**KBeacon SDK**

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Change Description** | **Author** |
| V1.0 | 2019/4/1 | first version | Ning |
| V0.5 | 2019/10/11 | add KSesnor function | Ning |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**C O N F I D E N T I A L**

This document is the property of KKM Co.Ltd. KKM retains all rights pertaining to industrial property including patent applications. This document is only for the recipient(s) which authorized by KKM. It contains confidential information and any use, dissemination, distribution, or reproduction of this message by unintended recipients is not authorized and may be unlawful. **Catalogue**

[1. Introduction 3](#_Toc21899548)

[2. Android demo 4](#_Toc21899549)

[3. Import SDK to project 4](#_Toc21899550)

[3.1 Prepare 4](#_Toc21899551)

[3.2 Import SDK 4](#_Toc21899552)

[4. How to use SDK 5](#_Toc21899553)

[4.1 Scanning device 5](#_Toc21899554)

[4.2 Connect to device 7](#_Toc21899555)

[4.3 Configure parameters 8](#_Toc21899556)

[4.3.1 Advertisment type 8](#_Toc21899557)

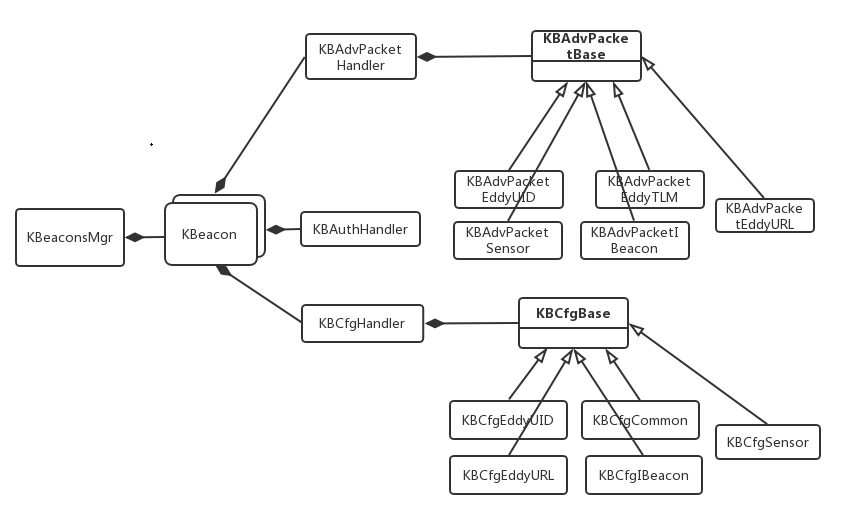
[4.3.2 Get device parameters 9](#_Toc21899558)

[4.3.3 Update device parameters 10](#_Toc21899559)

[5. Special instructions 12](#_Toc21899560)

# Introduction

With this SDK, you can scan and configure the KBeacon device. The SDK include follow main class:

* KBeaconsMgr: Global definition, responsible for scanning KBeacon devices advertisment packet, and monitoring the Bluetooth status of the system;
* KBeacon: An instance of a KBeacon device, KBeaconsMgr creates an instance of KBeacon while it found a physical device. Each KBeacon instance has three properties: KBAdvPacketHandler, KBAuthHandler, KBCfgHandler.
* KBAdvPacketHandler: parsing advertisement packet. This attribute is valid during the scan phase.
* KBAuthHandler: responsible for the authentication operation with the KBeacon device after the connection is established.
* KBCfgHandler：responsible for configuring parameters related to KBeacon devices

Scanning Stage

in this stage, KBeaconsMgr will scan and parse the advertisement packet about KBeacon devices, and it will create "KBeacon" instance for every founded devices, developers can get all advertisements data by its allAdvPackets or getAdvPacketByType function.

Connection Stage

After a KBeacon connected, developer can make some changes of the device by modifyConfig.

