Qdio - System Design Document

Hugo Cliffordson[1], Alrik Kjellberg[1], Melker Veltman[1], & Oskar Wallgren[1]

Group 25

[1] *TDA 367*

Chalmers University of Technology

2018

Contents

1 Introduction

This is a really introduction to the system design document. Although it might be rewritten in a later stage.'

2 System architecture

The Qdio application features two separate use cases in the form of a room host and a room guest. However the logic is mostly kept the same for both the host and the guest.
The application involves at least one Android device, but there is no upper limit for the amount of devices connected to one room. The communication between devices is managed by *Google Nearby Communications*. See Dependencies.

Qdio consists of several separated modules divided by business logic and responsibility in line with the single responsibility principle. An example of this is the *information* module, which is the only place the system communicates with the Spotify information API. All other modules relying on information from Spotify gets it through this module.
Other modules including communication, playback, model, room and view, can be read about under corresponding subsections.
All the following diagrams have been generated using PlantUML**plant˙uml** source code and renderer. All the source code for the diagrams can be found in the git repository**git˙repo**

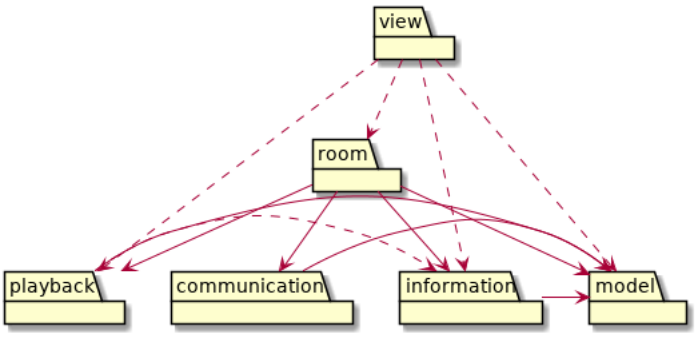

Figure 1: General package dependencies

Figure 1 describes the general dependencies between the modules of the system. Notice that there are relatively few dependencies between the strictly separated modules such as playback, communication and information. There is also nothing that depends on the view and the model is completely free from dependencies.

The view logic of the application is written using the *Model-View-ViewModel*

structure, featuring extensive usage of observables in the form of *LiveData*, see View model & LiveData.

The technical flow of creating a room starts with the owner advertising the device using *Google Nearby Connections*, after that is done the application instantiates a new headless Android fragment which sets up the connection to the Spotify playback API. After that is done the Room class is instantiated and takes control over data flow in the application. The room owner is now taken to the main screen and guests are now able to connect to the room.

LÄGG TILL DATA FLÖDE FÖR USER

2.1 External dependencies & Libraries

Several dependencies have been used throughout the project since the implementation of these parts would have been outside of the application scope.

2.1.1 ViewModel & LiveData

The Android Architecture Components Library**AndroidLifecycle** is used in the application to achieve easier usage of observables in the view logic.

2.1.2 Google Nearby Connections

Google nearby connections**nearbyconnections** is used for the communication between devices. The library uses a peer-to-peer connection method abstracted away from the technologies that are used. By using this library the application does not have to use a server component to achieve its purpose.

2.1.3 Spotify App Remote

To be able to play music on the Android device through the Spotify application, the Spotify app remote SDK **spotifyappremote** library is used. The library creates an interface with the local Spotify application on the device and abstracts several parts of the logic required otherwise.

2.1.4 Spotify web API

Spotify also exposes information about their tracks, albums and artists through a RESTful API. Qdio uses a wrapper library around the API called Spotify Web API Android**spotifywebapi** since writing a wrapper specific for the application was out of the project scope.

2.1.5 Testing Libraries

To be able to unit test the application, JUnit4**junit4** has been used in conjunction with Mockito**mockito** and PowerMock**powermock** The purpose of

Mockito in a unit testing environment is to be able to mock methods of classes outside the scope of the unit being tested. PowerMock has been used to be able to mock static methods. To test methods with sample data, the library Faker**javafaker** was used to create a test data utility.

