Custom App Project HD – Spike: Mood Music

Goals:

The aim of this task is to explore beyond the scope of this unit through the creation of a custom app. I have decided to work on a music player app, which I've titled 'Mood Music', that behaves like a music app should. In this report I will detail the different parts of Mood Music and what purpose they play in the app's functions. Specifically, I will mention:

- The high level design
- The browser service and client
- Behaving audio: Audio Focus change listener and noisyReceiver
- Building notifications

Github Link: https://github.com/Xavier-Gibbon/MoodMusic

Tools and Resources Used

- Android Studio
- Picasso
- Audio playing overview: https://developer.android.com/guide/topics/media
  - Building an audio app: https://developer.android.com/guide/topics/media-apps/audio-app/building-an-audio-app
  - Building a MediaBrowserService: https://developer.android.com/guide/topics/media-apps/audio-app/building-a-mediabrowserservice
  - Building a MediaBrowserClient: https://developer.android.com/guide/topics/media-apps/audio-app/building-a-mediabrowser-client
  - MediaSession Callbacks: https://developer.android.com/guide/topics/media-apps/audio-app/mediasession-callbacks
  - Managing audio focus: https://developer.android.com/guide/topics/media-apps/audio-focus
  - Don't be noisy: https://developer.android.com/guide/topics/media-apps/volume-and-earphones#becoming-noisy

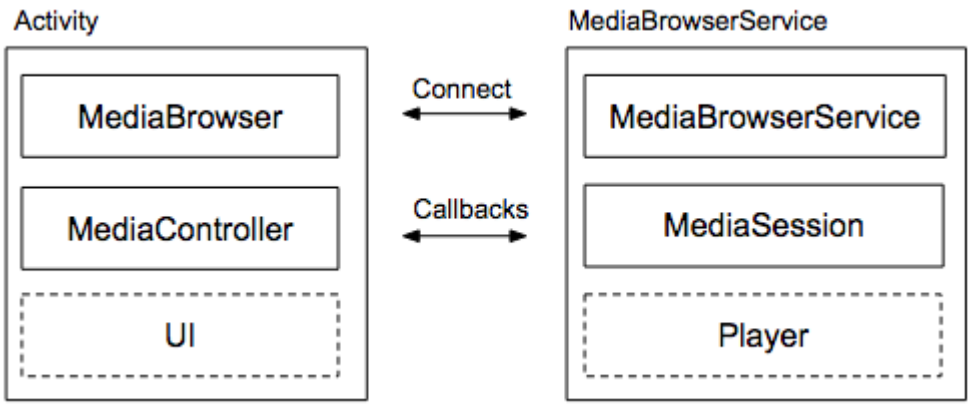
Knowledge Gaps and Solutions

For this app I refer to both the MediaBrowserService and the MediaBrowserClient. While these are the classes that I refer to, I actually implement MediaBrowserServiceCompat and MediaBrowserClientCompat. These two classes implement their counterparts, but also handle a lot of the handshakes that often happens when this design is used.

The high level design

Mood music implements the 'MediaBrowserService' design, taken from android development documentation. The idea behind this design is to create two classes, a MediaBrowserService which is a service that handles the media playing and a MediaBrowserClient which is an activity that connects to the service and controls the player.

This sort of design decouples the actual audio playing from the activity, allowing the music to be played in the background and allows other sources not provided by us to control the music, such as user pressing media buttons or events triggered from the phone. It also clearly separates the UI code from the Music code, which makes scaling this app easier to do. Below is a generic diagram from the android documentation of this architecture, and Appendix 1 shows a more detailed UML diagram of Mood Music.



The browser service and client

The MediaBrowserSerivce (implemented as MoodMusicBrowserService) is responsible for playing the audio and managing everything that either relies or effects the audio playing. This ranges from the audioFocusChangeListener to the notification.

Firstly, the MediaBrowserService gets the music to list off and to play from the phone's storage. This is done through a Content Resolver, using a URI that explicitly finds music files on disk. For each file the resolver finds using this URI, it constructs a media description object using the files id, title, artist and album id. This description will be used throughout the entire app, from handling the music playing to displaying the current song being played.

