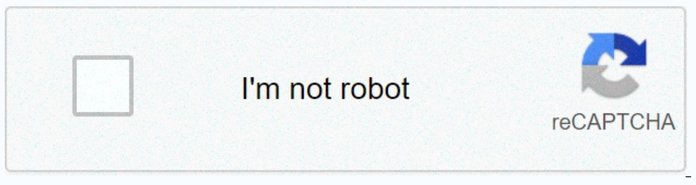

Continue

Android zip apk

Easy way to manage your programs All you need to handle compressed files Easily send files to your friends Send files to your friends Official smartphone file manager developed on Android Using a simple but powerful file manager for Android compression, unzip files on your Android device with a powerful file management tool zip application and the first Android archiver AndroZip File Manager allows you to copy, delete, move, unzip/unzip, compress encrypted ZIP files, search, and organize files, music, photos, and folders like you do on your PC. It is optimized for both phones and tablets. With more than 20,000,000 downloads and support for ZIP, encrypted ZIP, RAR, TAR, GZIP and BZIP2, AndroZip File Manager is the definitive leader among similar applications on the market. Fully supports decompression of encrypted ZIP files (standard, AES-128, AES 256-bit), compatible with WinRaR and other PC archive features, and also supports compression of encrypted ZIP files. AndroZip supports standard ZIP file encryption compatible with WinRaR and other PC archives (AES 128 and AES 256-bit encryption available for a fee). In addition to the archiving feature, you can also send files by e-mail, etc. It is optimized specifically for both mobile phone and tablet displays. For many additional features such as drag-and-drop functionality, AndroZip File Manager is preferred among tablet users. It is also used to back up applications, install new APKs on your phone, and manage phone memory and resources through Task Manager. AndroZip can be moved to your SD card to free up internal storage space. AndroZip: What others say about ✔ all comes to the rescue, providing File Explorer, Task Manager, Archive Manager, app backup tools, and more, all within a simple and intuitive interface. ✔ Of all the file managers I've tried for the Android platform, standing on top of the rest is the free AndroZip file manager. Requested permissions (like winzip or winrar (Eric, AndroZip user) such as Jack Warren, TechRepubub (Jack Warren, Tech ✔Repubub), device and app history: is required to view the app where our task killer is running. Photos/Media/Files: Needed for our app to manage these files and cache the images needed for our app's interface. Wi-Fi connection information: Required to display ads in the free version of the app. Close other apps: Required for task killer feature. Install Shortcuts: This is necessary if you want to use our app to add shortcuts to your phone's home screen. Run at Startup: Required to send analytics back to AVG to improve the app. Full network access: Required to display ads and send analytics back to AVG to improve the app. Show network connectivity: Required to show ads and send analytics back to AVG to improve your app. PreventFrom Sleep: Required to ensure that tasks such as dipping files are completed before the device goes to sleep. AVG AndroZip suggests other apps you might like based on information you have collected about apps already installed on your device. By installing/updating this app, you agree that the use of it is governed by these terms: feature: file manager with multiple/batch deletion, move, copy, ZIP, task killer, application manager, APK installer and short tap and long tap menu. Download now for free! New apps larger than 150 MB must use either feature delivery playback or asset delivery playback. Google Play must have no more than 100 MB of compressed APKs for users to download. For most apps, this is enough space for all your app's code and assets. However, some apps require more space to use high fidelity graphics, media files, or other large assets. Previously, if your app's compressed download size exceeded 100 MB, you had to host and download additional resources yourself when the user opened the app. Hosting and delivering extra files is costly and the user experience is not ideal. To make this process easier and more comfortable for users, Google Play allows you to attach two large extensions that complement the APK. Google Play hosts extensions for your app and provides them to your device at no charge. Extension files are stored in the device's shared storage location (also known as SD card or USB mountable partition, external storage) and can be accessed by the app. On most devices, Google Play downloads the extension as soon as you download the APK, so your app has all the information you need the first time the user opens it. However, you may need to download files from Google Play when you start the app. If you don't want to use extensions, and your app's compressed download size is greater than 100 MB, you're required to upload your app using the Android app bundle and use a compressed download size of up to 150 MB. In addition, app bundles delay APK generation and signing Google Play, so users download optimized APKs using only the code and resources required to run the app. Users can make downloads smaller and more optimized when they no longer need to create, sign, or manage multiple APKs or extensions. Summary Each time you upload an APK using the Google Play console, you have the option to add one or two extensions to the APK. Each file can be up to 2 GB, and you can choose any format, but we recommend that you use compressed files to save bandwidth during the download. Conceptually, each extension file plays a different role: the main extension is the primary extensionSee the additional resources your app needs. Patch extension files are optional and assume small updates to the main extension. The two extensions can be used in any way, but the main extension provides the primary asset, rarely updated, and rarely updated. Patch extension files are smaller and function as patch carriers and must be updated for each major release or as needed. However, even if you have a new patch extension file that is required for app updates, you still need to upload a new APK that contains the updated versionCode in the manifest. (In the Play console, you can't upload an extension to an existing APK.) Note: Patch extension files are meaningfully the same as the main extension, and each file can be used in any way. File name format The extension file you upload can be in any format (ZIP, PDF, MP4, etc.). You can also use the JOBB tool to encapsulate and encrypt that set of resource files and subsequent patches. Regardless of the file type, Google Play sees it as an opaque binary blob and renames the file using the main>patch scheme. <expansion-version>.