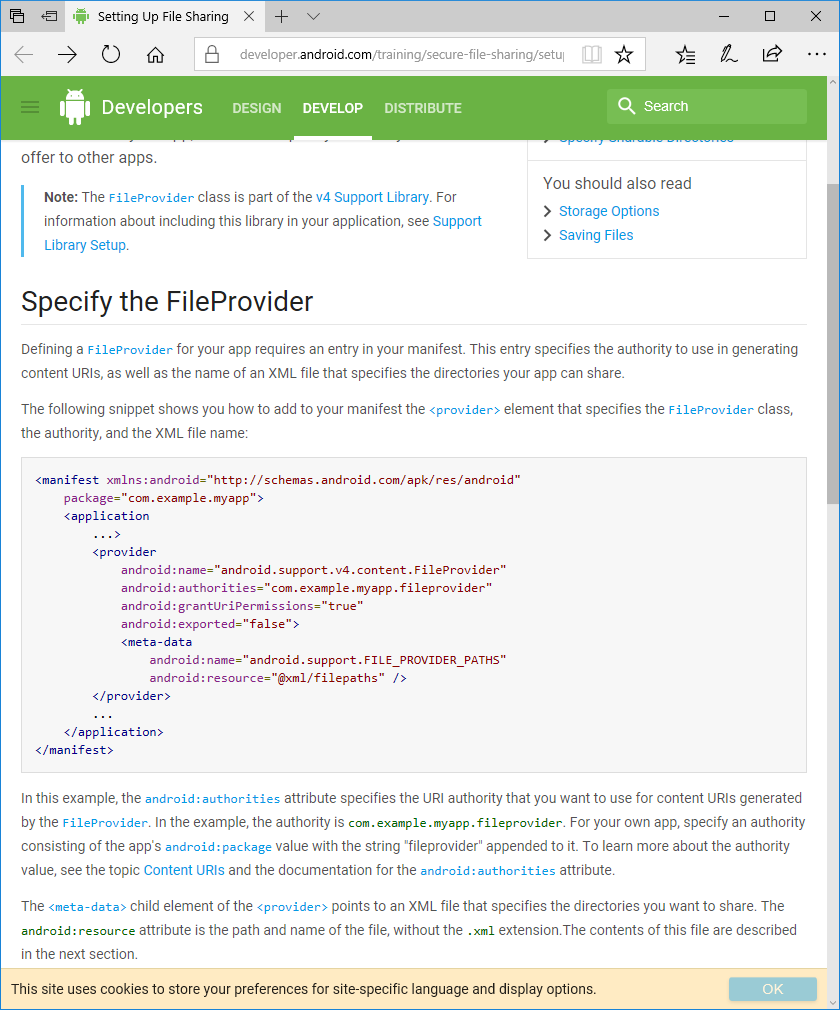
**File Sharing**

**File Sharing Set Up**

**Specifying the FileProvider**

Defining a FileProvider 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> element that specifies the FileProvider class, the authority, and the XML file name:

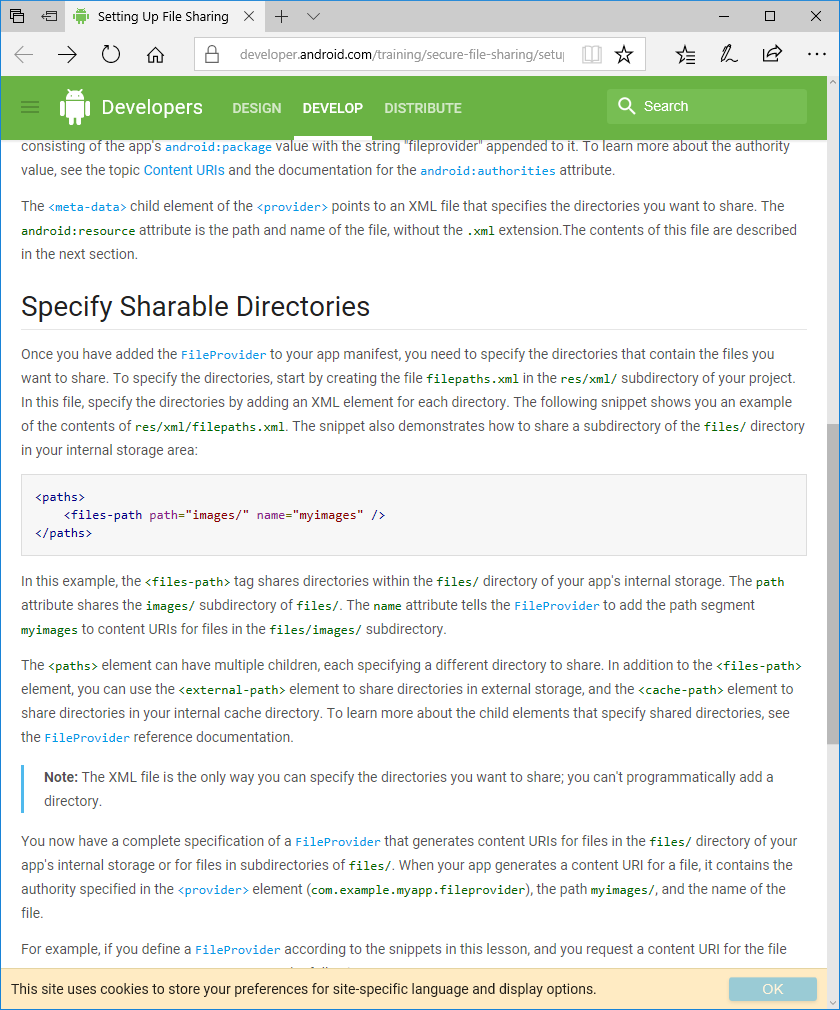


In this example, the android:authorities attribute specifies the URI authority that you want to use for content URIs generated by the FileProvider. In the example, the authority is com.example.myapp.fileprovider. For your own app, specify an authority consisting of the app's android:package value with the string "fileprovider" appended to it. To learn more about the authority value, see the topic Content URIs and the documentation for the android:authorities attribute.

The <meta-data> child element of the <provider> 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 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 file filepaths.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:



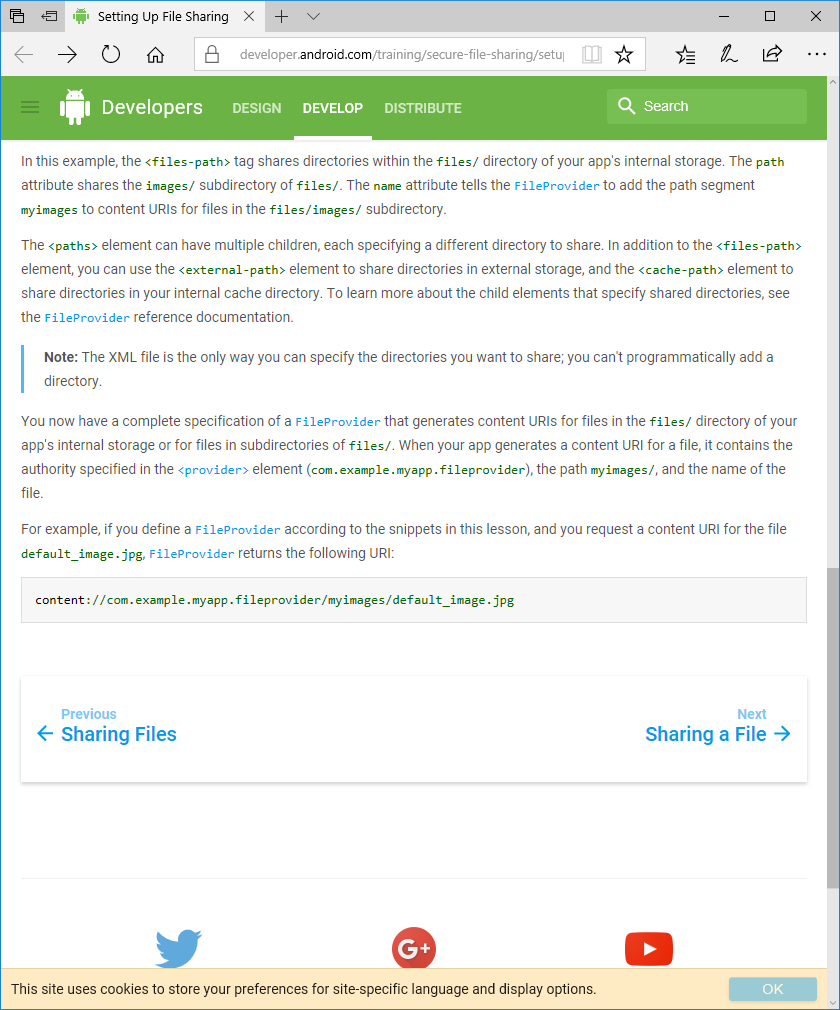
In this example, the <files-path> tag shares directories within the files/ directory of your app's internal storage. The path attribute shares the images/ subdirectory of files/. The name attribute tells the FileProvider to add the path segment myimages to content URIs for files in the files/images/ 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 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 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> element (com.example.myapp.fileprovider), the path myimages/, and the name of the file.

For example, if you define a FileProvider according to the snippets in this lesson, and you request a content URI for the file default\_image.jpg, FileProvider returns the following URI:



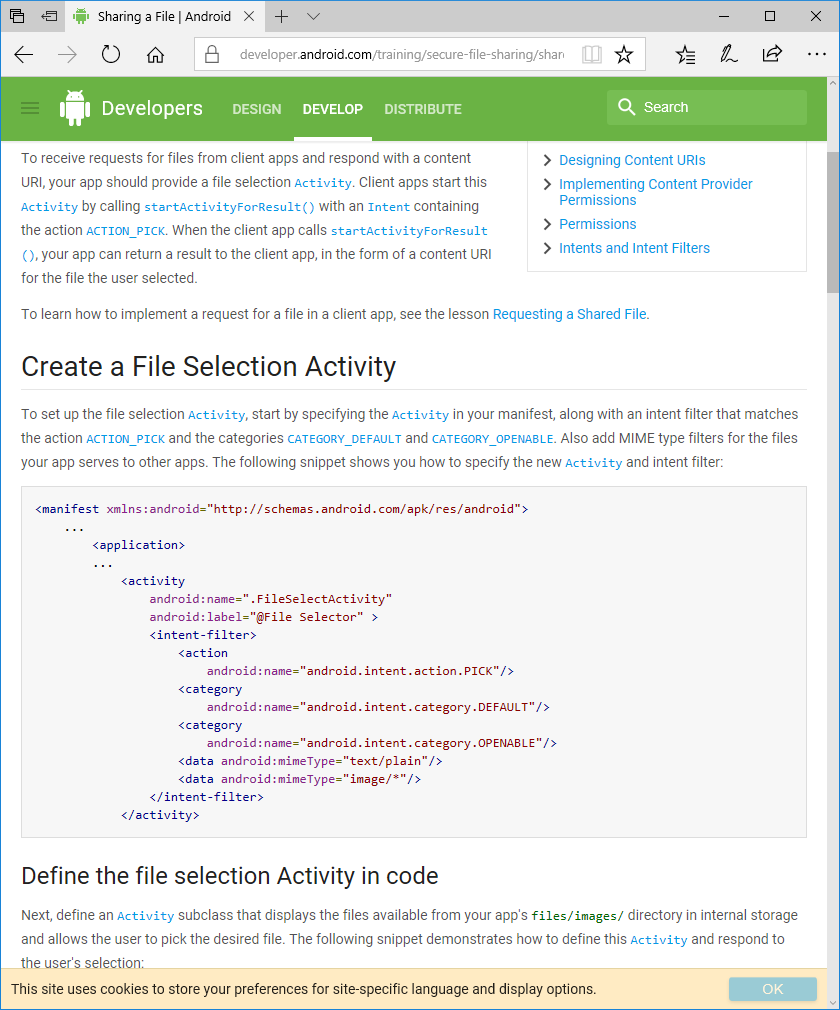
**Sharing a file**

**Receiving File Requests**

To receive requests for files from client apps and respond with a content URI, your app should provide a file selection Activity. Client apps start this Activity by calling startActivityForResult() with an Intent containing the action ACTION\_PICK. When the client app calls startActivityForResult(), your app can return a result to the client app, in the form of a content URI for the file the user selected.

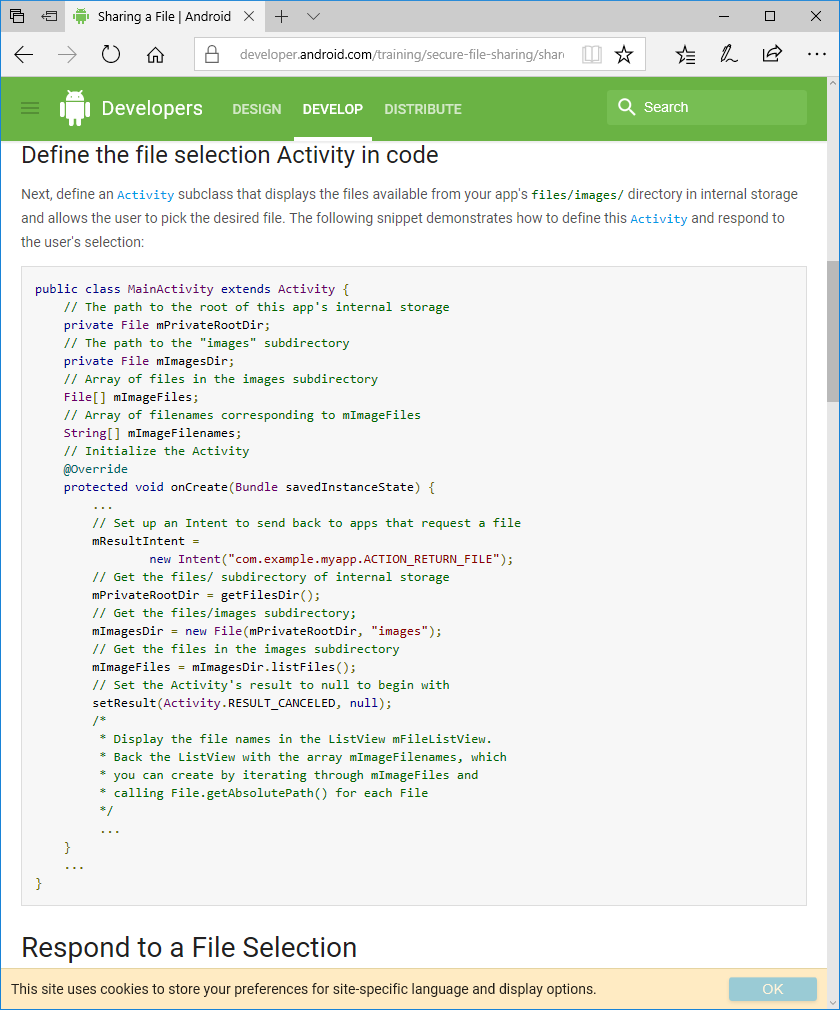
**Create a File Selection Activity**

To set up the file selection Activity, start by specifying the Activity in your manifest, along with an intent filter that matches the action ACTION\_PICK and the categories CATEGORY\_DEFAULT and 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 and intent filter:



**Respond to a file selection**

Next, define an Activity subclass that displays the files available from your app's files/images/ directory in internal storage and allows the user to pick the desired file. The following snippet demonstrates how to define this Activity and respond to the user's 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 displays the list of available files in a ListView, when the user clicks a file name the system calls the method onItemClick(), in which you can get the selected file.