2.2 Android application

To be able to verify the the quality of the application, STAN**stan** PMD**PMD** and FindBugs**FindBugs** has been used. STAN analyzed the source code of the application and reported no circular dependencies and verified our general package dependency diagram. PMD reported completely without errors using an already defined ruleset**PMD˙ruleset** FindBugs only reported errors about the text encoding on strings not being specified, this was interpreted more as a warning than an error.

2.2.1 Information

The information module is responsible for the fetching of information such as track, artist and playback data from Spotify.

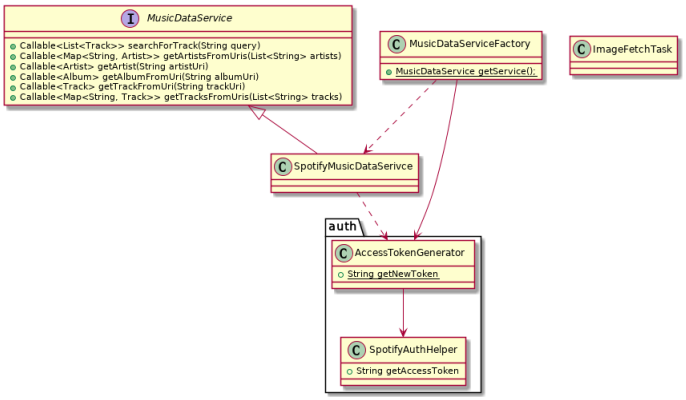

Figure 2: Information package dependencies

When music data is fetched, it is done through the Spotify web API. The API has been wrapped in the SpotifyMusicDataService class to suit the application's needs since the use of AsyncTasks is required. This is due to the calling of web requests not being permitted on the main thread. The module also contains an image fetch task which is used to acquire the album covers as bitmap. Similar to the music data service, the image fetch task also uses AsyncTask due to the use of webrequests.

The implementation of the MusicDataService interface is package-private and only reached through the Factory, in line with the dependency inversion principle and open-closed principle.

When the application uses the MusicDataService, the factory ensures that the SpotifyMusicDataService is instantiated and has a valid access token. If it does not have a valid token, it is generated from the AccessTokenGenerator class. The AccessTokenGenerator delegates the functionality of sending the web requests to get the tokens to the SpotifyAuthHelper class.

2.2.2 Communication

The communication module handles all communication with external devices when a device has either connected to a room or opened a new room.

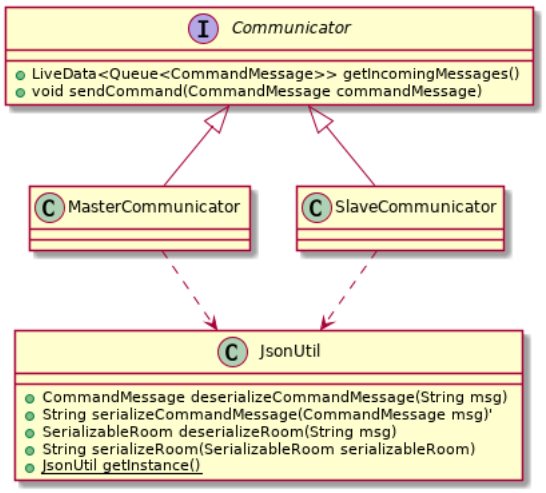

Figure 3: Communication package dependencies

It is through the Communicator interface the separate Room implementations sends and receives their commands. The Room implementations are completely independent from the Communicator implementation being used, however the MasterCommunicator is used for the MasterRoom and the SlaveCommunicator is used for the SlaveRoom.
The communicator package does not know what it is sending or receiving. This is in line with the separation of concerns principle and induces low coupling.
The JsonUtil class is used to serialize or deserialize data in JSON format.

2.2.3 Playback

The playback module takes care of the music playback and the Spotify app remote authentication.