<package-name>.obb This scheme has three main or patch components: whether the file is the main or patch extension. There can be only one main file and one patch file for each APK. <expansion-version> This is an integer that matches the version code of the APK with which the extension was first associated (matches the app's android:versionCode value). The Play Console allows you to reuse the uploaded extension in the new APK, but emphasizes First because the extension file is not renamed and retains the version that was applied when the file was first uploaded. <package-name> Java-style package name for the app. For example, let's say the APK version is 314159 and the package name is com.example.app. When you upload the main extension, the file's name is main.314159.com.app.app.obb Storage location When Google Play downloads the extension to your device, it's stored in the system's shared storage location. To ensure correct behavior, do not delete, move, or rename the extension. If your app needs to download from Google Play, you'll need to save the file in exactly the same location. The getObbDir() method returns the specific location of the extension in the following format: <shared-storage>/Android/obb/<package-name>/ There are no more than one extension file in this directory for each app. One is the main extension, and the other is the patch extension (if necessary). The previous version will be overwritten when you update the app with the new extension. Starting with Android 4.4 (API level 19), apps can read OBB extension files without external storage permissions. However, some </package-name></shared-storage></package-name></expand-version></package-name></expand-name>/expand-version>In Android 6.0 (API level 23) and later versions, permissions are still required, so you must declare READ_EXTERNAL_STORAGE permissions in the app manifest and request permissions at run time <uses-permission android:name_android.permission.READ_EXTERNAL_STORAGE>/uses-permission> However, some Android implementations do not require permission to read OBB files. The following code snippet shows how to verify read access before requesting permissions for external storage obb_filename READ_EXTERNAL_STORAGE open_failed open_failed > open_failed obb_filename. Try open_failed open_failed open_failed-valued file, <a0></a0>false;<a1></a1><a2></a2><a3></a3><a4></a4><a5></a5><a5></a5><a5></a5><a5></a5><a5></a5><a5></a5><a5></a5></a1><a2></a2><a4></a4><a5></a5><a5></a5></a1><a2></a2><a4><a4></a1><a2></a2><a4></a5><a5></a5><a5></a5></a (open_failed) , you can request permissions before reading the // OBB READ_EXTERNAL_STORAGE. If you need to unpack the contents of the deployment file, do not delete the OBB extension later and do not store the unpacked data in the same directory. The unzipped file must be stored in the directory specified by getExternalFilesDir(). However, if possible, we recommend that you use an extended file format that can be read directly from the file instead of unziping the data. For example, we provided a library project called the APK Extension Zip Library that reads data directly from a Zip file. Note: Unlike APK files, files stored in shared storage can be read by users and other apps. For this to work, do not perform any additional compression on the media files when you create the ZIP package. For example, if you use the zip tool, you must use the -n option to specify a file suffix that you do not want to compress. ogg main_expansion media_files download process In most cases, Google Play downloads the APK to the device and downloads and saves the extension at the same time. However, in some cases, you may not be able to download the extension, or you may have deleted a previously downloaded extension. To address this situation, your app must be able to download the file itself using the URL provided by Google Play at the beginning of its main activity. The download process from a high level looks like this: the user chooses to install the app from Google Play. In the case of Google You can download the extension (for most devices) and download it along with the APK. If you can't download the extension on Google Play, download only the APK. When the user starts the app, the app must check to see if the extension is already stored on the device. If yes, the app is ready. If no, the app will need to download the extension via HTTP from Google Play. The app must send a request to the Google Play client using the app licensing service on Google Play. Use this information to download the file and save it to the appropriate location. Note: If the file doesn't already exist on your device when you start the app, you'll need to include the code you need to download the extension from Google Play. As described in the next section on downloading extension files, we have greatly simplified this process and provided a library that performs downloads from the service with minimal code. Here's an overview of the tasks you're performing to use extension files in your development checklist app: The space is valuable and you need to keep your total download size as small as possible. If your app uses more than 100 MB to provide multiple versions of graphic assets for multiple screen densities, consider publishing multiple APKs that contain only the assets required for the target screen in each APK. For best results when publishing to Google Play, upload an Android app bundle that contains all of your app's