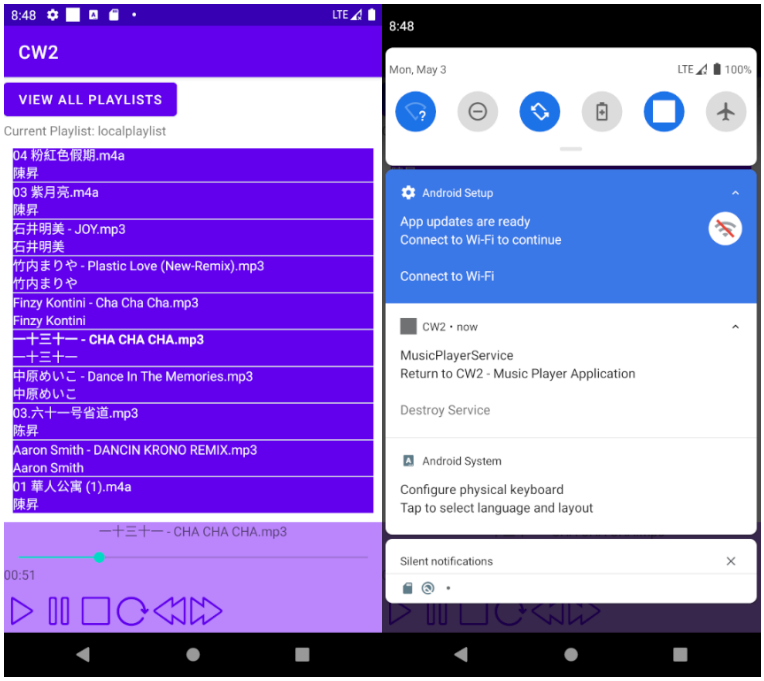
COMP3059 – Coursework 2 Report

Content

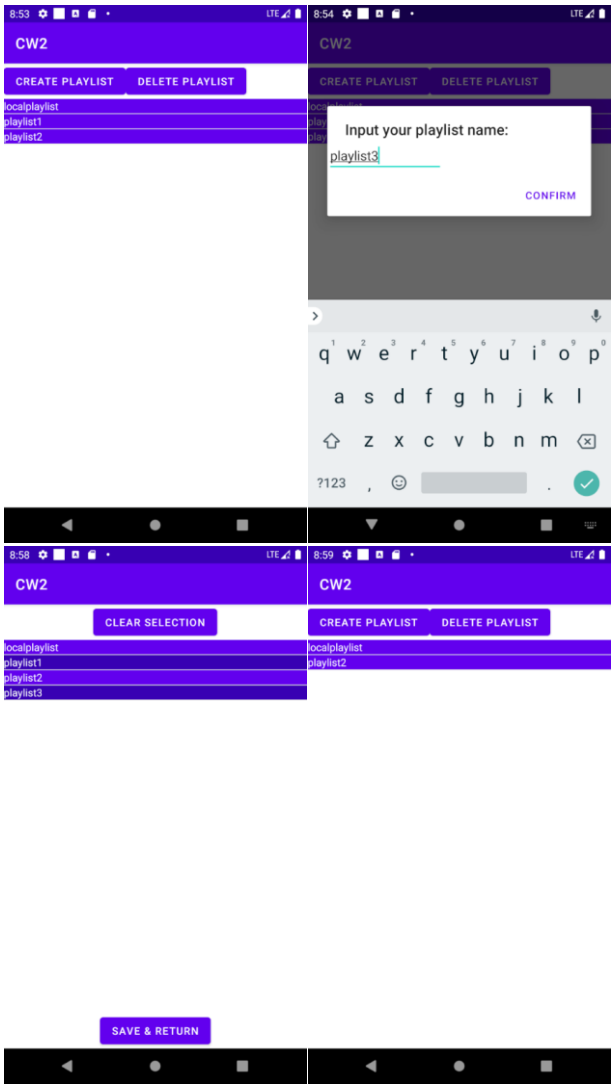
COMP3059 – Coursework 2 Report 1
Demonstration 2
Class Structure 5
Implementation 7
Permission 7
Music Loading 7
Implicit Intent 8
Two-way Communication 9
Notification & Components Lifecycle 12
Database 13

Demonstration

On starting the activity, a notification is created, and the default local play list is displayed. Users can choose to start, pause, stop, restart a music; or play previous/next music. When the application is terminated, the music play will still proceed. Users can click “Destroy Service” button in the notification to terminate the music play or click the notification context to return to the application.