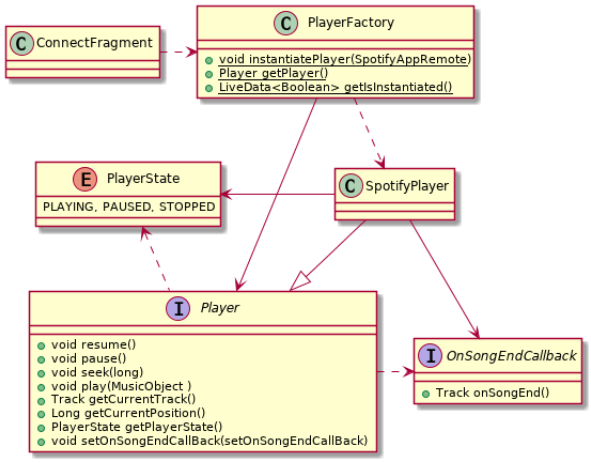

Figure 4: Playback package dependencies

When the application is told to open a new room, the ConnectFragment class is instantiated and begins the authorization with the Spotify App Remote SDK. The ConnectFragment is a *Headless Fragment* meaning that it gets attached to an Android activity but has no view at all.
The module features a Player interface which is implemented in *SpotifyPlayer* class. The implementation of the interface is using the same principles as the implementation of *MusicDataService* resulting in information and implementation hiding while maintaining open-closed principle.

OnSongEndCallback is used to attach a callback to a player. The callback is executed when a song has ended.

The *PlayerFactory* features static methods to instantiate, get and check if the Player has been instantiated. PlayerFactory is also the entry point of this module.

2.2.4 Model

The model package is where the applications domain parts are located. It consists of critical objects that define the state of the application. This package has no dependencies as described in the System Architecture section and shown in the general UML package diagram.

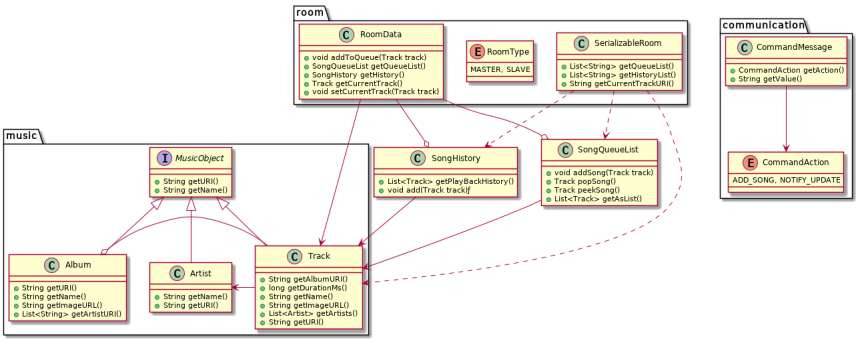

Figure 5: Model package dependencies

The application features its own implementation of a Track, Album and Artist object that differ slightly from the implementations found in both the web API and the app remote API. All these implement the MusicObject interface to simplify searching and queuing due to them all having a similar URI. There are also objects for the song queue and the history of recently played songs. Even though these classes currently lack large amounts of logic, they exist so the possibility of extension of functionality is more feasible. Furthermore featured in the model package is the CommandMessage object which is used in every message sent between devices. Whenever a message is sent, a CommandMessage is created with a CommandAction. This message contains either ADD_SONG or NOTIFY_UPDATE and its value is passed to the instantiated communicator. Some of the Room related logic is placed in the model package as well. The data used in the implementation of Room is stored by using a RoomData instance. The data may be simplified using the SerializableRoom class. The sole purpose of this class is to be converted do JSON and sent using a CommandMessage.

2.2.5 Room

The room module contains the implementation for the different types of rooms used depending on the type of user. The type of room defines restrictions and the manner in which the application communicates.