compiled code and resources, but you'll be postponed from generating APKs and signing Google Play. Determine which app resources you want to isolate from the APK and package them in a file to use as the main extension. Typically, if you want to update the main extension, use only the second patch extension. However, if the resources in the main extension exceed the 2 GB limit, you can use the patch file for the remaining assets. Develop your app so that you can use the extended file resources in the device's shared storage location. Do not delete, move, or rename the extension. If your app doesn't require a specific format, we recommend that you create a ZIP file for the extension and read the file using the APK extension ZIP library. Add logic to your app's main activity to see if the extension is on the device at startup. If the file is not on your device, use Google Play's app licensing service to request, download, and save the extension URL. To significantly reduce the amount of code, you need to create and ensure that you have good usersDuring the download, we recommend that you use the downloader library to implement the download behavior. Note that if you want to build your own download service instead of using the library, you must rename the extension and save it to the appropriate location. When you're done developing your app, follow the testing extension guide. Rules and Restrictions Adding APK extensions is a feature that you can use to upload apps using the Play console. When you upload an app for the first time or update an app that uses an extension file, you should be aware of the following rules and limitations: To download an extension from Google Play, the user must have the app from Google Play. Google Play doesn't provide an extension URL if the app is installed in any other way. When you run a download from within an app, the URL that Google Play provides for each file is unique for each download and expires immediately after it is granted to the app. If you want to update your app with a new APK, or upload multiple APKs to the same app, you can select an extension file that you uploaded for the previous APK. The extension file is not renamed, and the version received by the APK with which the file was originally associated is retained. If you use an extension file in conjunction with multiple APKs to provide different extension files for different devices, you must upload a separate APK for each device to provide a unique version code value and declare a different filter for each APK. You can't modify an extension file by itself and publish updates to your app. If only changes to the assets in the extension file apply, you can update the APK simply by changing the version code (and possibly the version name). Do not store other data in the obb/ directory. If you need to unpack the data, save it to the location specified by getExternalFilesDir(). Do not delete or rename the .obb extension unless you are updating it. That way, Google Play (or the app itself) will download the extension repeatedly. If you want to update the extension manually, you must delete the previous extension. Download the extension In most cases, Google Play installs or updates the APK and at the same time downloads the extension and saves it to your device. This allows you to use the extension file the first time you launch the app. However, the app must download the extension itself by requesting the extension from the URL provided in response from the app licensing service on Google Play. The basic logic that you need to download the extension file is:To shared storage location <package-name> (in Android/obb// directory). If an extension exists, everything is set up and you can continue with the app. If you don't have an extension: Use an app license on Google Play to make a request and get the extension file name, size, and URL of the app. To download the extension and save the extension, use the URL provided by Google Play. You must save the file to a shared storage location (Android/obb /<package-name>/) and use the exact file name provided in the Google Play response. Note: The URL that Google Play provides for the extension is unique for each download and expires immediately after it is granted to the app. If your app is free (not a paid app), you may not be using the app's licensing service. Primarily to apply app licensing policies so that users have the right to use the app (users who have paid legitimately on Google Play). To facilitate extended file functionality, licensing services have been enhanced to provide a response to apps that contain the URL of an extension file for an app hosted on Google Play. Therefore, even if your app is free to users, you must include a License Validation Library (LVL) to use the APK extension. Of course, if the app is free, you don't need to force license verification. Note: The extension URL is returned only if the user gets the app from Google Play, regardless of whether the app is free or not. In addition to LVL, you need a code set that downloads the extension over an HTTP connection and stores it in the appropriate location on the device's shared storage. When you include this step in your app, your device may not have enough space for the extension, so you should review it before you start downloading and warn the user if there is not enough space. To avoid blocking user action, the background service should download the file so that the user leaves the app when the download is complete. During requests and downloads, various errors can occur that need to be handled properly. Because the network connection can change during the download, you should process these changes and restart the download whenever possible if interrupted. While the download is in the background, you need to provide a notification of the progress of the download, notify the user when it's complete, and then return the user to the app when selected. To simplify this task, we have built a downloader library that requests an extension URL through the licensing service, downloads the extension, and performs all of the above tasks. </package-name> </package-name>You can use <a0> to pause and resume downloads in an activity.