

Android APK Expansions Exhaustive Tutorial

Introduction

This tutorial fully covers the Android APK Expansions topic: we will set up the libraries, come up with a working solution to download expansions, successfully test the implementation and rollout the written code to production. Sounds promising, isn't it?

After spending endless weeks of googling and stackoverflow seeking, the hope and desire to find a fully explanatory article regarding APK Expansions topic has irrevocably gone. Unsatisfied with the result, I decided to share my own APK Expansions implementation in this *tiny tutorial. Hope you will find it helpful.

If you need more information, please consider reading these pages:

- [Official documentation](#)
- [How to set up necessary libraries in Android Studio](#)

Please note that the tutorial is written in **Java** language. Also it is worth mentioning that the code was tested in [Android Studio](#) 4.0 under [Ubuntu](#) 20.04.

The provided solution works in a production environment. (!)

Checkout [Fractal Zoomer](#) where the expansions are implemented according to this tutorial.

You can download the **code** [here](#).

And without further ado, let's start!

Overview

Let me hazard a guess. You have developed a brand new, elaborate app with flawless graphics, complex physics and stunning sound effects. After testing it thoroughly, you were ready to present it to an outer world. You have taken a deep breath and created a new release, but after pressing the "BROWSE FILES" button to upload an APK, something has gone horribly wrong...

Upload failed

You need to reduce your APK file size to 100MB or use APK Expansion Files.

UPLOAD ANOTHER FILE

Worries aside! Every android developer sooner or later faces the issue of uploading an app to Google Console that exceeds the ~~terrifying~~ magical size limit.

First things first. What in a world is an Expansion File?

APK Expansions are no more than special files needed to expand the size limit of **100MB** for APKs and **150MB** for App Bundles.

The reason why your app exceeds the size limit is probably the “assets” folder that keeps all the textures, music, models, etc. Now you will have to use Expansion Files to store these assets instead.

There are two types of Expansion Files at your disposal: **main** and **patch**. Each file size can be up to **2GB**. You can use them together or apart. There is no semantic difference between the files, but it is recommended to store constant (that are not likely to change) assets into the main file and variable assets into the patch file. (See “Updating” section for more details).

Any rules regarding the files?

Quite a few!

File name rules

The file name must be:

[[main](#) | [patch](#)].<expansion-version>.<package-name>.obb

Where:

- [main](#) or [patch](#)

Is the type of the expansion file. There can be only one main file and one patch file for each APK.

- <expansion-version>

Is an integer that usually **matches** the **version code** of your app (located in “[Module build.gradle](#)” → “[versionCode](#)”). (See “Updating” section for more details).

- <package-name>

Is your app package (probably the package of your [MainActivity](#)).

Example:

- [main](#) expansion file is used.
- APK version code is [1](#).
- Your package is [arvolear.expansions_tutorial](#).

File name:

[main.1.arvolear.expansions_tutorial.obb](#)

In order to create an Expansion File with such a suspicious [.obb](#) extension, you will **ZIP archive** the “assets” folder and change [.zip](#) extension to [.obb](#) one. That simple.

*actually you can use any type of format you choose (ZIP, TAR.GZ, PDF, MP4...), however this tutorial takes advantage of ZIP.

Important

If you are willing to archive music or video assets, do **not** compress them.

You may use [-0](#) flag to create an archive without compression:

[zip -r -0 expansion.zip assets/](#)

Or [-n](#) flag to not compress specified files (by suffixes) only:

[zip -r -n .m4a;.mp4 expansion.zip assets/](#)

* [-r](#) flag is used to recursively compress the “assets” directory.

Storage location rules

When you surf the Android Play Store and finally find a perfect game that would

hook you from the very first launch, have a look at its size. If the size is more than 100MB, the game probably uses expansions. Google automatically provides the clients with necessary Expansion Files when they download the app. Yet these files are stored separately on users devices, apart from the APK.

More exactly expansions are stored in the device's **shared storage** that is available to the user (the user can do whatever they want to with the files stored there).

Exact directory location is:

`<shared-storage>/Android/obb/<package-name>/`

- `<shared-storage>`

Is a path to the shared storage (you can get it by calling static `getExternalStorageDirectory()` on the `Environment` class)

- `<package-name>`

Is your app package (probably the package of your `MainActivity`).

Important

Please note that Google **doesn't** always provide the clients with expansions. It is a developer's responsibility to **check** and **download** them manually (if necessary). Also it is very important to ensure that your app **doesn't** modify, change or rename anything in the "obb" directory. Or in other words, you are only allowed to download (and delete) **your** Expansion Files there. Nothing more.

This restriction stems from the fact that each Android app has to have its own, **unique** package name (aka `Bundle Id`). Android developers follow this rule by naming the app's top package with their own Developer Name (Google ensures its uniqueness). So, what might happen if an Expansion File is renamed? You are totally right – collision. And both of us know that collisions are bad guys.

Enough theory! Let's set up the libraries!

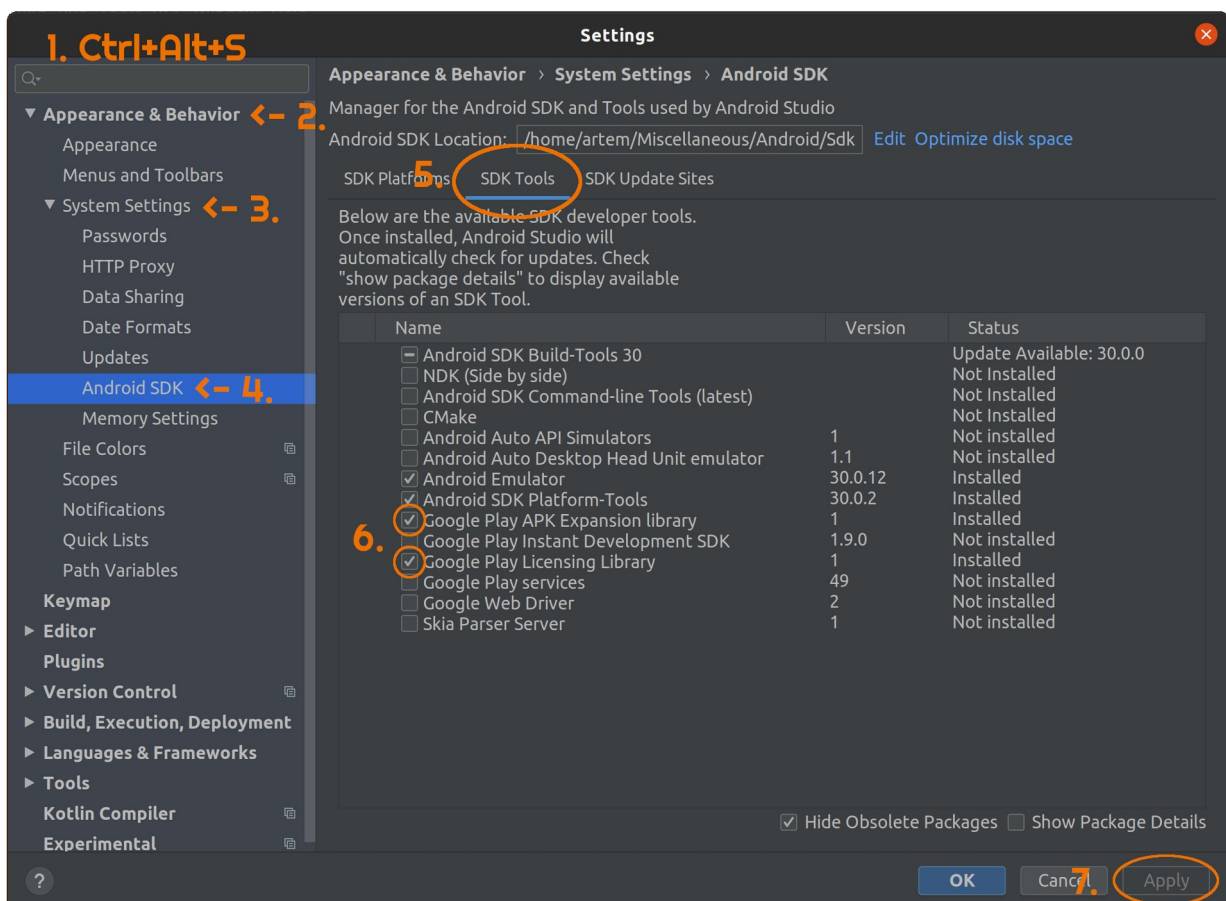
Setting up the libraries

We will use the **Downloading Library**, **License Verification Library** and **APK Expansion Zip Library** to implement our expansions solution.