The MediaBrowserSerivce manages the audio playing through the callback methods it implements in its mediaCallback object. The callback manages more than playing the audio: it also manages the audio focus and noisy receiver to make the audio behave with the app and it updates the metadata, the playback state and the notification to ensure that the data that the service provides is always up to date. The callback also ensures that the service stays alive if there are no bound clients while its playing as well as setting up the playlist for the player. Anything outside of this callback should refer to it when attempting to control the music instead of directly using the music player due to these different factors that the callback sets.

The callback object also contains a MoodMusicPlayerManager class. This class actually has the MediaPlayer object and handles all of the logic and functions around it, such as creating the MediaPlayer when needed and checking if the player has music ready to play.

The MediaBrowserClient (implemented as MoodMusicBrowserClient) is an interface that allows the user to startup and interact with the service. The client, when started, will connect to the service (if one doesn't exist, the service will be created). Once connected, the media buttons on the activity will have their 'onClick' functions set (which will use the services methods to control the music) and the client will subscribe to the service. The service will return a list of music to the subscribed client, and the client will use that information to fill out the recycler view. Here, the user can interact with the service through the client by setting the songs to be played as well as playing, pausing or skipping songs. The client can update its interface with the services metadata and playback state through its onMetadataChanged (not currently implemented) and onPlaybackStateChanged methods in its controller callback. These get called when the mediaSession in the service has its metadata or playback state set.

```
val am = applicationContext.getSystemService(Context.AUDIO_SERVICE) as AudioManager

// Request audio focus for playback, this registers the afChangeListener
audioFocusRequest = AudioFocusRequest.Builder(AudioManager.AUDIOFOCUS_GAIN).run { this: AudioFocusRequest.Builder
    setOnAudioFocusChangeListener(focusChangeListener)
    setAudioAttributes(AudioAttributes.Builder().run { this: AudioAttributes.Builder
        setContentType(AudioAttributes.CONTENT_TYPE_MUSIC)
        build() ^run
    })
    build() ^run
}

// Try to get the audio focus, we don't play anything if we can't have the audio focus
val result = am.requestAudioFocus(audioFocusRequest)
if (result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {
    Log.d(this::class.qualifiedName,  msg: "Audio focus granted, playing music")
    // Start the service (prevents the service from dying when the activity is closed)
    startService(Intent(baseContext, MoodMusicBrowserService::class.java))
    mediaSession.isActive = true

    // start the player
    player.play()

    // Start the noisy receiver and update the metadata and playback state
    registerReceiver(noisyReceiver, IntentFilter(ACTION_AUDIO_BECOMING_NOISY))
    updateMetadataAndPlaybackState(
        PlaybackStateCompat.STATE_PLAYING,
        PlaybackStateCompat.ACTION_PAUSE
    )
    // Put the service in the foreground, post notification
    updateNotification()
}
```

```
// This sets up the buttons in the activity and updates the UI using the metadata and playback state
// TODO: Should we have the buttons disabled by default, then enabled here?
private fun buildTransportControls() {
    val mediaController = MediaControllerCompat.getMediaController( activity: this@MoodMusicBrowserClient)
    val adapter = findViewById<RecyclerView>(R.id.list_music).adapter as MusicAdapter
    // Grab the view for the play/pause button
    findViewById<ImageView>(R.id.btn_play_pause).setOnClickListener { it: View!
        val pbState = mediaController.playbackState.state
        if (pbState == PlaybackStateCompat.STATE_PLAYING) {
            mediaController.transportControls.pause()
        } else {
            mediaController.transportControls.play()
        }
    }

    findViewById<ImageView>(R.id.btn_next).setOnClickListener { it: View!
        mediaController.transportControls.skipToNext()
    }
    findViewById<ImageView>(R.id.btn_prev).setOnClickListener { it: View!
        mediaController.transportControls.skipToPrevious()
    }
```

Behaving audio: Audio Focus change listener and noisyReceiver

It is important that the music that is being played works well with the rest of the phone and that it doesn't behave unexpectedly to our user. This is why I have implemented two components: an audio focus change listener and a noisy receiver.