</a0> By adding a download library and some code hooks to your app, almost all of the work of downloading extensions has already been coded. Therefore, to provide the best user experience with minimal effort, we recommend that you use the Downloader Library to download the extension. The information in the next section describes how to integrate libraries into your app. If you want to develop your own solution for downloading extensions using Google Play URLs, you must perform a license request according to your app's license documentation to obtain the extension file name, size, and URL from the response extra. The license policy must use the APKExpainsionPolicy class (included in the license validation library). About the Downloader Library To use APK extensions in your app and provide

the best user experience with minimal effort, we recommend that you use the downloader library that is included with the Google Play APK extension library package. The library downloads the extension in a background service, displays a user notification of the download status, processes network connectivity loss, and, if possible, resumes the download. To implement an extension file download using the Downloader Library, you need to extend a special service subclass and a BroadcastReceiver subclass that requires a few lines of code, each of which requires a few lines of code. Add logic to the main activity to check if the extension has already been downloaded and call the download process to display the progress UI. Implement a callback interface that includes several methods for the main activity that receives updates about the progress of the download. The following sections describe how to set up your app using the Downloader Library. To use the Ready Downloader Library to use the Downloader Library, you must download two packages from the SDK Manager and add the appropriate libraries to your app. First, open the Android SDK Manager (Tools > SDK Manager) and select the Android SDK Appearance and Behavior > System Settings > SDK Tools tab to download. For each library: Select New > New Module > File. In the Create New Module window, select Android Library, and then select Next. Specify an app/library name, such as Google Play License Library or Google Play Downloader Library, select Minimum SDK Level, and then select Finish. Select File > Project Structure. Select the Properties tab, and then click Library RepositoryDirectory (for play_licensing/license validation libraries, or for downloader library play_apk_expansion/downloader_library//) Select OK to create a new module. Note: The downloader library depends on the license validation library. Add the license validation library to the project properties of the download library. Alternatively, update the project to include the library from the command line <sdk>. To add both the lvL and the downloader library to your project、-- run an Android update project using the <a0> library </a0> option. For example, if you add both the Android Update Project --path-/Android/MyApp--Library-/android_sdk/Extra/Google/market_licensing--Library-/android_sdk/Extra/Google/market_apk_expansion/downloader_library License Verification Library and Downloader Library to your app, you can quickly integrate the functionality of downloading extension files from Google Play. The format you choose for extended files and how to read files from shared storage is another implementation that you need to consider based on your app's needs. Tip: The APK extension package includes a sample app that demonstrates how to use the Downloader Library in your app. This sample uses a third library that can be used in an APK extension package called the APK Extension Zip Library. If you want to use a ZIP file for your extension, we recommend that you add the APK Extension Zip library to your app. For more information, see the following sections on using the APK Extension Zip Library: Declaring user rights To download an extension file, the downloader library must have several permissions. They are <manifest ...> <!-- Required to access Google Play License --> <uses-permission android:name-com.android.vending.CHECK_LICENSE></uses-permission> <!-- Required to download files from Google Play --> <uses-permission android:name-android.permission.INTERNET><!-- uses-permission><uses-permission> alive while downloading files (NOT to keep screen wake) --> <uses-permission android:name-android.permission.WAKE_LOCK></uses-permission> <!-- Required to poll the state of the network connection and response to changes --> <uses-permission android:!-- android.permission.ACCESS_NETWORK_STATE name to check which Wi-Fi is enabled --> <uses-permission android:name-android.permission.ACCESS_WIFI_STATE></uses-permission> <!-- Required to read and write the exposure files on shared storage --> <uses-permission android:android.permission.WRITE_EXTERNAL_STORAGE name </manifest> Note: By default, the download library must have API level 4, but the APK extension Zip library must have API level 5. To run downloads in the background, the downloader library provides its own service subclass called DownloaderService that needs to be extended. In addition to downloading </sdk>The extension, DownloaderService, also: Registers a BroadcastReceiver that listens for changes to the device's network connection (CONNECTIVITY_ACTION broadcast), pauses the download if necessary (for example, loss of connection), and resumes the download if possible (the connection is retrieved). Schedule a backup alarm to retry the download RTC_WAKEUP service dies. Create a custom notification that displays the progress of the download and any errors or status changes. You can manually pause and resume the download in the app. Verify that shared storage is