- The [Downloader Library](#) is a bit obsolete, but it does what it has to do – handle the expansions downloading process.
- The [License Verification Library](#) is needed for the Downloader Library to actually start downloading expansions. (It sends the downloading request to Google and receives necessary URLs).
- The [APK Expansion Zip Library](#) is used to read the expansions easily - those strange .obb files we previously created. The library basically creates a virtual file system over ZIP so we could read an archive without unpacking it. That's neat.

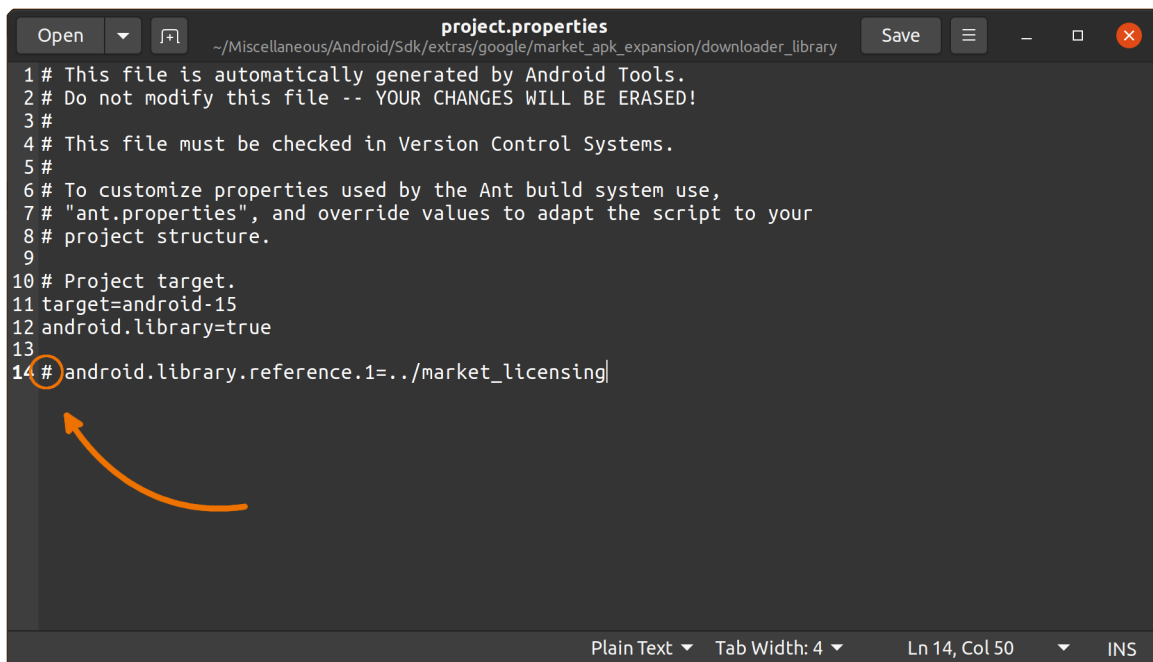
Downloading the libraries

- Open Android Studio → press “Ctrl+Alt+S” (settings window shall appear) → select “Appearance & Behavior” → “System Settings” → “Android SDK” → “SDK Tools” (on the top).
- Tick “Google Play APK Expansion Library” and “Google Play Licensing Library”. Press “Apply” and accept the license to install the libs.



Importing Downloader Library

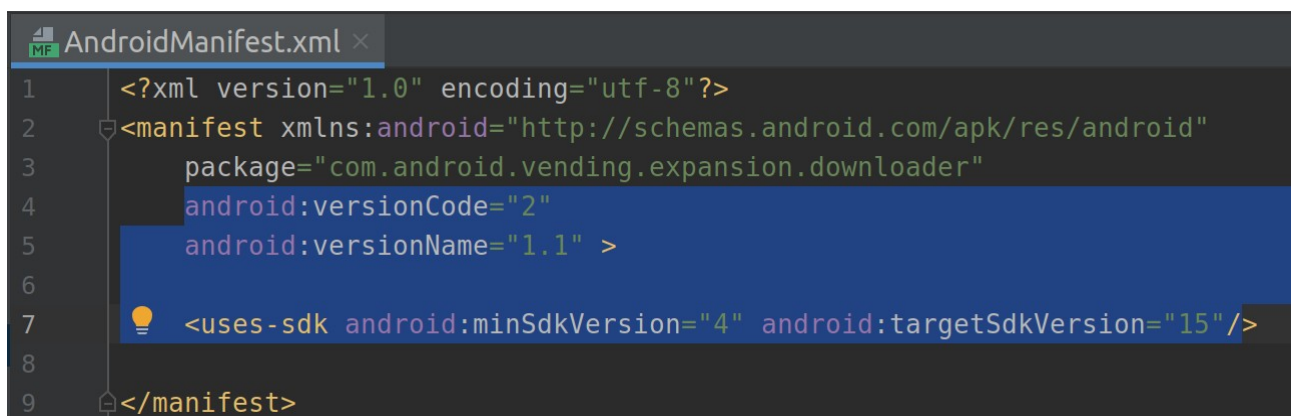
- Open the “Android/Sdk/extras/google/market_apk_expansion/downloader_library/project.properties” file on your PC.
- Comment out the last line
“`android.library.reference.1=../market_licensing`”.



```
project.properties
~/Miscellaneous/Android/Sdk/extras/google/market_apk_expansion/downloader_library

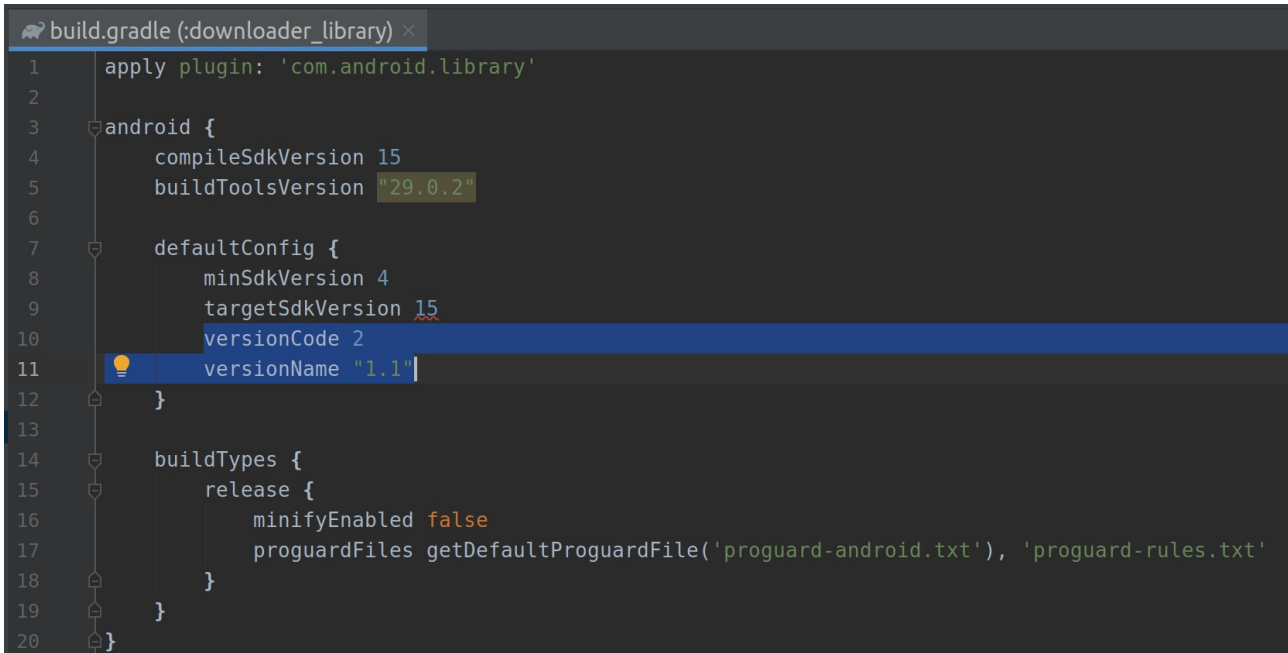
1 # This file is automatically generated by Android Tools.
2 # Do not modify this file -- YOUR CHANGES WILL BE ERASED!
3 #
4 # This file must be checked in Version Control Systems.
5 #
6 # To customize properties used by the Ant build system use,
7 # "ant.properties", and override values to adapt the script to your
8 # project structure.
9
10 # Project target.
11 target=android-15
12 android.library=true
13
14 # android.library.reference.1=../market_licensing
```

