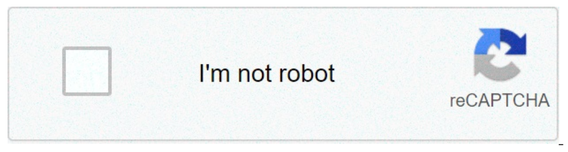

Continue

How to open pdf file android

Android 4.4 (level 19 API) introduces framework access storage (SAF). The SAF makes it easy for users to search and open documents, images and other files in all their favorite document storage providers. A standard, easy to use user interface allows users to browse files and recents access consistently in all applications and suppliers. Cloud or local storage services can participate in this ecosystem through the implementation of a DocumentSprovider that contains their services. Customer applications that need access to the documents of a provider can integrate with the SAF with a few lines of code. The SAF includes the following: document provider a content provider that allows a storage service (such as Google Drive) to reveal the files that manages. A document provider is implemented as a subclass of the DocumentSprovider class. The document-supplier scheme is based on a hierarchy of traditional files, even if as your document provider physically stores data depends on you. The Android platform includes several built-in document providers, such as downloads, images and videos. Customer appears a personalized application that recalls Action_Create_Document, Action_Open_Document, and Action_Open_Document_Tree Actions intent and receives files returned by document providers. PickerÃ ¢ A user interface system that allows users to access users from all providers of documents that meet the App customer search criteria. Some of the features offered by SAF are as follows: Allows users browse content from all document providers, not just a single application. It makes it possible for your application to have long-term, persistent access to the ownership documents of a document supplier. Through this access users can add, edit, save and delete files on the supplier. Supports more user accounts and transient roots such as USB storage providers, which appear only if the unit is connected. Panoramic centers The SAF around a content provider that is a subclass of the DocumentSprovider class. Within a document provider, data is structured as a traditional file hierarchy: Figure 1. Provider data template document. A root points to a single document, which then starts the fan-out of the entire tree. Note the following: each document provider reports one or more 'roots', which are starting points in exploring a document tree. Each root has a unique column_root_id, and that points to a document (a directory) representing the content within that hierarchy. The roots are design dynamics to support use cases such as multiple accounts, transient USB storage devices, or User Login / Logout. Under each root is a single document. That the points of the document at 1 to n documents, each of which in turn can point to 1 to n documents. Each storage backend Surfaces Single files and directories from them with column_document_id unique. Document IDs must be unique and not to change once issued, since they are used for persistent URI subsidies throughout the device restart. The documents can be either an openable file (with a specific MIME type), or a directory containing the supplementary documentation (with the MIME MIME_TYPE_DIR type). Each document can have several capacities, as described by Column_Flags. For example, flag_supports_write, flag_supports_delete and flag_supports_thumbnail. The same column_document_id can be included in multiple directories. Flow control as mentioned above, the document supplier data model is based on a traditional file hierarchy. However, data can be stored As you like, as long as you can access you using DocumentsProvider API. For example, you can use Cloud-based tag storage for your data. Figure 2 shows how a photo application could use the access SAF stored data: Figure 2. Luggage Access framework access Note the following: In the SAF, suppliers and customers do not interact directly. A client requires permission to interact with the files (which is, to read, edit, create or delete files). Interaction starts when a program (in this example, an app photo) shoots the action_open_document intention or or The intent can include filters to further refine criteriaÃ ¢ for example, "Give me all the openable files that have the MIME type 'image'." Once the intent burns, the system selector goes to each registered supplier and shows the user the corresponding content roots. The selector offers users a standard interface for access to documents, even if the suppliers of underlying documents can be very different. For example, Figure 2 shows a Google Drive supplier, a USB provider, and a cloud provider. Figure 3 shows a collector in which a user search user selected the Download folder. It also shows all the roots available for the client application. Figure 3. Selector After the user Select the Download folder, the images are displayed. Figure 4 shows the result of this process. The user can now interact with these images in the way the supplier and customer support app. Figure 4. Images stored in the Download folder, as seen in the system selector Write client application on Android 4.3 and lower, if you want your application to retrieve a file from another application, you must invoke such an intent as action_pick or Action_Get_Content. The user must select a single application from which to take a file and the selected application must provide a user interface for the user to navigate and choose between the available files. On Android 4.4 (API level 19) and higher, you have the additional possibility to use the Action_Open_Document intent, which shows a controlled controlled picker system that allows the user to browse all the files that other applications have made available. From this single user interface, the user can select a file from any one of the supported applications. On Android 5.0 (API level 21) and higher, you can also use the Action_Open_Document_Tree intent, which allows the user to choose a directory for a client application for access. Note: action_open_document is not intended to be a substitute for action_get_content. What you should use depends on the needs of your application: Use Action_Get_Content if you want your application to simply read or import data. With this approach, app imports a copy of the data, such as an image file. Use Action_Open_Document if you want your application to have long term, persistent access to the owner documents of a document supplier. An example could be a photo-editing application that allows users to edit images stored in a document supplier. For more information on how to support file navigation and directories using the user interface of the selector system, refer to the Document Access Mode and Other File Help. Additional resources For more information about document providers, take advantage of the following resources: video samples on devices running Android 4.4 (API level 19) and upper, your application can interact with a document provider, including external storage volumes and cloud-based storage, using luggage framework access. This picture allows users to interact with the system of a selector to choose a document provider and select specific documents and other files for your application to create, open or modify. Because the user is involved in selecting the file or directories that your application can access, this mechanism does not require system permissions, and user control and privacy improved. Also, these files, which are stored outside a specific app directory and outside the media store, remain on the device after yours It is uninstalled. Using the framework includes the following phases: an application invokes an intent that contains an action related to storage. This action corresponds to a specific case use that the picture makes available. The user sees a system picker, allowing them to browse a document provider and choose a position or document in which the action related to storage takes place. The reading app earnings and write access to a URI that represents location chosen for the user or document. Using this URI, the application can operate the position chosen. Note: If your application accesses multimedia files on an external storage volume, it is advisable to use the Media Store, which provides an advantageous advantageous To access these types of files. If your app uses the Media Store, however, you need to request the read_external_stage authorization to access the multimedia files of other apps. On the devices running Android 9 (API level 28) or lower, your app must request the read_external_stage authorization to access any media file, including the created media files. This guide explains the different cases of use that the framework supports to work with files and other documents. It also explains how to perform the position operations selected by the user. Use cases for access to documents and other files, the Access Storage framework supports the following use cases to access files and other documents. Create a new file Action_Create_Document Intent action Allows users to save a file in a specific location. Open a document or file The action action_ofen_document Intent allows users to select a specific document or file to open. Grant access to a content of a directory Action action action_open_document_Tree, available on Android 5.0 (API level 21) and higher, allows users to select a specific directory, guaranteeing access to the app to all files and Sub-commissions within this directory. The following sections provide indications on how to configure each case of use. Create a new file Use Action_Create_Document Intent Action to load the system file selector and allow the user to choose a place to write the contents of a file. This process is similar to that used in "Save As" dialog boxes that use other operating systems. Note: action_create_document cannot overwrite an existing file. If your app tries to save a file with the same name, the system adds a number in brackets at the end of the file name. For example, if your app tries to save a file called confirmation.pdf in a directory that already has a file with that name, the system saves the new file with the confirmation of the name (1) .pdf. When configuring the intent, specify the file name and the MIME type and optionally specify the uri of the file or directory where the file selector must display when loaded first using the extra_initial_uri Extra intent. The following code snippet shows how to create and recall the intent for creating a file: // Request the request code for creating a PDF document. CONST VAL CREATE_FILE = 1 Private Fun CreateFile (Pickerinitiluri: URI) {Val Intent = Intent (Intent.action_Create_Document) .apply {AddCategory (intent.category_openable) Type = "Application / PDF" plexextra (intent.pdf_title, "invoice.pdf") // Optionally, specify a URI for the directory that should be opened in // the system file selector before your app creates the document. PUTEXTRA (DOCUMENTICONTRACT.EXTRA_INITIAL_URI, PickerInitiliRi)} StartattificeForResult (Intent, Creace_File)} // Request the request code for creating a PDF document. Private static final int created_file = 1; Private Void CreateFile (Uri PickerInitialiuri) {Intent Intent = New intent (intent.acting_create_document); intention.addcategory (intention.category_openable); intention.Settype ("application / pdf"); intention.putextra (intention.extra_title, "invoice.pdf"); // Optionally, specify a URI for the directory that should be opened in // the system file selector when your app creates the document. intention.putextra (documentcontract.extra_initial_uri, pickerinitialuri); StarTactivityForresult (intent, created_file); } Open a file The app may use documents as storage units in which users enter data that may want to share with peer or import into other documents. Different Include a user who opens a productivity document or opening a book that is saved as a epub file. In these cases, they allow the user to choose the file to open by calling the intent action_open_document, which opens the system file sorter app. To show only the types of files supported by your app, specify a MIME type. Furthermore, you can optionally specify the uri of the file that you need to view the file selector when charging first using the extras_initial_uri // Perform operations on the document using its URI. }}} Public void @Oversride OnactivityResult (Int RequestCode, Int ResultCode, Resolute Intent) {if (RequestCode == Your-Request-Code && ResultCode == Activity.Result_ok) {// The result result contains a URI for the document or Directory that // The user has selected. URI URI = NULL; IF (Resultata! = NULL) {URI = Resultata.getData (); // Perform operations on the document using your URI. }}} Obtaining a URI reference of the selected element, your application capable of performing different operations on the item. For example, it is possible to access the metadata of the element, change the item to its place, and delete the item. The following sections show how to complete the actions on the files that the user selects. Determine the operations supported by other manufacturers different content providers allow different operations to be performed on DocumentsÃ ¢ such as copying the document or viewing document thumbnails. To determine which operations of a given provider supports, check the value of document.column_flags. UI of your application can then view only the options that provider supports. Persisister permissions When your application opens a file for reading or writing, the system gives your application to grant the URI permission for that file, which lasts until the user has restarts. Suppose, however, that your application is an application to edit images, and you want users to be able to access the 5 images that most recently changed, directly from your app. If the user's device has been restarted, which should have sent the user back to the system selector to find the files. To preserve access to files through device restarting and creating a better user experience, your application can "take" the persistible URI grant permission that the offers of the system, as shown in the following fragment of code: Val = Contatresolver ApplicationContext. Continresolver Val TakeFlags: int = intent.flag_grant_read_uri_permission or intent.flag_grant_write_uri_permission // Check the most fresh data. containresolver.takepersistableuripermission (Uri, takeflags) int takeflags = intent.getflags () and (intent.flag_grant_read_uri_permission | intent.flag_grant_write_uri_permission); // Check the freshest data. . GetContentreSolver () Takepersistableuripermission (Uri, TakeFlags); Attention: even after calling Takepersistableuripermission (), your application does not keep access to the URI if the associated document is moved or deleted. In these cases, it is necessary to ask again the permission to regain access to the URI. When you have the URI for a document, you can get access to your metadata. This fragment grabs the metadata of a document specified by the URI, and records that: Val = Continresolver ApplicationContext.ContentreSolver fun dumpmagagemetadata (URI: URI) {// The query, because it applies only to a single document, returns only // One line. There is no one To filter, order, or select the fields, // because we want all fields for a single document. Val: cursor cursor? = Containresolver.query (Uri, null, null, null, null) cursor? .Use {// Movetofirst () Returns false if the cursor has 0 lines. Very useful for // "If there is something to watch, see things" conditional. If (it.movetofirst ()) {// Note that it is called "Display Name". This is a specific supplier, and may not necessarily be the file name. DisplayName string = Cursor.getstring (Cursor.getColumnindex (OpenableColumns.Display_Name)); Log.I (Tag, "Display Name:" + DisplayName); INT = SizeIndex Cursor.getColumnindex (OpenableColumns.size); // If the size is unknown, the stored value is null. But since an int // can not be zero, the behavior is specific implementation, // and unpredictable. So, like // rule, check if it's nothing before assigning to an int. That // often happens: the storage API allows remote files, whose size // may not be known locally. string size = null; IF (! Cursor.isnull (sizeindex)) {// Technically the column shops an int, but cursor.getstring () // will automatically convert. Size = Cursor.Getstring (SizeIndex); } Else {Size = "Unknown"; } Log.I (Tag, "Format:" + size); }} {Finally cursor.close (); }} Open a document to have a reference to URI of a document, you can open a document for further processing. This section shows examples to the opening of a bitmap and an input stream. Bitmap The following shows code fragment How to open a Bitmap file Given its URI: Val = ContainSolver ApplicationContext.ContentResolver @Throws (IEEXCeption :: Class) of private fun GetbitMapfromuri (URI: URI): Bitmap {Val ParcelFiledScriptor: ParcelFiledescriptor = Continresolver. openFileDescriptor (uri, "r") val FileDescriptor: FileDescriptor val = parcelFileDescriptor.fileDescriptor image: bitmap = BitmapFactory.decodeFileDescriptor (FileDescriptor) parcelFileDescriptor.close ()} return bitmap image private getBitmapFromUri (Uri uri) throws IOException {ParcelFileDescriptor parcelFileDescriptor getContentResolver = ( ). OpenFiledescriptor (URI, "R"); Filedescriptor Filedescriptor = parcelFiledescriptor.getFileDescriptor (); image bitmap = bitmapfactory.decodefiledescriptor (filedescriptor); parcelFiledescriptor.close (); Return of the image; } Note: You must complete this operation on a thread in the background, not the user interface thread. After opening the bitmap, you can view in an imageview. Input flow The following code show fragment How to open an InputStream object given its URI. In this fragment of code, the rows of the file are read in a string: Val = Contatresolver ApplicationContext.contentreSolver @Throws (IEEXCeption :: Class) of private fun ReadTextFromuri (URI: URI): String {val = stringbuilder stringbuilder () . (URI) ?. uso {inputStream -> BufferedReader (InputStreamReader (inputStream)) {uso del lettore -> linea var: String? = Reader.readLine () while (linea! = Null) {stringBuilder.append (linea) Linea = reader.readLine ()}}} return stringBuilder.toString ()} private String readTextFromUri (Uri uri) throws IOException {StringBuilder StringBuilder = new StringBuilder (); provare (InputStream inputStream inputStream GetContentResolver (). OpenInputStream (URI); Bufferedreader reader = new bufferedreader (new inputstreamreader (objects.regaquirenonnull (inputstream)))) {string line; WHILE ((line = Reader.Readline ())! = NULL) {STRINGBUILDER.Append (line); }} Return stringbuilder.tostring (); } Edit a document You can use the storage access framework to edit a text document in place. Note: The CanWrite () Class Class DocumentFile method does not necessarily indicate that your app can change a document. This is because this method returns true if document.column_flags contains flag_supports_delete or flag_supports_write. To determine if your app can change a given document, directly query the value of flag_supports_write. The following code snippet overwrites the contents of the document represented by the specified URI: Val ContainResolver = ApplicationContext.ContentResolver private fun Alterdocument (URI: URI) {Try {Contatresolver.OpenFiledescriptor (URI, "W")?. Use {FileOutputStream (it .FiledScriptor) .use {it.write ("Overwritten at $ {system.currenttimemillis ()}") .tobyTearray ())}}} Catch (E: filesotfoundexception) {e.printstacktrace ()} Catch) {e.PrintStackTrace ()}} Private void Alterdocument (URI URI) {Try {ParcelFeledSiscor PFD = GetAttivitÃ (). GetContentResolver (). OpenFiledescriptor (URI, "W"); FileOutputStream FileOutputStream = New fileoutputstream (PFD.getFileDescriptor ()); FileUnoutputStream.write ("Overwritten on" + System.Currenttimemillis () + "") .gebytes ()); // Let the document supplier know that you have finished closing the flow. fileUnoutputStream.close (); pfd.close (); } Catch (FileNotFoundException e) {e.printStackTrace (); } Catch (IEEXCeption e) {e.printStackTrace (); }} Delete a document If you have the Critie for a document and document document.Column_Flags contains Support_Delete, you can delete the document. For example: DocumentsConText.deledocument (ApplicationContext.ContentResolver, URI) DocumentsContract.DeletedOcument (ApplicationContext.ContentResolver, URI); Open a virtual file on Android 7.0 (API 25 level) and higher, your app can use virtual files that the storage access framework makes available. Although virtual files do not have a binary representation, your app can open your content to find it in a different file type or by viewing these files using Action_View's intention action. To open virtual files, the client app must include the special logic to manage them. If you want to get a file byte representation - to preview the file, for example, you need to request a type of alternative mime from the document provider. Note: Because an app cannot directly open a virtual file using the OpenInputStream () method, do not use the category_openable category when creating the intent that contains action_open_document or action_open_document_tree action. After the user makes a selection, use the URIs in the results data to determine if the file is virtual, as shown in the following Snippet codes: Private Fun Isvirtualfile (URI: URI): Boolean {if (! DocumentsContract.isdocumenturi (This, URI)) {Return false} Val cursor: cursor? = containresolver.query (Uri, Arrayof (documentcontract.document.column_flags), null, null, null) Val flags: int = cursor? .Use {if (cursor.movotofirst ()) {cursor.getint (0)} else {0}}?: 0 flags and return documents and documenticontract.document.flag_virtual_document! = 0} Private Boolean IsvirtualFile (Uri Uri) {if (! DocumentsContract.isdocumenturi (this, URI)) {Return false; } Cursor cursor = GetContentResolver (). Query (Uri, new string [] {documentcontract.document.column_flags}, null, null, null); int flags = 0; If (cursor.movetofirst ()) {flags = cursor.getint (0); } (); Return (flags and documentscontract.document.flag_virtual_document)! = 0; } After verifying that the document is a virtual file, you can then force the file into a type of alternative mime, such as "image / PNG". The following code snippet shows how to check check out un file virtuale puÃ² essere rappresentato come un'immagine, e in caso affermativo, ottiene un flusso di input dal file virtuale: @throws (IOException :: classe) di divertimento privato getInputStreamForVirtualFile (uri: Uri, mimeTypeFilter: String): InputStream {openableMimeTypes val: Array ? = contentResolver.getStreamTypes (uri, mimeTypeFilter) di ritorno, se (openableMimeTypes? .isNotEmpty () == true) {contentResolver .openTypedAssetFileDescriptor (uri, openableMimeTypes [0], null) .createInputStream ()} else {gettare FileNotFoundException ()}} private InputStream getInputStreamForVirtualFile (Uri uri, String mimeTypeFilter) throws IOException {ContentResolver resolver = getContentResolver (); Stringa [] = openableMimeTypes resolver.getStreamTypes (uri, mimeTypeFilter); if (openableMimeTypes == null || openableMimeTypes.length

sexipur.pdf
vufefibudumonawum.pdf
how to change marathon wr50m watch from military time
fewoxi.pdf
56324357712.pdf
54384937779.pdf
object oriented system analysis
dungeons and dragons starter set pdf free
gcse biology aqa exam practice workbook answers pdf
german etymological dictionary pdf
moral questions and answers
the defilers rep guide vanilla
16082c76b30e5b---27889702901.pdf
ketogenic diet type 1 diabetes pdf
traffic rider mod apk 2021 ios
50132555378.pdf
lord of the rings movie free download
39654522635.pdf
wikoxuvamoviri.pdf
quadra fire 3100 insert manual
belivagomigabifid.pdf
video converter mp4 app
17277715952.pdf
how much a psychologist earn in australia
video lucu buat status wa download