The audioFocusChangeListener's purpose is to listen for any change to the audio focus. This is caused by other functions on the phone attempting to gain the audio focus, like an app wanting to play a video, a notification wanting to briefly play a sound or a call arriving on the phone. There are many different ways for our app to lose focus, but it leads to two actions which depend on whether or not the app can duck the audio. If the app can duck the audio, the app will continue to play the audio but with a lowered audio (used when notifications play a sound). If the app cannot duck the audio, then the audio will be paused.

```
private val focusChangeListener: AudioManager.OnAudioFocusChangeListener =
    AudioManager.OnAudioFocusChangeListener { focusChange ->
        when (focusChange) {
            AudioManager.AUDIOFOCUS_GAIN -> {
                Log.i(MoodMusicBrowserService::class.qualifiedName, msg: "Audio Focus gained")
                mediaCallback.onPlay()
            }
            AudioManager.AUDIOFOCUS_LOSS -> {
                Log.i(MoodMusicBrowserService::class.qualifiedName, msg: "Audio Focus lost")
                mediaCallback.onPause()
            }
            AudioManager.AUDIOFOCUS_LOSS_TRANSIENT -> {
                Log.i(MoodMusicBrowserService::class.qualifiedName, msg: "Audio Focus lost: transient")
                mediaCallback.onPause()
            }
            AudioManager.AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK ->  {
                Log.i(MoodMusicBrowserService::class.qualifiedName, msg: "Audio Focus lost: transient and can duck")
                // Audio ducking is done automatically, nothing to do on this end
            }
        }
    }
```

The noisyReceiver is a broadcast receiver that is designed to listen to the 'becoming noisy' action. This receiver gets registered when the service starts playing music, and stops when the music is paused. This action is triggered whenever the phone is becoming noisy, usually by the user removing headphones from the device. When this happens, the app will pause the music that is currently playing.

```
// noisyReceiver is used to pause the music if the phone becomes noisy
// e.g. if the headphones are unplugged
private val noisyReceiver :BroadcastReceiver  = object : BroadcastReceiver() {
    override fun onReceive(context: Context?, intent: Intent?) {
        if (intent?.action == ACTION_AUDIO_BECOMING_NOISY) {
            onPause()
        }
    }
}
```

These changes are necessary for any app that wants to play audio outside the app, as failing to do this will result in an awful user experience. Ignoring these focus changes will cause the app to play over other sounds that the user will want to hear, and ignoring the noisy action will cause the app to continue to play while the user is removing headphones. This does not only sound awful, but it is unexpected for the user and it will cause the user to stop using the app due to it misbehaving and being a huge hassle.

Building the notification

For this app I also created a notification. The notification is implemented through first initializing the notification builder when the service is created. This sets up the parts of the notification that won't change a lot, such as creating its look and setting up the different actions it has available. However, the notification does not get created until the updateNotification method gets called. In this method, the rest of the notification gets created, which includes displaying the current music playing as well as updating the state of the play/pause button. Finally, the notification gets pushed to the phone using the services function startForeground. The builder class gets reused each time the notification needs to get updated.

It is important to have another method of controlling the service available to the user that do not require the phones constant access, as it gives them an easy way to control the current music playing without the need to switch activities for a task as small as this one. It also gives them a way to control the music while the phone is locked, which makes the app more usable.

Open Issues and Recommendations

There are still some things that I would like to add to this app, and some areas where this app can be improved.