- Open Android Studio → press “File” → “New” → “Import Module...”.
- Find “Android/Sdk/extras/google/market_apk_expansion/downloader_library” → press “Ok” → “Next” → “Finish”.
- Browse to “downloader_library” module → select “manifests” → “AndroidManifest.xml”. Remove the lines “`android:versionCode=“2”`”, “`android:versionName=“1.1”`” and “`<uses-sdk ... />`”:



```
AndroidManifest.xml
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="com.android.vending.expansion.downloader"
4     android:versionCode="2"
5     android:versionName="1.1" >
6
7     <uses-sdk android:minSdkVersion="4" android:targetSdkVersion="15"/>
8
9 </manifest>
```

- Browse to “Gradle Scripts” → select “downloader_library build.gradle”. Add the lines “`versionCode 2`” and “`versionName “1.1”`” to the “`defaultConfig`” section and press “Sync Now”.



```

1  apply plugin: 'com.android.library'
2
3  android {
4      compileSdkVersion 15
5      buildToolsVersion "29.0.2"
6
7      defaultConfig {
8          minSdkVersion 4
9          targetSdkVersion 15
10         versionCode 2
11         versionName "1.1"
12     }
13
14     buildTypes {
15         release {
16             minifyEnabled false
17             proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.txt'
18         }
19     }
20 }

```

Importing License Verification Library

- Press “File” → “New” → “Import Module...”.
- Find “Android/Sdk/extras/google/market_licensing/library” → press “Ok” → “Next” → “Finish”.
- Rename the Module (the module only) from “library” to “market_licensing”.
- Select “market_licensing” module → “manifests” → “AndroidManifest.xml”. Remove the lines “`android:versionCode=“2”`”, “`android:versionName=“1.5”`” and “`<uses-sdk ... />`”.
- Browse to “Gradle Scripts” → select “market_licensing build.gradle”. Add the lines “`versionCode 2`” and “`versionName “1.5”`” to the “`defaultConfig`” section and press “Sync Now”.

Importing APK Expansion Zip Library

- Press “File” → “New” → “Import Module...”.

- Find “Android/Sdk/extras/google/market_apk_expansion/zip_file” → press “Ok” → “Next” → “Finish”.
- Select “zip_file” module → “manifests” → “AndroidManifest.xml”. Remove the lines “`android:versionCode=“2”`”, “`android:versionName=“1.1”`” and “`<uses-sdk ... />`”.
- Browse to “Gradle Scripts” → select “zip_file build.gradle”. Add the lines “`versionCode 2`” and “`versionName “1.1”`” to the “`defaultConfig`” section and press “Sync Now”.

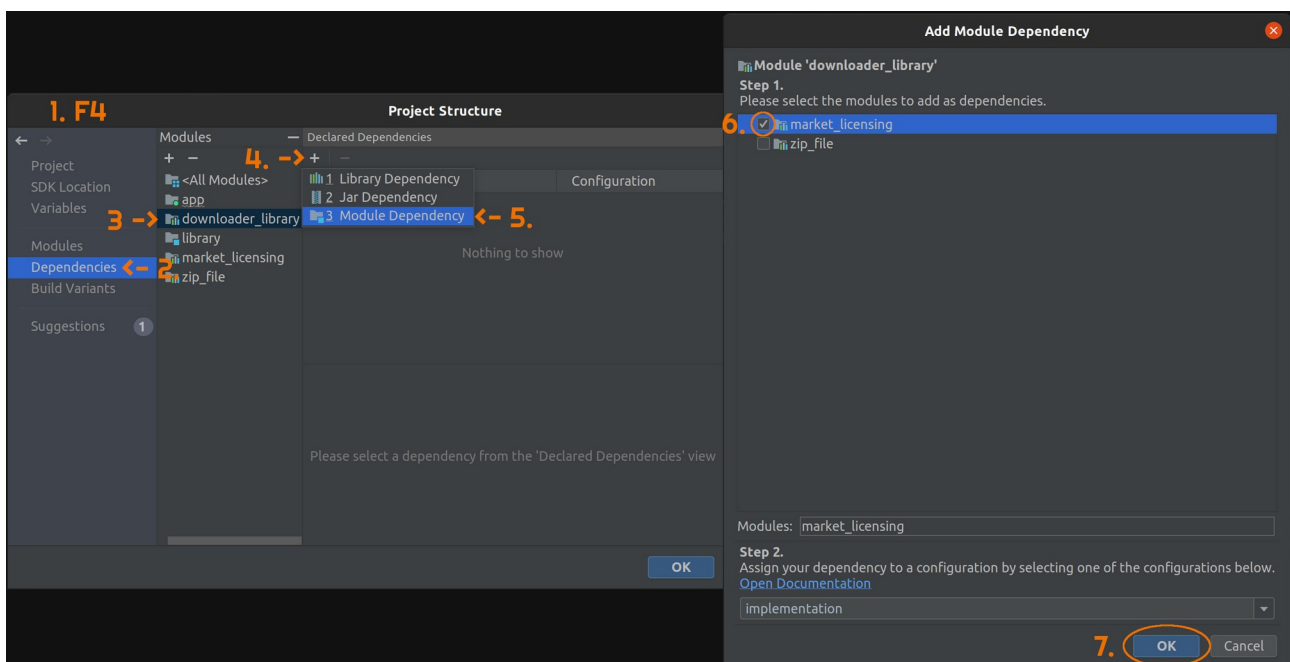
If everything is butter smooth up to this point – we go on!

Setting up the dependencies

If you build the project now, unfortunately you may receive the following error:
“`error: package com.google.android.vending.licensing does not exist`”.

Fix it by following these two steps:

- Open Android Studio → choose any module and press “F4” (projects structure window shall appear) → select “Dependencies” tab (on the left) → choose “downloader_library” module → press “+” sign → select “3 Module Dependency” → in the opened window tick “market_licensing” → press “Ok”.