When using an intent to send a file's URI from one app to another, you must be careful to get an URI that other apps can read. Doing so on devices running Android 6.0 (API level 23) and later requires special care because of changes to the permissions model in that version of Android, particularly READ\_EXTERNAL\_STORAGE's becoming a dangerous permission, which the receiving app might lack.

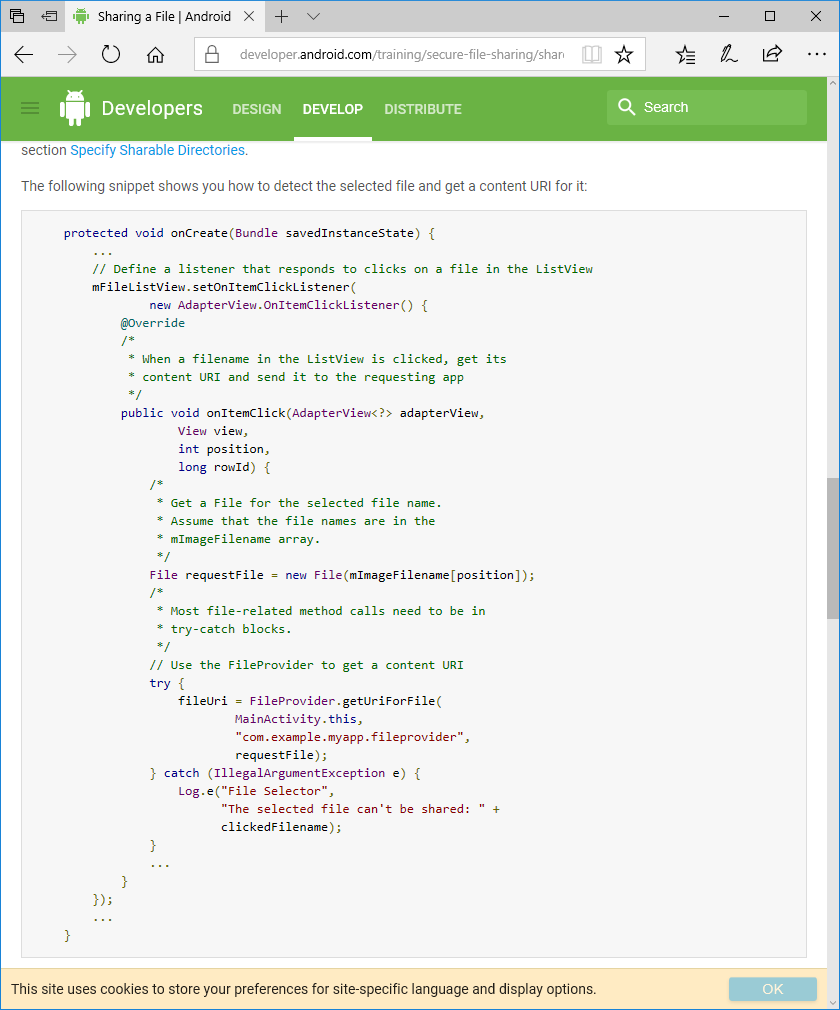
With these considerations in mind, we recommend that you avoid using Uri.fromFile(), which presents several drawbacks. This method:

* Does not allow file sharing across profiles.
* Requires that your app have WRITE\_EXTERNAL\_STORAGE permission on devices running Android 4.4 (API level 19) or lower.
* Requires that receiving apps have the READ\_EXTERNAL\_STORAGE permission, which will fail on important share targets, like Gmail, that don't have that permission.

Instead of using Uri.fromFile(), you can use URI permissions to grant other apps access to specific URIs. While URI permissions don’t work on file:// URIs generated by Uri.fromFile(), they do work on URIs associated with Content Providers. The FileProvider API can help you create such URIs. This approach also works with files that are not in external storage, but in the local storage of the app sending the intent.

