# Android example :

## 1. Prerequisite

- Android Studio ➔ Installation : https://developer.android.com/studio/install

## 2. Android API

Kalima provides an API for Android in the form of an aar library, allowing to launch a service which connects to the blockchain, and which provides the necessary elements to communicate with that service.

Depending on the need, one can follow the Client Tutorial and simply use the Kalima JAR. However, the advantage of a service is that it can run as a background task and allow data reception even when the Android app is closed (to keep on receiving notifications, for example).

In this document, we are going to explain the different steps to set up that service in order to connect to the Kalima blockchain.

Please note the project needs to be in Android X in order to function with the service.

### a. Adding the library in the project

First, you need import the Kalima Android service library in your project.

Place kalima-android-lib-release.aar in the « libs » folder of your module. For example, if your module is called app, place the library in app/libs/. Then, add in the Gradle of your module:

```
repositories {
    flatDir {
        dirs 'libs'
    }
}
```

As well as :

```
dependencies {
    implementation (name: 'kalima-android-lib-release', ext: 'aar')
}
```

### b. Kalima Service Preferences

KalimaServicePreferences is an object which will allow us to configure a number of elements via the Android shared preferences. It's the equivalent of the configuration file we use with the Kalima JAR.

Below is an example of configuration :

```
// Create android preferences to configure KalimaService for this app
KalimaServicePreferences kalimaServicePreferences =
KalimaServicePreferences.getInstance(getApplicationContext(), APP_NAME);
kalimaServicePreferences.setNodeName("ExampleNode");
// You need to choose an unused port
kalimaServicePreferences.setServerPort(9100);
// Notaries address and ports (ipAddress:port), separated with ","
kalimaServicePreferences.setNotariesList("62.171.131.154:9090,62.171.130.233:9090
,62.171.131.157:9090,144.91.108.243:9090");
// Directory for Kalima files storage
kalimaServicePreferences.setFilesPath(Environment.getExternalStorageDirectory().t
oString() + "/Kalima/jit/KalimaAndroidExample/");
// Path for log directory
kalimaServicePreferences.setLogPath(Environment.getExternalStorageDirectory().toS
tring() + "/Kalima/jit/KalimaAndroidExample/");
kalimaServicePreferences.setLedgerName("KalimaLedger");
// Set list of CachePaths we want for notifications
kalimaServicePreferences.setNotificationsCachePaths(new
ArrayList<String>(Arrays.asList("/StAubin/Underfloor_1/alarms")));
// Set class path of notification receiver
// You will receive broadcast in this receiver when new data arrives in
CachesPaths you choose with setNotificationsCachePaths
// Then you can build a notification, even if the app is closed
kalimaServicePreferences.setNotificationsReceiverClassPath("org.kalima.kalimaandr
oidexample.NotificationReceiver");
```

The utilities of the different configuration are :

- setNodeName ➔ Use a name which allows you to recognize your node
- setServerPort ➔ In principle, this port isn't used on the client nodes. You can choose any port you want.
- setNotariesList ➔ Allows you to configure the list of addresses and ports of the Notary Nodes which you want to connect to, in the form of ip: port, separated by commas.
- setFilesPath ➔ Allows you to choose where the necessary files for Kalima will be stored, as well as the possible logs.
- setNotificationsCachePath ➔ Allows you to choose the cache paths on which you want to be notified (this principle will be explained in detail later).
- setNotificationsReceiverClassPath ➔ Allows you to indicate the classpath which will manage the notifications to the service (this principle will be explained in detail later).

### c. Kalima Cache Callback

The KalimaCacheCallback object will allow us to take action when there are interactions with the blockchain. Below is an example of instantiation :

```
kalimaCacheCallback = new KalimaCacheCallback.Stub() {
  @Override
  // Callback for incoming data
  public void onEntryUpdated(String cachePath, KMsgParcelable kMsgParcelable)
  throws RemoteException {
    Log.d("onEntryUpdated", "cachePath="  + cachePath + ", key=" +
    kMsgParcelable.getKey());
  }

  // Callback for deleted data
  @Override
  public void onEntryDeleted(String cachePath, String key) throws RemoteException
  {
    Log.d("onEntryDeleted", "cachePath="  + cachePath + ", key=" + key);
  }

  @Override
  public void onConnectionChanged(int i) throws RemoteException {
  }
};
```