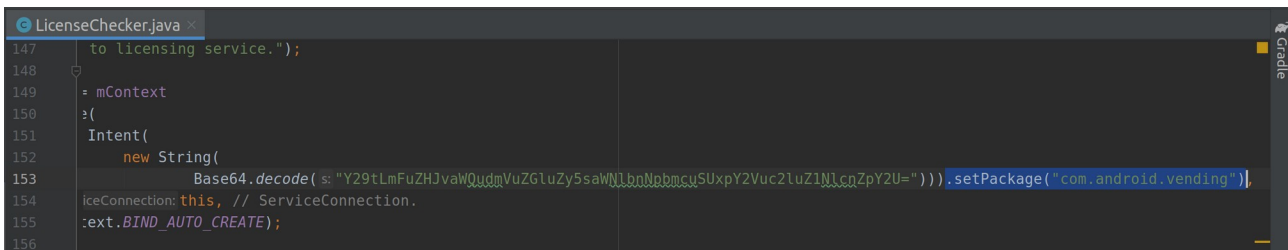


- Choose “app” module → press “+” sign → select “3 Module Dependency” → in the opened window tick “downloader_library” and “zip_file” → press “Ok” → “Apply” → “Ok”.

Cleaning up the damn code

Yep, you will have to hotfix the libraries’ code to avoid running into trouble when we start writing our own expansions handling solution... That is sad, I know.

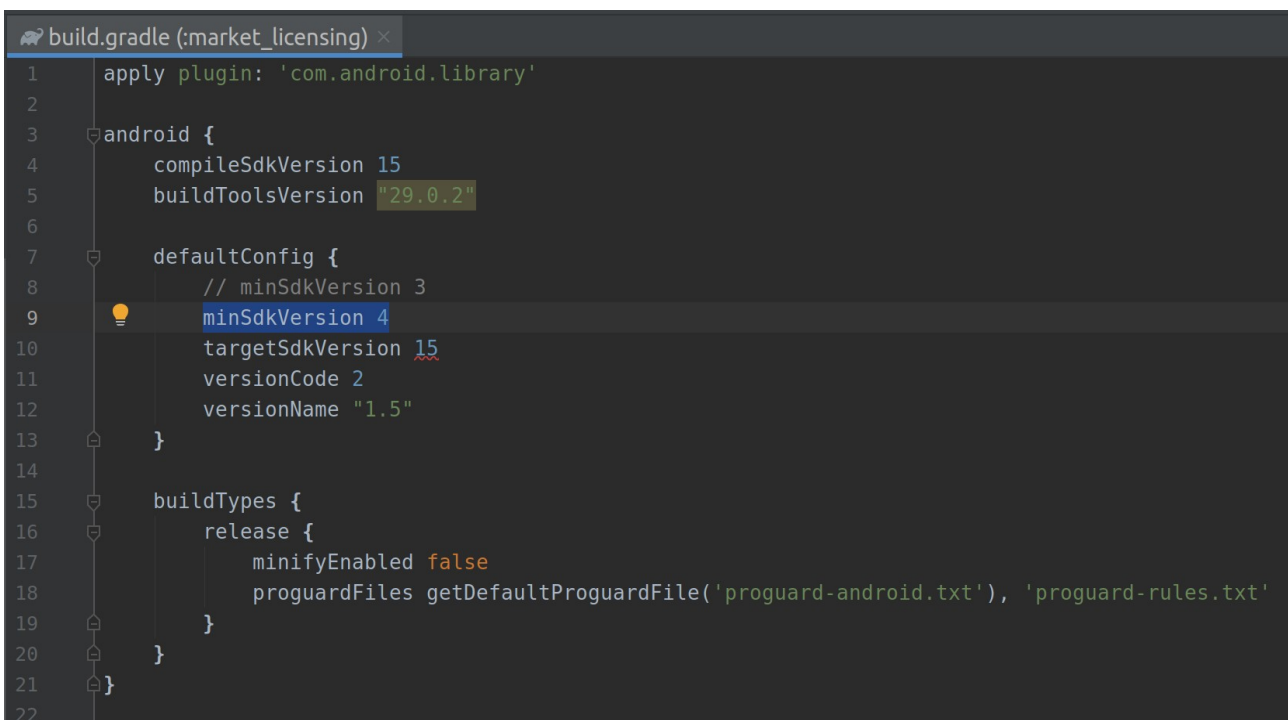
- Open Android Studio → navigate to “market_licensing” module → “java” → “com.google ...” → select “LicenseChecker” java class.
- In the “LicenseChecker.java” class scroll down to the line **153**. Append “.setPackage(“com.android.vending”)” after the **third** closing bracket:



```

147 to licensing service.");
148
149 = mContext
150 :(
151 Intent(
152     new String(
153         Base64.decode( s: "Y29tLmFuZlZlZGluZy5saW1lbnNpbmNlcjUxY2Vuc2luZ1NlcnZpY2U=")))
154         .setPackage("com.android.vending"));
155 iceConnection: this, // ServiceConnection.
156 :ext.BIND_AUTO_CREATE);
  
```

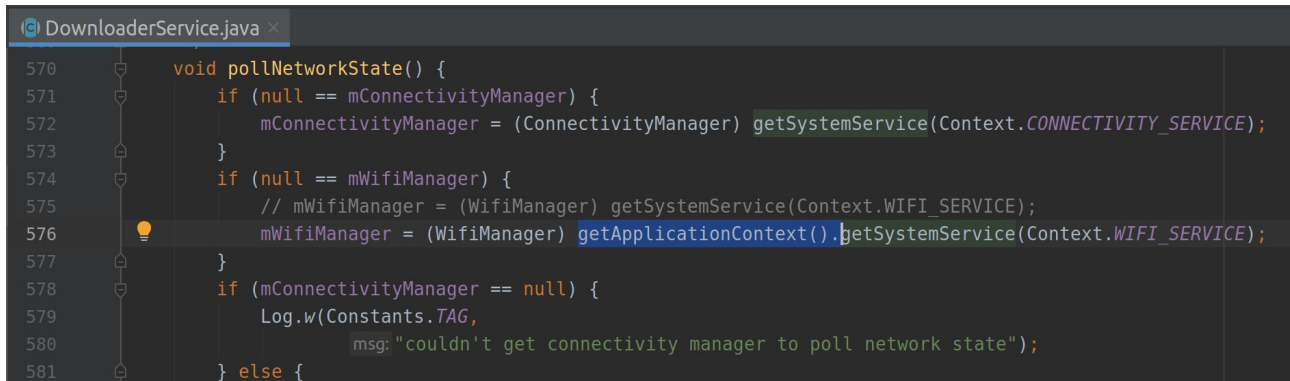
- Browse to “Gradle Scripts” → select “market_licensing build.gradle”. Change “minSdkVersion” to 4 and press “Sync Now”:



```

1  apply plugin: 'com.android.library'
2
3  android {
4      compileSdkVersion 15
5      buildToolsVersion "29.0.2"
6
7      defaultConfig {
8          // minSdkVersion 3
9          minSdkVersion 4
10         targetSdkVersion 15
11         versionCode 2
12         versionName "1.5"
13     }
14
15     buildTypes {
16         release {
17             minifyEnabled false
18             proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.txt'
19         }
20     }
21 }
22
  
```

- Navigate to “downloader_library” module → select “java” → “com.google ...” → “impl” → “Downloader Service” java class.
- In the “DownloaderService.java” class scroll down to the line 575. Insert “`getApplicationContext()`.” after “`(WifiManager)`”:



```

570     void pollNetworkState() {
571         if (null == mConnectivityManager) {
572             mConnectivityManager = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
573         }
574         if (null == mWifiManager) {
575             // mWifiManager = (WifiManager) getSystemService(Context.WIFI_SERVICE);
576             mWifiManager = (WifiManager) getApplicationContext().getSystemService(Context.WIFI_SERVICE);
577         }
578         if (mConnectivityManager == null) {
579             Log.w(Constants.TAG,
580                 msg: "couldn't get connectivity manager to poll network state");
581         } else {

```

Please don’t worry about permissions, they will feel fine after the next section.