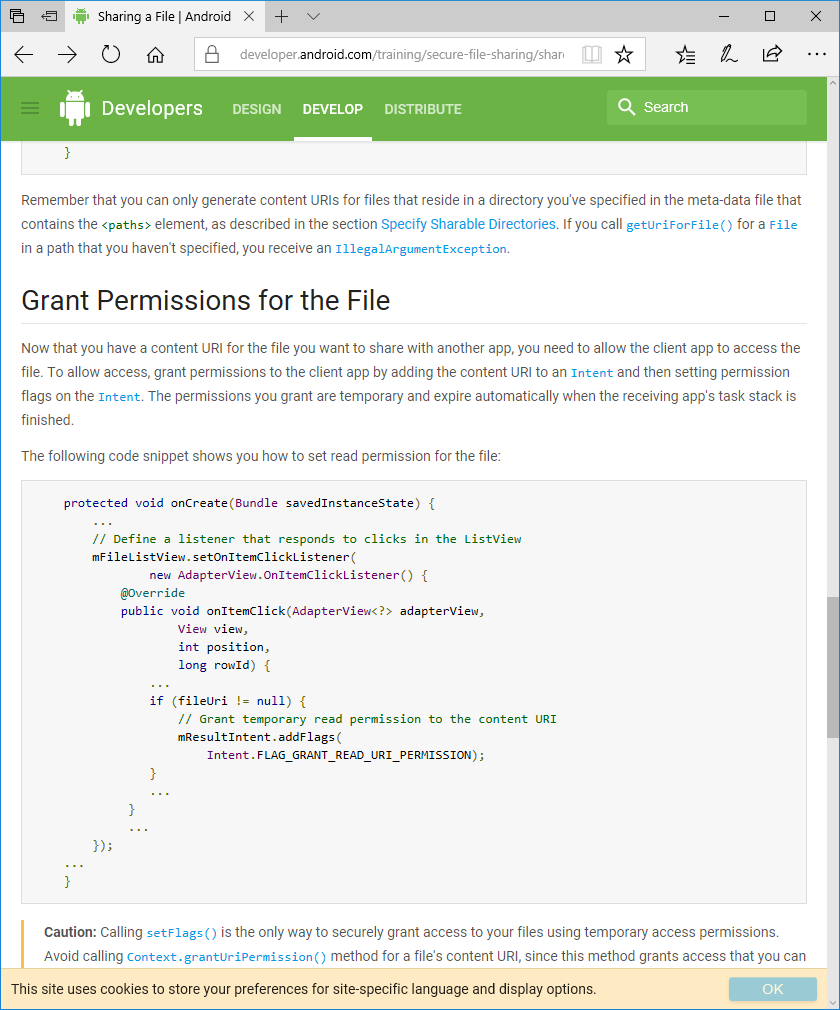
In onItemClick(), get a File object for the file name of the selected file and pass it as an argument to getUriForFile(), along with the authority that you specified in the <provider> element for the FileProvider. The resulting content URI contains the authority, a path segment corresponding to the file's directory (as specified in the XML meta-data), and the name of the file including its extension. How FileProvider maps directories to path segments based on XML meta-data is described in the section Specify Sharable Directories.

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



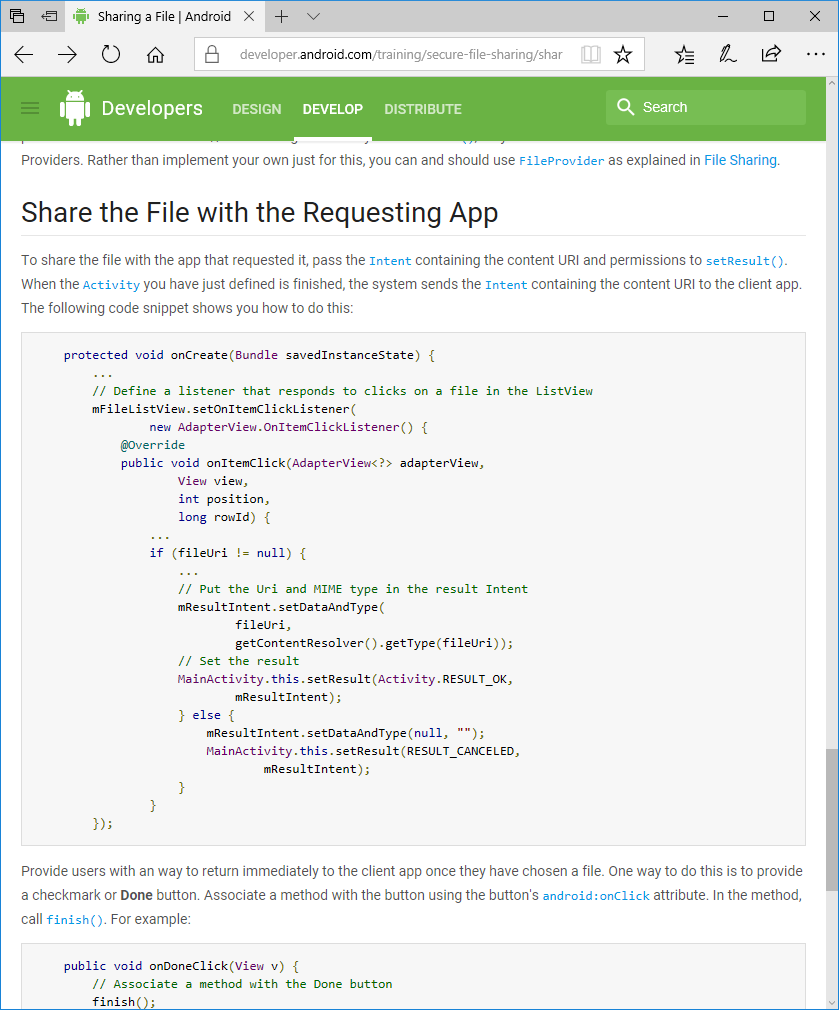
**Granting permissions for a file**

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 and then setting permission flags on the Intent. The permissions you grant are temporary and expire automatically when the receiving app's task stack is finished.

