

WiFi Audio SDK for Android

Version 5.1.0

SDK Guide

Table of Contents

Native development settings	1
Android settings	1
Permissions requested	1
Optional configuration	3
Set up your own Firebase project	3
Initial setup of a WiFi Audio app in Android Studio	4
Create project	4
Import the WiFi Audio SDK	5
Exploring the sample application	7
Definition of the main SDK class: ExxtractorConnection	7
Some commonly used SDK classes and methods	15
Scanning for Wi-Fi Audio systems in the local area network	15
IP connection (Direct connection)	17
Get the channel list	17
Notification builder registration	19
Set default loudspeaker value	19
Configurable Loudspeaker	19
Start playback of channel	22
Stop playback of channel	22
Loading of channel information	23

Native development settings

Android NDK version	21.0.6113669 (r21)
Supported ABIs	<ul style="list-style-type: none">• armeabi-v7a• arm64-v8a• x86• x86_64

Android settings

Min SDK version	26
Target SDK version	34
Compile SDK version	34
Android X	1.0.0

Permissions requested

- android.permission.INTERNET
- android.permission.WAKE_LOCK
- android.permission.ACCESS_NETWORK_STATE
- android.permission.ACCESS_WIFI_STATE
- android.permission.CHANGE_WIFI_MULTICAST_STATE
- android.permission.MODIFY_AUDIO_SETTINGS
- android.permission.FOREGROUND_SERVICE
- android.permission.READ_PHONE_STATE
- android.permission.WRITE_EXTERNAL_STORAGE
- android.permission.FOREGROUND_SERVICE_MEDIA_PLAYBACK
- android.permission.FOREGROUND_SERVICE_LOCATION
- android.permission.CAMERA
- android.permission.ACCESS_FINE_LOCATION
- android.permission.ACCESS_BACKGROUND_LOCATION
- android.permission.BLUETOOTH
- android.permission.BLUETOOTH_SCAN
- android.permission.BLUETOOTH_CONNECT
- android.permission.BLUETOOTH_ADMIN
- android.permission.BLUETOOTH_ADVERTISE
- android.permission.BLUETOOTH_CONNECT
- android.permission.POST_NOTIFICATIONS

Optional configuration

Currently, the main application ListenWiFi scans and opens dynamic links generated for the Listen Technologies environment. For the usage of another company, a new Firebase project to support and develop the dynamic links functionality should exist and link it to the example app.

Important note: The Dynamic Link functionality is now deprecated but if you have a previous configuration set, it can be used until August 2025. [More information](#)

Set up your own Firebase project

- Add in the StreamCatcher/build.gradle file the credentials needed that Firebase provides:

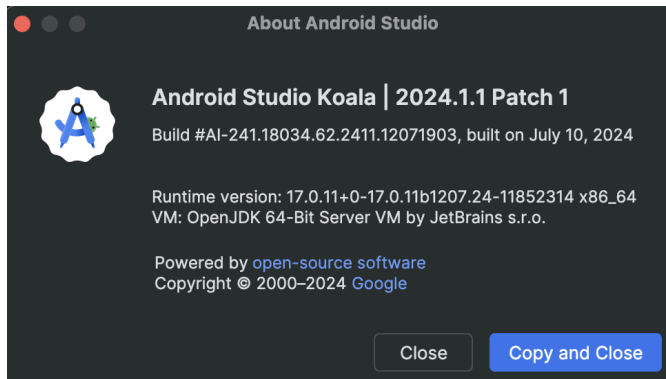
```
signingConfigs {  
    ExampleReleaseConfig {  
        keyAlias 'Example Google Key'  
        keyPassword 'example'  
        storeFile file('Example-Key.keystore')  
        storePassword 'example'  
    }  
}
```

- Add the Example-Key.keystore in the project.
- Replace the google-services.json file that Firebase provides in the folder StreamCatcher/