mounted and available, that the file does not already exist, and that there is sufficient space before downloading the extension. Then, if any of these are not true, notify the user. All you need to do is create a class in your app that extends the DownloaderService class and override three methods that provide specific app details. getSALLYT() It must return an array of random bytes that the license policy uses to create the desoperation catter. Salt makes the uninterpreted SharedPrefences file where license data is stored unique and undethoditable. getAlarmReceiverClassName() Should return the class name of the broadcastReceiver in the app that needs to receive an alarm that the download needs to be resumed (which can occur if the downloader service stops unexpectedly). For example, the full implementation of DownloaderService: // You must use a public key that belongs to the const val BASE64_PUBLIC_KEY of the publisher account. -106, -107, -33, 45, -1, 84) Class Sample Downloader Service: DownloaderService() Overriding getPubublKey (string: BASE64_PUBLIC_KEY override fun getSALT(): ByteArray , SALT override fun alarm receiver class name (: string: SampleAlarmReceiver: class.java.name , public class SampleDownloaderService extension service BASE64_PUBLIC_KEY// public key must be used) Account Public Static Final String BASE64_PUBLIC_KEY SALT , NEW BYTES[] 1, 42, -12, -1, 54, 98, -100, -12, 43, 2, -8, -4, 9, 5, -106, -107, -33, 45, -1, @Override BASE64_PUBLIC_KEY 84 @Override Public string getPubububkey @Override() Note: BASE64_PUBLIC_KEY value must be updated to a public key that belongs to the publisher account.You can find the key in the Developer Console under the profile information. This is also necessary if you want to test the download. Don't forget to declare the service in the manifest file <app ...> <service android:name SampleDownloaderService></service> : . </app> Alarm Receiver Implementation To monitor the progress of file downloads and resume downloads as needed, DownloaderService schedules an RTC_WAKEUP alarm to deliver intents to BroadcastReceiver in your app. You must define a broadcastReceiver that checks the status of the download and, if necessary, calls the API from the download library to restart. You need to override the onReceive() method to call downstar client marshal. Example: Class Sample Alarm Receiver : BroadcastReceiver() Intents: Intents) , try the downloader client Marshall.startDownloadServiceIfRequired (context, intent, sample overder service::class.java @Override) . Note that this is a class that needs to return a name in the getAlarmReceiverClassName() method of the service (see previous section). Don't forget to declare the receiver in the manifest file <app ...> <receiver android:name SampleAlarmReceiver></receiver> : . </app> Start download The main activity of the app (launched by the launcher icon) is responsible for checking whether the extension is already on the device and, if not, starting the download. To start a download by using the Download Library, you need to do the following: The Downloader Library contains several APIs in helper classes that assist with this process. For example, the sample app provided in the APK extension package (string file name, long file size) calls the following method in the activity's onCreate() method to see if the extension file already exists on the device > >. Helper.doFileExist (this file name, xf.file size, false)) is false. returns false. Returns In this case, each XAPKFile object holds the version number and file size of the known extension and a Boolean value whether it is the main extension. (For more information, see sampleDownloaderActivity class in the sample app.) If this method returns false, the app should start downloading. Start downloading by calling a static method Downer client Marshall.startDownloadServiceIfRequired (context c, pending intent notification client, class <?> service class). This method receives the following parameters: the context of the app. Notification client: Pending intent to start the main activity. This is used in notifications created by DownloaderService to indicate the progress of the download. When the user selects a notification, the system must launch the PendingIntent specified here and open an activity that indicates the progress of the download (usually the activity that initiated the download). serviceClass: The class object required to implement DownloaderService. This method returns an integer that indicates whether a download is required. Possible values are NO_DOWNLOAD_REQUIRED: Returned if the file already exists or if the download is already in progress. LVL_CHECK_REQUIRED: Returned when license validation is required to obtain the extension URL. DOWNLOAD_REQUIRED: Returned if the URL of the extension file is already recognized but not downloaded. LVL_CHECK_REQUIRED behavior DOWNLOAD_REQUIRED behavior is essentially the same, and you don't usually have to worry about it. The main activity that calls startDownloadServiceIfRequired() simply NO_DOWNLOAD_REQUIRED the response is correct. If the NO_DOWNLOAD_REQUIRED is not available, the download library must start the download and update the activity UI to show the progress of the download (see next step). If the NO_DOWNLOAD_REQUIRED, the file is available and the app can be launched. Example: Overriding fun (saved instance state: bundle?) (!expandFilesDerive()) If you want to build an intent to start this activity from a notification intent, make sure that the deployment file is available (main activity::class Intent.FLAG_ACTIVITY_CLEAR_TOP Intent.FLAG_ACTIVITY_NEW_TASK.PendingIntent.FLAG_UPDATE_CURRENT java).