As we can notice, 3 callbacks are available :

- onEntryUpdated ➔ This callback will be called every time a piece of data will be added/modified in cache memory. The callback has two parameters : the cachePath which allows to know on which cache the piece of data has arrived, and the message in the form of KMsgParcelable (which allows retrieving data of the message). An example of a simple use of that Callback would be the update of a list of messages. A message has been added or modified > so you update the list of messages.
- onEntryDeleted ➔ This callback will be called every time a message will be deleted in cache memory. It has two parameters: the cachePath on which the message has been deleted, as well as the key to the deleted message. An example of a simple use of that callback would be the update of a list of messages. A message is deleted, which means you have removed it from your display list.
- onConnectionChanged ➔ This callback will be called every time there is a disconnection or a reconnection with one of the Notary Nodes. This Callback can be useful for example to alert users that it's not connected to the blockchain, or that one of the notary nodes isn't responding anymore.

To function correctly, we have to implement our Callback during the connection between our activity and the service (which will be detailed later), thanks to the line below:

```
kalimaServiceAPI.addKalimaCacheCallback(kalimaCacheCallback);
```

When our activity isn't at the forefront anymore, we'll need to delete the callback's implementation thanks to the line below. Example: it can also be used, in the « onPause » feature of an activity's lifecycle:

```
kalimaServiceAPI.unregisterKlimaCacheCallback(kalimaCacheCallback);
```

### d. Connection between an activity and the service

Before being able to communicate with the service from an activity, they have to be « connected » together. To do so, you have to call bindService at the beginning of the activity, in the onResume feature for example:

```
Intent intent = new Intent(this, KalimaService.class);
intent.putExtra(KalimaService.APP_NAME, APP_NAME);
bindService(intent, serviceConnection, BIND_AUTO_CREATE);
```

And this, by instantiating serviceConnection beforehand :

```
serviceConnection = new ServiceConnection() {

  @Override
  public void onServiceConnected(ComponentName name, IBinder service) {
    kalimaServiceAPI = KalimaServiceAPI.Stub.asInterface(service);

    try {

      kalimaServiceAPI.addKalimaCacheCallback(kalimaCacheCallback);

    } catch (RemoteException e) {
      e.printStackTrace();
    }

  }

  @Override
    public void onServiceDisconnected(ComponentName name) {
  }
};
```

The onServiceConnected feature will be called when your activity will be connected correctly to the service. Then, it is in that feature that you have to initialize kalimaServiceAPI (which will be discussed later) and add our possible callback(s). You can also add code at that moment, to initialize a display list with the data present in cache memory at that instant for example.

### e. Kalima Service API

The KalimaServiceAPI object will allow to communicate with the service to interact with the blockchain. Those are the main features it's composed of :

- get(String cachePath, String key) ➔ Returns the message with the « key » key present in the "cachePath" cache, via an object like KMsgParcelableRetourne
- getAll(String cachePath, boolean reverseDirection, KMsgFilter) ➔ Returns an ArrayList of KMsgParcelable of all the messages present in the "CachePath" cache. We can change the order of the ArrayList if needed via "reverseDirection" and add a filter via KMSgFilter.
- set(String cachePath, String key, byte[] body, String tll) ➔ Allows to create a transaction on the blockchain. The TTL (Time To Live) allows choosing the lifesapn of the message in seconds (-1 to have the message being always active)
- delete( String cachePath, String key) ➔ Creates a transaction on the blockchain to delete the "key" message in the "cachePath" cache.

