Developer Study Guide

An introduction to Bluetooth® Beacons – Android

Release: 1.3.2

Document Version : 1.3.2

Last Updated: 21st December 2020

Running time: 90 to 120 minutes

CONTENTS

Revision History

Version	Date	Author	Changes
1.0.0	16th December 2016		Initial version
1.1.0	1st August 2018	Kai Ren Bluetooth SIG	Raspberry Pi Tutorial: removed all the BDS references in the Raspberry Pi Tutorial. Raspberry Pi Tutorial: user doesn't need to download shell file on the host computer, user can just download the shell file by command on R Pi3 through SSH remote access or local access. Android Tutorial: add a "template" Android project for user, they can start copy and paste on it. User doesn't need to create a new Android Studio project. Android Tutorial: add a "fullSolution" Android project for user, user can build and install the App on their Android devices. This project is for user to debug if user has any problem for the implementation on "template" project. Android Tutorial: update Android project gradle, make the project can be compiled by latest Android Studio. Android Tutorial: double checked, the Android projects can be built on Windows and macOS host computer. iOS Tutorial: update some APIs which are deprecated. iOS Tutorial: optimize the work flow.
1.2.0	18th December 2018	Kai Ren Bluetooth SIG	Name changed to "Develoepr Study Guide" Fixed an Android Studio compilation warning.
1.3.0	11th October 2019	Kai Ren Bluetooth SIG	iOS retrofit the whole study guide to make it be compatible with Swift, instead of Objective-C. add new template Swift source code Raspberry Pi: Add latest Raspberry Pi4 support Update BlueZ installation from v5.49 to v5.50
1.3.1	28th October 2020	Martin Woolley Bluetooth SIG	Updated Android tutorial to incorporate permissions requests and be compatible with Android 10. Minimum Android version supported is now 6.0.
1.3.2	21st December	Martin Woolley	Language changes

	2020	Bluetooth SIG	

Overview

In this guide, we will show you how to create a simple Android application that will discover nearby Bluetooth® beacons. The application you build will show a list of nearby beacons with these details:

- manufacturer UUID (first 16 bytes of BEACON ID)
- additional data (arguments) from AltBeacon (last 4 bytes of BEACON ID)
- current RSSI and reference RSSI

Beacons and AltBeacons

Beacons are low-power Bluetooth devices that advertise a data packet containing some unique identifier and secondary bits of information. The AltBeacon specification is an open and interoperable specification for proximity beacons using Bluetooth technology. AltBeacons, by design, are not tied to any particular manufacturer or chipset.

Prerequisites

This tutorial has been tested under the following conditions:

- Android Studio v4.0. The Android developer site and documentation will walk you through setting up the Android Studio developer environment on your Windows PC or Mac OS X computer.
- Android SDK for API 23 or higher i.e. Android 6.0 with support for Bluetooth Low Energy (LE).
- Various smartphone makes and models.
- An AltBeacon. See the first tutorial in this set for instructions and a hardware list to get a Raspberry Pi3 set up and working as an AltBeacon.

Running Time

The two exercises in this lab should take 1-2 hours to complete, depending on your experience with Android development.

How this works

Follow the exercises, tasks and steps below to get to the finished product. If you want to skip ahead, or if you want to check your work, or if you get stuck, refer to the completed source project provided as part of the guide.

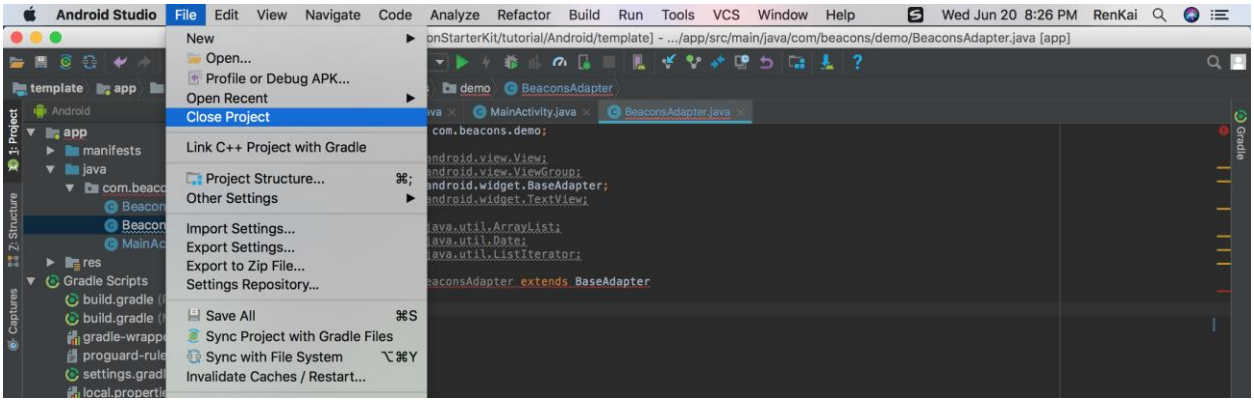
Please note: To complete this lab you will need to be comfortable with Android Studio.