// This is where you set the download // progress (next step) to be displayed.return / If you don't need a download, fall down to launch the app > startApp() // Launch the app if an extension is available Intent.FLAG_ACTIVITY_NEW_TASK @Override. Intent.FLAG_ACTIVITY_CLEAR_TOP);... Pending Intents , Pending Intents.getActivity (this, 0, notification intent, PendingIntent.FLAG_UPDATE_CURRENT). Start the download service (if necessary) Start Results , Download Client Marshall.Start DownloadServiceIfRequired (this is a pending intent, sample downloader service.class). If the download has started, initialize this activity to display the progress of the download (startResult DownloaderClientMarshaller.NO_DOWNLOAD_REQUIRED! Return; If you don't need to download it, launch the app, startApp(). If an extension is available and the startDownloadServiceIfRequired() method returns a value other than NO_DOWNLOAD_REQUIRED, call DownloaderClientMarshaller.CreateStub(IDownloaderClient client, class <?> downloader service to create an instance of IStub. IStub provides an activity-to-activity binding to the downloader service so that the activity receives a callback about the progress of the download. To instantiate an IStub by calling CreateStub(), you must pass an implementation of the IDownloaderClient interface and a DownloaderService implementation. The next section on receiving download progress typically describes the IDownloaderClient interface that must be implemented in the activity class so that the activity UI can be updated when the download state changes. We recommend that you instantiate IStub by calling CreateStub() after starting the download during the onCreate() method of the activity. For example, in the previous code sample onCreate(), you can respond to the result of startDownloadServiceIfRequired() this@MainActivity: Pending intent, SampleDownloaderService::class.java) // If the download starts, initialize the activity and display the progress (startResult!-DownloaderClientMarshaller.NO_DOWNLOAD_REQUIRED) / / Create a member instance of the IStub DownloaderClientStub.Back to // Start the Download Service (if necessary) Start Results . . . . . If the download starts, initialize the activity and display the progress (startResult!-DownloaderClientMarshaller.NO_DOWNLOAD_REQUIRED) / / IStub DownloaderClientStub , DownloaderClientMarsaller.CreateStub(this, SampleDownloaderService.class); // If you want to inflate the layout that shows the download progress R.layout.downloader_ui, it indicates progress. After the onCreate() method returns, the activity receives a call to onResume(). Conversely, you need to call disconnect() in the activity's onStop() callback. Resume overriding fun() . . . . . connect(this) Super.onResume()' is a fun overwrite stop() Disconnect (this) super.onStop() @Override protected void onResume()@Override If you call connect() on a protected disabled stop() (null!-downloader client stub) in IStub, the activity is bound to DownloaderService, so a callback for changes to the download state is passed to the activity through the IDownloaderClient interface. To receive download progress receive download progress updates and interact with DownloaderService, you must implement the IDownloaderClient interface in the Downloader Library. Typically, the activity used to start a download must implement this interface to view the progress of the download and send requests to the service. The interface methods required for IDownloaderClient are: onServiceConnected(Messenger m) after instantiating IStub in the activity, you will receive a call to this method: To send a request to the service, such as pausing or resuming a download, you must call DownloaderServiceMarsaller.CreateProxy() to receive the IDownloaderService interface connected to the service. The recommended implementation would be: Private Bar Remote Service: IDownloaderService?. Overwriting Fun Service Connection (m: Messenger) CreateProxy(m).Apply . . . . . Messenger.. In addition, the Messenger -> On-Client Update (Messenger) is a private IDownloader @Override Service remote service; CreateProxy(m);Remote Service.On-Client Update (Downter Client Stub.get Messenger());If the IDownloaderService object is initialized, you can send commands to the downloader service, such as:Resume the download (download () and continue the request ()). The onDownloadStateChanged (int newState) download service calls this when a download state change occurs, such as the start or completion of a download. The newState value is one of several possible values, as specified by one of the STATE_ constants of the IDownloaderClient class. To provide a useful message to the user, you can request a string for each state by calling a helper. This returns the resource ID of one of the strings bundled with the downloader library. For example, the string Pause download while roaming corresponds STATE_PAUSED_ROAMING the following: The onDownloadProgress (DownloadProgressInfo Progress) download service provides a variety of information about the progress of the download, including estimated download progress time, current speed, overall progress, and totals, so that you can update the download progress UI. Call this to deliver the DownloadProgressInfo object. Tip: For an example of these callbacks that update the download progress UI, see SampleDownloaderActivity in the sample app that came with the Apk extension package. Some public methods of the IDownloaderService interface that you might find useful are: Resumes a download that has paused the execution of the download. SetdownloadFlags(int flag) Set the user settings for the type of network for which you want to download the file. The current implementation supports FLAGS_DOWNLOAD_OVER_CELLULAR flags, but you can also add other flags. By default, this flag is not enabled, so the user must be using Wi-Fi to download the extension. You may want to provide user settings to enable downloads over your thy network. In that case, you can make the following call: Remote Services - Downloader Service Marshall. CreateProxy(m).apply Set up IDownloaderService.FLAGS_DOWNLOAD_OVER_CELLULAR IDownloaderService.FLAGS_DOWNLOAD_OVER_CELLULAR a0> If you want to build your own downloader service instead of using the Google Play Downloader library that uses APKExpansionPolicy, you