### f. The start of the service

Once started, the service doesn't stop. Therefore it's best to start it :

- At the start of the phone thanks to a BroadcastReceiver ➔ https://www.dev2qa.com/how-to-start-android-service-automatically-at-boot-time/#:~:text=Start%20Android%20Service%20When%20Boot,android%20monitor%20log%20cat%20console.
- At the launch of an application, meaning in your first activity:

Service start example :

```
@Override
protected void onResume() {
  super.onResume();
  if(permissionStorage) {
    Intent intent = new Intent(this, KalimaService.class);
    intent.putExtra(KalimaService.APP_NAME, APP_NAME);
    if(!isMyServiceRunning()) {
      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        startForegroundService(intent);
      } else {
        startService(intent);
      }
    }
    bindService(intent, serviceConnection, BIND_AUTO_CREATE);
  }
}
```

isMyServiceRunning feature detail :

```
private boolean isMyServiceRunning() {
  ActivityManager manager = (ActivityManager)
  getSystemService(Context.ACTIVITY_SERVICE);
  if (manager != null) {
    for (ActivityManager.RunningServiceInfo service :
    manager.getRunningServices(Integer.MAX_VALUE)) {
      if (KalimaService.class.getName().equals(
      service.service.getClassName())) {
        return true;
      }
    }
  }
return false;
}
```

### g. Authorizations

To run correctly, the service needs authorizations for storage. That's why the main activity asks for the following authorization to the user :

```
private void checkPermissions(){
  if(ContextCompat.checkSelfPermission(getApplicationContext(),
  Manifest.permission.WRITE_EXTERNAL_STORAGE)!=
  PackageManager.PERMISSION_GRANTED){
    ActivityCompat.requestPermissions(this, new
    String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE}, 0);
  } else {
    permissionStorage = true ;
  }
}
```

Also, some permissions need to be added to the manifest :

```
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
```

### h. Notifications

The service is conceived to react to transactions even when the application is closed (to create notifications, for example). We saw earlier that we can choose the cache paths for which we want to be notified, as well as the class managing the notifications.

In fact, the service will send a broadcast when the message arrives on one of the chosen cache paths. To create notifications, you have to create a class which comes from the Broadcast receiver (the class of which you have passed the path via KalimaServicePreferences).

This class needs to be declared in the Manifest in the following manner :

```
<receiver android:name=".NotificationReceiver"
  android:exported="true">
  <intent-filter >
    <!-- always set name to org.kalima.NOTIFICATION_ACTION -->
    <action android:name="org.kalima.NOTIFICATION_ACTION"/>
  </intent-filter>
</receiver>
```

Here’s an example of implementation of the Broadcast Receiver:

```
public class NotificationReceiver extends BroadcastReceiver {

    /*
    With this class, you can launch notifications even if your app is
closed.
    To achieve that, you need to declare this receiver in your Manifest
file.
    See example in AndroidManifest.xml.
     */

    private final String CHANNEL_ID =
"org.kalima.kalimandroidexample.notifications";
    private static int NOTIFICATION_ID = 0;

    @Override
    public void onReceive(Context context, Intent intent) {
        KMsgParcelable kMsgParcelable =
intent.getParcelableExtra(KalimaService.EXTRA_MSG);
        String cachePath =
intent.getStringExtra(KalimaService.EXTRA_CACHE_PATH);
        Log.d("notification", "cachePath=" + cachePath + " key=" +
kMsgParcelable.getKey());

        // You can choose an activity to start, when user click on
notification
        Intent dialogIntent = new Intent(context, MainActivity.class);
        PendingIntent pendingIntent = PendingIntent.getActivity(context,
0, dialogIntent, 0);
        createNotificationChannel(context);

        NotificationCompat.Builder builder = new
NotificationCompat.Builder(context, CHANNEL_ID)
                .setSmallIcon(android.R.drawable.stat_sys_warning)
                .setContentTitle(cachePath)
                .setContentText(kMsgParcelable.getKey())
                .setContentIntent(pendingIntent)
                .setAutoCancel(true)
                .setPriority(NotificationCompat.PRIORITY_HIGH);

        NotificationManagerCompat notificationManager =
NotificationManagerCompat.from(context);
        notificationManager.notify(NOTIFICATION_ID, builder.build());
        NOTIFICATION_ID++;
    }

    private void createNotificationChannel(Context context) {
        // Create the NotificationChannel, but only on API 26+ because
        // the NotificationChannel class is new and not in the support
library
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            String description = "Alarms channel";
            int importance = NotificationManager.IMPORTANCE_HIGH;
            NotificationChannel channel = new
NotificationChannel(CHANNEL_ID, CHANNEL_ID, importance);
            channel.setDescription(description);
            // Register the channel with the system; you can't change
the importance
            // or other notification behaviors after this
            NotificationManager notificationManager =
context.getSystemService(NotificationManager.class);
            notificationManager.createNotificationChannel(channel);
        }
    }
}
```

## 3. Execution, installation

You can directly test your application on your computer via the Android emulator, or test it on your own Android device, or generate an APK to install afterwards on any Android device.

### a. Android Emulator

To create a virtual Android device on your computer, please follow this tutorial: https://developer.android.com/studio/run/managing-avds

Then you can test your application on the virtual device :

https://developer.android.com/studio/run/emulator

### b. Installation on a physical device :

To install the application directly on your device :

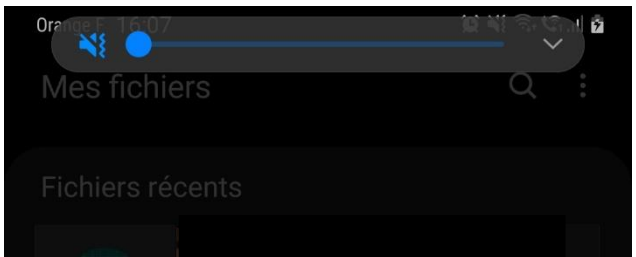
https://developer.android.com/studio/run/device

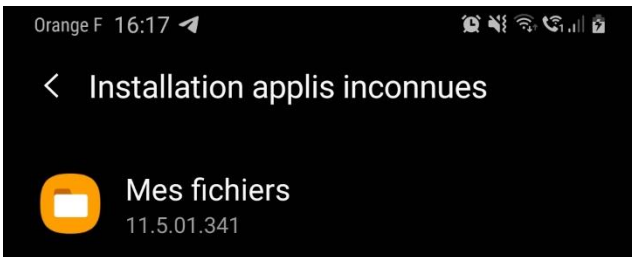
### c. Generating the APK

You can also generate an APK to test your application. To do so, you need to install that APK on a device to install the application.

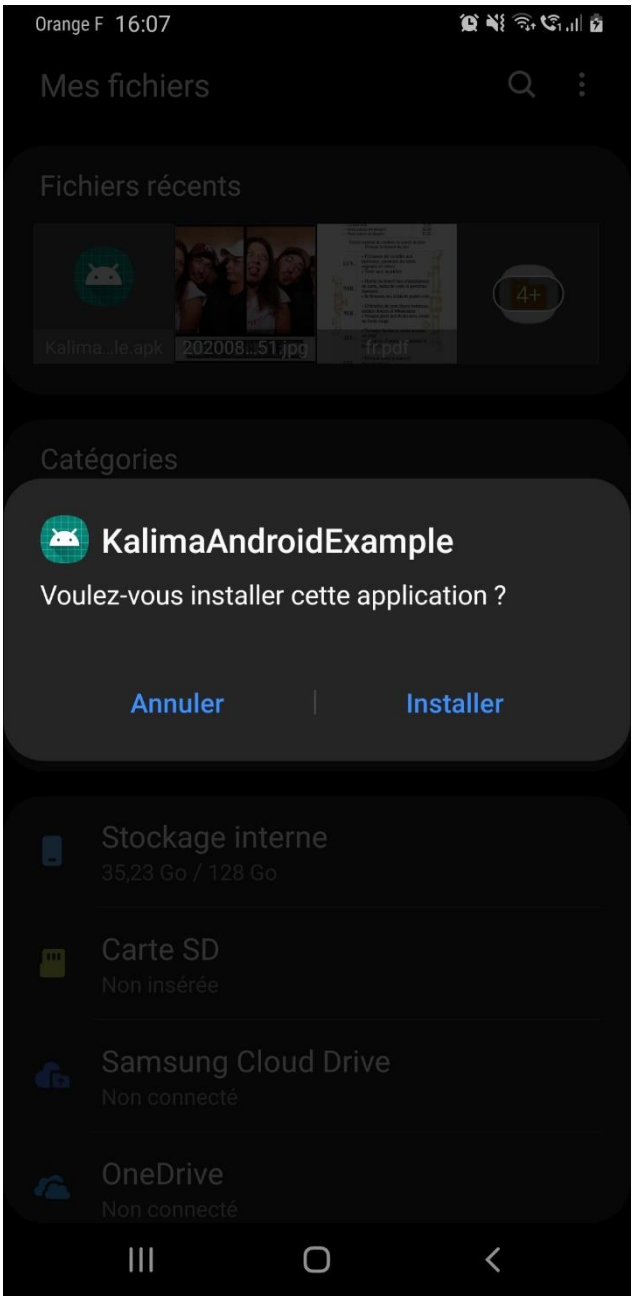
To generate the APK : https://medium.com/@ashutosh.vyas/create-unsigned-apk-in-android-325fef3b0d0a

Then, paste the APK on your device, and go to its location via the “My Files” application. Click on the APK to install it. The installation of applications via « My Files » needs to be authorized. If a message appears at that point, click on parameters and allow the installation (see screenshots below).



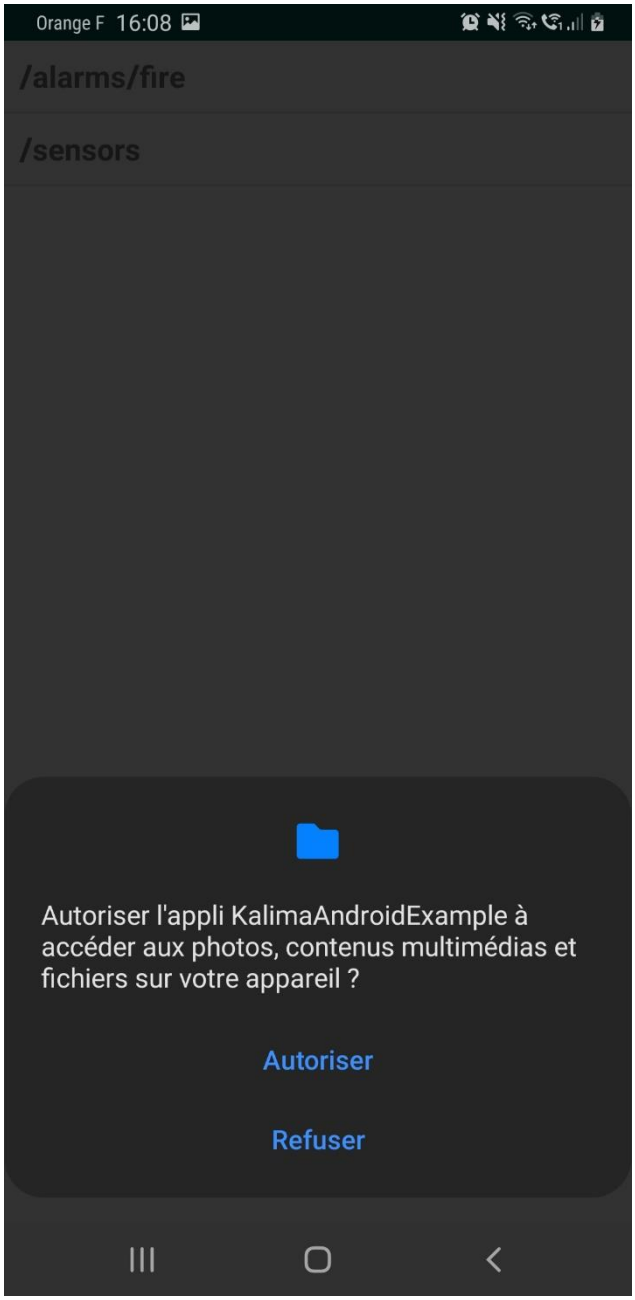



Lastly, you can install the application :



## d. Operation of the application

At the first launch, the application asks for an authorization to have access to your files. This authorization is necessary for the storage of files useful and needed for Kalima. Once the authorization is accepted, a page will open automatically. That page allows to configure your serialId. The serialId needs to be provided by Kalima. It allows you to be authentified on the blockchain.




On the application's main page, you can see the list of the existing tables (/alarms/fire and /sensors). At the bottom of the page, you can send a text. That text is sent in /alarms/fire with the unique key called "key". If you are authorized on the blockchain, the sending and reception should work correctly. You'll receive a notification at each sending. Moreover, if you click on /alarms/fire, you'll see the last message received.