Building our own solution

Whoo... That was a bit of work from your side! Now let’s quickly jump into the implementation before your inspiration hasn’t vanished.

Reminder: the **full code** is waiting for you [here](#).

Declaring manifest permissions

Downloader Library requires some permissions to be declared in your app’s manifest file. Without them the lib won’t be able to handle the expansions downloading.

Declare these permissions in the manifest’s “`<manifest ... >`” section:

```

<uses-permission android:name="com.android.vending.CHECK_LICENSE"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

```

The [official documentation](#) states that “`READ_EXTERNAL_STORAGE`” permission is not always a necessity, however I’ve faced immediate app crashes on **android 6** when the permission has **not** been declared and requested at runtime.

Important manifest file modification

Our Downloader Library is based on the outdated Apache library (HTTP client) that has been declared **deprecated** since **Android 9** (API level 28). In order to Android 9+ devices could still download expansions, we must add these lines to manifest’s “<application ... >” section:

```
<uses-library
    android:name="org.apache.http.legacy"
    android:required="false">
</uses-library>
```

Extending necessary classes

Please note that our code has its own, separate package called “`expansions`”.

Downloader Service

The Downloader Library provides a special [Service](#) that we have to extend. It is called “`Downloader Service`” and its main purpose is to provide us with expansions. So generous. (More info [here](#)).

The complete code:

```
public class ExpansionDownloaderService extends DownloaderService
{
    private static final String BASE64_PUBLIC_KEY = "YOUR KEY GOES HERE"; // TODO
    must change with our own key

    private static final byte[] SALT = new byte[]
    {
        }; // TODO fill this in with random values [-128; 127]

    /* The public key goes from your Android Market publisher account.
     * You may find one on Google Console -> your app -> development tools
     */
    @Override
    public String getPublicKey()
    {
```

```

    return BASE64_PUBLIC_KEY;
}

/* The salt has to be unique across other applications,
 * please generate it carefully
 */
@Override
public byte[] getSALT()
{
    return SALT;
}

/* Setting up the alarm receiver */
@Override
public String getAlarmReceiverClassName()
{
    return ExpansionAlarmReceiver.class.getName();
}
}

```

There are two main things in the code that have been left shrouded in secrecy and need to be mentioned separately: the magical “[PUBLIC_KEY](#)” and “[SALT](#)”.

The **public key** is a special thing that Google uses to identify you as a publisher. So you must change the provided String with your own public key. You will find one in Google Console → your app → “Development tools” → “Services & APIs” → “Your license key for this application”. Just copy-paste it and forget.

The **salt** is needed for proper licensing. **The salt has to be unique among other Play Store apps.** Simply generate **20** or more random byte type values and fill them into the array. The chances of collision will be literally miserable.

Important

The [ExpansionDownloaderService](#) class we have created is basically a Service descendant, so we have to declare it in the app’s manifest file.

Add this line into the manifest’s “<application ... >” section.

```
<service android:name=".expansions.ExpansionDownloaderService"/>
```

Alarm Receiver

This little class monitors all the changes that occurred while downloading the expansions.

The code:

```
/* This class handles callbacks when downloading expansions */
public class ExpansionAlarmReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        try
        {
            DownloaderClientMarshaller.startDownloadServiceIfRequired(context, intent,
ExpansionDownloaderService.class);
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

Important

Please add this line into the app's manifest “<application ... >” section.

```
<receiver android:name=".expansions.ExpansionAlarmReceiver"/>
```

Starting the download

As it was mentioned earlier (I count that you remember), a developer must check expansions availability and download them if necessary. It is well recommended to perform the checking mechanism on **each** application launch.

Here comes the MainActivity → onCreate(...) code:

```
/* App's entry point. Here we must check expansions and download the ones if necessary */
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    /* Important note. Here we set activity_main layout */
    setContentView(R.layout.activity_main);

    /* Creating new expansions controller. It is the class which
    * fully embraces expansions
    */
}
```

```

expansionController = new ExpansionController(this);

/* Checking for expansions accessibility */
if (!expansionController.downloadContent())
{
    /* Checking for external storage read permission */
    if (expansionController.checkPermission())
    {
        /* Expansions are in place and permission is already granted,
        * nothing to download so launch the app
        */
        launchTheApp();
    }
}
}

```

This function is an entry point of our program. We set the [ContentView](#) of our app to match the view of the [MainActivity](#), create the [ExpansionsController](#) instance (the class which encapsulates expansions), check whether we need to download the Expansion Files, check for external storage reading permission and if everything is OK, launch an app (you may launch another activity or update the UI with downloaded textures).

Now, what is the [ExpansionController](#) class?

Here is how we download expansions [ExpansionController](#) → [downloadContent\(\)](#) code:

```

/* This function is called from the MainActivity onCreate() method.
 * It checks whether we need to start downloading the expansions or
 * the expansions are already here
 */
public boolean downloadContent()
{
    /* Check expansion delivery */
    if (!expansionFilesDelivered())
    {
        /* Pending intent is used to open this activity from notification.
        * Especially when the app is closed and downloading is complete
        */
        Intent notifierIntent = new Intent(activity, MainActivity.class);
        notifierIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
        Intent.FLAG_ACTIVITY_CLEAR_TOP);

        PendingIntent pendingIntent = PendingIntent.getActivity(activity, 0, notifierIntent,
        PendingIntent.FLAG_UPDATE_CURRENT);

        try
        {
            /* Trying to start downloading */

```

```

        int startResult = DownloaderClientMarshaller.startDownloadServiceIfRequired(activity,
pendingIntent, ExpansionDownloaderService.class);

        /* If response code is not NO_DOWNLOAD_REQUIRED, we must download the
expansion */
        if (startResult != DownloaderClientMarshaller.NO_DOWNLOAD_REQUIRED)
        {
            /* Prepare user interface */
            initUI();
            return true; // Yes, we started downloading the expansion
        }
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}

return false; // No, we didn't start downloading -> expansion is OK
}

```

First of all we check the expansions availability, if the expansions are already in place, there is no special need to download them once again, right? Then we create required notification intents (notification will launch the app when it is closed and downloading is complete) and start downloading. We must start downloading expansions only and only when the received code is **not** `NO_DOWNLOAD_REQUIRED`.

How do we know whether expansions feel fine?