# Android demo

To make your development easier, we have two android demos in github. They are:

* eddystonedemo: The app can scan KBeacon devices and configure Eddystone URL, TLM, UID related parameters. this SDK are introduced with reference to this demo.
* ibeacondemo: The app can scan KBeacon devices and configure iBeacon related parameters.

# Import SDK to project

## Prepare

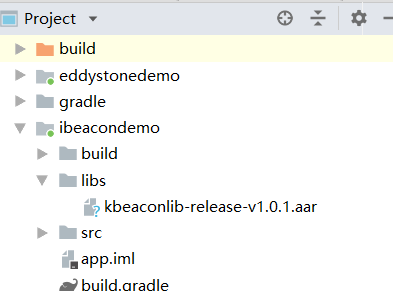
Development environment:

Android Studio

minSdkVersion 21

## Import SDK

1. Copy the development kit kbeaconlib-release-xx.aar file into the libs directory and add dependencies in build.gradle. As shown below:



2. Edit "build.gradle" file under the APP project

**a. Add follow line in build.gradle:**

repositories {

flatDir {

dirs 'libs' // aar lib directory

}

}

**b. add kbeaconlib to dependencies**

dependencies {

…

compile(name:'kbeaconlib-release-v1.0.1', ext:'aar')

}

3. Add the Bluetooth permissions and the corresponding component registration under the AndroidManifest.xml file. As follows:

<**uses-feature  
 android:name="android.hardware.bluetooth\_le"**

**android:required="true"** />  
  
<**uses-permission android:name="android.permission.BLUETOOTH"**/>  
<**uses-permission android:name="android.permission.BLUETOOTH\_ADMIN"**/>  
<**uses-permission android:name="android.permission.ACCESS\_COARSE\_LOCATION"** />

# How to use SDK

## Scanning device

1. Get KBeaconMgr instance in Activity, also your application should implementation the scanning callback.

@Override  
**public void** onCreate(Bundle savedInstanceState) {

//other code

*//get KBeacon central manager instance***mBeaconsMgr** = KBeaconsMgr.*sharedBeaconManager*(**this**);  
**if** (**mBeaconsMgr** == **null**)  
{  
 toastShow(**"Make sure the phone supports BLE funtion"**);  
 **return**;  
}  
**mBeaconsMgr**.**delegate** = **beaconMgrDeletate**;

//…

}

2. Request permission:

In Android-6.0 or later, Bluetooth scanning requires location permissions, so the app should request permission before start scanning like follows:

**if** (ContextCompat.checkSelfPermission(**this**, Manifest.permission.***ACCESS\_COARSE\_LOCATION***)  
 != PackageManager.PERMISSION\_GRANTED) {  
 ActivityCompat.*requestPermissions*(**this**,  
 **new** String[]{Manifest.permission.***ACCESS\_COARSE\_LOCATION***}, PERMISSION\_COARSE\_LOCATION);  
}

3. Start scanning

**int** nStartScan = **mBeaconsMgr**.startScanning();  
**if** (nStartScan == 0)  
{  
 Log.*v*(***TAG***, **"start scan success"**);  
}  
**else if** (nStartScan == KBeaconsMgr.***SCAN\_ERROR\_BLE\_NOT\_ENABLE***) {  
 toastShow(**"BLE function is not enable"**);  
}  
**else if** (nStartScan == KBeaconsMgr.***SCAN\_ERROR\_NO\_PERMISSION***) {  
 toastShow(**"BLE scanning has no location permission"**);  
}  
**else**{  
 toastShow(**"BLE scanning unknown error"**);  
}