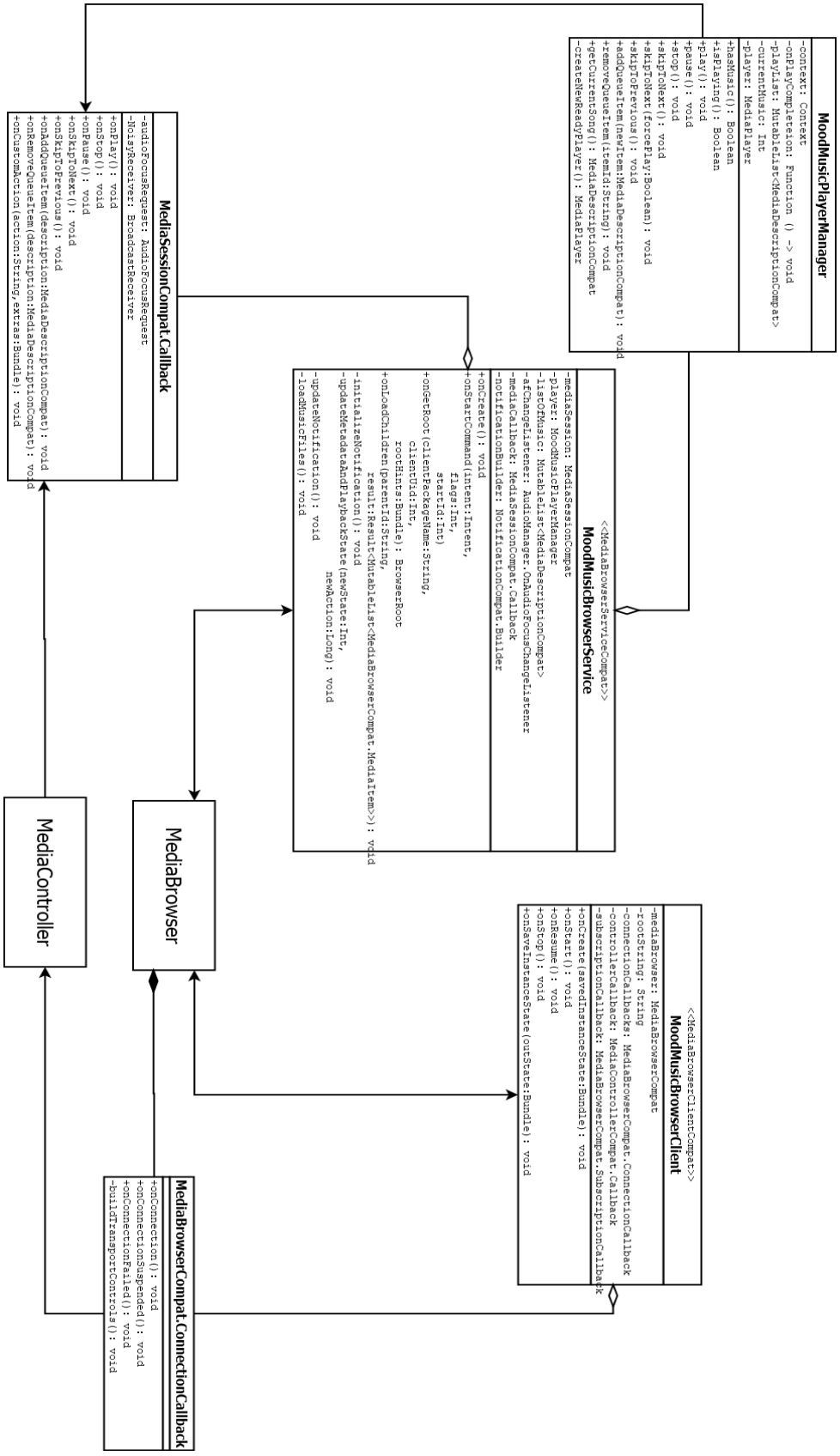
While a user can create a playlist and have the service play through the music in that playlist, there is no way to save any of these playlists. This is a feature that should be core to any music app, and I only didn't get around to it due to time restrictions. I think the proper way to go about this is by using both the service's 'onLoadChildren' method to push the saved playlists from the service to the client and the session callbacks 'onCustomAction' to save the created playlist from the client to the service.
The onLoadChildren method can return media items that are not playable but browsable. With browsable items, you can create a structure where the user can navigate into playlists to view the playable songs in that playlist.
These playlists can be edited or created by the user in the client. Once the user has completed their edits/creation then the client should notify the service using the onCustomAction method. This action should expect a playlist id and a list of music in the extra's parameter, and the service should construct and save a playlist using these two variables. The playlist should be saved to disk using internal storage in a JSON format, and should ideally only save the ids of the playlists and the songs as everything else should be gotten in the app.

The second thing that can be improved is the error checking. There are several spots in the code right now that currently assume that the best outcome always occurs. This includes the client connecting to and maintaining the connection of the mediaBrowser as well as the service requesting permission to read external storage and grabbing the music files using a content resolver. Each of these cases should be investigated individually and each case should have an appropriate response to any sort of error, either by disabling the media buttons or by alerting the user on the sort of problem that they have encountered.

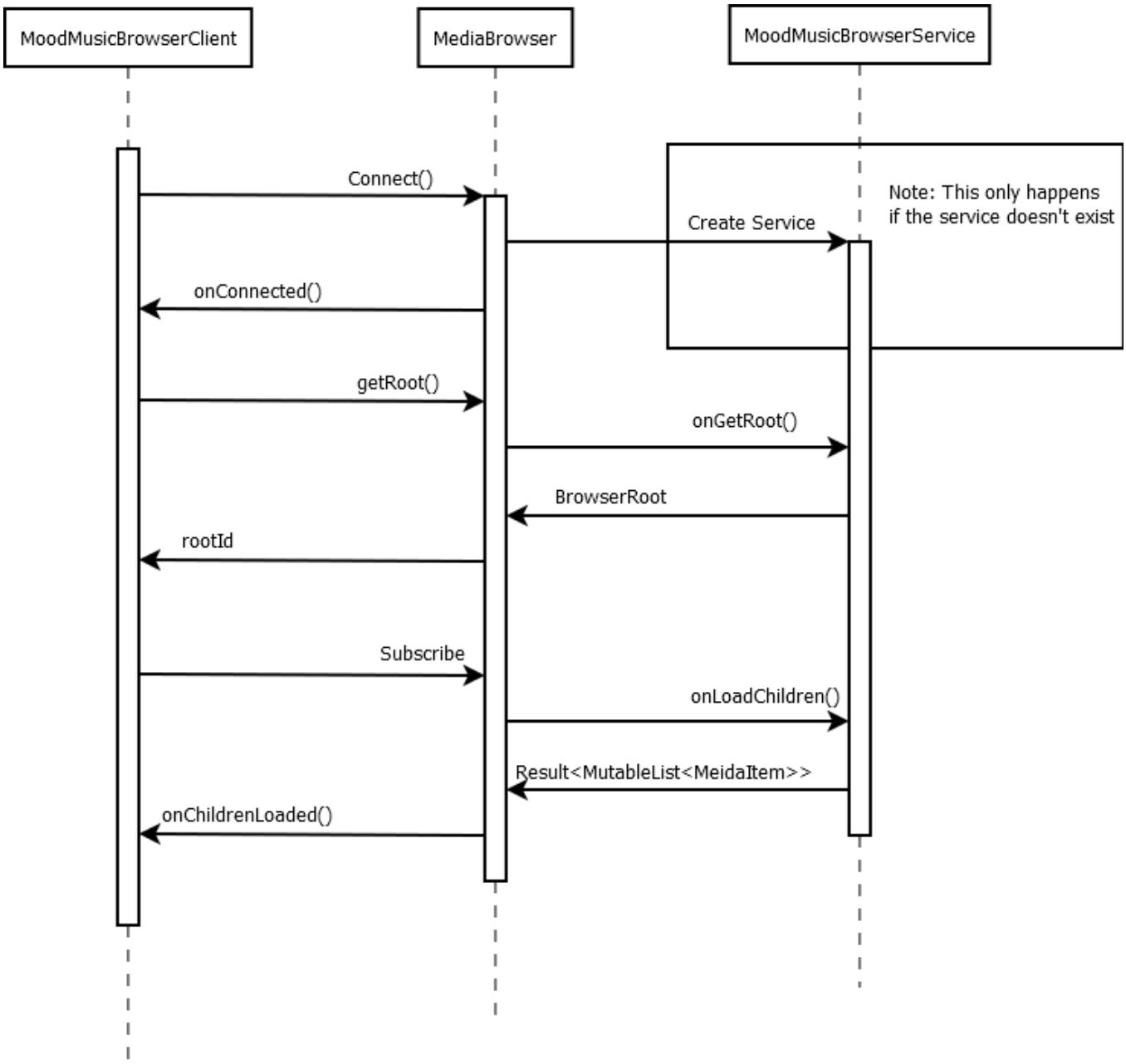
The last thing that could be improved is the media player itself. There is a potentially better fit for the media player in the ExoPlayer library. While I haven't investigated this a whole lot, the ExoPlayer has features that put it ahead of the media player, namely supporting dynamic streaming over HTTP and built in playback position finding. Something that is ultimately worth a look over.