```

/* Simply check the expansion file availability.
 * You might also want to check the patch expansion file here
 */
private boolean expansionFilesDelivered()
{
    /* Firstly check existence of expansion parental directory +
    * delete everything except current expansion files there
    */
    checkExpansions();

    /* Get expansion name from its type and version */
    String expName = Helpers.getExpansionAPKFileName(activity, EXP_IS_MAIN,
EXP_VERSION);

    /* Check whether file with exact name and size exists or delete on mismatch */
    return Helpers.isFileExists(activity, expName, EXP_SIZE, true);
}

```

Here we call `checkExpansions()` function to create missing parent directories for our

expansions and delete unused or misused files there. This situation may well occur when the user deletes (what he was even thinking about?) the whole “obb” directory or copies anything else but expansions to “<our package>” dir. Then we call `getExpansionAPKFileName()` function to generate the path to our expansion file. We use the generated filename and `doesFileExist()` function to check the expansions availability, deleting anything found on mismatch. Not too shabby.

Here is `checkExpansions()` function code:

```
/* This Function:
 * 1) creates parental directories for the expansion file
 * 2) deletes unused files in the parental directory
 */
/* Parental name is actually the name of the package
 */
private void checkExpansions()
{
    File packageFile = activity.getObbDir();

    /* If parental dir does not exist. Also checking for the "obb" directory */
    if (!packageFile.exists())
    {
        /* Create the dirs */
        packageFile.mkdirs();
    }
    else
    {
        /* Listing files in parental dir */
        File[] expansionFiles = packageFile.listFiles();

        if (expansionFiles != null)
        {
            /* Getting current expansion file (you may also need to
             * check the patch one)
             */
            String expName = Helpers.getExpansionAPKFileName(activity, EXP_IS_MAIN,
EXP_VERSION);

            /* Loop through files */
            for (File file : expansionFiles)
            {
                /* If name mismatches -> delete the file */
                if (!file.getName().equals(expName))
                {
                    file.delete();
                }
            }
        }
    }
}
```


We get the abstract parental [path](#) to our expansions, check the path's existence and create the missing directories. However if parental directory exists, we loop through its files and delete everything that mismatches with our expansions.

And the static variables:

```
public static final boolean EXP_IS_MAIN = true;
public static final int EXP_VERSION = 1; // TODO change for your needs
public static final long EXP_SIZE = 57199; // TODO change for your needs
```

You will find the [Expansion File](#) I use in the root folder of the provided [project](#).

The `initUI()` function is right here:

```
/* This function is called when we need to start downloading expansions,
 * it prepares the downloading UI
 */
private void initUI()
{
    /* Create downloader client */
    downloaderClientStub = DownloaderClientMarshaller.CreateStub(this,
ExpansionDownloaderService.class);

    /* Set new layout (downloading screen) */
    activity setContentView(R.layout.download);

    /* Initialize downloading screen */
    expansionPage = new ExpansionPage(activity, this);
}
```

This function is called right before we start downloading the expansions. It creates the **downloader client** and initializes the downloading UI.

Important

The `initUI()` function only creates the downloader client, it does **neither** connect, **nor** disconnect it. We have to perform these actions separately, ideally on activity's `onStart()` and `onStop()` callbacks.

The code:

```
/* On each onStart() we must connect the downloading client */
public void start()
{
```

```

if (downloaderClientStub != null)
{
    /* Connect the client */
    downloaderClientStub.connect(activity);

    /* UI update */
    expansionPage.start();
}

/* On each onStop() we must disconnect the client */
public void stop()
{
    if (downloaderClientStub != null)
    {
        /* Disconnect the client */
        downloaderClientStub.disconnect(activity);

        /* UI update */
        expansionPage.stop();
    }
}

```

`start()` function is called on each `onStart()` callback to **connect** the client. `stop()` function is called on each `onStop()` callback to **disconnect** the client. (`MainActivity` callbacks of course).

Getting notified with downloading progress

So far we have created the downloader client, started downloading the expansions and initialized the corresponding UI. That is a huge amount of work, I totally agree, however our UI has a little bug inside – it doesn't react to the download changes. The user might have lost the wi-fi connection or, on contrary, has fully downloaded the expansions. How could we get notified?

At first we must override this tiny-little function:

```

/* Must override. This is how the service knows which changes
 * it has to show back to the user
 */
@Override
public void onServiceConnected(Messenger m)
{
    remoteService = DownloaderServiceMarshaller.CreateProxy(m);
    remoteService.onClientUpdated(downloaderClientStub.getMessenger());
}

```

After `onServiceConnected()` function is called, we can start sending commands to the service, such as: “Pause the downloading please” or “Be a good boy, resume the downloading for me”, or “I am fine with cellular connection, please proceed”. You definitely got an idea.

And the function which reacts to these commands (cause we have to update the UI) is as follows:

```
/* This function is called when the downloading process changes its state.
 * We can track the change and react respectively
 */
@Override
public void onDownloadStateChanged(int stateId)
{
    /* 1) initially we don't want to show cellular connection message
     * 2) we are downloading, not paused
     * 3) we expect the downloading to end at some point of time
     */
    boolean showCellMessage = false;
    boolean paused = false;
    boolean indeterminate = false;

    switch (stateId)
    {
        case IDownloaderClient.STATE_IDLE:
        case IDownloaderClient.STATE_CONNECTING:
        case IDownloaderClient.STATE_FETCHING_URL:
            indeterminate = true;
            break;

        case IDownloaderClient.STATE_FAILED_CANCELED:
        case IDownloaderClient.STATE_FAILED:
        case IDownloaderClient.STATE_FAILED_FETCHING_URL:
        case IDownloaderClient.STATE_FAILED_UNLICENSED:
        case IDownloaderClient.STATE_PAUSED_BY_REQUEST:
        case IDownloaderClient.STATE_PAUSED_ROAMING:
        case IDownloaderClient.STATE_PAUSED_SDCARD_UNAVAILABLE:
            paused = true;
            break;

        case IDownloaderClient.STATE_PAUSED_NEED_CELLULAR_PERMISSION:
        case
IDownloaderClient.STATE_PAUSED_WIFI_DISABLED_NEED_CELLULAR_PERMISSION:
            paused = true;
            showCellMessage = true;
            break;

        case IDownloaderClient.STATE_DOWNLOADING:
            break;
    }
}
```

```

        case IDownloaderClient.STATE_COMPLETED:
            expansionPage.setFinished(); // set 100% downloaded
            launchTheGame(); // launch the game
            break;

        default:
            paused = true;
            indeterminate = true;
    }

    state = stateId;
    statePaused = paused;

    /* If cellular message state changes */
    if (cellularShown != showCellMessage)
    {
        cellularShown = showCellMessage;
        /* Either show cellular message or hide it */
        expansionPage.triggerCellular(cellularShown);
    }

    /* Simple UI updates */
    expansionPage.triggerIndeterminate(indeterminate);
    expansionPage.updateState(state);
    expansionPage.updateResumePauseButton(statePaused);
}

```

So we basically check (in the huge [switch-case](#)) every possible state that we may come across, change the state-associated variables and update the UI respectively.

Yet we need one more (the last but not the least) function that would notify us with the actual downloading progress.

Here it comes:

```

/* Function is called when downloading changes its progress.
 *
 * Note that we have to manually set 100% progress because
 * this callback function is not called when 100% is reached
 */
@Override
public void onDownloadProgress(DownloadProgressInfo progress)
{
    expansionPage.updateProgress(progress);
}

```

The [DownloadProgressInfo](#) class (our argument) provides us with very handy info: the downloading speed, the downloading progress and the overall total progress. We will use this data to update the UI. Please note that this function is **not** called when 100%

progress is reached. We will have to carry this event separately (see [STATE_COMPLETED](#) case in the previous function).

Yeah, I know this section wasn't a piece of cake, but I will still insist on checking out the complete [code](#). It will give you much wider angle of what is happening. Also don't be shy to open an [issue](#) on github if anything is left unclear. OK?

Testing

As my University dean once said, there are no programs without bugs, there are only programs where bugs haven't been found.

Enter the testing section, here we will try to ~~kill~~ eliminate as many bugs as possible.

At first, we will import the Expansion File to the emulator manually – that is how you will test expansions' reads and modifications locally, on a regular basis. And in the next section we will test the automatic downloading process from Google.

Testing Expansion File reads

If you have cloned the provided [project](#), it is unfairly simple for you to add the [Expansion File](#) to the emulator.

Just follow me:

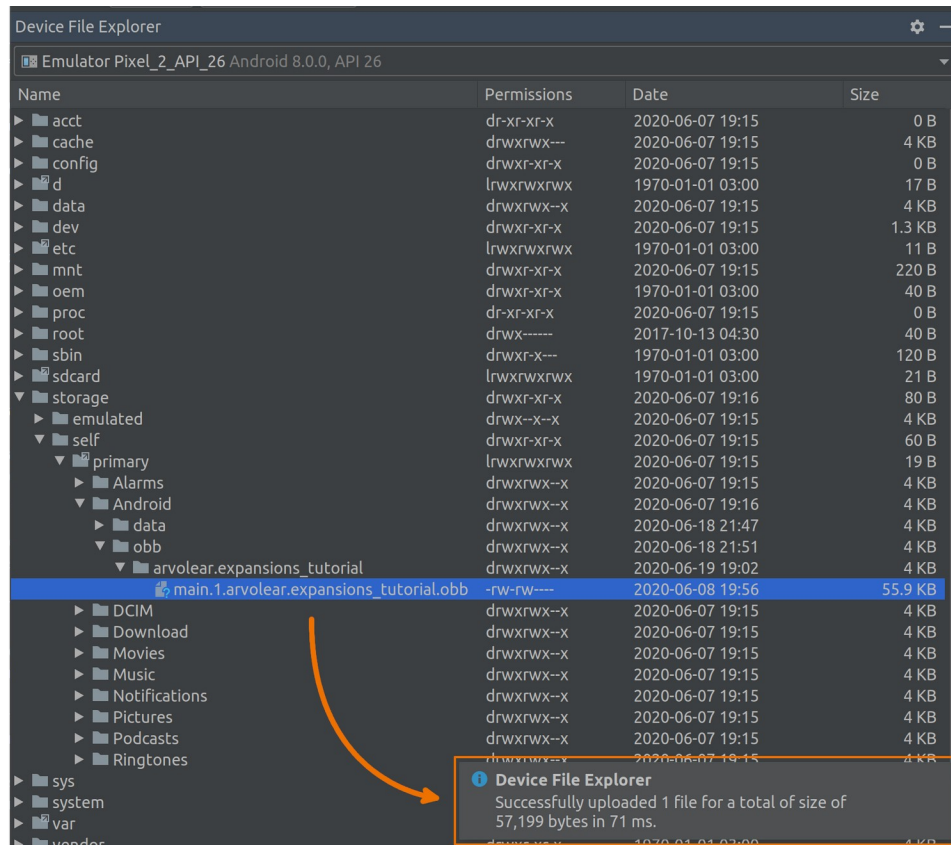
- Open Android Studio → select “View” (on the top) → “Tool Windows” → “Device File Explorer”. If the file system tree is not shown, try launching an emulator.

If you use a different device than I do (Google Pixel 2, API 26) the FS tree might vary a bit. Nevertheless we have taken the same quest – to upload an Expansion File to “obb” directory.

- Open “storage” → “self” → “primary” → “Android” → “obb”.
- Right click “obb” dir → “New” → “Directory”. Enter the name “[arvolear.expansions_tutorial](#)” → press “Ok”. Please, it is very **important** that you enter exactly the same name because it has to match the name of our package.

- Right click “[arvolear.expansions_tutorial](#)” dir → “upload”. Find the file “[main.1.arvolear.expansions_tutorial.obb](#)” in the project’s root directory → press “Ok”.

Your FS tree should end up looking somewhat similar to this one:



After we have successfully imported the expansion, it is high time we tested the access rights (more info [here](#)). As you might recollect, our [MainActivity](#) code is already capable of performing this check (the second [if](#)):

```
/* Checking for expansions accessibility */
if (!expansionController.downloadContent())
{
    /* Checking for external storage read permission */
    if (expansionController.checkPermission())
    {
        /* Expansions are in place and permission is already granted,
        * nothing to download so launch the app
        */
        launchTheApp();
    }
}
```

[checkPermission\(\)](#) function returns **true** if the “external storage read” permission is granted and an attempt to read the Expansion File was successful. When **false** is

returned, the app requests the user to grant the permission and proceeds from the corresponding callback function.

Da code:

```
/* Simple "external storage read" permission checker */
public boolean checkPermission()
{
    /* If permission is not available */
    if (ContextCompat.checkSelfPermission(activity, PERMISSION) ==
PackageManager.PERMISSION_DENIED)
    {
        /* Get expansion file */
        File obb = new File(Helpers.getExpansionAPKFileName(activity, EXP_IS_MAIN,
EXP_VERSION));

        /* Trying to read the expansion file */
        if (!obb.canRead())
        {
            /* Request permission if failed */
            ActivityCompat.requestPermissions(activity, new String[]{PERMISSION},
PERMISSION_CODE);
            return false; // Reading failed
        }
    }

    return true; // Everything is fine, go on
}
```

The constants:

```
public static final String PERMISSION = Manifest.permission.READ_EXTERNAL_STORAGE;
public static final int PERMISSION_CODE = 1; // This may be any number you wish
```

MainActivity permission callback function is as follows:

```
/* Request permissions callback function. It is called from expansionsController
 * when the user accepts or rejects "read external storage" permission request
 */
@Override
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions,
@NonNull int[] grantResults)
{
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);

    /* PERMISSION_CODE is an int that stands for "read external storage" permission.
     * Yet it is an arbitrary variable
     */
}
```

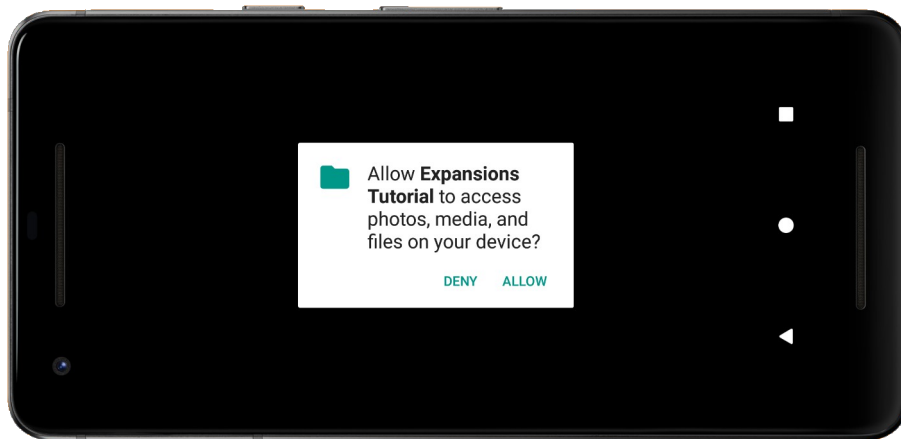
```

if (requestCode == ExpansionController.PERMISSION_CODE)
{
    /* We will launch the app if permission is granted */
    if (grantResults.length > 0 && grantResults[0] ==
PackageManager.PERMISSION_GRANTED)
    {
        launchTheApp();
        return;
    }

    Toast.makeText(this, "Unable to launch the game without permission",
Toast.LENGTH_LONG).show();
    finish(); // Close the app if permission is rejected
}
}

```

Basically `checkPermission()` function launches this window:



And `onRequestPermissionsResult()` function reacts to user's reply.

Reading expansions

So far so good. We have imported the Expansion File to the emulator and figured out how to get required access rights.

But how can we extract information from expansions?

Via [ZIP library](#). Yes, the one from the “Setting up the libraries” section.

Here is the code sample:


```
private ZipResourceFile expansionFile;

.
.
.