Exercise 1 – Setting up your Android device

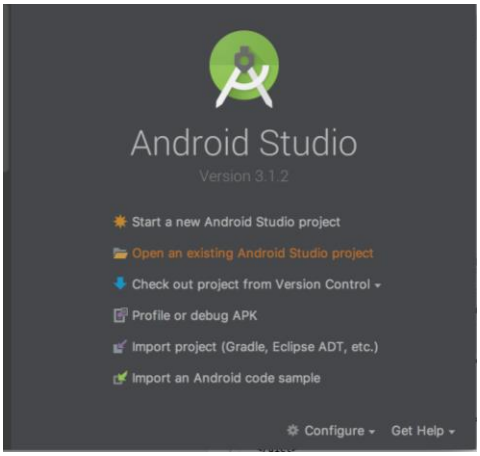
Task 1. open “template” project

Open Android Studio

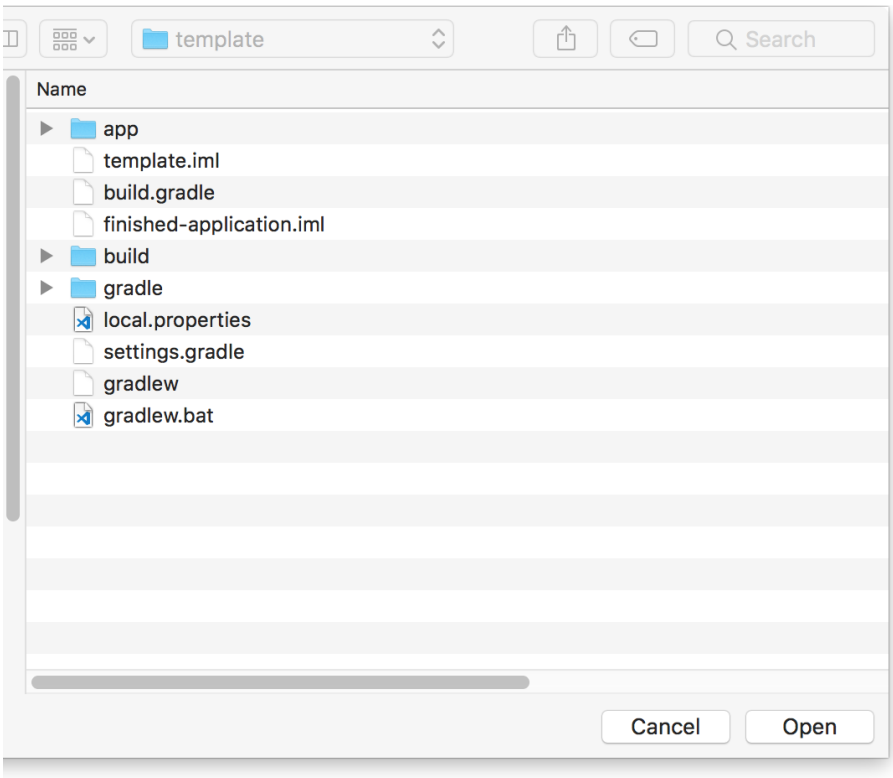
Open Android Studio on your Windows or macOS computer. If there is already a project opened by Android Studio, go to `File -> Close Project` to close current project.



Then, click “Open an existing Android Studio project” to import the template project.



Template project location: `./tutorial/Android/template/` like below picture shown and click “Open” button.



Task 2. Set up for Beacon data

Before doing work related to Beacons, we need to prepare the application for handling and displaying all the data we need.

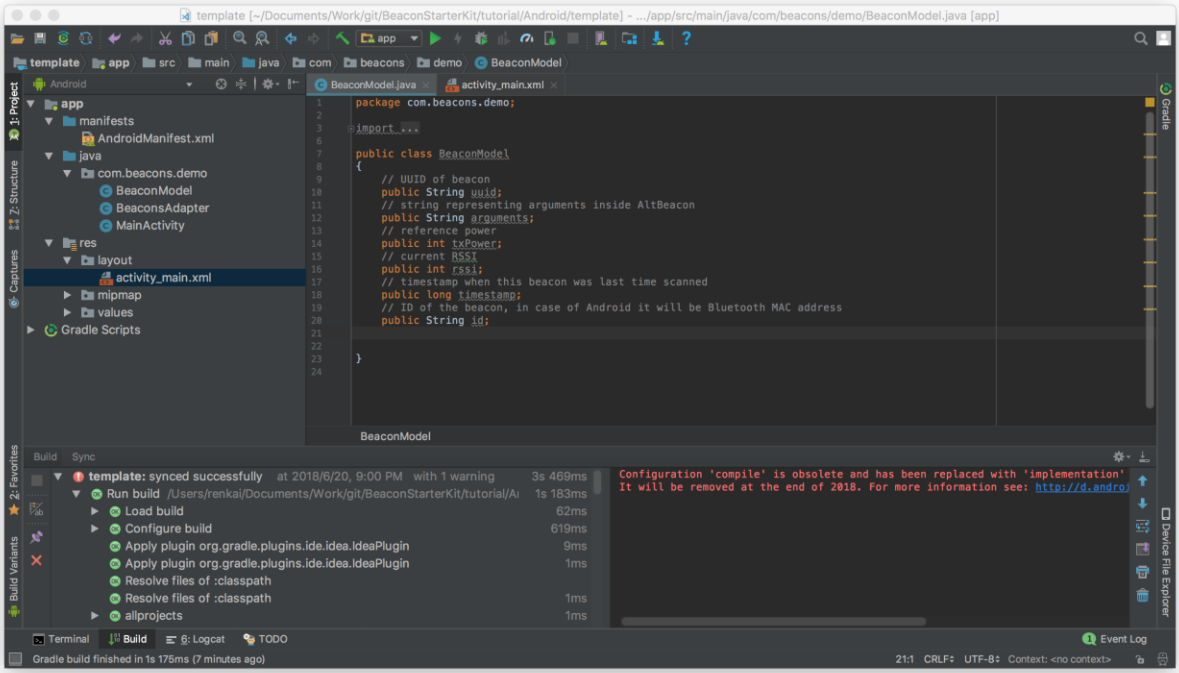
Prepare Beacon model

Open `BeaconModel.java` and add the following fields into BeaconModel class:

```
// UUID of beacon
public String uuid;
// string representing arguments inside AltBeacon
public String arguments;
// reference power
public int txPower;
// current RSSI
public int rssi;
// timestamp when this beacon was last time scanned
public long timestamp;
// ID of the beacon, in case of Android it will be Bluetooth MAC address
public String id;
```

You can add some additional helper methods to this model, but for now this is all that is required. The last two are metadata, and they will not be presented in the UI. id will be used to identify beacons and

timestamp will be used to find out if a beacon is still nearby (if timestamp is older than N seconds it means that beacon was not seen for N seconds and that could mean it is out of range).