must use the APKExpansionPolicy included in the license confirmation library. The APKExpansionPolicy class is similar to ServerManagedPolicy (available in the Google Play License Verification Library), but includes additional processing of responses for APK extension files. Note: As described in the previous section, if you use the downloader library, you do not need to use this class directly because the library performs all interactions with APKExpansionPolicy. The <a0></a0> class contains methods to help you get the necessary information about the available extension files.If you're not using the download library, see the documentation for adding licenses to your app. Read Extension When the APK extension is saved to the device, the way the file is read depends on the type of file used. As described in the overview, extension files can be any type of file, but they are renamed using a specific file name format and saved to <shared-storage>, /Android/obb//<package-name>. Regardless of how you read the file, you should always make sure that the external storage is readable. You may have storage mounted on your computer via USB, or you may have actually removed the SD card. Note: When your app starts, you should always make sure that external storage space is available and readable. It returns one of several possible strings that represent the state of the external storage. To make it readable in your app, you need MEDIA_MOUNTED return value. Get the file name The APK extension is saved using the specific file name format [main>patch], as described in Overview. To get the location and name of the <expansion-version>.<package-name>.obb extension file, you must use the getExternalStorageDirectory() and getPackageName() methods to construct the path to the file. Gets an array that contains the full path to both extensions using the methods available in the app. Patch Version: Int): Array <String> ,val Package Name , ctx.packageName val ret , mutableListOf<String>() (Environment.getExternalStorageState(Environment.MEDIA_MOUNTED) Build the full path to the root : Environment.getExternalStorageDirectory() val exp Path - file (root.toString() + EXP_PATH + package name) // Verify that the extended file path exists (expPath.exists() 0) <a0> val strMainPath </a0> $expPath.$mainVersion.$packageName.obb val - file (strMainPath) (main.isFile) . . . . . 0) $expPath.$mainVersion.$packageName.obb val main file (strPatchPath) main.isFile EXP_PATH) . . . . . , int main version, int patch version) (Environment.getExternalStorageState() .equals(Environment.MEDIA_MOUNTED)) // File root to build the full path to the extension of the app, environment.getExternal storage directory();file expPath gt;</String></package-name></expand-version></package-name></shared-storage>+ EXP_PATH + package name); make sure that the path to the extension file exists (main version > 0) s string strMainPath - expPath + file.separator + main version + + + package name + .obb ; file main - new file (strMainPath); If (patch version > 0) , the string strPatchPath - expPath + file.separator + patch. + main version + . + package name + .obb ; file main - new file (str patch path). if (main.isFile())) The string[] retArray returns a new string [ret.size()];. if there are no more than one. When you call this method, you pass the context of the app and the version of the extension that you want. There are many ways to check the version number of an extension file. One easy way is to save the version to the SharedPrefences file when the download starts by querying the extension file name using the getExpainsionFileName(int index) method of the APK extension policy class. You can then retrieve the version code by reading the SharedPrefences file when accessing the extension. For more information about reading from shared storage, see the data storage documentation. Using the APK Extension Zip Library, the Google Market APK extension package contains a library called APK Extension Zip Library <sdk>(/extras/Google/google_market_apk_expansion/zip_file/). This is an optional library to help you read the extension file when you save it as a ZIP file. This library makes it easy to read resources from a ZIP extension as a virtual file system. The APK Extension Zip library contains the following classes and APIs: APKExpainsionSupport Provides several methods for accessing extension file names and ZIP files. Returns a ZipResource file that represents the sum of both the main file and the patch file. In other words, if you specify both mainVersion and patchVersion, ZipResourceFile, which provides read access to all data, is returned and the data in the patch file is merged on top of the main file. ZipResourceFile represents a ZIP file on shared storage and performs all the work that provides a virtual file system based on the ZIP file. To get an instance, use APK Extension Support .getAPKExpansionZipFile() or ZipResourceFile by passing the path to the extension. This class contains a variety of useful methods, but you usually don't need to access most of them. Some important methods are: getInputStream (string asset path) provides an input stream for reading files in a ZIP file: assetPath must be the path to the desired file relative to the root of the contents of the ZIP file. Descriptor (string) </sdk>Provides an asset file descriptor for a file in a ZIP file. assetPath must be the path to the desired file relative to the root of the contents of the ZIP file. This is useful for certain Android APIs that require asset file descriptors, such as some media player APIs. APEZProvider Most apps don't need to use this class. This class defines a ContentProvider that marshals data from a ZIP file through a content provider URI and provides file access to specific Android APIs that expect URI access to media files. This is useful, for example, if you want to play a video in the video view .setVideoURI(). Even if you are using an extension file to save media files, zip files can use Android media playback calls that provide offset and length controls (such as MediaPlayer.setDataSource() and SoundPool.load().