/* Here we use zip_file library.
 * Trying to get the expansion file with the specified version
 */
expansionFile = APKExpansionSupport.getAPKExpansionZipFile(activity,
ExpansionController.EXP_VERSION, ExpansionController.EXP_VERSION);
```

Reading an image asset:

```
/* Reading from expansions file */
stream = expansionFile.getInputStream(path);

/* Creating bitmap from the stream of bytes */
Bitmap newBitmap = BitmapFactory.decodeStream(stream);

/* Closing the stream */
stream.close();
```

Reading a media asset:

```
/* Getting the file descriptor */
AssetFileDescriptor descriptor = expansionFile.getAssetFileDescriptor(path);

/* Specifying descriptor's starting position and ending position */
mediaPlayer.setDataSource(descriptor.getFileDescriptor(), descriptor.getStartOffset(),
descriptor.getLength());

descriptor.close(); // Don't forget to deallocate resources
```

At first we declare a `ZipResourceFile` instance – imagine it describing a virtual file system over a specified ZIP archive. Then we call static `getAPKExpansionZipFile()` function to get our Expansion File and assign it to the `expansionFile` variable. In order to extract an image asset from the expansion, we decode a byte stream indicated by `path` variable. And to extract a media asset, we use the associated file descriptor.

Please note that `path` has to be **relative** to the root folder of the ZIP archive.

No more code, I promise!

Testing Expansion File downloads

In the previous section we have tested expansions' reads by manually creating necessary directories and uploading the file to the emulator. But you know, it is not a real world scenario. Our code has to be robust enough to complete these steps automatically. Unsupervised.

Unfortunately this section won't cover all the possible problems you may encounter with this piece of ... But it would definitely help you proceed if you got stuck.

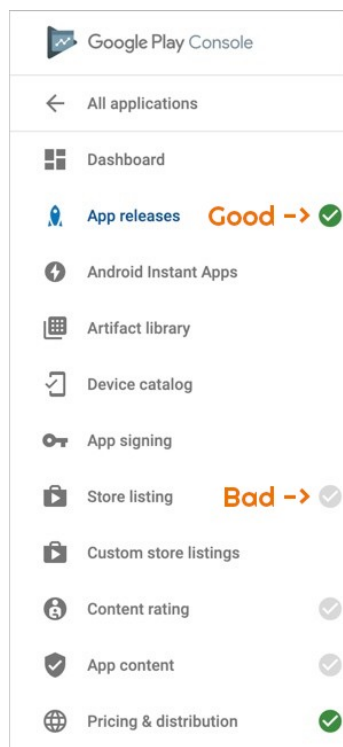
Starting with the big G, that is what Google [has](#) regarding the File Downloads. Pretty shallow as for me... Some elbow grease to dig the truth out is definitely required.

In the first place, we need to create an “[Internal test track](#)” if your app hasn't been published yet. In other words if your app is already in “beta”, “release” or anything else you will only need an update, **not** the internal test track.

Creating Internal test track

If you haven't created an application on Google Console [main page](#) yet, please tell me, what are you waiting for?

Choose your app in Google Console and fill in everything that is required to fulfill those “ticks in bubbles” on the left. You won't be able to publish an app if any of them are left grayish. Very strange explanation but still...



Navigate to “Release management” → “App releases” → scroll down to “Internal test track” → click “MANAGE” → click “CREATE RELEASE” on the top.

Important

Before clicking ~~seary~~ familiar “BROWSE FILES” button, please prepare your app, i.e. create an Expansion File (according to the rules), move (or delete) the “assets” directory to reduce the app’s size and test everything thoroughly.

Click the “BROWSE FILES” button and find your **signed** APK. Wait for it to upload and press the little-tiny “+” sign on the right → under “Use expansion file” drop down menu choose “Upload a new file” → find and select your Expansion File in the opened window. Press the “Save” button. Fill in “What’s new” field if you want to and click “SAVE” → “REVIEW”. Scroll down and click “START ROLLOUT TO INTERNAL TEST” → confirm everything.

Please note that if it is your first rollout, it may take up to a **week** for Google to accept the app.

What else to consider?

[Internal test track](#) is a special release branch that is only available to a particular group of users. You can create a list of testers in “Release management” → “App releases” → “Internal test track” → “Manage testers”.

Press “CREATE NEW LIST” and follow the instructions.

Important

You don’t need to add yourself to the testers list, but you will need to **log in** to your Google account on the testing (emulator) device.

Also bear in mind that the downloading process will start **only** and **only** when one has downloaded an app from Play Store. **No “Shift+F10” – Play Store only.**

If you have followed along this tutorial accurately, your app should start downloading

the Expansion Files flawlessly. Suppose you will have to delete them at first, after downloading the app from Play Store. Just don't delete too much and everything will be perfecto del mundo.

Anything left unclear? Don't beat around the bush to leave the comments below, I will try to sort the things out.

Updating

You have gone extremely far, please accept my respect. However we have one more topic to be discussed – app updating (more info [here](#)).

From Google point of view, the process of updating an app with expansions is essentially the same as creating an Internal test track (see previous topic). Basically when the app is ready to be updated, you will enter a new `versionCode` in the app's module build gradle, create a signed APK and upload everything to Google Console.

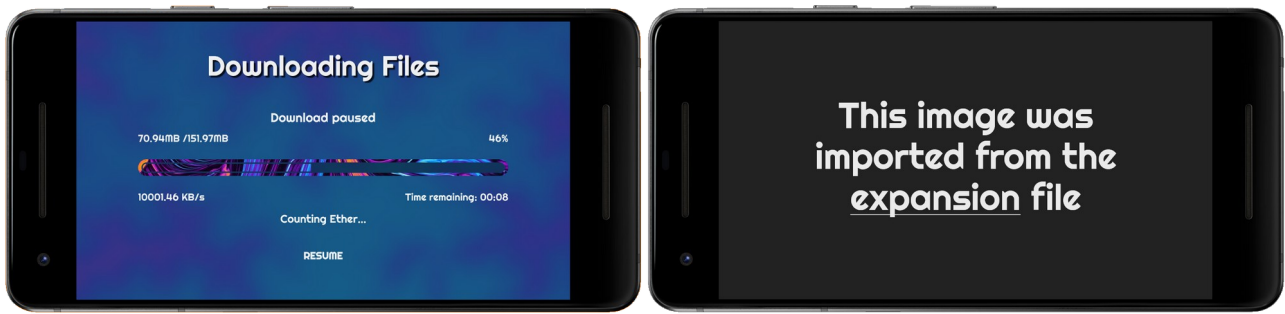
But there are two thing to consider:

The first thing is that Google Console **allows** developers to **reuse** previously uploaded Expansion Files. This feature is extremely helpful, because if you decide to upload a new Expansion File (with new version number) all your users will have to download this new file with an app's update. Just imagine this file being 2GB... Whoof... So what you have to do is reuse expansions whenever it is possible. No code modifications are needed, just the old file in Google Console. Yet if you decide to reuse expansions, the versions of the Expansion Files will differ from the version of your app, however it is a very common situation.

And the second thing is that if you need to update an Expansion File, you will still have to create a new signed APK (with new version code). Please bear in mind that uploading a new Expansion File means **not** saving users' bandwidth because of the downloading necessity. That's why we have **main** and **patch** Expansion Files: the first one for permanent assets and the second one for regular updates. Also note that when one updates an app, previous expansions will be **overwritten**.

Result

Oh dear, results time!



On the left you can see the downloading screen and on the right – successful Expansion File reading.

That's it! I hope you enjoyed this exhaustive tutorial and found inspiration to write better code!

*feel free to leave a star [here](#) if you like the article