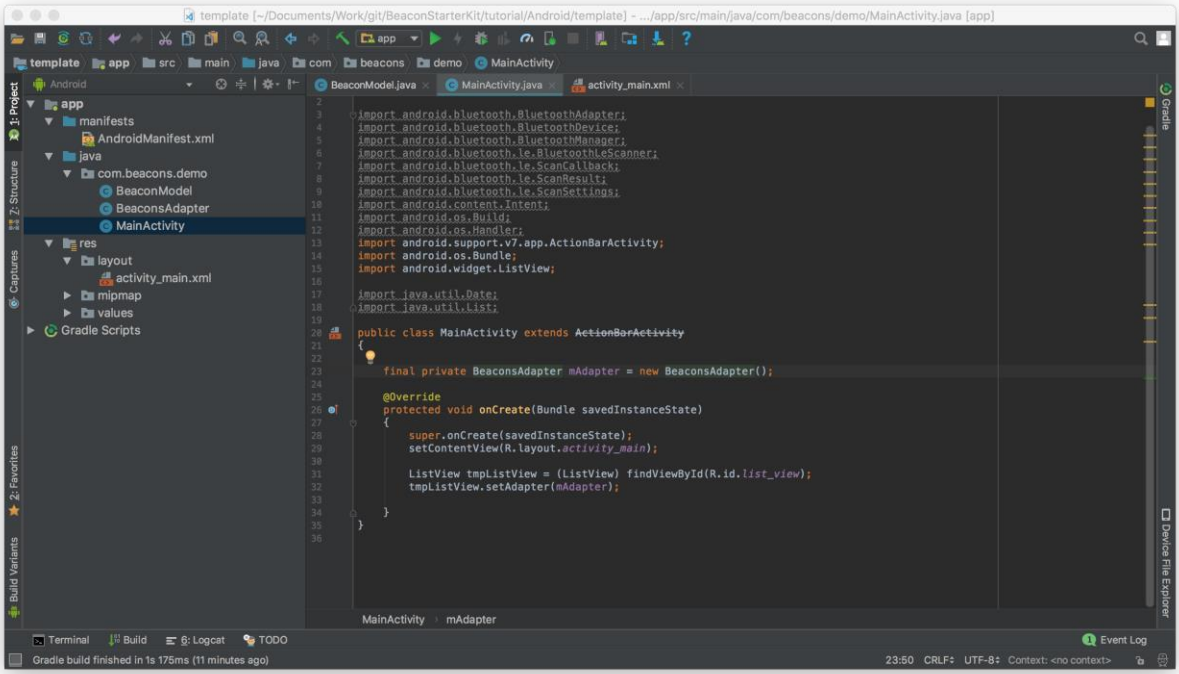
Add an adapter for our list view

Now modify `MainActivity.java`. Add a private reference to the instance of the adapter.

```
final private BeaconsAdapter mAdapter = new BeaconsAdapter();
```

And finally, extend the `onCreate` method as follows:

```
ListView tmpListView = (ListView) findViewById(R.id.list_view);
tmpListView.setAdapter(mAdapter);
```



Extend adapter so it can keep a list of BeaconModels and also return valid Views for our list

Here we will add an `ArrayList<BeaconModel>` field to our adapter and use it to implement the adapter's methods (leave getView empty for now) and then declare an `ArrayList` of `BeaconModel` objects inside `BeaconsAdapter` class:

```
private final ArrayList<BeaconModel> mBeacons = new ArrayList<>();
```

Make sure we are returning the size of the array and individual `BeaconModel` items from it, by completing two of the overrides in `BeaconsAdapter` like this:

```
    @Override
    public int getCount() {
        return mBeacons.size();
    }

    @Override
    public BeaconModel getItem(int position) {
        return mBeacons.get(position);
    }
```

And two additional methods which will be required later, to manipulate the content of the list:

```
    public void addNewBeacon(final BeaconModel beacon)
    {
        mBeacons.add(beacon);
        notifyDataSetChanged();
    }
```

```
    public BeaconModel findBeaconWithId(final String id)
    {
        for(final BeaconModel beacon : mBeacons) {
            if(beacon.id.equals(id)) return beacon;
        }
        return null;
    }
```

And finally, add skeleton implementations of the remaining BaseAdapter methods:

```
    @Override
    public long getItemId(int i) {
        return 0;
    }

    @Override
    public View getView(int i, View view, ViewGroup viewGroup) {
        return null;
    }
```

Now we will work on the cell's view itself. We will use the two_line_list_item layout defined by Android itself and will put all data into two `TextViews` provided by that layout. We will use the widely known recycled views pattern and view holder pattern here. So, let's add a new inner class to `BeaconsAdapter` to implement holder for our views into our private class inside `BeaconsAdapter`:

```
private class ViewHolder {
    public TextView text1;
    public TextView text2;

    public ViewHolder(final View target) {
        text1 = (TextView) target.findViewById(android.R.id.text1);
        text2 = (TextView) target.findViewById(android.R.id.text2);
    }

    public void updateAccordingToBeacon(final BeaconModel beacon) {
        text1.setText(beacon.uuid);
        String secondLine = String.format(
                "%s RSSI: %d TxPower: %d",
                beacon.arguments,
                beacon.rssi,
                beacon.txPower);
        text2.setText(secondLine);
    }
}
```

Our view holder class has some additional functionality for cleaner code. It can configure the UI according to the `BeaconModel`, which makes `getView` method shorter and focused on recycling views and setting up view holders if needed.

Now update the `getView` method to make use of our `ViewHolder` implementation:

```
@Override
public View getView(int position, View convertView, ViewGroup parent)
{
    if (convertView == null) {
        convertView = View.inflate(parent.getContext(),
                                   android.R.layout.two_line_list_item, null);
    }

    ViewHolder holder = (ViewHolder) convertView.getTag();
    if (holder == null) {
        holder = new ViewHolder(convertView);
        convertView.setTag(holder);
    }

    holder.updateAccordingToBeacon(getItem(position));

    return convertView;
}
```

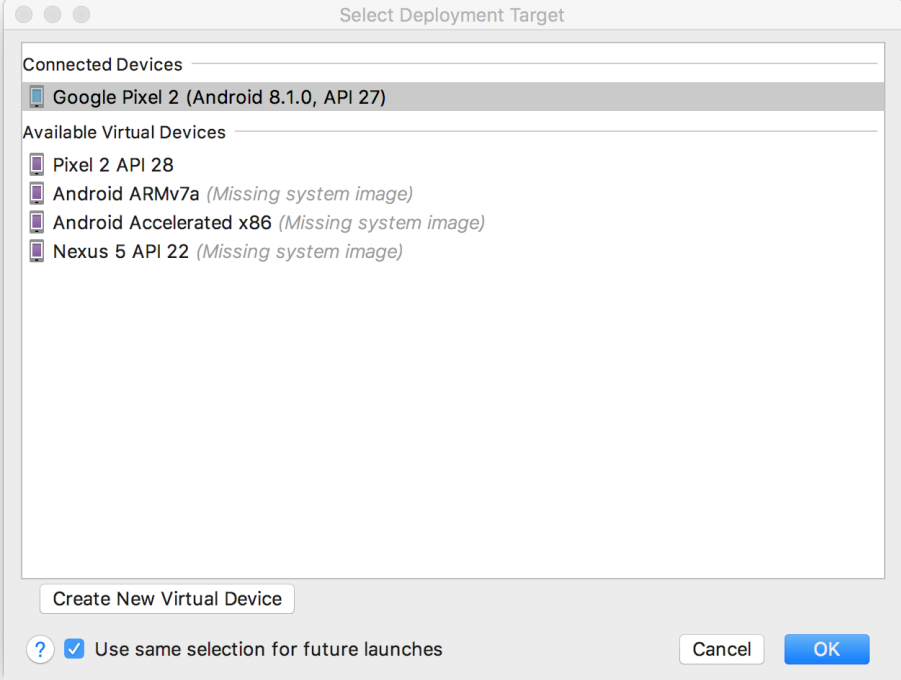
Test your application on a device

Inside the `MainActivity` class add a temporary custom implementation of the `onResume()` callback to add a dummy beacon to our list:

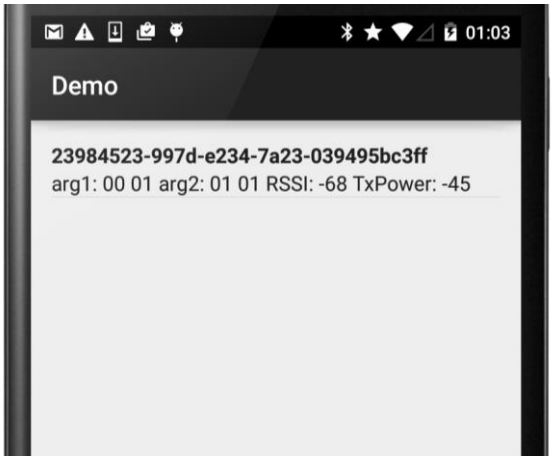
```
@Override
protected void onResume()
{
    super.onResume();

    BeaconModel test = new BeaconModel();
    test.rssi = -68;
    test.txPower = -45;
    test.uuid = "23984523-997d-e234-7a23-039495bc3ff";
    test.arguments = "arg1: 00 01 arg2: 01 01";
    mAdapter.addNewBeacon(test);
}
```

In Android Studio, select `Run -> Run 'app'` to start the application. Android Studio will ask you where you'd like to run the application. This has to be a real device supporting Bluetooth LE. Select the physical device connected to your development machine and mark "`Use same device ...`" to remember your choice.



Run the application and you should see something like the screen below:



Exercise Complete

Well done! You have completed Exercise 1.

Remove the test version of `onResume()` method, take a short break and after that we will start working with Bluetooth hardware and a real AltBeacon.

Exercise 2 – Working with AltBeacon

Task 1. Detect and enable Bluetooth® hardware

Decide how and where you are going to interact with Bluetooth Hardware.

There are many ways you can interact with Bluetooth technology on your device. In many real-world scenarios you would want to create a Service that runs in the background. In this lab, however, we will have only a single activity, and we want the application to scan only when it is opened. In order to make this tutorial easier to understand, we will interact with the Bluetooth APIs directly inside our Activity code. This is not best practice for an Android application which uses Bluetooth technology, and you should always plan carefully for how you want to use Bluetooth technology in a real-world app, considering power management and performance.

Detect if device has Bluetooth® enabled and request for it if needed

The first task is easy: we need to make sure that Bluetooth hardware is available and enabled on the device. For that we need to add one field to `MainActivity` class and a function which will check if Bluetooth technology is available and enabled.

First, add below code snippet inside the `MainActivity` class:

```
private BluetoothAdapter mBtAdapter = null;

private boolean isBluetoothAvailableAndEnabled() {
    BluetoothManager btManager = null;
    btManager = (BluetoothManager) getSystemService(BLUETOOTH_SERVICE);

    mBtAdapter = btManager.getAdapter();
    return mBtAdapter != null && mBtAdapter.isEnabled();
}
```

We also would like to have a method which will allow us to get user consent for Bluetooth technology to be used and ask the user to turn on the Bluetooth adapter if necessary. In order to do that we will start a new activity with intent to enable Bluetooth hardware (and this is why we added the BLUETOOTH_ADMIN permission to the manifest). Define an id for this request and add a method to make request itself:

```
final private static int BT_REQUEST_ID = 1;

private void requestForBluetooth() {
    Intent request = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(request, BT_REQUEST_ID);
}
```

We would also like to find out if the user has enabled the Bluetooth hardware, so we need to override the `onActivityResult()` method, check if this is the result for our request and check if Bluetooth was enabled[1]. Add this code to your activity:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == BT_REQUEST_ID) {
        if (isBluetoothAvailableAndEnabled()) {
            startScanning(); // we will declare that function shortly
        }
    }
    else {
        super.onActivityResult(requestCode, resultCode, data);
    }
}
```

If Bluetooth is now enabled, we just start scanning once again. This time it should pass check for Bluetooth and really start scanning for beacons.

Please note: at this point the compiler will complain about startScanning(), but don't worry – we will fix this in a moment.

Task 2. Start/stop scanning callback methods

Location Permissions

Android requires certain permissions to be granted when Bluetooth APIs are used. The required permissions are listed in the AndroidManifest.xml file supplied with the tutorial template project.

```
    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

The ACCESS_FINE_LOCATION permission is needed to allow Bluetooth scanning to take place. The Android APIs classify Bluetooth scanning as a location-related action because *active scanning* reveals the Bluetooth device address of the scanning device to other in-range Bluetooth devices. It should be noted, that Bluetooth also supports something called *passive scanning* which does not involve the transmission of any packets over the air, only listening for packets and in addition, most smartphones implement Bluetooth *private resolvable addresses* which hide the real Bluetooth address of the smartphone and instead use a periodically changing address which makes its packets look like they were transmitted by a series of different devices. Those other devices which have been paired with the smartphone can determine the true address of the transmitting device but those which have not, cannot do this and therefore the privacy of the user is protected.

[1] There are lots of reports, for different devices and Android versions, that checking `resultCode == RESULT_OK` does not always work properly. Hence, we will check directly to see if Bluetooth is now on.

Nevertheless, Android requires the ACCESS_FINE_LOCATION to be granted and this must be accomplished through an interactive request made to the user the first time scanning is to be invoked. The following code should be added to the MainActivity class. This is more or less standard example code from Android developer documentation.

Make the following declarations:

```
    // location permissions
    private static final int REQUEST_LOCATION = 0;
    private static String[] PERMISSIONS_LOCATION =
{Manifest.permission.ACCESS_FINE_LOCATION};
    private boolean permissions_granted=false;
    private Toast toast;
```

Then add the following methods:

```
    private void requestLocationPermission() {
        if
(shouldShowRequestPermissionRationale(Manifest.permission.ACCESS_FINE_LOCATION)){
            final AlertDialog.Builder builder = new AlertDialog.Builder(this);
            builder.setTitle("Permission Required");
            builder.setMessage("Please grant Location access so this application can
perform Bluetooth scanning");
            builder.setPositiveButton(android.R.string.ok, null);
            builder.setOnDismissListener(new DialogInterface.OnDismissListener() {
                public void onDismiss(DialogInterface dialog) {
                    requestPermissions(new
String[]{Manifest.permission.ACCESS_FINE_LOCATION}, REQUEST_LOCATION);
                }
            });
            builder.show();
        } else {
            requestPermissions(new
String[]{Manifest.permission.ACCESS_FINE_LOCATION}, REQUEST_LOCATION);
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[]
permissions, @NonNull int[] grantResults) {
        if (requestCode == REQUEST_LOCATION) {
            // Check if the only required permission has been granted
            if (grantResults.length == 1 && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
                // Location permission has been granted
                permissions_granted = true;
            }else{
            }
        } else {
            super.onRequestPermissionsResult(requestCode, permissions, grantResults);
```

```
        }
    }
```

Scanning control in onPause and onResume

Our application will be scanning from the moment it is being opened, until user close it. So we should start scanning in `onResume()` and stop it in `onPause()` lifecycle's callbacks inside activity code. Let's add two methods `startScanning()` and `stopScanning()` and put invocations of them in our overriding callbacks:

```
@Override
//this should replace our previous dummy onResume from Exercise 1
protected void onResume()
{
    super.onResume();
    if (permissions_granted) {
        startScanning();
    }
}

@Override
protected void onPause()
{
    if (permissions_granted) {
        stopScanning();
    }
    super.onPause();
}

private void startScanning() {

}

private void stopScanning() {

}
```

Scanning callbacks

We will implement the above two methods later (they are really simple and short). Firstly, we need to provide the most important part of our interaction with Bluetooth hardware. This is a scan callback, which is called every time our Android device discovers a nearby Bluetooth device. In addition, we must handle an API change in Android 5.0: you can declare variables of classes introduced in Android 5.0, but to call any function from this new API (even constructor) you need to put this code inside conditional check, like this:

```
if(Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
    // android 4.4.4 and earlier code

}
else {
    // android 5.0 and later code

}
```

Both APIs define slightly different callbacks for scanning, as well as different functions to start scanning. First let's define variables which will hold references to our callback. Depending on the Android version on the device only one of them will be used.

Add this code to the MainActivity class.

```
// callback for Android before Lollipop
private BluetoothAdapter.LeScanCallback mLeOldCallback = null;
// callback used on Lollipop and later
private ScanCallback mLeNewCallback = null;

private void initializeCallback() {
    if(Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
        // android 4.4.4 and earlier

    }
    else {
        // android 5.0 and later

    }
}
```

We also added a helper function to initialize the correct version of the callback. Firstly, we'll implement the callback for the pre-Lollipop API. Add the code below to your project inside the first (i.e. older) branch of the initializeCallback method:

```
// android 4.4.4 and earlier
mLeOldCallback = new BluetoothAdapter.LeScanCallback()
{
    @Override
    public void onLeScan(BluetoothDevice device, int rssi, byte[] scanRecord) {

    }
};
```

As you can see, there is only one function, which will be called from the background when a Bluetooth device is discovered. The Lollipop version will look quite similar. Put this in the second branch:

```
// android 5.0 and later
mLeNewCallback = new ScanCallback() {
    @Override
    public void onScanResult(int callbackType, ScanResult result) {
```

```
    }

    @Override
    public void onBatchScanResults(List<ScanResult> results) {

    }
};
```

Now we will complete the onBatchScanResults method which will return information about more than one device. For purposes of this lab we will just call the callback for every item returned in the batch:

```
@Override
public void onBatchScanResults(List<ScanResult> results) {
    for(final ScanResult result : results) {
        onScanResult(0, result);
    }
}
```

Add the following snippet to the end of the initializeCallback method to check and if necessary, request the ACCESS_FINE_LOCATION permission:

```
        if (ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
            permissions_granted = false;
            requestLocationPermission();
        } else {
            permissions_granted = true;
        }
```

Your current code connected for handling Bluetooth device discovery should look similar to this:

```
    private void initializeCallback() {
        if(Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
            // android 4.4.4 and earlier
            mLeOldCallback = new BluetoothAdapter.LeScanCallback()
            {
                @Override
                public void onLeScan(BluetoothDevice device, int rssi, byte[] scanRecord) {
                    handleNewBeaconDiscovered(device, rssi, scanRecord);
                }
            };

            permissions_granted = true;

        } else {
// android 5.0 and later
            mLeNewCallback = new ScanCallback() {
                @Override
                public void onScanResult(int callbackType, ScanResult result)
                {
                    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP)
                    {
                        handleNewBeaconDiscovered(
                                result.getDevice(),
                                result.getRssi(),
                                result.getScanRecord().getBytes());
                    }
                }

                @Override
                public void onBatchScanResults(List<ScanResult> results) {

                }
            };
            if (ContextCompat.checkSelfPermission( context: this, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
                permissions_granted = false;
                requestLocationPermission();
            } else {
                permissions_granted = true;
            }
        }
    }
```

Finally, we need to call our callback initialization method. Do this at the end of your `onCreate()` method, so it looks like this:

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    ListView tmpListView = (ListView) findViewById(R.id.list_view);
    tmpListView.setAdapter(mAdapter);

    initializeCallback();
}
```

Task 3. Start scanning for beacons

Start scanning on Android before Lollipop

In order to start scanning on the older API implement `startScanning()` as follows:

```
private void startScanning() {
    if (!permissions_granted) {
        return;
    }
    if (!isBluetoothAvailableAndEnabled()) {
        requestForBluetooth();
        return;
    }

    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
        mBtAdapter.startLeScan(mLeOldCallback);
    }
    else {
        // code for Lollipop and later
    }
}
```

As you can see the code is short and simple – and you can ignore any warnings about deprecation. We just check if Bluetooth is enabled or not we ask for it. Then we have a conditional check for Android version and we use the proper API accordingly. For the older API it is just a single line:

mBtAdapter.startLeScan(mLeOldCallback);

Start scanning: Lollipop or later

Scanning on Lollipop is a little bit more involved, but still easy, and it gives more freedom with scanning parameters (whereas there are no options to change scanning behavior on previous API). Put the code below into the Lollipop branch of the `startScanning()` method:

```
// code for Lollipop and later
BluetoothLeScanner scanner = mBtAdapter.getBluetoothLeScanner();
if (scanner != null) {
    ScanSettings settings = new ScanSettings.Builder()
            .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
            .setReportDelay(0)
            .build();

    scanner.startScan(null, settings, mLeNewCallback);
}
```

In the code above, we get the `BluetoothScanner` object from Bluetooth Adapter, and then we use that object to start scanning, using our own custom settings. We defined that we want to have latency as low as possible (in most cases you would more likely use a different mode to reduce battery usage) and ask to be told about every discovery as soon as possible (again, in a real-world application you would be more careful). We are not filtering devices, so the first argument to the `startScanning()` method is null.

Stopping discovery mode

To stop scanning you have to use branching based on the API level of current device, determined at runtime. This time both versions will be short and self explanatory:

```
private void stopScanning() {
    if (permissions_granted) {
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
            mBtAdapter.stopLeScan(mLeOldCallback);
        } else {
            BluetoothLeScanner scanner = mBtAdapter.getBluetoothLeScanner();
            if (scanner != null) {
                scanner.stopScan(mLeNewCallback);
            }
        }
    }
}
```

The startScanning and stopScanning methods should now look like this:

```
    private void startScanning() {
        if (!permissions_granted) {
            return;
        }
        if (!isBluetoothAvailableAndEnabled()) {
            requestForBluetooth();
            return;
        }

        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
            mBtAdapter.startLeScan(mLeOldCallback);
        } else {
// code for Lollipop and later
            BluetoothLeScanner scanner = mBtAdapter.getBluetoothLeScanner();
            if (scanner != null) {
                ScanSettings settings = new ScanSettings.Builder()
                        .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
                        .setReportDelay(0)
                        .build();

                scanner.startScan( filters: null, settings, mLeNewCallback);
            }
        }
    }

    private void stopScanning() {
        if (permissions_granted) {
            if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLIPOP) {
                mBtAdapter.stopLeScan(mLeOldCallback);
            } else {
                BluetoothLeScanner scanner = mBtAdapter.getBluetoothLeScanner();
                if (scanner != null) {
                    scanner.stopScan(mLeNewCallback);
                }
            }
        }
    }
```

Task 4. Handle data returned inside callbacks

Declare and use a function for parsing data from callbacks

You won't be able to run your application as is, because we need to implement a few functions. Even if you could, you wouldn't see anything in the UI. That is of course because we have only an empty

implementation of our callbacks. We need to process the data returned in the callbacks, convert them into our `BeaconModel` and push to our list view.

First, let's unify how both versions of APIs parse the data, as both have different sets of information returned in the callback, and we don't want to write two subtly different pieces of parsing code.

The method below will do real parsing/handling of the data and will be called by both callbacks. Add it to the `MainActivity`:

```
private void handleNewBeaconDiscovered(final BluetoothDevice device,
                                       final int rssi,
                                       final byte[] advertisement)
{

}
```

Now we need to make sure that both versions of the API are calling that method with the same parameters. For Android 4.4 and earlier we want to update the callback, like below, add `handleNewBeaconDiscovered()` into `initializeCallback()` .

```
// android 4.4.4 and earlier

mLeOldCallback = new BluetoothAdapter.LeScanCallback()
{
    @Override
    public void onLeScan(BluetoothDevice device, int rssi, byte[] scanRecord) {
        handleNewBeaconDiscovered(device, rssi, scanRecord);
    }
};
```

For Lollipop and later, we will need to get the same data from the ScanResult parameter, so update the `OnScanResult()` declaration inside the Lollipop branch of `initializeCallBack()`, just add it like this:

```
@Override
public void onScanResult(int callbackType, ScanResult result)
{
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP)
    {
        handleNewBeaconDiscovered(
                result.getDevice(),
                result.getRssi(),
                result.getScanRecord().getBytes());
    }
}
```

As you can see, we again put the code inside an API level check for the current device. This is because it is a callback, so we never know where it could come from and when it could be called. The system cannot be sure it won't be called on a pre-Lollipop device, so you need to check for it once again inside

the callback directly. Additionally, this will be reported back to our app in background thread, so that is another good reason to make that check again. This leads us to a very important part of our `handleNewBeaconDiscovered()` implementation: as soon as you would like to make any change in the UI, you need to make sure you have switched to the UI/main thread. To do that we will use the `runOnUiThread()` function, which will handle everything for us.

So here you need to update the `handleNewBeaconDiscovered()` function, as follows:

```
private void handleNewBeaconDiscovered(final BluetoothDevice device,
                                       final int rssi,
                                       final byte[] advertisement)
{
    /*do as much as possible here (it is running in background thread, so is not
blocking UI*/

        runOnUiThread(new Runnable() {
        @Override
        public void run() {
            // and here we can put code modifying UI without worrying about threads
        }
    });
}
```

Check if the peripheral is an AltBeacon

Now we need to decide if what we get from the Scanner is an AltBeacon device or not. We will put most of this code into the `BeaconModel` class, so switch the editor to `BeaconModel.java` and add these static fields:

```
private static final int PROTOCOL_OFFSET = 3;
private static final int AD_LENGTH_INDEX = 0 + PROTOCOL_OFFSET;
private static final int AD_TYPE_INDEX = 1 + PROTOCOL_OFFSET;
private static final int BEACON_CODE_INDEX = 4 + PROTOCOL_OFFSET;
private static final int UUID_START_INDEX = 6 + PROTOCOL_OFFSET;
private static final int UUID_STOP_INDEX  = UUID_START_INDEX + 15;
private static final int ARGS_START_INDEX = UUID_STOP_INDEX + 1;
private static final int TXPOWER_INDEX = ARGS_START_INDEX + 4;

private static final int AD_LENGTH_VALUE = 0x1b;
private static final int AD_TYPE_VALUE = 0xff;
private static final int BEACON_CODE_VALUE = 0xbeac;
```

These values follow the AltBeacon specification which you can find here.

We are particularly interested in the following fields:

- AD_LENGTH, which should be always 0x1B
- AD_TYPE, which should be always 0xFF
- BEACON_CODE, which should be always 0xBEAC

- BEACON_ID, which we split into three parts:
  - BEACON UUID – first 16 bytes
  - ARGUMENT1 – another two bytes
  - ARGUMENT2 – another two bytes
- REFERENCE_RSSI – reference TxPower (should represent signal strength at 1m distance from the peripheral)

Inside the advertisement data we receive in Android, we also get three additional bytes at the beginning, which are AD Flags. This is why our indexes are higher from those mentioned in AltBeacon Protocol Diagram; we want to accommodate these three additional bytes.

To determine if we are seeing an AltBeacon, we will test the AD_LENGTH, AD_TYPE and BEACON_CODE fields. If they do not match our criteria for an AltBeacon, we will stop processing that device, add below code snippet in `BeaconModel.java`:

```
public static boolean isAltBeacon(final byte[] data) {
    if ((data[AD_LENGTH_INDEX] & 0xff) != AD_LENGTH_VALUE) return false;

    if ((data[AD_TYPE_INDEX] & 0xff) != AD_TYPE_VALUE) return false;

    final int code = ((data[BEACON_CODE_INDEX] << 8) & 0x0000ff00) |
                     ((data[BEACON_CODE_INDEX + 1]) & 0x000000ff);
    if(code != BEACON_CODE_VALUE) return false;

    return true;
}
```

Now let's add that to our handling method (over in `MainActivity`) to filter out everything that is not an AltBeacon:

```
private void handleNewBeaconDiscovered(final BluetoothDevice device,
                                       final int rssi,
                                       final byte[] advertisement)
{
    if (!BeaconModel.isAltBeacon(advertisement)) return;

    // rest of implementation as in previous step
   ....
```

Parse data and update the UI (list view)

Now we would like to be able to get all the data needed in our `BeaconModel`. But first we need to find out if the device just reported wasn't previously reported (which could mean it is already inside our list of beacons). If the item is already in our list, we only need to update the current RSSI value and timestamp (as other values are constant). If the item is new, we need to create a new object of class `BeaconModel` and fill out all the fields.

Add the following to `handleNewBeaconDiscovered()` in `MainActivity`, replacing placeholder source as needed:

```
private void handleNewBeaconDiscovered(final BluetoothDevice device,
                                       final int rssi,
                                       final byte[] advertisement)
{
    if (!BeaconModel.isAltBeacon(advertisement)) return;

      // rest of implementation as in previous step
      final BeaconModel beaconToAdd;
      BeaconModel beacon = mAdapter.findBeaconWithId(device.getAddress());
      if (beacon == null) {
          // new item
          beacon = new BeaconModel();
          beacon.updateFrom(device, rssi, advertisement);
          beaconToAdd = beacon;
      }
      else {
          // we have this in the list.. just update and notify adapter about changes
          beaconToAdd = null;
          beacon.rssi = rssi;
          beacon.timestamp = new Date().getTime();
      }

      ...
}
```

And in the UI thread we want to update the UI:

```
runOnUiThread(new Runnable() {
    @Override
    public void run() {
        if (beaconToAdd != null) {
            mAdapter.addNewBeacon(beaconToAdd);
        }
        else {
            // just notify about changes in underlying data
            mAdapter.notifyDataSetChanged();
        }
    }
});
```

You will notice we are missing the `updateFrom()` method inside `BeaconModel.japa`, which will parse all data and set proper values for our model. Let's add that method to `BeaconModel`:

```
public void updateFrom(final BluetoothDevice device,
                       final int rssi,
                       final byte[] advertisement) {
    this.rssi = rssi;
    this.id = device.getAddress();
```

```
    this.timestamp = new Date().getTime();
    this.txPower = (int) advertisement[TXPOWER_INDEX];
    this.arguments = String.format("arg1: %02x %02x  arg2: %02x %02x",
            advertisement[ARGS_START_INDEX],
            advertisement[ARGS_START_INDEX + 1],
            advertisement[ARGS_START_INDEX + 2],
            advertisement[ARGS_START_INDEX + 3]);

    StringBuilder sb = new StringBuilder();
    for(int i = UUID_START_INDEX, offset = 0; i <= UUID_STOP_INDEX; ++i, ++offset) {
        sb.append(String.format("%02x", (int)(advertisement[i] & 0xff)));
        if (offset == 3 || offset == 5 || offset == 7 || offset == 9) {
            sb.append("-");
        }
    }
    this.uuid = sb.toString();
}
```

Test your application on the device

Now you can run your application and you should be able to see listed any nearby beacons complying with the AltBeacon specification.

Task 5. Periodically check for beacon timestamp

As you move your phone or tablet around you may leave the beacon's broadcast range. When this happens, your app will simply stop receiving broadcasts; Android will not tell your app you are out of range, nor will your app get any callback about the missing beacon. Hence, you need to take care of this yourself. That's why we added a timestamp to our model. Every time a particular beacon is reported, we set its timestamp to the current time, so we can check every beacon to see when it was last seen by our Bluetooth hardware. Based on this timestamp and the time elapsed we can remove invalid beacons from our list.

Let's start with a function which will go through all the beacons reported by the adapter and remove those which haven't been discovered for a specified amount of time (the timeout). Add the following function and static variable to your `BeaconsAdapter` class:

```
private static final long BEACON_LIFE_DURATION = 1000; // 6 seconds

public boolean validateAllBeacons() {
    boolean anythingChanged = false;

    final long oldestTimestampAllowed = new Date().getTime() - BEACON_LIFE_DURATION;
    ListIterator<BeaconModel> iterator = mBeacons.listIterator();
    while (iterator.hasNext()) {
        final BeaconModel beacon = iterator.next();
        if (beacon.timestamp < oldestTimestampAllowed) {
            iterator.remove();
            anythingChanged = true;
```

```
        }
    }

    return anythingChanged;
}
```

We would like to call this function periodically, every N seconds. There are many ways of doing this on Android, but we will use a Handler object and its `postDelayed()` method. To make it work, we need also a Runnable object, which will implement our validation of beacons.

Add the following snippet into your `MainActivity.java` code:

```
final private Handler mHandler = new Handler();

private Runnable periodicValidationTask = new Runnable() {
    @Override
    public void run() {
        if (mAdapter.validateAllBeacons()) {
            mAdapter.notifyDataSetChanged();
        }
        // add it again to queue:
        startValidating(); // this function will be declared and defined in next step
    }
};
```

p.s. Handler above refers to android.os.Handler.

Now we can add code to trigger this task every N seconds (7 seconds in our example):

```
final private static long VALIDATION_PERIOD = 7000; // 7 seconds

private void startValidating() {
    mHandler.postDelayed(periodicValidationTask, VALIDATION_PERIOD);
}

private void stopValidating() {
    mHandler.removeCallbacks(periodicValidationTask);
}
```

We need to add invokes of those function inside onResume() and onPause() activity's lifecycle callbacks.

Add this to onResume():

```
startValidating();
```

And this line to onPause():

```
stopValidating();
```

And your final code should like this:

```
protected void onResume()
{
    super.onResume();
    if (permissions_granted) {
        startScanning();
        startValidating();
    }
}

@Override
protected void onPause()
{
    if (permissions_granted) {
        stopScanning();
        stopValidating();
    }
    super.onPause();
}
```

Then, please build your project and install the apk on your target device, this apk will include all the features in this tutorial.

Exercise Complete

Well done for completing Exercise 2 and this guide.

In Bluetooth Beacon Starter Kit, we also provide full solution which contain intact source code and project setting, you can open this project by Android Studio, build and install the apk on your Android device directly, the location: `./tutorial/Android/fullSolution/`

You should now have an understanding of how to work with AltBeacons on Android. Good luck!