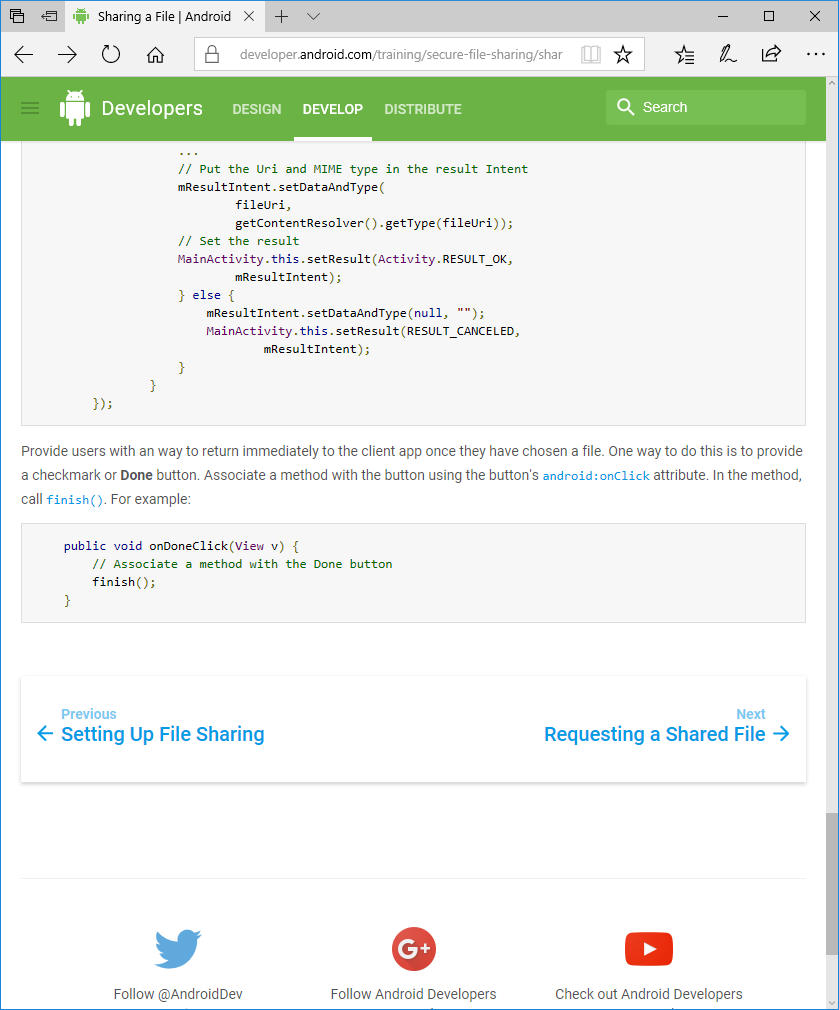


**Sharing files with the requesting app**

To share the file with the app that requested it, pass the Intent containing the content URI and permissions to setResult(). When the Activity you have just defined is finished, the system sends the Intent containing the content URI to the client app. The following code snippet shows you how to do this:



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 attribute. In the method, call finish(). For example:

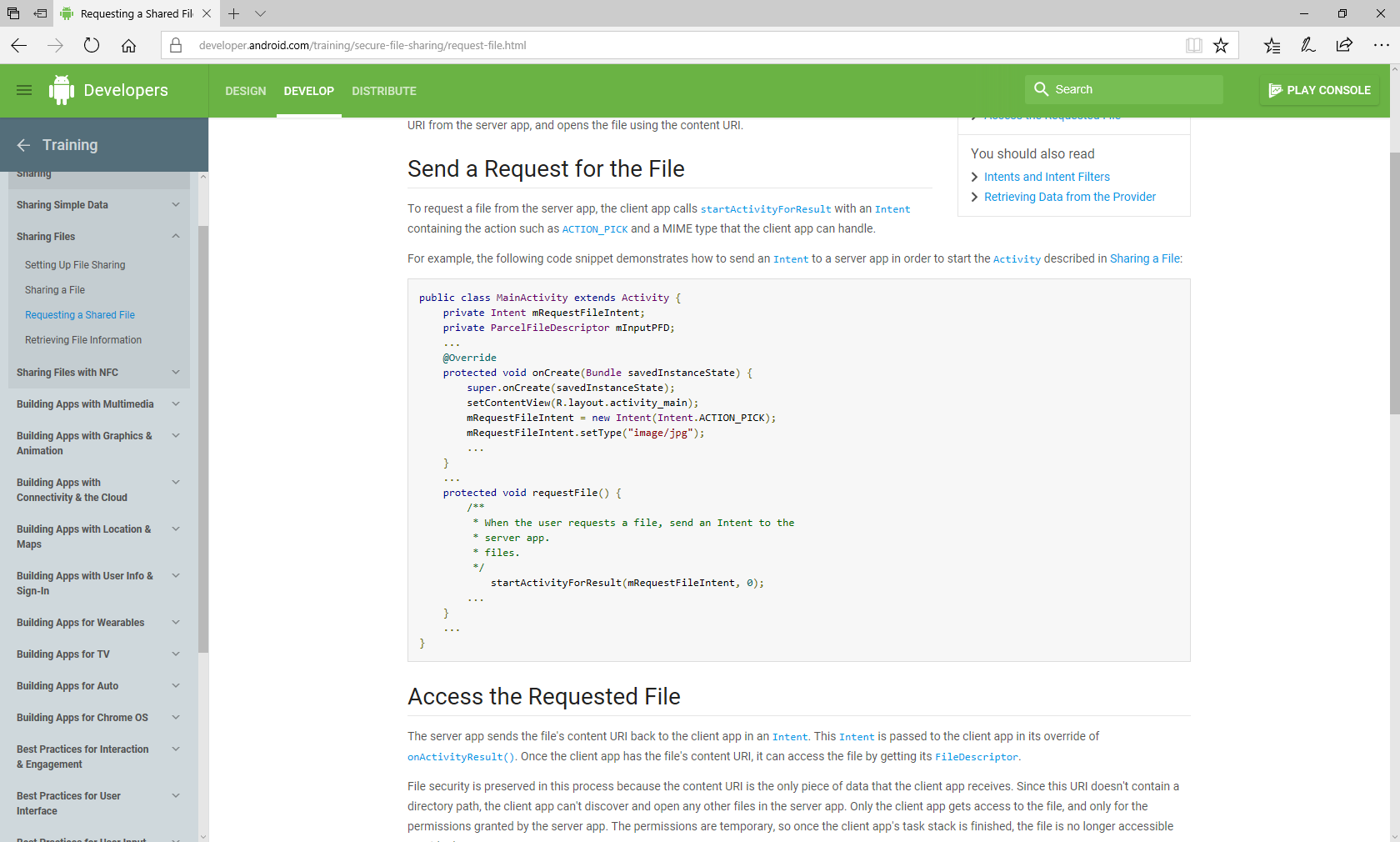


**Requesting a shared file**

**Sending a request for the file**

To request a file from the server app, the client app calls startActivityForResult with an Intent containing the action such as ACTION\_PICK and a MIME type that the client app can handle.

For example, the following code snippet demonstrates how to send an Intent to a server app in order to start the Activity described in Sharing a File:

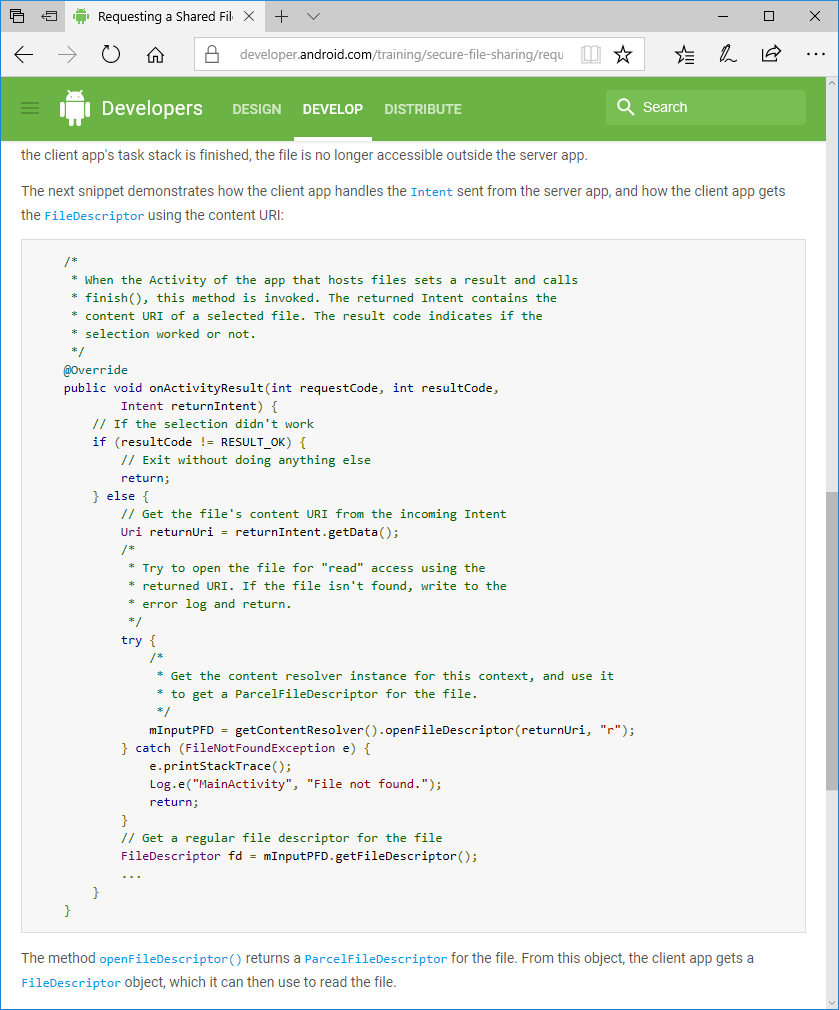


**Accessing the requested file**

The server app sends the file's content URI back to the client app in an Intent. This Intent is passed to the client app in its override of onActivityResult(). Once the client app has the file's content URI, it can access the file by getting its FileDescriptor.