3. Before start scanning, the app should implementation KBeaconMgr delegate for get scanning result

*//get advertisement packet during scanning callback*KBeaconsMgr.KBeaconMgrDelegate **beaconMgrDeletate** = **new** KBeaconsMgr.KBeaconMgrDelegate()  
{  
 *//get advertisement packet during scanning callback* **public void** onBeaconDiscovered(KBeacon[] beacons)  
 {  
 **for** (KBeacon beacon: beacons)  
 {  
 *//get beacon adv common info* Log.*v*(***LOG\_TAG***, **"beacon mac:"** + beacon.getMac());  
 Log.*v*(***LOG\_TAG***, **"beacon name:"** + beacon.getName());  
 Log.*v*(***LOG\_TAG***,**"beacon rssi:"** + beacon.getRssi());  
  
 *//get adv packet* **for** (KBAdvPacketBase advPacket : beacon.allAdvPackets())  
 {  
 **switch** (advPacket.getAdvType())  
 {  
 **case** KBAdvType.***KBAdvTypeIBeacon***:  
 {  
 KBAdvPacketIBeacon advIBeacon = (KBAdvPacketIBeacon)advPacket;  
 Log.*v*(***LOG\_TAG***,**"iBeacon uuid:"** + advIBeacon.getUuid());  
 Log.*v*(***LOG\_TAG***,**"iBeacon major:"** + advIBeacon.getMajorID());  
 Log.*v*(***LOG\_TAG***,**"iBeacon minor:"** + advIBeacon.getMinorID());  
 **break**;  
 }  
  
 **case** KBAdvType.***KBAdvTypeEddyTLM***:  
 {  
 KBAdvPacketEddyTLM advTLM = (KBAdvPacketEddyTLM)advPacket;  
 Log.*v*(***LOG\_TAG***,**"TLM battery:"** + advTLM.getBatteryLevel());  
 Log.*v*(***LOG\_TAG***,**"TLM Temperature:"** + advTLM.getTemperature());  
 Log.*v*(***LOG\_TAG***,**"TLM adv count:"** + advTLM.getAdvCount());  
 **break**;  
 }  
 }  
 }  
  
 **mBeaconsDictory**.put(beacon.getMac(), beacon);  
 }  
  
 **if** (**mBeaconsDictory**.size() > 0) {  
 **mBeaconsArray** = **new** KBeacon[**mBeaconsDictory**.size()];  
 **mBeaconsDictory**.values().toArray(**mBeaconsArray**);  
 **mDevListAdapter**.notifyDataSetChanged();  
 }  
 }  
  
 **public void** onCentralBleStateChang(**int** nNewState)  
 {  
 **if** (nNewState == KBeaconsMgr.***BLEStatePowerOff***)  
 {  
 Log.*e*(***LOG\_TAG***, **"BLE function is power off"**);  
 }  
 **else if** (nNewState == KBeaconsMgr.***BLEStatePowerOn***)  
 {  
 Log.*e*(***LOG\_TAG***, **"BLE function is power on"**);  
 }  
 }  
  
 **public void** onScanFailed(**int** errorCode)  
 {  
 Log.*e*(***LOG\_TAG***, **"Start N scan failed："** + errorCode);  
 **if** (**mScanFailedContinueNum** >= ***MAX\_ERROR\_SCAN\_NUMBER***){  
 toastShow(**"scan encount error, error time:"** + **mScanFailedContinueNum**);  
 }  
 **mScanFailedContinueNum**++;  
 }  
};

4. Handle KSensor data from advertisment packet

*//get advertisement packet during scanning callback***public void** onBeaconDiscovered(KBeacon[] beacons)  
{

KBAdvPacketSensor advSensor = (KBAdvPacketSensor)device.getAdvPacketByType(KBAdvType.***KBAdvTypeSensor***);  
**if** (advSensor != **null**)  
{  
 KBAccSensorValue accPos = advSensor.getAccSensor();  
 **if** (accPos != **null**) {  
 strAccValue = String.*format*(Locale.***ENGLISH***, **"x:%d; y:%d; z:%d"**,  
 accPos.**xAis**, accPos.**yAis**, accPos.**zAis**);  
 }

}

…

}

5. Clean scanning result and stop scanning

After start scanning, The KBeaconMgr will buffer all found KBeacon device. If the app want to remove all buffered KBeacon device, the app can:

**mBeaconsMgr**.clearBeacons();

If the app want stop scanning:

**mBeaconsMgr.** stopScanning();

## Connect to device

If the app want to change the device configruation, then it need connect to the device.

mBeacon.connect(password, max\_timeout, connectionDelegate);

paramaters:

* ***Password***: device password, the default password is 0000000000000000
* max\_timeout: max connection timer, uint is ms.
* ConnStateDelegate**:** connection callback.