Initial setup of a WiFi Audio app in Android Studio

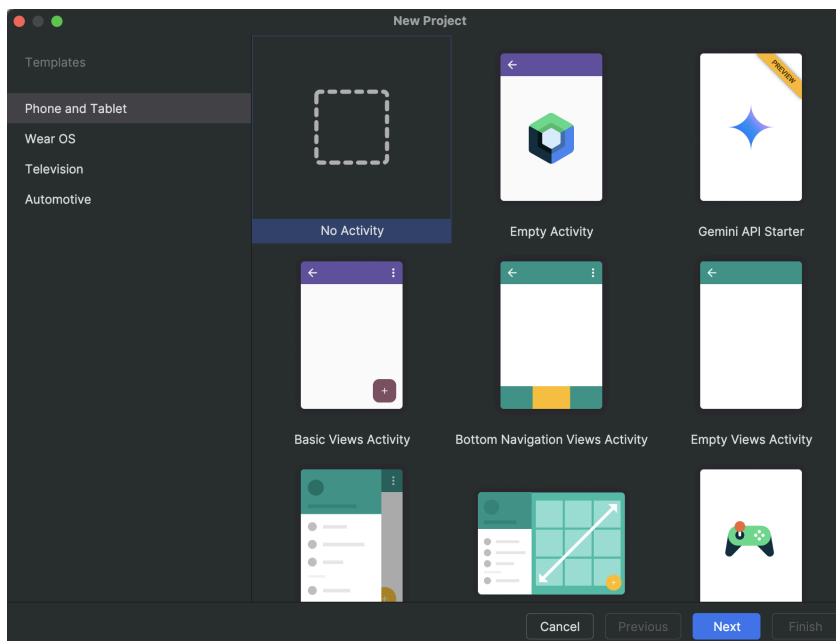
Create project

In this section, we will create a new project and set it up so it works with WiFi Audio SDK for Android, for this we are using the following Android Studio version:

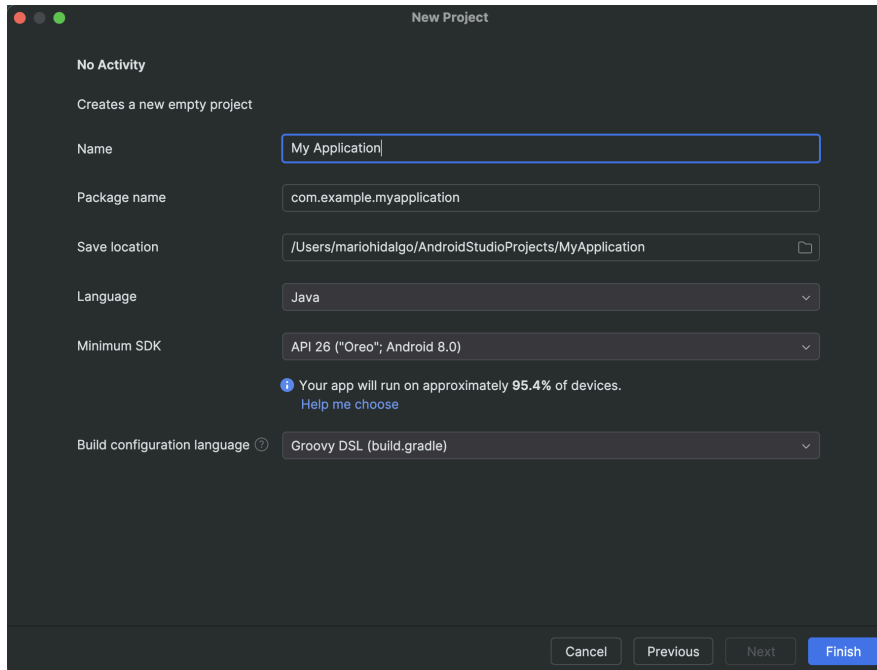


Follow the steps below:

1. Open Android Studio.
2. Go to *File | New | New Project*; select an activity type for the application. For this guide, we will use a No Activity. Click **Next**.



3. Enter the application and package name, add the location for the application files, and select the minimum SDK API 26 (the minimum supported by WiFi Audio SDK) or higher. Click **Next**.



The screenshot shows the 'New Project' dialog in Android Studio. The dialog has a dark theme and contains the following fields and options:

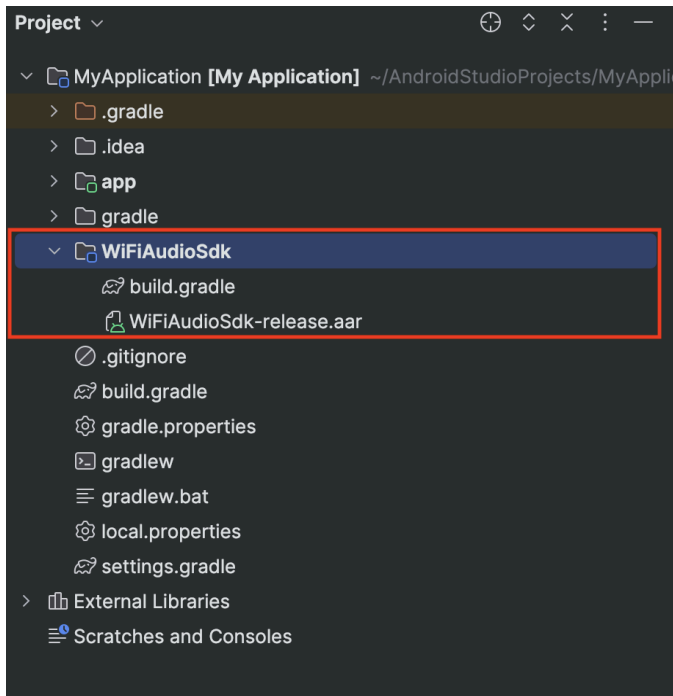
- Name:** My Application
- Package name:** com.example.myapplication
- Save location:** /Users/mariohidalgo/AndroidStudioProjects/MyApplication
- Language:** Java
- Minimum SDK:** API 26 ("Oreo"; Android 8.0)
- Build configuration language:** Groovy DSL (build.gradle)