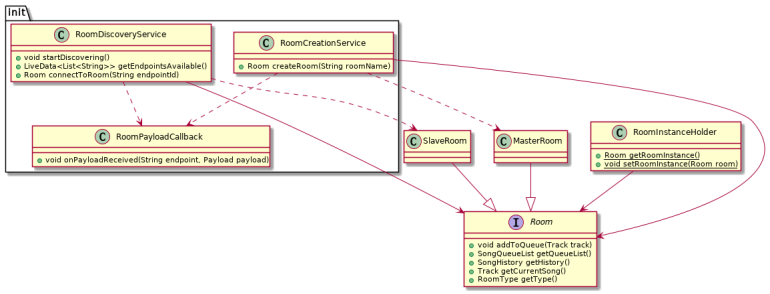

Figure 6: Room package dependencies

There are two different types of rooms where the used type depends on if the user is a host or a guest. When the application is entered as a host, the *MasterRoom* is instantiated and used throughout the application lifecycle. The MasterRoom fetches an instance of the Player and is therefore able to manipulate its current state by calling play when a song is added to an empty queue and setting the *onSongEndCallback*. The communicator in the MasterRoom is set to send all connected devices a *CommandMessage* containing the *NOTIFY_UPDATE* CommandAction and a *SerializedRoom* object with the current song, song queue, and song history in a JSON format.

If the application however is entered as a guest user, the *SlaveRoom* is instantiated and used. There is no dependency towards the Player and it is therefore not possible to manipulate. The SlaveRoom has its own representation of the current song, song queue and song history that updates when a NOTIFY_UPDATE CommandMessage is recieved. When calling the *addToQueue* method, a CommandMessage is sent using the communicator with the *ADD_SONG* CommandAction and the track URI to the host MasterRoom. Both types of room implements the Room interface which is used as a middle hand between the rooms and their communicators.
Since every instance of the application only contains one single room, there's a RoomInstanceHolder that make the current room easy to access. It contains a method that returns the current room instance or throws an exception if no room is yet instantiated.
The RoomDiscoveryService is the service that handles both discovering of open rooms for guests and connection to an open room if the guest decides to connect to it. To be able to observe on the open rooms from a ViewModel, they are exposed through a LiveData object.
The RoomCreationService handles the creation of a new room, and all the calls to the Google Nearby Connections API that is needed.

2.2.6 View

The view logic in the application is featured in the view module. Here resides all activities and fragments used throughout the application.

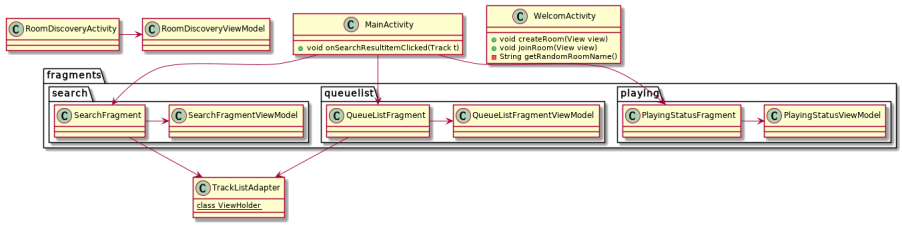

Figure 7: View package dependencies

Qdio features extensive usage of the *Model-View-ViewModel* pattern, resulting in a separation of concern in the view logic that otherwise may become highly coupled to the business logic behind it. Therefore every class except the ones that barely contain any view logic also features a viewmodel that exposes observables in the form of LiveData.
To separate the logic even further, the application uses Android fragments that are self contained view objects that may feature their own viewmodel. Search, QueueList and PlayingStatus are the fragments used in the application

3 Access control

The application Qdio has no permissions that are managed on account level. In other words there is no functionality that specifies, for instance, an admin user vs a regular user. Instead the different roles, the application is built up by the implementation of the interface Room by using the method getType. This method returns either master or slave.

The two different roles is visible to the end user only in the context of a slave being connected to a room or a master having opened a room. For the master the play/pause button and the seek bar will be visible and available for use while they are hidden for the slave. The reason for this is so that only the user playing the music has the possibility to manipulate it.