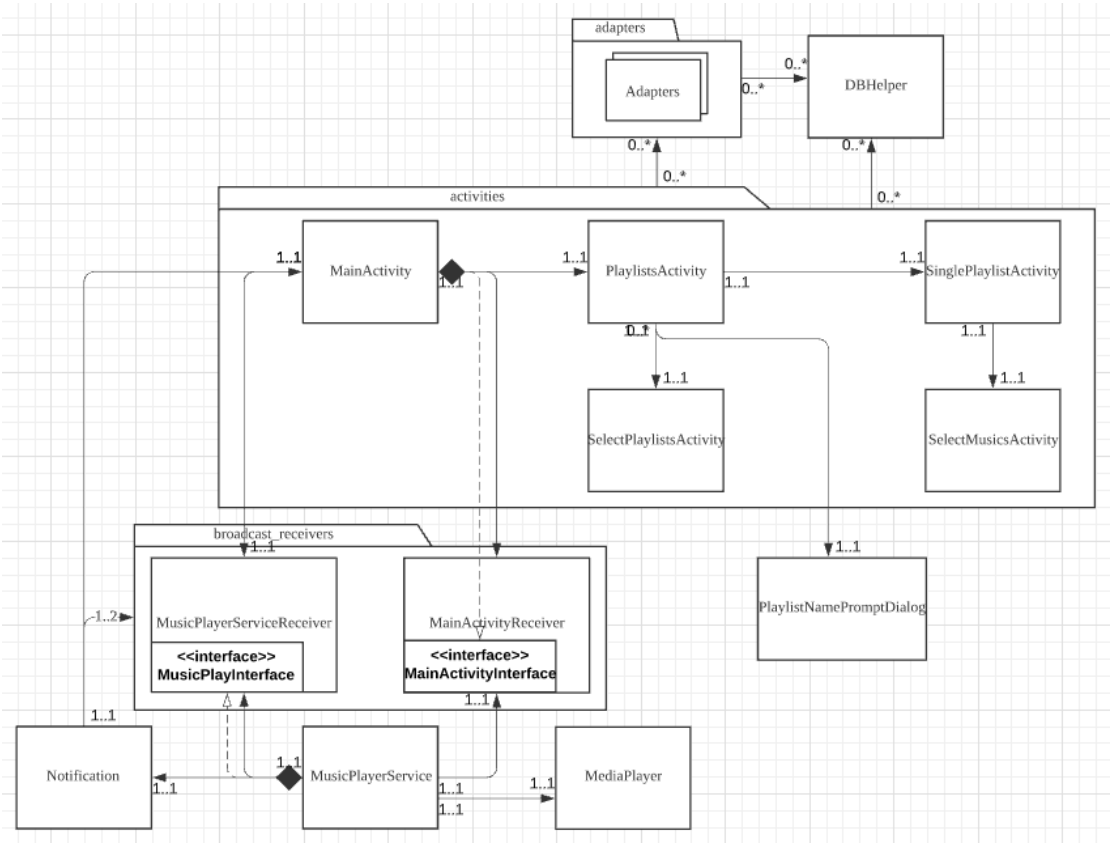
Users can view all playlists by clicking the button. In the redirected interface, users can create a playlist with specified name or delete multiple playlists. For creation, users can create a playlist a dialog with name replication detected. For deletion, users are redirected to a new interface to select playlists to delete. Users can click “CLEAR SELECTION” to clear all selected playlists and can click “SAVE & RETURN” to return results.



Then, users can open a playlist by clicking a card. In the redirected interface, users can insert/delete multiple instances of music to/from the playlist by a new interface similar to that in above. When users decide to play a music, they can select a music in the playlist and click “PLAY” button to play the current playlist in the main interface. If there is no music in the playlist, no music will be played. If users do not select a music, the first music will be played by default, otherwise the selected music will be played.

Additionally, users can choose to open a music file with this application and play it. The views can be adjusted to rotation.

Class Structure



The above diagram is a simplified class diagram of this project. Functionalities of main classes are introduced below:

1. `MainActivity`: Display a playlist; send broadcast to `MusicPlayerService` to play, pause, stop or restart musics; play previous/next music in the playlist; navigate to `PlaylistsActivity`.
2. `MainActivityReceiver` and `MusicPlayerServiceReceiver`: Receive broadcasts for corresponding activity and execute commands in that activity.
3. `MusicPlayerService`: Use `MediaPlayer` to play, pause, stop, or restart a music; send broadcast to `MainActivity` to inform current progress of the playing music; start `Notification`.
4. `PlaylistsActivity`: Display all playlists; Open `PlaylistNamePromptDialog`; navigate to `SelectPlaylistsActivity`; navigate to `SinglePlaylistActivity`.
5. `PlaylistNamePromptDialog`: Prompt user to input a playlist name and send it back to `PlaylistsActivity`.

6. `SelectPlaylistsActivity`: Select a list of playlists to be deleted and send it back to `PlaylistsActivity`.
7. `SinglePlaylistActivity`: Display a list of musics in the playlist; navigate to `SelectMusicsActivity`.
8. `SelectMusicsActivity`: Select a list of musics to be inserted/deleted and send it back to `SelectMusicsActivity`.
9. `DBHelper`: Execute database commands.
10. `Notification`: Return to `MainActivity` or destroy `MainActivity` and `MusicPlayerService`.

In general, the functionalities of playing music/ viewing playlist, viewing/operating on all playlists, and selecting music a playlist to play are separated into three activities. The service is started in the `MainActivity` and continues its lifecycle until the “Destroy Service” button in the notification is clicked. Other implementation like data extraction from underlying storage is introduced in the following sections in detail.

Implementation

In this section, detailed implementations are introduced.

Permission

Since the application should be able to read music files from SD card or implicit intents, the read external storage permission should be granted. Likewise, since the application should be able to open a notification, the service started by the application should be granted the permission to run in the foreground. Therefore, in `AndroidManifest.xml`, following permission usage requests are declared.

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

Then in the `MainActivity`, these permissions are checked and requested for granting with `Dexter`, a dynamic permission checking library. If requested permissions are not granted, the application will be terminated.

```
// check permission in runtime and determine next moves
protected void checkPermission() {
    Dexter.withActivity(this)
            .withPermission(Manifest.permission.READ_EXTERNAL_STORAGE)
            .withListener(new PermissionListener() {
                @SuppressLint("SetTextI18n")
                @RequiresApi(api = Build.VERSION_CODES.KITKAT)
                @Override
                public void onPermissionGranted(PermissionGrantedResponse response) {
```

Music Loading

If the permissions are granted, in the initialization phase, a list of `Music` object is loaded. For implementation, a cursor is created by querying the local media directory. Then, cursor is iteratively probed to the end for extracting information from the music files. `Music` objects are created and added to the array list.

```java
// load music from sdcard/Music/
private ArrayList<Music> loadMusics(Context context) {
    ArrayList<Music> musicList = new ArrayList<~>();

    Uri uri = MediaStore.Audio.Media.EXTERNAL_CONTENT_URI;
    String selection = MediaStore.Audio.Media.IS_MUSIC + "!= 0";
    @SuppressLint("Recycle") Cursor c = context.getContentResolver().query(uri, projection: null, selection, selectionArgs: null, sortOrder: null);
    if(c != null) {
        while(c.moveToNext()) {
            // create music object and add music object to the list
            String url = c.getString(c.getColumnIndex(MediaStore.Audio.Media.DATA));
            String artist = c.getString(c.getColumnIndex(MediaStore.Audio.Media.ARTIST));
            String musicName = c.getString(c.getColumnIndex(MediaStore.Audio.Media.DISPLAY_NAME));
            int duration = c.getInt(c.getColumnIndex(MediaStore.Audio.Media.DURATION));

            Music music = new Music(url, musicName, artist);
            music.setDuration(duration);
            musicList.add(music);
        }
    }
    return musicList;
}
```

Implicit Intent

To receive implicit intent to play music, initially a intent filter is registered for the `MainActivity` in `AndroidManifest.xml`.

```xml
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.BROWSABLE" />
    <category android:name="android.intent.category.DEFAULT" />

    <data android:scheme="file" />
    <data android:scheme="http" />
    <data android:scheme="https" />
    <data android:scheme="content" />

    <data android:mimeType="audio/mpeg" />
    <data android:mimeType="audio/mp3" />
    <data android:mimeType="audio/mpeg3" />
    <data android:mimeType="audio/x-mpeg-3" />
</intent-filter>
```

Then, the application will listen to implicit intents from opening music files.

```
// listen to implicit intent
@RequiresApi(api = Build.VERSION_CODES.KITKAT)
private void listenImplicitIntent(Intent inIntent) {
    if (Objects.equals(inIntent.getAction(), b: "android.intent.action.VIEW")) {

        // get posted music
        Uri uri = inIntent.getData();
        MusicLoader musicLoader = new MusicLoader( context: this);
        Music postedMusic = musicLoader.loadMusicByUrl(uri);

        if (postedMusic != null) {
             // play music
            String postedMusicName = postedMusic.getMusicName();
            for (Music music: localMusicList) {
                if (music.getMusicName().equals(postedMusicName)) {
                    currentMusic = music;
                }
            }
            startPlay( view: null);

            // update music name text view
            musicNameTextViewUI.setText(currentMusic.getMusicName());

            // configure seek bar maximum value
            seekBarUI.setMax(currentMusic.getDuration());
        }
    }
}
```

Two-way Communication

To achieve the requirement in the specification sheet, I created a service with `startService` method in the `MainActivity`. However, compare to `bindService` method, the service started in this way cannot communicate directly with activities. I use broadcast receiver component as a media to realize communication.

A two-way communication should be established that the activities should send the current music and music instruction (play, pause, stop, and restart) to the service as well as that the service should send back current progress of the playing music to configure the seek bar in `MainActivity`.

For activity-service communication, in my `MainActivity`, there are several button-clicking listening methods to play, pause, stop, and restart the selected music. They ought to send different signals to the service to execute different operations with the media player. Hence, different intent actions are defined for sending the broadcast.

```
// send broadcast to music service
private void sendMusicBroadcast(String filterName) {
    Intent intent = new Intent();
    intent.setAction(filterName);

    Bundle bundle = new Bundle();
    if (currentMusic != null) {
        bundle.putSerializable("5", currentMusic);
        intent.putExtras(bundle);
    }

    sendBroadcast(intent);
    Log.i(TAG, msg: filterName + " broadcast is sent!");
}
```

```
// pause playing music
public void pausePlay(View view) { sendMusicBroadcast( filterName: "@string/pause_filter"); }
```

Next, after initialized in `MusicPlayerService` service, a broadcast receiver will listen to intent with specific actions.

```
public MusicPlayerServiceReceiver(Context context) {
    super();
    // configure filters
    IntentFilter filter = new IntentFilter();
    filter.addAction("@string/play_filter");
    filter.addAction("@string/pause_filter");
    filter.addAction("@string/restart_filter");
    filter.addAction("@string/stop_filter");
    context.registerReceiver( receiver: this, filter);
}
```

The broadcast receiver has an inner interface declaring the music operations, which must be implemented in the `MusicPlayerService`.

```
public interface MusicPlayInterface {
    void playMusic(Music music);
    void pauseMusic(Music music);
    void restartMusic();
    void stopMusic();
    void startBroadcastBack(Music music);
}
```

Once the broadcast receiver receives an intent, it will call implemented methods in the service.

```
// extract music from bundle
music = (Music) inBundle.getSerializable("5");

// music actions
MusicPlayerService musicPlayerServiceContext = (MusicPlayerService) context;
switch (intent.getAction()) {
    case "@string/play_filter":
        musicPlayerServiceContext.playMusic(music);
        break;
    case "@string/pause_filter":
        musicPlayerServiceContext.pauseMusic(music);
        break;
    case "@string/restart_filter":
        musicPlayerServiceContext.restartMusic();
        break;
    case "@string/stop_filter":
        musicPlayerServiceContext.stopMusic();
        break;
}

// send music progress back
musicPlayerServiceContext.startBroadcastBack(music);
```

In detail, it will tell the service to perform corresponding operations of media player[1] according to the intent action. Then, it will tell the service to broadcast back current music progress to the main activity for setting seek bar position.

[1] Reference to: https://developer.android.com/reference/android/media/MediaPlayer

```
public void restartMusic() {
    // paused/started -> started
    mediaPlayer.seekTo( msec: 0);
    mediaPlayer.start();
}
```

```
// start broadcast back music progress
@Override
public void startBroadcastBack(Music music) {
    // update music progress and broadcast back to main
    Thread thread = new Thread(new Runnable() {
        @Override
        public void run() {
            while (true) {
                // update progress
                music.setMusicProgress(mediaPlayer.getCurrentPosition());

                // broadcast back to main activity
                broadcastBackProgress(music);
            }
        }
    });

    // start thread
    if (!thread.isAlive()) {
        thread.start();
    }
}
```

The implementation for service-activity communication follows the similar pattern. After the broadcast receiver receives the intent, it will tell `MainActivity` to configure seek bar position according to music progress passed.

Notification & Components Lifecycle

Since the service is started by `startService` method rather than `bindService` method, the service can be maintained after application being killed.

The notification is created in the service with following codes referenced to lecture

codes. A button named “Destroy Service” is created for closing the application and the service in the notification. When it is clicked, a broadcast is sent for the `MainActivity` and `MusicPlayerService`. Besides, the context is set clickable to navigate back to the application.

```
// reference: Week 3 lecture materials
private void createNotification() {
    NotificationManager notificationManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    // Create the NotificationChannel, but only on API 26+ because
    // the NotificationChannel class is new and not in the support library
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        CharSequence name = "channel name";
        String description = "channel description";
        int importance = NotificationManager.IMPORTANCE_DEFAULT;
        NotificationChannel channel = new NotificationChannel(CHANNEL_ID, name, importance);
        channel.setDescription(description);
        // Register the channel with the system; you can't change the importance
        // or other notification behaviors after this
        notificationManager.createNotificationChannel(channel);
    }

    Intent intent = new Intent( packageContext: MusicPlayerService.this, MainActivity.class);
    intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TASK);
    PendingIntent pendingIntent = PendingIntent.getActivity( context: this,  requestCode: 0, intent,  flags: 0);

    Intent actionIntent = new Intent( action: "@string/stop_service_filter");
    PendingIntent pendingActionIntent = PendingIntent.getBroadcast( context: MusicPlayerService.this,  requestCode: 0, actionIntent, PendingIntent.FLAG_UPDATE_CURRENT);

    NotificationCompat.Builder mBuilder = new NotificationCompat.Builder( context: this, CHANNEL_ID)
            .setSmallIcon(R.drawable.ic_launcher_background)
            .setContentTitle("MusicPlayerService")
            .setContentText("Return to CW2 - Music Player Application")
            .setContentIntent(pendingIntent)
            .addAction(R.drawable.ic_launcher_foreground,  title: "Destroy Service", pendingActionIntent)
            .setPriority(NotificationCompat.PRIORITY_DEFAULT);

    //notificationManager.notify(NOTIFICATION_ID, mBuilder.build());
    startForeground(NOTIFICATION_ID, mBuilder.build());
```

In `MainActivity`, the call will terminate the entire application. In `MusicPlayerService` correspondingly, the `onDestroy` method is called.

Database

A database is created to contain tables representing a list of playlists. I implemented a `DBHelper` class to encapsulate SQL related operations.

Initialization

In the initialization phase of the application, a default playlist (local playlist) is created. For database operations, a table is created with the following columns.

LocalPlaylist

Refresh table Live updates

	_id	musicName	artist	url
1	1	tkzc.mp3	久石譲	/storage/10EE-1004/Music/tkzc.mp3
2	2	naushika.mp3	細野晴臣	/storage/10EE-1004/Music/naushika.mp3
3	3	pclhz.mp3	紅鼬	/storage/10EE-1004/Music/pclhz.mp3

Insertion of A User-specified Playlist

In the `PlaylistsActivity`, users could insert a playlist with specified name input in a dialog. For database operations, tables are created according to a `tableName`. The return value indicates whether the creation is successful.

```
// create a new table
public boolean createNewTable(String tableName) {
    SQLiteDatabase db = DBOpenHelper.getWritableDatabase();

    // check if table already exists
    @SuppressLint("Recycle") Cursor c = db.rawQuery( sql: "SELECT count(*) FROM sqlite_master WHERE type='table' AND name = '" + tableName +"'", selectionArgs: null);

    if (c.moveToNext()) {
        int count = c.getInt( columnIndex: 0);
        if (count > 0) {
            // table exists -> return negative response
            return false;
        }
        else {
            // table not exists -> create new playlist and return positive response
            String sql = "CREATE TABLE "+ tableName +"(_id integer PRIMARY KEY AUTOINCREMENT,musicName varchar(128),artist varchar(128),url varchar(128) UNIQUE)";
            db.execSQL(sql);
            DBContract.getInstance().addTableName(tableName);
            return true;
        }
    }
    return false;
}
```

Deletion of Multiple Playlists

In the `PlaylistsActivity`, users can delete multiple playlists. For implementation, on clicking the delete button, the user will be directed to an activity for selecting multiple playlists. Positions of selected playlists will be stored in a `playlistPositions` array list to be sent back to the `PlaylistsActivity`.

```
// listen to list view item click event
public void listenListViewItemClick(ListView playlistsListViewUI) {

    // put the item to the selected music list
    playlistsListViewUI.setOnItemClickListener(new AdapterView.OnItemClickListener() {
        @SuppressLint("ResourceAsColor")
        @Override
        public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
            playlistPositions.add(position);
            view.setBackgroundColor(R.color.white);
        }
    });
}
```

Once the `PlaylistsActivity` successfully receives selected positions, it will get the playlist names from the UI according to the sent positions. Then it uses the `DBHelper` to delete the tables by playlist names. When database modifications are done, the UI

is updated accordingly.

```
// delete the table in the database according to its name
public void deleteTableByTableName(String tableName) {
    SQLiteDatabase db = DBOpenHelper.getWritableDatabase();

    String sql = "DROP TABLE " + tableName;
    db.execSQL(sql);
}
```

Insertion & Deletion of Multiple Musics

Insertion and deletion of multiple musics are realized in the similar way as the previous section.

```
// insert an item to a table
public void insertItemToTable(@Nullable String tableName, ContentValues values) {
    SQLiteDatabase db = DBOpenHelper.getWritableDatabase();

    long id = db.insert(tableName,  nullColumnHack: null, values);
}
```

For deletion, the item is located according to music URL.

```
// delete an item from a table according to url
public void deleteItemFromTableByUrl(@Nullable String tableName, String url) {
    SQLiteDatabase db = DBOpenHelper.getWritableDatabase();

    String sql = "DELETE FROM " + tableName + " WHERE " + DBContract.URL + " == '" + url + "'";
    db.execSQL(sql);
}
```

Update List View of Playlists

Every time the database is changed, the corresponding list view should be updated. This requires query operations on the database. For query of tables, the instance is simply quired by the index in the list view.

```
// query the table in the database according to its index
public Cursor queryTableByIndex(int index) {
    SQLiteDatabase db = DBOpenHelper.getReadableDatabase();

    Cursor c = db.rawQuery( sql: "SELECT name FROM sqlite_master WHERE type='table' AND name != 'android_metadata' AND name != 'sqlite_sequence'", selectionArgs: null);

    for (int i = 0; i < index; i++) {
        if (!c.moveToNext()) {
            break;
        }
    }
    return c;
}
```

Update List View of Musics

The update on list of musics takes an indirect measure. For database operations, the received index of item is firstly queried to the primary key `_id`, then the item is queried according to the primary key.

```
// query an item from a table according to index
public Cursor queryItemFromTableByIndex(@Nullable String tableName, int index) {
    SQLiteDatabase db = DBOpenHelper.getWritableDatabase();

    int _ID = query_IdFromTableByIndex(tableName, index);
    Cursor c = db.query(tableName, columns: null, selection: DBContract._ID + " == " + _ID, selectionArgs: null, groupBy: null, having: null, orderBy: null);
    return c;
}

// query a table for _id according to index
private int query_IdFromTableByIndex(@Nullable String tableName, int index) {
    SQLiteDatabase db = DBOpenHelper.getReadableDatabase();

    int _ID = 1;
    @SuppressLint("Recycle") Cursor c = db.rawQuery( sql: "SELECT *, (SELECT COUNT(*) FROM " + tableName + " b WHERE a._id >= b._id) AS cnt FROM " + tableName + " a", selectionArgs: null);
    if (c.moveToNext()) {
        int ctn = c.getInt( columnIndex: 4);
        do {
            if (c.getInt( columnIndex: 4) == index + 1) {
                _ID = c.getInt( columnIndex: 0);
            }
        } while(c.moveToNext());
    }
    return _ID;
}
```

```
// destroy the service
@Override
public void onDestroy() {
    // unregister receiver
    unregisterReceiver(receiver);

    // close notification
    NotificationManager notificationManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    notificationManager.cancel(NOTIFICATION_ID);

    // stop thread and release music player
    exit = true;
    mediaPlayer.stop();
    mediaPlayer.release();

    Log.i(TAG, msg: "Music service destroyed!");
    super.onDestroy();
}
```