Appendix

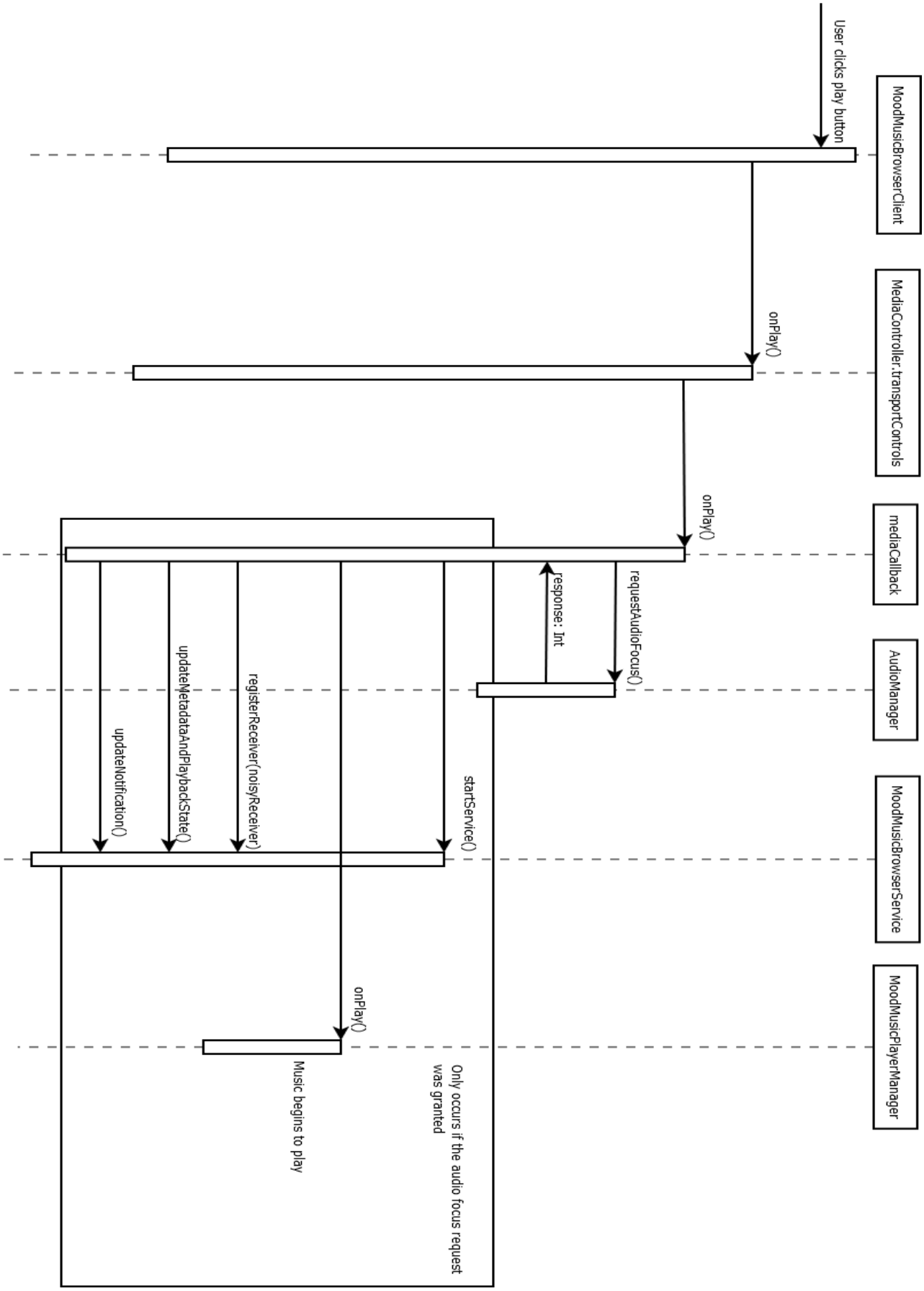
Appendix 1 - UML Diagram (can be found on github):



Appendix 2 – Client startup sequence diagram



Appendix 3 – User presses play button sequence diagram



Appendix 4: App screen and notifications