Below the 'Minimum SDK' field, there is a blue information icon and the text: "Your app will run on approximately 95.4% of devices." with a link "Help me choose".

At the bottom right, there are four buttons: "Cancel", "Previous", "Next", and "Finish". The "Finish" button is highlighted in blue.

Import the WiFi Audio SDK

1. Create a new directory in the project and add the WiFiAudioSdk-release.aar file and a build.gradle file like the following structure:



2. Add the SDK information in the new WiFiAudioSdk/build.gradle file.

```
configurations.maybeCreate("default")
artifacts.add("default", file('WiFiAudioSdk-release.aar'))
```

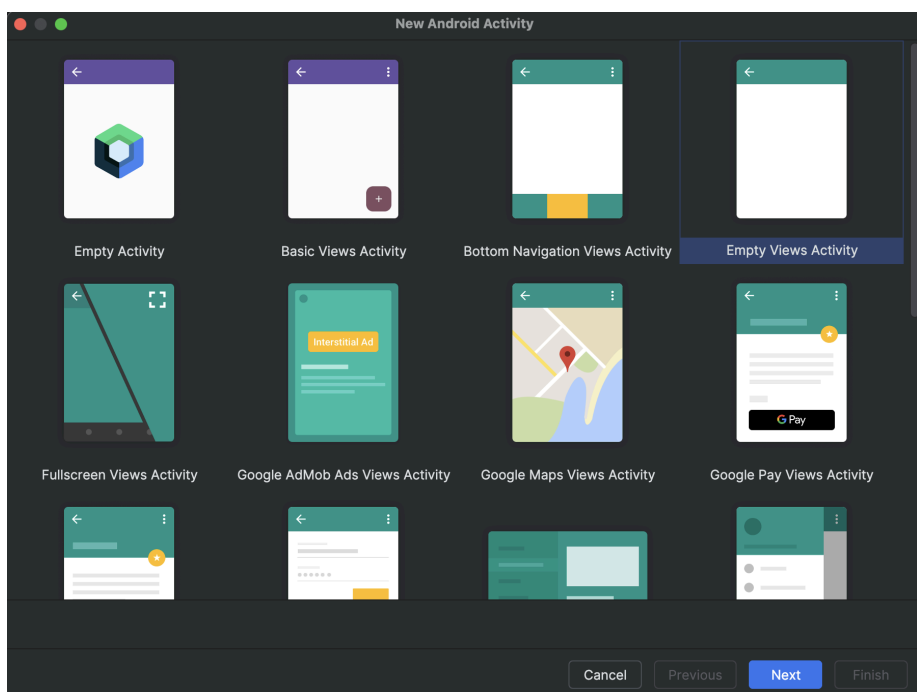
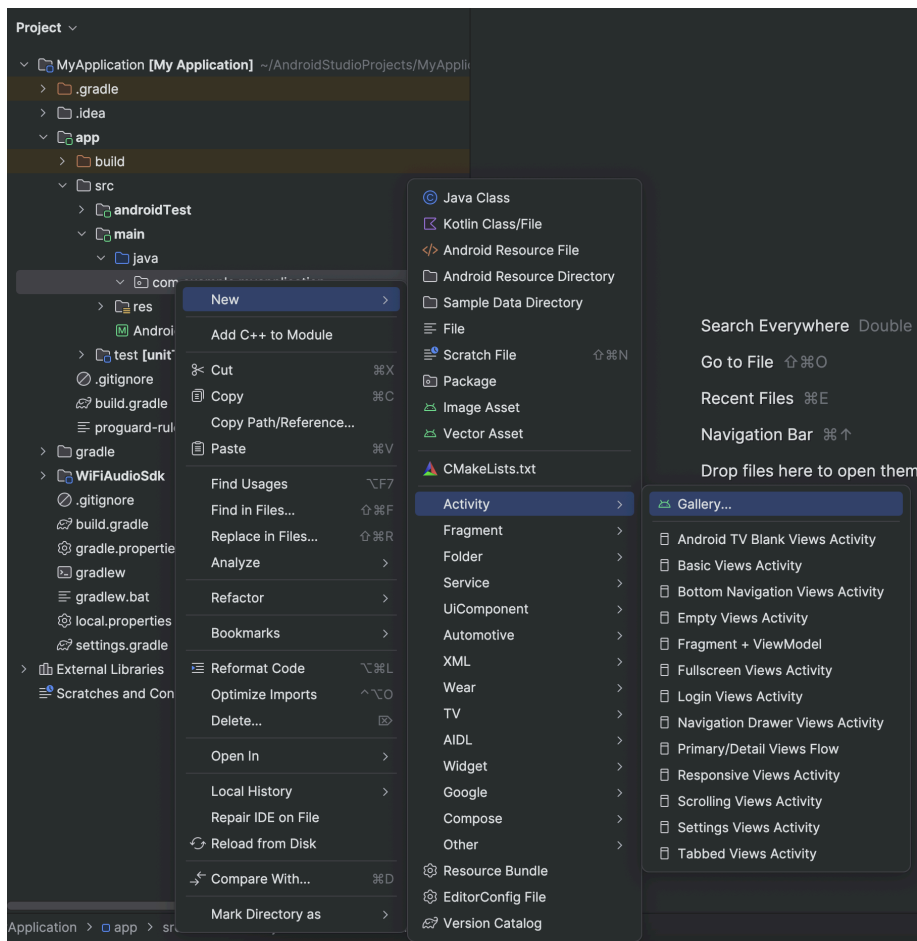
3. In the settings.gradle file include the SDK.

```
include ':app', ':WiFiAudioSdk'
```

4. Add the WiFi Audio SDK as a dependency in your app's *build.gradle* file and some dependencies like dagger for injection.

```
dependencies {
    implementation project(':WiFiAudioSdk')
    implementation libs.appcompat
    implementation libs.material
    implementation 'androidx.multidex:multidex:2.0.0'
    api 'com.google.dagger:dagger:2.29.1'
    annotationProcessor 'com.google.dagger:dagger-compiler:2.29.1'
    implementation 'org.altbeacon:android-beacon-library:2+'
}
```


5. Add a new activity to the project and then you can access the WiFiAudioSdk.



New Android Activity

Empty Views Activity
Creates a new empty activity

Activity Name
MainActivity

☒ Generate a Layout File

Layout Name
activity_main

☐ Launcher Activity

Package name
com.example.myapplication

Source Language
Java

Cancel Previous Next Finish

Exploring the sample application

Definition of the main SDK class: ExxtractorConnection

ExxtractorConnection is the entry point for the functionality provided by the WiFi Audio SDK. One way to access it is through dependency injection, as you may be using this all over the place. The following steps explain how this is done in the sample application provided in the WiFi Audio SDK package.

1. Create a new empty interface class with the following code to be used for the dagger

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import javax.inject.Qualifier;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface ApplicationContext {}
```

2. Create a new class and a new interface that will perform the injection of the SDK classes into the classes that need it when required as a singleton instance. You can review more about Dagger and the injection [here](#).

```
import android.content.Context;
import com.exxothermic.audioeverywheresdk.ExxtractorConnection;
import javax.inject.Singleton;
import dagger.Module;
import dagger.Provides;

@Module
public class AudioEverywhereModule {

    public AudioEverywhereModule() {}

    @Provides
    @ApplicationContext
    @Singleton
    Context providesContext() {
        return AudioEverywhereApplication.getContext();
    }

    @Provides
    @Singleton
```

```

    ExxtractorConnection providesAudioEverywhere(@ApplicationContext Context
context) {
        return new ExxtractorConnection(context, false);
    }

    @Provides
    @Singleton
    ExxtractorConnectionFacade providesConnectionFacade(@ApplicationContext
Context context) {
        return new ExxtractorConnectionFacade(context);
    }
}

```

```

import javax.inject.Singleton;
import dagger.Component;

@Singleton
@Component(modules = {AudioEverywhereModule.class})
public interface AudioEverywhereComponent {
    void inject(MainActivity mainActivity);
    void inject(ExxtractorConnectionFacade exxtractorConnectionFacade);
}

```

3. Create a new class that extends from *android.app.Application*, which will be used to retrieve the context and the object graph that the SDK will use for multiple references

```

import android.app.Application;
import android.content.Context;
import androidx.multidex.MultiDex;

public class AudioEverywhereApplication extends Application {

    private static AudioEverywhereApplication mContext;
    private static AudioEverywhereComponent mComponent;
    @Override
    public void onCreate() {
        super.onCreate();
        mContext = this;
        mComponent = buildComponent();
    }

    @Override
    protected void attachBaseContext(Context base) {
        super.attachBaseContext(base);
    }
}

```

```

        MultiDex.install(this);
    }

    protected AudioEverywhereComponent buildComponent() {
        return DaggerAudioEverywhereComponent.builder().audioEverywhereModule(
            new AudioEverywhereModule()).build();
    }

    public static Context getContext() {
        return mContext.getBaseContext();
    }

    public static AudioEverywhereComponent getComponent() {
        return mComponent;
    }
}

```

4. Set the created application class as the name for the application definition in the *AndroidManifest.xml* file including the package reference of the class, enable the required permissions, and other services (check the *AndroidManifest.xml* in the sample app for more details).

```

<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.CHANGE_WIFI_MULTICAST_STATE"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
android:maxSdkVersion="18"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE_MEDIA_PLAYBACK"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE_LOCATION"/>
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION"/>
<uses-permission android:name="android.permission.BLUETOOTH_SCAN"/>
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT"/>
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE" />
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
<uses-permission android:name="android.permission.POST_NOTIFICATIONS" />

<application
    android:name="org.exxothermic.streamcatcher.ui.AudioEverywhereApplication"
    android:allowBackup="true"

```

```

        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme"
        android:largeHeap="true"
        android:usesCleartextTraffic="true">
        <activity
            android:name=".MainActivity" android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        ...
    </application>

```

5. Any place you need a reference to the singleton *ExXtractorConnection* object you will need to add the following code like in this *ExxtractorConnectionFacade* file:

```

import android.content.Context;
import com.exxothermic.audioeverywheresdk.ExxConnection;
import com.exxothermic.audioeverywheresdk.ExxtractorConnection;
// Import of the Inject annotation to be used for the injection of the
// singleton object
import javax.inject.Inject;

public class ExxtractorConnectionFacade {
    // Inject the object using dagger
    @Inject
    ExxtractorConnection mRealExxtractorConnection;

    private ExxConnection mExxtractorConnection;
    private Context mContext;

    public ExxtractorConnectionFacade(Context context) {
        // OnCreate of the item where you are including the singleton you need
        // to inject all
        // the objects that are using the reference
        AudioEverywhereApplication.getComponent().inject(this);
        mExxtractorConnection = mRealExxtractorConnection;
        mContext = context;
    }
}

```

6. In your main activity you can access the ExxtractorConnectionFacade and for example call the connectServer method with the handle response like follows:

```
import com.exxothermic.audioeverywheresdk.AudioEverywhereException;
import com.exxothermic.audioeverywheresdk.AudioEverywhereResponseHandler;
import javax.inject.Inject;

public class MainActivity extends AppCompatActivity {

    @Inject
    ExxtractorConnectionFacade mConnectionFacade;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        AudioEverywhereApplication.getComponent().inject(this);
        super.onCreate(savedInstanceState);
        EdgeToEdge.enable(this);
        setContentView(R.layout.activity_main);

        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main), (v, insets) -> {
            Insets systemBars =
insets.getInsets(WindowInsetsCompat.Type.systemBars());
            v.setPadding(systemBars.left, systemBars.top,
systemBars.right, systemBars.bottom);
            return insets;
        });
        String ipAddress = "192.168.0.6";

        mConnectionFacade.mRealExxtractorConnection.connectToServer(ipAddress, 5,
new AudioEverywhereResponseHandler() {
            @Override
            public void onSuccess() {
            }

            @Override
            public void onSuccess(Object o) {
            }

            @Override
            public void onFailure(AudioEverywhereException e) {
            }
        });
    }
}
```

7. For your main activity, it is required to include the permission to read the phone state and then initialize the Telephony Manager if permission is already granted by introducing the following code:

```
//request READ_PHONE_STATE permission for ExxChannelManager in the sdk
if (ContextCompat.checkSelfPermission(this,
Manifest.permission.READ_PHONE_STATE) != PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(
        this,
        new String[]{Manifest.permission.READ_PHONE_STATE},
        READ_PHONE_STATE
    );
} else
{
    mConnectionFactory.initializeTelephonyManager(this.getBaseContext());
}
```

8. Also is required the Bluetooth permission for beacon scanning, so you can request it by introducing the next method in code:

```
// Check Location permissions for monitoring beacons
private void requestBluetoothPermissions() {
    try {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
            if (ContextCompat.checkSelfPermission(this.getActivity(),
Manifest.permission.BLUETOOTH_SCAN)
                != PackageManager.PERMISSION_GRANTED) {
                ActivityCompat.requestPermissions(
                    this.getActivity(),
                    new String[]{Manifest.permission.BLUETOOTH_SCAN,
Manifest.permission.BLUETOOTH_CONNECT, Manifest.permission.BLUETOOTH_ADMIN,
Manifest.permission.BLUETOOTH_ADVERTISE
                        , Manifest.permission.BLUETOOTH_PRIVILEGED,
Manifest.permission.BLUETOOTH},
                    PERMISSION_REQUEST_BLUETOOTH_SCAN
                );
            }
        }
    } catch (Exception exception) {
        Log.i(LOG_TAG, "Unable to check bluetooth permissions");
    }
}
```

9. Override the request permission result callback to continue asking for any other missing permissions or initialize the Telephony Manager if the user accepts the

READ_PHONE_STATE permission access, for more detail you can check the method [onRequestPermissionsResult](#) in the MainActivity.java in the example app.

```
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions,
int[] grantResults)
{
    switch (requestCode)
    {
        case READ_PHONE_STATE:
        case PERMISSION_REQUEST_BLUETOOTH_SCAN:
        case PERMISSION_REQUEST_FINE_LOCATION:
        case PERMISSION_REQUEST_BACKGROUND_LOCATION:
        default:
            return;
    }
}
```

Some commonly used SDK classes and methods

Scanning for Wi-Fi Audio systems in the local area network

The process that will scan for a Wi-Fi Audio system on the network is triggered by the method `scanNetworkForExxtractors` of the *ExxtractorConnection* object (all the methods to use the *ExxtractorConnection* in the example app are located in *ExxtractorConnectionFacade.java*). The steps are as follows:

1. Having the *ExxtractorConnection* reference, start the scanning, by calling the *scanNetworkForExxtractor* method. You need to pass the *timeout* seconds and an *AudioEverywhereResponseHandler* object with the implementation required to handle the result of the process. The following is an example of how to do it:

```
mExxtractorConnection.scanNetworkForExxtractors(SCANNING_TIMEOUT_IN_SECONDS,
new AudioEverywhereResponseHandler() {
    @Override
    public void onSuccess() {
        if (mExxtractorConnection.isLocationEnabled()) {
            mExxtractorConnection.updateExctrLocation(manager.getDeviceType(),
                new AudioEverywhereResponseHandler<Location>() {
                    @Override
                    public void onSuccess() {}

                    @Overridepublic void onSuccess(Location location) {
                        //here you put the code that you want to run when the
connection is completed; update ui, start the retrieve of the list of channels or
others
                    }
                }
            );
        }
    }
});
```

```

    }
    @Override
    public void onFailure(AudioEverywhereException exception) {
        // This can be called due multiple causes.
        // Check the error type and act accordingly
        switch (exception.getReason()) {
            case INVALID_LOCATION:
                // The location field is not present, but the
feature is enabled,so we throw an error
                Messages.displayInfoDialog("Connection error",
                    "Invalid Location",);
                break;
            case INVALID_PARTNER:
                // for an invalid partner, we may not receive the
location
                // but the SDK stores it so we can extract
information from it
                Messages.displayInfoDialog("Connection error",

```

IP connection (Direct connection)

In some cases the connection to the server is made directly through the IP address (Not scanning the network), in these cases it is required to retrieve the server's information right after a successful connection.

```

AudioEverywhereResponseHandler connectionResponseHandler = new
AudioEverywhereResponseHandler() {
    @Override
    public void onSuccess() {
        mRealExxtractorConnection.updateExctrLocation(2, new
AudioEverywhereResponseHandler<Location>() {
        @Override
        public void onSuccess() {
        }
        @Override
        public void onSuccess(Location location) {
            mRealExxtractorConnection.startSyncingChannelWithExxtractor(2);
        }
        @Override
        public void onFailure(AudioEverywhereException exception) {
        }
    });
}
@Override
public void onSuccess(Object list) {
}
@Override
public void onFailure(AudioEverywhereException exception) {

```

```

    }
};
mRealExxtractorConnection.connectToService(serverHostname,
                                           serverIpAddress, "Unknown",
connectionResponseHandler, 30);

```

Get the channel list

This is a 2 steps process:

1. Start the asynchronous process.

```

mExxtractorConnection.startSyncingChannelWithExxtractor(getDeviceType());
// getDeviceType is a helper method that returns an int with 2 for phone and 3
for tablet
mExxtractorConnection.getChannelList();
// get the channel list

```

2. Registering a broadcast receiver to listen for the messages sent from the SDK and act accordingly:

```

BroadcastReceiver mManagerListener = new BroadcastReceiver() {
    @Override
    public void onReceive (Context context, Intent intent) {
        final String action = intent.getAction();
        switch (action) {
            case AudioEverywhereMessages.LIST_UPDATED:
                // this is the action triggered when the list of channels is
updated from
                // the SDK. For example we can change the notification message
for the
                // currently selected channel
                if (mExxtractorConnection.getCurrentAudioChannel() != null) {
mExxtractorConnection.updateNotificationAndRemoteControlText (
                    String.format(NOTIFICATION_FORMAT,
mExxtractorConnection.getCurrentAudioChannel().getTitle()));
                }
                break;
            case AudioEverywhereMessages.PLAYBACK_STATE_CHANGE:
                // The playback for the current channel has changed. Update the
UI accordingly
                break;
            case AudioEverywhereMessages.SERVICE_STATE_CHANGED:

```

```

        // The service may be disconnected, or some issue may happen to
the current
        // connection, filter the reason for it
        StateChangeInformation information = (StateChangeInformation)
intent.getSerializableExtra(AudioEverywhereMessages.CHANGE_INFORMATION);
        // implemented method to handle the action depending of the type
        handleStateChange(information);
        break;
    default:
        // we received an unknown message. Process should not continue
        break;
    }
}
};
mExxtractorConnection.registerBroadcastReceiver(mManagerListener);
...

private void handleStateChange(StateChangeInformation information) {
    switch (information.getReason()) {
        case UnreachableLan:
        case ServiceNotAvailable:
        case UnknowError:
            // More likely, the service is not available
            // At this point you can enable the user to perform another scan of
the

```

Notification builder registration

This is required if you want to show the message of the current channel reproduction at the notification bar; a NotificationBuilder object must be set to the ExxtractorConnection object.

```

final NotificationBuilder notificationBuilder = new NotificationBuilder(
    getString(R.string.app_name),
    R.drawable.ic_notify,
    MainActivity.class);
mExxtractorConnection.setNotificationBuilder(notificationBuilder);

```

Set default loudspeaker value

Using the shared preferences, set up the "ShouldUseLoudspeaker" to TRUE if audio should be played from the loudspeaker. Or set it to FALSE if audio should be played from the earpiece.

```

SharedPreferences preferences = getSharedPreferences("AppPreferences",
Context.MODE_PRIVATE);
SharedPreferences.Editor editor = preferences.edit();

```

```
editor.putBoolean("ShouldUseLoudspeaker", Boolean.TRUE);
editor.commit();
```

Configurable Loudspeaker

This section is for future implementation. Currently is not present in the example application.

IMPORTANT:

For a better handling of the audio session rerouting. The changes about using or not the loudspeaker, should be available ONLY when a channel is streaming. If the application is not streaming the component to change the loudspeaker value should be hidden or disabled.

Another scenario that is very important to mention is for devices that don't have the earpiece, like tablets. For those cases the reroute configuration shouldn't be available. You can use a method to set up this feature when it's the case. For example:

```
private void setupSpeakerControl(View view) {
    // Validates speaker configuration and switch behavior.
    AudioManager mAudioManager = (AudioManager)
    getContext().getSystemService(Context.AUDIO_SERVICE);

    SharedPreferences preferences =
    getActivity().getSharedPreferences("AppPreferences",
        Context.MODE_PRIVATE);
    //get value of the Loudspeaker use
    final boolean shouldUseLoudspeaker =
    preferences.getBoolean(Constants."ShouldUseLoudspeaker",
        getResources().getBoolean(R.bool.useLoudspeakerDefaultValue));

    //Create a flag for earpiece
    boolean mEarpiecePresent = false;

    if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.M)
    {
        //look for the earpiece on the device.
        AudioDeviceInfo[] adi =
        mAudioManager.getDevices(AudioManager.GET_DEVICES_OUTPUTS);
        for (int i = 0; i < adi.length; i++) {
            switch (adi[i].getType()) {
                case AudioDeviceInfo.TYPE_BUILTIN_EARPIECE:

                    mEarpiecePresent = true;
                    break;
            }
        }
    }
}
```

```

    }
}
}
//create component to show loudspeaker configuration
fab = view.findViewById(R.id.speakerControl);
fabContainer = (LinearLayout) view.findViewById(R.id.speakerControlContainer);

if (fab != null) {
//use an image for the component
    fab.setImageResource(shouldUseLoudspeaker ? R.drawable.loudspeaker_on :
R.drawable.loudspeaker_off);
//Validate earpiece is present
    if (mEarpiecePresent) {
        fab.setOnClickListener(new View.OnClickListener(){
            @Override
            public void onClick(View view) {
//Add code for loudspeaker configuration and then start stream
            }
        });
    } else {
// If there is no earpiece, hide the component to configure
        fab.setVisibility(View.GONE);
        fabContainer.setVisibility(View.GONE);
    }
}
}
}

```

- A.** Every time the loudspeaker value is changed the streaming should be restarted to reconfigure the reroute of the audio as requested.

For example, if you have defined a component for this configuration. In the onClick method, you should start the stream again.

```

@Override
public void onClick(View view)
{
    final boolean usingLoudspeaker =
preferences.getBoolean("ShouldUseLoudspeaker",
    getResources().getBoolean(R.bool.useLoudspeakerDefaultValue));

    mAudioManager.setMode(usingLoudspeaker ?
AudioManager.MODE_IN_COMMUNICATION : AudioManager.MODE_NORMAL);
}

```

```

        SharedPreferences.Editor editor = preferences.edit();
        editor.putBoolean(Constants.APP_PREFERENCES_SHOULD_USE_LOUDSPEAKER,
            !usingLoudspeaker);
        editor.commit();

        AudioChannel currentChannel = connection.getCurrentAudioChannel();

        if (currentChannel != null) {
            connection.startStreamOfChannel(currentChannel, "Currently listening to
%s. ", currentChannel.getTitle()), !usingLoudspeaker, null, shouldPlayStereoFlag);
        }
    }
}

```

The **mAudioManager** object should be defined in a previous configuration method or on the creation of the current view.

```

AudioManager mAudioManager = (AudioManager)
getContext().getSystemService(Context.AUDIO_SERVICE);

```

- B. Another important suggestion is to implement a notification on the list of channels to refresh the UI when there is an interruption of the audio (like the case when the Bluetooth or wired headset is disconnected) because the audio is stopped when that happens. This will allow the user to know that the audio needs to be restarted manually to get the audio from the correct output. To handle the interruptions use **mManagerListener** described in point 2 [here](#).

Start playback of channel

This method receives an **AudioChannel** object (maybe from the list adapter) and tries to start the streaming.

```

public void startPlaybackOfChannel(AudioChannel channel) {
    final boolean shouldPlayInStereo = true;
    // This variable indicates if the audio should be routed to the loudspeaker
    // If useLoudspeaker is set to false the app will try to route the audio
    // to the earpiece (if it is available)
    final boolean useLoudspeaker = true;
    mExxtractorConnection.startStreamOfChannel(
        channel,

```

```

String.format("Currently listening to %s.", channel.getTitle()),
shouldPlayInStereo,
new AudioEverywhereResponseHandler() {
    @Override
    public void onSuccess() {
        updateUI();
    }

    @Override
    public void onSuccess(Object list) {
        Log.i(LOG_TAG, "StartStreamOfChannel onSuccess with List");
    }

    @Override
    public void onFailure(AudioEverywhereException exception) {
        Log.i(LOG_TAG, "StartStreamOfChannel onFailure", exception);
    }
}, useLoudspeaker
);
}

```

Stop playback of channel

This method tries to stop the streaming of the current selected channel if there is one.

```

public void stopCurrentStream() {
    final AudioChannel activeChannel = mExtractorConnection.getCurrentAudioChannel();
    if (activeChannel != null) {
        // If the channel is playing, we can stop it now
        if (activeChannel.getState() == AudioChannel.ChannelStatus.PLAYING) {
            mExtractorConnection.stopStream(new AudioEverywhereResponseHandler() {
                @Override
                public void onSuccess() {
                    updateUI();
                }

                @Override
                public void onSuccess(Object list) {
                    updateUI();
                }

                @Override
                public void onFailure(AudioEverywhereException exception) {
                    updateUI();
                }
            });
        }
    }
}
}

```


Loading of channel information

Below you will find a list of the main methods of the `AudioChannel` object that can be used to retrieve the required information.

```
audioChannel.getTitle()
audioChannel.getSubtitle()
audioChannel.getDescription()
audioChannel.getTag()
audioChannel.getBackgroundColor()

// Must validate if the format is fine
audioChannel.getImageUrl()
audioChannel.getPlayingImageUrl()

audioChannel.getState()
AudioChannel.ChannelStatus.PLAYING
AudioChannel.ChannelStatus.STOPPED
AudioChannel.ChannelStatus.LOADING
AudioChannel.ChannelStatus.READY
AudioChannel.ChannelStatus.ISSUE
```

The following example is intended to show how to display channels information, by asynchronously loading channel images.

```
Ion.with(mContext).load(audioChannel.getImageUrl())
    .intoImageView(holder.channelBackground)
    .setCallback(new FutureCallback<ImageView>() {
        @Override
        public void onCompleted(Exception e, ImageView result) {
            if (result != null && e == null) {
                // Sets the background color to white for png images
                result.setBackgroundColor(Color.WHITE);
            }
        }
    });
```