< For this to work, do not perform any additional compression on the media files when you create the ZIP package. For example, if you use the zip tool, you must use the -n option to specify a file suffix that you do not want to compress. If main_expansion media_files use the Read ZIP Extension Zip library from an ogg zip file, reading a file from a ZIP usually needs: Main version, patch version) // Use the extract file .getInputStream(pathToFileInsideZip) to get the input stream of a known file in the extension file. Gets the input stream of a well-known file in the extension. The above code provides access to all files that exist in either the main extension file or the patch extension file by reading from the merged map from both files. The getAPKExpansionFile() method must be provided by the app's android.content.Context and the version numbers of both the main and patch extension files. If you are reading from a specific extension file, you can use the ZipResourceFile constructor with the path to the extension file: // Get the ZipResourceFile that represents the extension for a particular extension file. InputStream file stream to get the input stream of a well-known file in an extension fileFor more information about how to use this library for extension files, see the SampleDownloaderActivity class in the sample app. If you want to use this sample as the basis for your own implementation, you must declare the byte size of the extension file in the xAPKS array. Test extension files Before publishing an app, you need two tests: reading the extension file and downloading the file. Test files must test the app's functionality to read files from shared storage before uploading the app to Google Play. Just add the file to the appropriate location in the device shared storage and launch the app: on your device, Google Play will create the appropriate directory for the shared storage where you want to store the file. For example, if the package name is com.example.android, you must create the directory Android/obb/com.example.android/ in the shared storage area. (Connect the test device to your computer, mount shared storage, and manually create this directory.) Manually add the deployment files to the directory. Rename the file to match the format of the file name you want to use on Google Play. For example, regardless of the file type, the main extension of the com.example.android app must be main.0300110.com.example.android.obb. The version code can be any value. Keep in mind: the main extension always starts with the main, and the patch file starts with a patch. The package name always matches the name of the APK to which the file is attached on Google Play. The extension is now lying on the device, and you can install and run the app to test the extension. Here are some things to keep in mind when handling extensions: Do not delete or rename .obb extensions (even if you unpack the data to a different location): That way, Google Play (or the app itself) will download the extension repeatedly. Do not store other data in the obb/ directory. If you need to unpack the data, save it to the location specified by getExternalFilesDir(). Because the Test File Download app must manually download the deployment file the first time it is opened, it's important to test this process to make sure that the app can query the URL and download the file and save it to your device. To test your app's implementation of the manual download procedure, publish it to an internal test track so that it is available only to certified testers. If everything works as expected, the app should start downloading the extension as soon as the main activity starts. Note: Previously, you were testing your app by uploading an unpublished Write version. This feature is no longer supported. Instead, you can add this file to the <a0>T:SystemInternal, closed, or open test tracks. For more information, see The Write app is no longer supported. App updates One of the great benefits of using extensions in Google Play is that you can update your app without having to re-download all of your original assets. Google Play can provide two extensions per APK, so you can use the second file as a patch to update or provide new assets. This way, you don't have to re-download the main extension, which can be large and costly for users. The patch extension file is technically the same as the main extension, and both the Android system and Google Play are not running the actual patch between the main and patch extension files. The app code must run the required patch itself. When you use a ZIP file as an extension file, the APK extension Zip library included in the APK extension package includes the function of merging the patch file with the main extension. Note: Even if you only need to modify the patch extension, you'll still need to update your APK to run the update on Google Play. If your app doesn't need to change the code, you're needing to update the version code in the manifest. Users who previously installed the app will not download the main extension unless you change the main extension associated with the APK in the Play console. Existing users will only receive updated APKs and new patch extension files (keeping previous main extension files). Here are some issues to keep in mind when updating extension files: One main extension file and one patch extension. During a file update, Google Play deletes the previous version (the app must be removed when performing a manual update). When you add a patch extension file, the Android system doesn't actually patch the app or the main extension. You need to design your app to support patch data. However, the APK extension package includes a library for using ZIP files as extension files, allowing you to easily read all extended file data by merging patch file data into the main extension. Data.