File security is preserved in this process because the content URI is the only piece of data that the client app receives. Since this URI doesn't contain a directory path, the client app can't discover and open any other files in the server app. Only the client app gets access to the file, and only for the permissions granted by the server app. The permissions are temporary, so once the client app's task stack is finished, the file is no longer accessible outside the server app.

The next snippet demonstrates how the client app handles the Intent sent from the server app, and how the client app gets the FileDescriptor using the content URI:



The method openFileDescriptor() returns a ParcelFileDescriptor for the file. From this object, the client app gets a FileDescriptor object, which it can then use to read the file.

**File MIME (multipurpose internet mail extensions)**

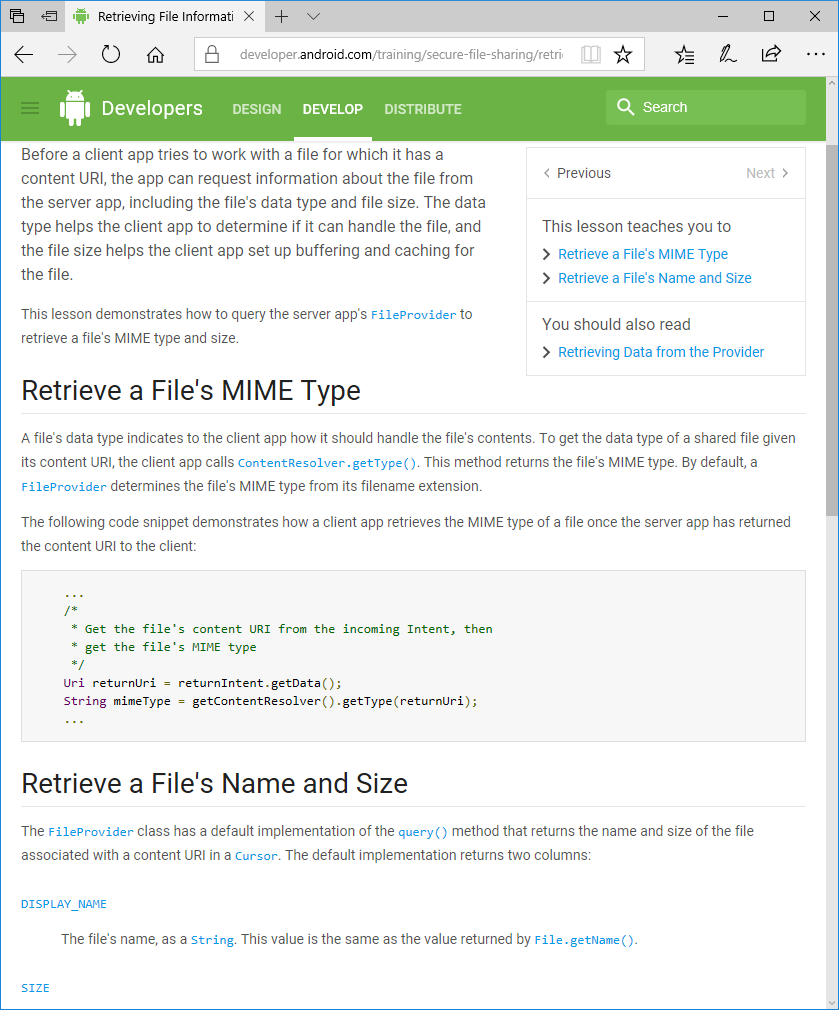
Before a client app tries to work with a file for which it has a content URI, the app can request information about the file from the server app, including the file's data type and file size. The data type helps the client app to determine if it can handle the file, and the file size helps the client app set up buffering and caching for the file.

**Retrieving file information**

**Retrieve a files MIME (multipurpose internet mail extensions)**

A file's data type indicates to the client app how it should handle the file's contents. To get the data type of a shared file given its content URI, the client app calls ContentResolver.getType(). This method returns the file's MIME type. By default, a FileProvider determines the file's MIME type from its filename extension.

The following code snippet demonstrates how a client app retrieves the MIME type of a file once the server app has returned the content URI to the client:



**Retrieve a files name and size**

The FileProvider class has a default implementation of the query() method that returns the name and size of the file associated with a content URI in a Cursor. The default implementation returns two columns:

DISPLAY\_NAME

The file's name, as a String. This value is the same as the value returned by File.getName().

SIZE

The size of the file in bytes, as a long This value is the same as the value returned by File.length()

The client app can get both the DISPLAY\_NAME and SIZE for a file by setting all of the arguments of query() to null except for the content URI. For example, this code snippet retrieves a file's DISPLAY\_NAME and SIZE and displays each one in separate TextView:

