Obb file downloader app

Obb file download app.

Discover 3 million app exciting without fee. EASY. Important: from August 2021, the new apps are required to publish with the Android app package on Google Play. The new larger apps of 150 MB are now supported by a delivery function of the playback function or from a balance sheet. Google Play requires the compressed APK that users download no more than 100 MB. For most apps, this is plenty of space for all app codes and resources. However, some apps need more space for high fidelity graphics, multimedia files or other large activities. Previously, if your app's compressed download size has exceeded 100 MB, you had to host and download additional resources alone when the user opens the app. Hosting and extra file service can be expensive and user experience is often less than ideal. To make this process easier for you and more pleasant to users, Google Play allows you to attach two large expansion files that integrate your APK. Google Play hosts the expansion files for your app and serves them to the device at no cost for you. The expansion files are saved in the shared storage position of the device (the SD card or the mountable USB partition; also known as "external" storage) in which your app can access it. On most devices, Google Play downloads the expansion files at the same time download the APK, so your app has everything you need when the user opens it for the first time. In some cases, however, your app has to download files from Google Play when your app starts. If you want to avoid using the expansion files and the size of the compressed download of your app is larger than 100 MB, you need to download your app instead using the Android app bundles that allows up to a dimension of compressed download from 150 MB. Also, since the use of App Bundles differs the APK generation and the signature on Google Play, users download the APKs optimized with only the code and resources they need to run your app. It is not necessary to build, sign and manage more apk or expansion files and users get smaller and optimized downloads. Overview Each time you load an APK using the Google Play console, you can add one or two expansion files to the APK. Each file can hold up to 2 GB and can be any format you choose, but we recommend using a compressed file to keep bandwidth during download. Conceptually, each expansion file plays a different role: the main expansion file is the primary expansion file for further resources required by your app. The patch expansion file is optional and intended for small updates to the main expansion file. While you can use the two expansion files in any way you want, we recommend that the main expansion file provides primary activities and they should rarely updated; The patch expansion file should be smaller and serve as a Ã ¢ â‚¬Å Patch Carrier ", to be updated with each major release or as necessary. However, even if your app update requires only a new file expansion of the patch, you still need to upload a new APK with a version updated version in the manifest. (the play console does not allow you to upload an expansion file to an existing APK.) NOTE: the expansion patch file Ã ¨ semantically the same as the main expansion file - you can use each file in any way you want. filename format each expansion file you upload can be any format you choose (zip, pdf, mp4, etc.). it can also use the Jobb for encapsulating and encrypting a set of resource files and subsequent patches for that set. Regardless of the type of file, Google Play considers Binary Binary opaque and rename the files using the following diagram: [Main | Patch]. . .obs or patch Specifies whether the file is the main expansion or patch file. There can only be a main file and a patch file for each APK. This is an integer that corresponds to the APK version code with which the expansion is first associated (corresponds to the Android app: Valentercode Value). "First" is emphasized emphasized Although the playback console allows you to reuse a loaded expansion file with a new APK, the name of the expansion file does not change - retains the version applied to it when the file is charged for the first time. Name of the Java style package of the app. For example, suppose your APK version is 314159 and your package name is com.Example.app. If you load a main expansion file, the file is renamed to: main.314159.com.example.app.OBB Storage position When Google Play downloads the expansion files to a device, save them to the system's shared storage position. To ensure proper behavior, you don't need to delete, move or rename the expansion files. In the event that your app has to download from Google Play itself, you need to save the files in the exact location. The Getobbadir () method returns the specific location for your expansion files in the following form: / Android / OBB / / for each app, there are never two expansion files in this directory . One is the main expansion file and the other is the patch expansion file (if necessary). Previous versions are overwritten when updating your app with new expansion files. Since Android 4.4 (API level 19), the apps can read the OBB expansion files without external storage permit. However, some implementations of Android 6.0 (API level 23) and later they still require permission, so it is necessary to declare the read_external_starage authorization in the Manest app and request permission at the execution as follows: For Android version 6 and later, the external storage authorization must be requested in runtime. However, some Android implementations do not require permission to read the OBB files. The following code snippet shows how to control access to readings before requesting external storage permission: Val OBB = file (OBB_FILENAME) VAR Open_Failed = false try {bufferedReader (FileReader (OBB)). Also {BR -> READABBFILE (BR)}} Catch (E: IOEXCEPTION) {Open_Failed = TRUE} IF (open_failed) {// Request dead_external_storage ammort before reading files OBB READBFILOWITHPERMISSION ()} File OBB = New file (OBB_FILENAME) file; Boolean open_faild = false; Test {bufferedreader br = new bufferedreader (new fileReader (obbe); open_failed = false; Readbfile (br); } Capture (IEEXCeption e) {Open_Failed = true; } If (open_failed) {// READ_EXTERNAL_STORAGE REQUEST Authorization before reading the OBB READOBFILEWINCIEWPERMISSION () file; } If you need to unpack the contents of the expansion files, do not delete the OBB expansion files later and do not save unpacked data in the same directory. You should save your unpacked files in the directory specified by GetExternalFilesDir (). However, if possible, it is better if you use an expansion file format that allows you to read directly from the file instead of requesting to unpack the data. For example, we have provided a library project called APK Expansion Zip Library that reads your data directly from the ZIP file. Attention: unlike the APK files, any file saved on the shared storage space can be read by the user and to other apps. Tip: If you pack your multimedia files in a zip, you can use playback calls on files with offset and length controls (such as mediaplayer.setDataSource () and soundpool.load ()) without the need for Unpack your zip. For this functions, you don't need to perform additional compression on the media files when creating ZIP packages. Example, when using the ZIP tool, you need to use the -n â € Manager SDK) and under aspect & behavior> System settings> Android SDK, select the SDK Tools tab to select and download: Google Play Licensing Library Librage Package Google Play APK Package Expansion Library Create A new library module for the license verification library and the downloader library. For each library: select File> New> New module. In Create new module, select Android Library, and then select Next. Specify an app / library name as "Google Play License Library" and "Google Play Downloader" library, select minimum SDK level, then select Finish. Select File> Project Structure. Select the Property tab and repository Library, enter the library from / Extras / Google / Directory (Play_licensing / for the License Verification Library or Play_APK_EXPANSION / DOWNLOADER_LIBRARY / for the Downloader Library). Select OK to create the new module. module. The downloader library depends on the license verification library. Be sure to add the license verification library to the Property of the Downloader Library project. Or from a command line, update the project to include libraries: change directories to the / Tools / Directory. Run the Android update project with the -Library option to add both the LVL library and the downloader to your project. For example: Android Update Project --Path ~ / Android / MyApp --Library ~ / Android_SDK / Extras / Google / Market_Licensing --Library ~ / Android_SDK / Extra / Google / Market_apk_Expansion / Downloader_Library with both license verification library And Library Downloader Add to your App, you will be able to quickly integrate the possibility of downloading expansion files from Google Play. The format you choose for the expansion files and the way you have them from the shared memory is a separate implementation that you should consider based on the needs of the app. Tip: The APK expansion package includes a sample app that shows how to use the downloader library in an app. The sample uses a third library available in the Apk expansion package called APK expansion ZIP library. If you plan to use zip files for expansion files, we suggest you also add the Apk Zip expansion library to your app. For more information, see the following section on using the APK expansion zip library. By declaring user permissions to download expansion files, the downloader library requires various permissions to be declared in the manifest file of your app. I am: ... Note: By default, the downloader library requires The API level 4, but the APK expansion library requires the API level 5. In order to perform download in the background, the Downloader library offers its own service subclass called Downloaderservice you should extend. In addition to downloading the expansion files for you, the downloaderservice also: records a broadcastreceiver listening to the changes to the device's network connectivity (transmission _action connectivity) to pause the download when necessary (as due to the loss of connectivity) and Resume the discharge whenever possible (the connectivity is acquired). Planning an RTC_Wakeup alarm to retry the download for cases where the service is killed. It builds a personalized notification that displays the progress of the download and any errors or state changes. Allows your app to pause manually and resume the download. Check that the shared storage is mounted and helpful, that the files do not already exist, and that there is enough space, everything before downloading the expansion files. So notify the user if one of these is not true. All you have to do is create a class in your app that extends the class And overwrite three methods to provide specific App details: GetPublicanKey () This must return a string that is the Base64 coded RSA public key for your Publisher account, available from the profile page on the playback console (see setting for licenses) . Getestal () This must return a series of random bytes that the license policy uses to create an obfuscator. The salt guarantees guarantees The obvious shared preferences file in which the data is saved license is unique and not detectable. GATAALARMRECEIVERCLASSNAME () This must return the name of the Broadcastreamer class in your app which should receive the alarm indicating that the download must be restarted (which could happen if the Downloader service stops unexpectedly). For example, here is a complete implementation of Downloaderservice: // you need to use the public key belonging to your account Publisher account cost val base64_public_key = "yourlvlkey" // you should also change this Salt Val Salt = ByteArrayof (1, 42, - 12, -1, 54, 98, -100, -12, 43, 2, -8, -1, 9, 5, -106, -4, 9, 5, -106, -107, -33, 45, -1, 84) Class SampleDownloadDerservice: DownloadService () {Override Fun GetPuplickkey (): String = base64_public_key override fun Getsalt (): ByteArray = salt override funn getalarmreceeverclassname (): string = samplelarmreceiver :: class.java.name} class.java .name} Public class SAMPLEDOWNLoadService Extend Downloaderservice {// You must use the public key belonging to your account Publisher Account Public Static Final String Base64_Public_Key = "Yourlvlkey"; // you should also modify this final static byte of salt [] salt = new byte [] {1, 42, -12, -1, 54, 98, -100, -12, 43, 2, -8, -4 , 9, 5, -106, -107, -33, 45, -1, 84}; @Override Public String GetPublican () {Return Base64_public_key; } @Override Public Byte [] getsalt () {return rooms; } @Override Public String GetalarmReceEverClassName () {Return sampleLarmReceeEiver.class.getName (); }} Notice: You must update the Base64_Public_Key value to be the public key belonging to your Publisher account. You can find the key to the developer console under your profile information. This is also necessary when your downloads occur. Remember to declare the service in your manifest file: ... Implementation of the alarm receiver To monitor file downloads and restart the download If necessary, Downloaderservice plans an RTC_Wakeup alarm that offers an intent to a broadcastreceiver in your app. It is necessary to define the BroadcastreCeiver to call an API from the downloader library that controls the status of the download and restart it if necessary. You simply have to overwrite the onrecive () method to call DownloadClientMarshaller.startownloadserviceifrequired (). For example: Class SampleAlarmReceiver: Broadcastreamer () {Override of Onreceive Fun (Context: Context, Intent Intent) {Try {DownloadDerClientMarShaller.startDownloadServiceIfrequired (context, intention, SampleDownloadService :: Class.java)} Catch (E: PackageManager.NamenotfoundException) {e.PrintStackTrace ()}}} Public Class SampleAlarmReceiver extends BroadcastreCeiver {Onreceive (context context, intent intent) @override public void {try {downloaderclientmarshrequired.startdownloadserviceifrequired (context, intention, sampledownloaderservice.class); } Catch (namenotfoundexception e) {e.printstacktrace (); }}} Note that this is the class for which you need to return the name in the GetAlarmReceVerClassName method of your service (see the previous section). Remember to declare the receiver in your manifest file: ... Starting the main activity download in your app (that started From your launcher icon) it is responsible for checking if the expansion files are already present on the device and start the download if they are not. Starting the download using the downloader library requires the following procedures: check if the files have been downloaded. The Downloader library includes some Helper class bees to help with this process: (Context, C, Boolean Mainfile, Int VersionCode) DoesFileExist (Context C, String Filename, Long Filesize) For example, the example applying Apk package expanding calls the following oncreate method of the activity () to check if the LA The files already exist on the device: fun expansionFilesDeliversed (): Boolean {xapks.freak {xf -> helpers.get.getexpansionapkfilename (this, xf.isbase, xf.filospersion) .also {filename -> IF (! Helper.doesfileexist (This, file name, xf.filesize, false)) Return false}} Return true} Boolean expansionFilesDeliversed () {for (XAPKFILE XF: XAPKS) {String Filename = Helpers.getExpansionapkFileName (this, XF.ISBASE, XF.FILAVERSION); If (! Helpers.doesfileExist (this file name, xf.filesize, false)) Return false; } Return True; } In this case, each XAPKFILE object contains the version number and size of the file of a known expansion file and a Boolean on the fact that it is the main expansion file. (See the SampleDownLoAderTivity class of the example app for details.) If this method returns false, then the app must start the download. Start download by calling the static method DownloadDerclientMarshaller.StartownloadServiceifRequered (Context C, Pendingint NotificationClient, Class ServicesClass). The method takes the following parameters: context: the context of the app. NotificationClient: a pendant to start your main activity. This is used in notification that the Downloaderservice creates to show the progress of the download. When the user selects the notification, the system calls the student you provide here and should open the activity that shows the progress of the download (usually the same activity that started the download). Serviceclass: The class object for the implementation of Downloaderservice, required to start the service and start the download if necessary. The method returns an integer that indicates whether the download is required or not. The possible values are: no_download_required: returned if the files exist or a download is already in progress. LVL_CHECK_REQUIRED: returned if a license check is required to capture the URLs of the expansion file. Download_Required: returned if the URLs of the expansion file are already known, but have not been downloaded. The behavior for lvl_check_required and download_required is essentially the same and normally you don't have to be worried about them. In your main activity that calls startdownloadserviceifrequequired (), you can simply check if the answer is nor _download_required. If the answer is something different from no_download_required, the downloader library starts the download and you should update your user interface activity to view the progress of the download (see the next step). If the answer is nodownload_required, the files are available and your app can start. For example: OVERRIDE FUN ONCREATE (SAVIENSTANCESTATE: BUNDLE?) {SUPER.ONCREATE (SAVIENSTANSTANCESTATE) // Check if the expansion files are available before going further if (! EXPANSIONFILESDELIVER ()) {VAL PENDINGINTENT = // Build an intent of Start this activity from the notification intent (this, MainActivity :: Class.java) .apply {flags = intention.flag_activity_new_task or intention.flag_activity_clear_top} .let {notifinintent -> pendingintent.gatività (this, 0, notifierintent, pendingent.flag_update_current)} / / Start the download service (if necessary) Val StartRresult: int = DownloadDerclientMattershaller.startownloadserviceifrequerato (this, PendingIntent, SampledownloadSerterservice :: Class.java) // If the download was started, initializes this activity to show // DOWNLOAD PROGRESS SE (STARTRESULT! = DownloadClientMarshaller.No_Download_Requored) {// here you set to view the download // progress (PAS later {// Check if the expansion files are Before going further if (! EXPANSIONFILESDELIVERED ()) {// EXPANSIONFILESDELIVERED ()) {// EXPANSIONFILESEDELIVERED ()) / Build an intent to start this activity from the intention intention notification notifierintent = New Intent (this, MainActivity.getClass ()) ; Notifierintent.setflags (intention.flag_tivitÃ _new_task | | ... pendingintent pendingintent = pendingintent.gativity (this, 0, notifiniant, pendintent.flag_update_current); // Start the download service (if required) int = Startresult DownloaderClientMarshaller.StarTownloadServiceIfrequered (this, pendentente, sampledownloaderervice.class); // If the download was started, it initializes this activity to show // download progress if (StartResult! = DownloadDerClientMarShaller.No_Download_requilored) {// Here is here you set to view the download / / progress (next step). ..; } // If the download has not been necessary, drop to start the app} STARTAPP (); // Expansion Files are available, start the app} when the startdownloadserviceifrequered () method returns something other than no_download_requilored, create an instant of iStub by calling DownloadEderClientMarShaller.Createstub (IdownloadClient client, Class Downloaderservice). The ISTUB provides a link between your activity to the downloader service so that your business receives callback on the progress of the download. To instantiate your iSTUB by calling CRETESTUB (), you need to transmit it an implementation of the IdownLoadAderclient interface and the DownloadService implementation. The next section on receiving the progress downloads discuss the hydrottadoloaderclient interface, which should usually be implemented in the activity class so you can update the activity user interface when the download status changes. It is advisable to call CREATESTUB () to instantiate your iSTUB during the Oncreate () method), after StartDownloadServiceIfRequered () start the download. For example, in the previous code sample for oncreate (), you can respond to the result of StartDownloadserviceifrequerod () as this: // Start the download service (if necessary) Val StartResult = DownloadDerclientMarshaller.StardownloadServiceifrequered (this @ Maineactity, Pendingent, SampleDownloaderService, SampleDownloaderService:: class.java) // If download has started, it initializes the activity to show progress if (Startresult! = DownloaderClientMarshaller.No_download_requilored) {// instantiate an instance member ISTUB DownloaderClientStub = DownloaderClientMarshaller. Caatestub (this, SampledownLoaderservice :: Class.java) // inflate layout showing the download progress setcontentview (r.layout.downloader_ui) Return} // Start the download service (if required) int startresult = downloadloaderclientmarshaller.stardownloadserviceifrequered (this, Pentingent, sampledownloaderservice.class); // If the download has started, it initializes the activity to show progress if (StartResult! = DownloadDerclientMarshaller.No_Download_Requilter) {// Instance an instance Member of Istub DownloadClientStub = DownloadClientMarchaller.caatestub (this, sampledownloaderservice.class); // inflates the layout showing the download progress setcontentview (r.layout.downloader_ui); Return; } After the oncreate () method returns, your business receives a call to onresume (), which is where you need to call Connect () on iStub, passing it the context of your app. Conversely, you should call the disconnection () in the onstop callback () of your business. Override Onresume () {DownloadClientStub? .Connect (this) super.onresume ()} Override fun onstop () {DownloadDerclientstub? .Disconnect (this) super.anstop ()} @Oversride protected void Onresume () {if (null! = DownloadRerclientstub) {DownloadClientStub.connect (this); } super.onresume (); } @Override protected empty onStop () {if (null! = DownloadClientStub) {DownloadClientStub.disconnect (this); } super.anstop (); } Connect () call on ISTUB Agazzina your activity to downloaderService so that your activity receives callbacks regarding changes to the download status through IDownloaderClient interface. Reception Download progress to receive updates related to the progress of the download and to interact with the downloaderservice, you need to implement the IdownloadClient interface of the downloader library. Usually, the activity you use to start downloading should implement implementation Interface to view the progress download and send requests to the service. The interface methods required for IdownLoadCellClient are: OnServiceConnected (Messenger M) After instant the iStub in your business, you will receive a call to this method, which passes a Messenger object that is connected with the instance of Downloaderservice. To send requests to the service, for example to pause and resume downloads, you need to call DownloaderserviceMattershaller.createroroxy () to receive the IdownLoaderservice interface connected to the service. A recommended implementation is similar to this: Private Var RemoteService: ideownloaderservice? = Null ... Override Fun OnServiceConnected (M: Messenger) {RemoteService = DownloaderVicemarshaller.createsProxy (M) .apply {DownloadClientStub? .Messenger? The remoteservice.onclientupdated (DownloadDerclientStub.getMessenger (); } With the initialized Idownloaderservice object, you can send commands to the Downloader service, for example to pause and resume download (RequestFreeDownload () and RequestContinuedownonwload ()). OnDownloadStateChanged (int Newstate) The download service calls this when a modification of the download status occurs, for example the download starts or complete. The NewState value will be one of the various possible values specified by one of the constants of the class_ * class condition. To provide a useful message to your users, you can request a corresponding string for each status by calling helpers.getdownloaderringringResourceDFromstate (). This returns the resource ID for one of the bundled strings with the downloader library. For example, the string "Download pauses because you're roaming" corresponds to state_paused_roaming. OnDownloadProgress (DownloadprogGreggressinfo Progress) The download service calls this to provide a DownloadProgressInfo object, which describes various information on the progress of the download, including the estimated time, the current speed, general progress and the total so you can update the user interface Download progress. Tip: For examples of these callbacks that update the download advance user interface, see the sampleDownloadDeartivity in the sample app provided with the APK expansion package. Some public methods for the Idownloaderservice interface you may find useful are: RequestpaSedewnownload () Pause download. RequestContinIndownload () resumes a pause download. Setdownloadflags (int flags) Sets user preferences for network types on which it is ok to download files. The current implementation supports a flag, flags_download_over_cellular, but others can be added. By default, this flag is not enabled, so the user must be on Wi-Fi to download expansion files. You may want to provide a user preference to enable downloads on the cellular network. In which case, you can call: remoteService = DownloadErservicemarShaller.createproxy (M) .apply {... setdownloadflags (idrowloadterservice.flags_download_over_cellular)} remoteService .SEnloadFlags_Download_over_cellular.flags_download_over_celular); Using APKExpansionsPolicy if you decide to create your own downloader service instead of using the Google Play Downloader Library, you can still use the apkexpansionpolicy provided in the license verification library. The ApkexPansionPoly class is almost identical to ServermanagedPolicy (available in the Google Play license verification library) but includes additional management for the response extras of the APK expansion file. Note: if you use Downloader library as discussed in the previous section, the library performs all the interactions with apkexpansionpolicy so as not to use this class directly. The class includes methods to help you get the necessary information on the available expansion files: getexpansionlcount () GETEXPANSIONURL (INT (INT GETEXPANSIONFILENAME (INT INDEX) GETEXPANSIONFILESIZE (INT INDEX) For more information on how to use APKExpansionSpoly when you do not use the downloader library, consult the documentation to add licenses to your app, which explains how to implement a license policy like this one. Reading the expansion file After saved APK expansion files on the device, how to read your files depends on the type of file you used. As discussed in the overview, your expansion files can be any type of file you want, but are renamed using a particular file name format and are saved to / Android / OBB / /. Regardless of how to read your files, you should always check that external storage is available for reading. There is a possibility that the user has the archiving mounted on a computer on USB or actually removed the SD card. Note: when your app starts, it is always necessary to check if the external storage space is available and readable by calling GetExternalStageAgeAgeState (). This returns one of the different possible strings that represent the status of the external memory. In order to be readable from your app, the return value must be media_mounting. Get file names as described in overview, APK expansion files are saved using a specific file name format: [Main | Patch]. . .OBB To get the position and names of your expansion files, you need to use the methods GetPeXternalStorageDedRectory () and getpackagename () to build the path path. Here is a method you can use in your app to get an array containing the full path for both of your expansion files: fun GetapKeXPansionSionFiles (CTX: Context, Mainversion: int, PatchVersion: int): Array {Val Packagename = CTX .packageename val ret = mutablelistof () if (environmental.getexternalstoragestate () == environment.media_mounted) {// build the full path for the app expansion files val root = environment.getexternalstagedegereptory () val expeppath = File (root.toString () + Exp_Path + Packagename) // Check that the path to the expansion file exists if (ExpoTh.exists ()) {if (MAINVERSION> 0) {Val STMAINPATH = "$ expeppath $ {file.separator Lead {file.separator} patch. $ mainversion. $ packaname. obbb "Val main = file (strpatchpath) If (strpatchpatch) {RET + = strPatchpath}}}} Return Ret.totypedarray ()} // The shared route to all Apps Expa File NSION String Static Final Privatic Exp_Path = "/ Android / OBB /"; Static string [] GetapKeXPansionFiles (Context CTX, INT MAINVERSION, INT PATCHVERSION) {STRING PACKAGENAME = CTX.getPackagename (); Vector Ret = New Vector (); IF (environment.getexternalStagestate () .Equals (environment.media_mounted)) {// Build the full path for app files of the Root File Root = Environment.GetExternalStageDeectory (); File exteppath = new file (root.tostring () + Exp_Path + Packagename); // Verify that the path to the expansion file exists if (espath.exists ()) {if (mainversion> 0) {string STMAINPATH = EXPPATH + FILE.SEARATOR + "MAIN." + MAINVERSION + "." + Packagename + ".obb"; File main = new file (STMAINPATH); if (main.isfile ()) {Ret.add (STMAINPATH); }} If (PatchVersion> 0) {STRING STRPATCHPATH = EXPPATH + FILE.SEARATOR + "PATCH." + MAINVERSION + "." + Packagename + ".obb"; File main = new file (strpatchpath); if (main.isfile ()) {Ret.add (StrtchPatch); }}}} String [] REARRAY = new string [RET.SIZE ()]; Ret.torray (rearray); Return Rearray; } Call this method by passing the context of the app and the version of the desired expansion file. There are many ways where you can determine the version number of the expansion file. A simple way is to save the version in a sharedPreferences file when the download starts, questioning the name of the expansion file with the ApkexPansionPoly Class GetExPansionFileMe (int indication) Index) You can then get the version code by reading the ShareDPreferences file when you want to access the expansion files. For more information on reading from shared memory, see the data storage documentation. Using the Apk Expansion Zip Library The Google Market Apk expansion package includes a library called APK Expansion Zip Library (located in / Extra / Google / Google_market_apk_expansion / zip_file /). This is an optional library that allows you to read your expansion files when they are saved as a zip file. Using this library allows you to read the resources easily from ZIP expansion files as a virtual file system. The APK Expansion Zip Library includes the following classes and API: ApkexPansionsupport provides some methods to access the names of expansion files and zip files: GuapKeXPansionFiles () The same method shown above that returns the full file path for both files of expansion. GetapKeXPansionzipFile (Context CTX, INT MAINVERSION, INT PATCHVERSION) Returns a ZipresourceFile that represents the sum of the main file of the main file. That is, if you specify both the mainversion and the patchversion, this returns a zipresourcefile that provides access to reading to all data, with the patch file data to top the main file. ZipresourceFile represents a zip file on the shared archive and performs all the work to provide a virtual file system based on zip files. You can get an instance using apkexpansionsupport.getapkexpansionzipfile () or with zipresourcefile passing the path to your expansion file. This class includes a variety of useful methods, but generally you do not need access to many of them. A couple of important methods are: getInputStream (String AssetPath) provides an input to read a file inside the zip file. The assetpath must be the path of the desired file, relative to the root of the zip file content. GATESassetFileDescriptor (String AssetPath) provides a assetfiledescriptor for a file inside the ZIP file. The assetpath must be the path of the desired file, relative to the root of the zip file content. This is useful for certain Android APIs that require an AssetFileDescriptor, like some mediaPlayer APIs. Appezprovider Most apps don't need to use this class. This class defines a ContentProvider travelers of the ZIP file data through a URI of the content provider to provide file access for certain Android APIs that expect access URI to the media files. For example, this is useful if you want to play a video with VideoView.setVideouri (). If you use expansion files to store media files, a ZIP file allows you to use Android media playback calls that provide offset and length commands (such as MediaPlayer.SetDataSource () and SoundPool.Load ()). For this functions, you don't need to perform additional compression on the media files when creating ZIP packages. For example, when using the ZIP tool, you need to use the -N â €