**private** KBeacon.ConnStateDelegate **connectionDelegate** = **new** KBeacon.ConnStateDelegate()  
{  
 **public void** onConnStateChange(KBeacon var1, **int** state, **int** nReason)  
 {  
 **if** (state == KBeacon.***KBStateConnected***)  
 {  
 Log.*v*(***LOG\_TAG***, **"device has connected"**);  
 **nDeviceLastState** = state;  
 }  
 **else if** (state == KBeacon.***KBStateConnecting***)  
 {  
 Log.*v*(***LOG\_TAG***, **"device start connecting"**);  
 **nDeviceLastState** = state;  
 }  
 **else if** (state == KBeacon.***KBStateDisconnecting***)  
 {  
 Log.*v*(***LOG\_TAG***, **"device start disconnecting"**);  
 **nDeviceLastState** = state;  
 }  
 **else if** (state == KBeacon.***KBStateDisconnected***)  
 {  
 **if** (nReason == KBConnectionEvent.***ConnAuthFail***) {  
 toastShow(**"password error"**);  
 } **else if** (nReason == KBConnectionEvent.***ConnTimeout***) {  
 toastShow(**"connection timeout"**);  
 } **else** {  
 toastShow(**"connection other error, reason:"** + nReason);  
 }  
  
 **nDeviceLastState** = state;  
 Log.*e*(***LOG\_TAG***, **"device has disconnected:"** + nReason);  
 }  
 }  
};

// disconnec from the device.

mBeacon.disconnect();

## Configure parameters

### Advertisment type

KBeacon devices support sending multiple beacon advertisment packet in parallel.

Example, Advertisment period was set to 500ms. Advertisment type was set to “iBeacon + URL + UID + KSensor”, then the advertisment packet is like follow

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time | 0 | 500 | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 |
| Adv type | KSensor | UID | iBeacon | URL | KSensor | UID | iBeacon | URL |

If the advertisment type include TLM, the TLM advertisment interval is fixed to 10. It means the TLM will advertisement every 10 packet.

Example:Advertisment period was set to 500ms. Advertisment type was set to “URL + TLM”, then the advertisment packet is like follow

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time | 0 | 500 | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 |
| Adv type | URL | URL | URL | URL | URL | URL | URL | URL | URL | TLM | URL |

### Get device parameters

After the app connect to KBeacon success. The KBeacon will get current paramaters from physical device. so the app can update UI and show the paramaters.

**private** KBeacon.ConnStateDelegate **connectionDelegate** = **new** KBeacon.ConnStateDelegate()  
{  
 **public void** onConnStateChange(KBeacon var1, **int** state, **int** nReason)  
 {  
 **if** (state == KBeacon.***KBStateConnected***)  
 {  
 Log.*v*(***LOG\_TAG***, **"device has connected"**);

updateDeviceToView();  
 }  
 }  
};

*//update device's configuration to UI***public void** updateDeviceToView()  
{  
 **boolean** isTLMEnable = **false**, isUIDEnable = **false**, isUrlEnable = **false**;  
 KBCfgCommon commonCfg = (KBCfgCommon) **mBeacon**.getConfigruationByType(KBCfgType.*KBConfigTypeCommon*);  
 **if** (commonCfg != **null**) {  
  
 *//print basic capibility* Log.*v*(***LOG\_TAG***, **"support iBeacon:"** + commonCfg.isSupportIBeacon());  
 Log.*v*(***LOG\_TAG***, **"support eddy url:"** + commonCfg.isSupportEddyURL());  
 Log.*v*(***LOG\_TAG***, **"support eddy tlm:"** + commonCfg.isSupportEddyTLM());  
 Log.*v*(***LOG\_TAG***, **"support eddy uid:"** + commonCfg.isSupportEddyUID());  
 Log.*v*(***LOG\_TAG***, **"support ksensor:"** + commonCfg.isSupportKBSensor());  
 Log.*v*(***LOG\_TAG***, **"beacon has button:"** + commonCfg.isSupportButton());  
 Log.*v*(***LOG\_TAG***, **"beacon can beep:"** + commonCfg.isSupportBeep());  
 Log.*v*(***LOG\_TAG***, **"support accleration sensor:"** + commonCfg.isSupportAccSensor());  
 Log.*v*(***LOG\_TAG***, **"support humidify sensor:"** + commonCfg.isSupportHumiditySensor());  
 Log.*v*(***LOG\_TAG***, **"support max tx power:"** + commonCfg.getMaxTxPower());  
 Log.*v*(***LOG\_TAG***, **"support min tx power:"** + commonCfg.getMinTxPower());  
  
 *//get support trigger* Log.*v*(***LOG\_TAG***, **"support trigger"** + commonCfg.getTrigCapibility());  
  
 *//device model* **mBeaconModel**.setText(commonCfg.getModel());  
  
 *//device version* **mBeaconVersion**.setText(commonCfg.getVersion());  
  
 *//current advertisment type* **mEditBeaconName**.setText(commonCfg.getAdvTypeString());  
  
 *//advertisment period* **mEditBeaconAdvPeriod**.setText(String.*valueOf*(commonCfg.getAdvPeriod()));  
  
 *//beacon tx power* **mEditBeaconTxPower**.setText(String.*valueOf*(commonCfg.getTxPower()));  
  
 *//beacon name* **mEditBeaconName**.setText(String.*valueOf*(commonCfg.getName()));  
  
 *//check if Eddy TLM advertisement enable* isTLMEnable = ((commonCfg.getAdvType() & KBAdvType.***KBAdvTypeEddyTLM***) > 0);  
 **mCheckboxTLM**.setChecked(isTLMEnable);  
  
 *//check if Eddy UID advertisement enable* isUIDEnable= ((commonCfg.getAdvType() & KBAdvType.***KBAdvTypeEddyUID***) > 0);  
 **mCheckboxUID**.setChecked(isUIDEnable);  
 **mUidLayout**.setVisibility(isUIDEnable? View.***VISIBLE***: View.***GONE***);  
  
 *//check if Eddy URL advertisement enable* isUrlEnable= ((commonCfg.getAdvType() & KBAdvType.***KBAdvTypeEddyURL***) > 0);  
 **mCheckBoxURL**.setChecked(isUrlEnable);  
 **mUrlLayout**.setVisibility(isUrlEnable? View.***VISIBLE***: View.***GONE***);  
  
 *//check if iBeacon advertisment enable* Log.*v*(***LOG\_TAG***, **"iBeacon advertisment enable:"** + ((commonCfg.getAdvType() & KBAdvType.***KBAdvTypeIBeacon***) > 0));  
  
 *//check if KSensor advertisment enable* Log.*v*(***LOG\_TAG***, **"iBeacon advertisment enable:"** + ((commonCfg.getAdvType() & KBAdvType.***KBAdvTypeSensor***) > 0));  
 }  
  
 *//get eddystone URL paramaters* KBCfgEddyURL beaconUrlCfg = (KBCfgEddyURL) **mBeacon**.getConfigruationByType(KBCfgType.*KBConfigTypeEddyURL*);  
 **if** (beaconUrlCfg != **null**) {  
 **mEditEddyURL**.setText(beaconUrlCfg.getUrl());  
 }  
  
 *//get eddystone UID information* KBCfgEddyUID beaconUIDCfg = (KBCfgEddyUID) **mBeacon**.getConfigruationByType(KBCfgType.*KBConfigTypeEddyUID*);  
 **if** (beaconUIDCfg != **null**) {  
 **mEditEddyNID**.setText(beaconUIDCfg.getNid());  
 **mEditEddySID**.setText(beaconUIDCfg.getSid());  
 }  
  
 //other  
}

### Update device parameters

After app connect to device success, the app can update update paramaters about KBeacon device.

Example1: update advertisment period, tx power, device name

**public void** simpleUpdateDeviceTest() {  
 **if** (!**mBeacon**.isConnected()) {  
 **return**;  
 }  
  
 *//get current paramaters* KBCfgCommon newCommomCfg = **new** KBCfgCommon();  
 **try** {  
 newCommomCfg.setAdvPeriod(1000);  
 newCommomCfg.setTxPower(-4);  
 newCommomCfg.setName(**"KBeaconDemo"**);  
 } **catch** (KBException excpt) {  
 toastShow(**"input data invalid"**);  
 excpt.printStackTrace();  
 }  
  
 ArrayList<KBCfgBase> cfgList = **new** ArrayList<>(1);  
 cfgList.add(newCommomCfg);  
 **mDownloadButton**.setEnabled(**false**);  
 **mBeacon**.modifyConfig(cfgList, **new** KBeacon.ActionCallback() {  
 @Override  
 **public void** onActionComplete(**boolean** bConfigSuccess, KBException error) {  
 **mDownloadButton**.setEnabled(**true**);  
 **if** (bConfigSuccess)  
 {  
 clearChangeTag();  
 toastShow(**"config data to beacon success"**);  
 }  
 **else** {  
 toastShow(**"config failed for error:"** + error.**errorCode**);  
 }  
 }  
 });  
}

Sometimes we need to configure multiple advertisment type parameters at the same time. We recommend that the app check whether the parameters change before upload. If the paramaters valis is no change, the app do not need to send the configuration.

Example2: check if the paramaters was change and update paramaters

*//read user input and download to KBeacon device***void** updateViewToDevice()  
{  
 **if** (!**mBeacon**.isConnected())  
 {  
 **return**;  
 }  
  
 KBCfgCommon oldCommonCfg = (KBCfgCommon)**mBeacon**.getConfigruationByType(KBCfgType.*KBConfigTypeCommon*);  
 KBCfgCommon newCommomCfg = **new** KBCfgCommon();  
 KBCfgEddyURL newUrlCfg = **new** KBCfgEddyURL();  
 KBCfgEddyUID newUidCfg = **new** KBCfgEddyUID();  
 **try** {  
 *//check if user update advertisement type* **int** nAdvType = 0;  
 **if** (**mCheckBoxURL**.isChecked()){  
 nAdvType |= KBAdvType.***KBAdvTypeEddyURL***;  
 }  
 **if** (**mCheckboxUID**.isChecked()){  
 nAdvType |= KBAdvType.***KBAdvTypeEddyUID***;  
 }  
 **if** (**mCheckboxTLM**.isChecked()){  
 nAdvType |= KBAdvType.***KBAdvTypeEddyTLM***;  
 }  
 *//check if the parameters changed* **if** (oldCommonCfg.getAdvType() != nAdvType)  
 {  
 newCommomCfg.setAdvType(nAdvType);  
 }  
  
 *//adv period, check if adv period changed* Integer changeTag = (Integer)**mEditBeaconAdvPeriod**.getTag();  
 **if** (changeTag > 0)  
 {  
 String strAdvPeriod = **mEditBeaconAdvPeriod**.getText().toString();  
 **if** (Utils.*isPositiveInteger*(strAdvPeriod)) {  
 Integer newAdvPeriod = Integer.*valueOf*(strAdvPeriod);  
 newCommomCfg.setAdvPeriod(newAdvPeriod);  
 }  
 }  
  
 *//tx power* changeTag = (Integer)**mEditBeaconTxPower**.getTag();  
 **if** (changeTag > 0)  
 {  
 String strTxPower = **mEditBeaconTxPower**.getText().toString();  
 Integer newTxPower = Integer.*valueOf*(strTxPower);  
 **if** (newTxPower > oldCommonCfg.getMaxTxPower() || newTxPower < oldCommonCfg.getMinTxPower()) {  
 toastShow(**"tx power not valid"**);  
 **return**;  
 }  
 newCommomCfg.setTxPower(newTxPower);  
 }  
  
 *//device name* String strDeviceName = **mEditBeaconName**.getText().toString();  
 **if** (!strDeviceName.equals(oldCommonCfg.getName()) && strDeviceName.length() < KBCfgCommon.***MAX\_NAME\_LENGTH***)

{  
 newCommomCfg.setName(strDeviceName);  
 }  
  
 *//uid config* **if** (**mCheckboxUID**.isChecked())  
 {  
 KBCfgEddyUID oldUidCfg = (KBCfgEddyUID)**mBeacon**.getConfigruationByType(KBCfgType.*KBConfigTypeEddyUID*);  
 String strNewNID = **mEditEddyNID**.getText().toString();  
 String strNewSID = **mEditEddySID**.getText().toString();  
 **if** (!strNewNID.equals(oldUidCfg.getNid()) && KBUtility.*isHexString*(strNewNID)){  
 newUidCfg.setNid(strNewNID);  
 }  
  
 **if** (!strNewSID.equals(oldUidCfg.getSid()) && KBUtility.*isHexString*(strNewSID)){  
 newUidCfg.setSid(strNewSID);  
 }  
 }  
  
 *//url config* **if** (**mCheckBoxURL**.isChecked())  
 {  
 KBCfgEddyURL oldUrlCfg = (KBCfgEddyURL)**mBeacon**.getConfigruationByType(KBCfgType.*KBConfigTypeEddyURL*);  
 String strUrl = **mEditEddyURL**.getText().toString();  
 **if** (!strUrl.equals(oldUrlCfg.getUrl())){  
 newUrlCfg.setUrl(strUrl);  
 }  
 }  
 }**catch** (KBException excpt)  
 {  
 toastShow(**"config data is invalid:"** + excpt.**errorCode**);  
 excpt.printStackTrace();  
 }  
  
 ArrayList<KBCfgBase> cfgList = **new** ArrayList<>(3);  
 cfgList.add(newCommomCfg);  
 cfgList.add(newUidCfg);  
 cfgList.add(newUrlCfg);  
 **mDownloadButton**.setEnabled(**false**);  
 **mBeacon**.modifyConfig(cfgList, **new** KBeacon.ActionCallback() {  
 @Override  
 **public void** onActionComplete(**boolean** bConfigSuccess, KBException error) {  
 **mDownloadButton**.setEnabled(**true**);  
 **if** (bConfigSuccess)  
 {  
 clearChangeTag();  
 toastShow(**"config data to beacon success"**);  
 }  
 **else** {  
 **if** (error.**errorCode** == KBException.***KBEvtCfgNoParamaters***)  
 {  
 toastShow(**"No data need to be config"**);  
 }  
 **else** {  
 toastShow(**"config failed for error:"** + error.**errorCode**);  
 }  
 }  
 }  
 });  
}

# Special instructions

1. AndroidManifest.xml of SDK has declared to access Bluetooth permissions.

2. After connecting to the device successfully, we suggest delay 1 second to sending configure data, otherwise the device may not return data normally.

3. If you app need running in background, We suggest that sending and receiving data should be executed in the "Service". There will be a certain delay when the device returns data, and you can broadcast data to the "Activity" after receiving in the "Service".