**JUKE**

****

Complete Documentation

CSCI 201 – Principles of Software Development

Prepared by Rama Gosula, Jason Roodman, Chris Cognetta, Nayha Kamboj, Katie Park, and Meghan Mehta

Table of Contents

Overall Concept 3

High-Level Requirements 4

Windows 4

User Functionality 5

Technical Specifications 6

Detailed Design 13

Testing 21

Deployment 29

Overall Concept

Juke is an android mobile application where the user is able to form a collective playlist with other users in their vicinity. The app allows a host user to open a playlist and choose songs to add and remove from the playlist. Other users are able to connect to the host user’s playlist, view all current songs in the playlist, add songs from the host user’s device to the playlist, and up vote or down vote on songs that are currently in the playlist. The number of votes on a particular song determines the song’s position in the playlist.

**High Level Requirements**

Juke should be broken into different windows, each of which will be described below. It will also have different functionality based on what kind of user is logged in – guest or registered.

**Windows**

Login Window

The Login Window will allow a user to login with their email and a password. A user can also choose to sign in as a Guest and receive limited functionality compared to registered users. The login window will also support register functionality, allowing a new user to become a registered user. To register, a user must supply a username and password. Upon logging in, the user will be brought to the Main Menu window.

Main Menu

The Main Menu will display two options: Start Playlist and Join Playlist. A user will be brought to a unique screen when each option is individually selected

Start Playlist

Start Playlist allows a current user to act as a host and manually add their own music to a new playlist. Other users are able to look up the host playlist by typing in the name of the playlist in the Join Playlist option.

Join Playlist

Join Playlist allows a user to select a host user’s playlist to contribute songs and vote on songs already on the playlist. Users will be able to locate a specific host’s playlist by manually entering a user’s email. After a host user has been selected, a Playlist Window will appear displaying the contents of the host’s current playlist.

Playlist Window

The Playlist Window will display the current songs in a host user’s playlist. Users will be able to select new songs from the host user’s music library to add to the playlist and up/down vote songs. The order of the music queue will change according to the number of up/down votes a song receives. A separate Song Addition window will be used to append new songs to the playlist. A user will be able to access this window until the host user decides to end the playlist on their device.

Song Addition

In the Song Addition window a user will be able to select one or more songs from the host user’s device. Once songs are selected, a submit button will be used to add those songs to the host user’s playlist and return the user to the Playlist Window. If the user adds a song that is already in the current playlist, instead of adding double adding the song the playlist, the song will be treated as if it had been “up voted” by that user. If a song has already been played the user is add it to the playlist again.

**User Functionality**

Guest User

A Guest User will experience significantly less functionality compared to a registered user. Namely, a Guest User will be limited to the option of Join Playlist in the Main Menu. A guest user will not be able to host a playlist, add/manage friends, or view saved playlists.

Registered User

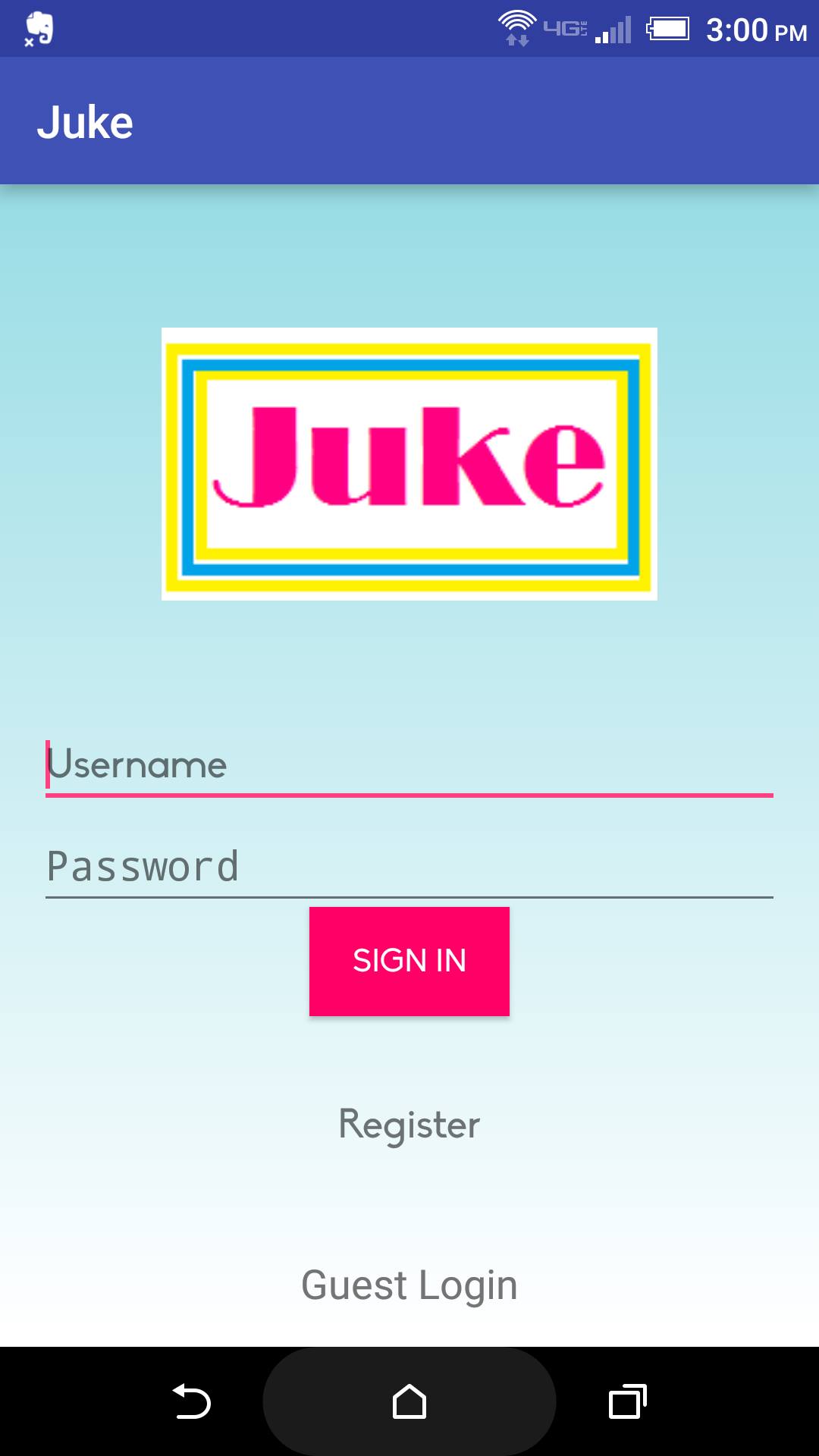
A Registered User will have access to the full functionality of Juke. A Registered User has access to every function in the Main Menu – Start Playlist and Join Playlist. When choosing to act as a playlist host, registered users will have a level of super-user control over the playlist. Namely, the registered host user will be able to skip songs, remove songs from the playlist, and pause the playlist.

Technical Specifications

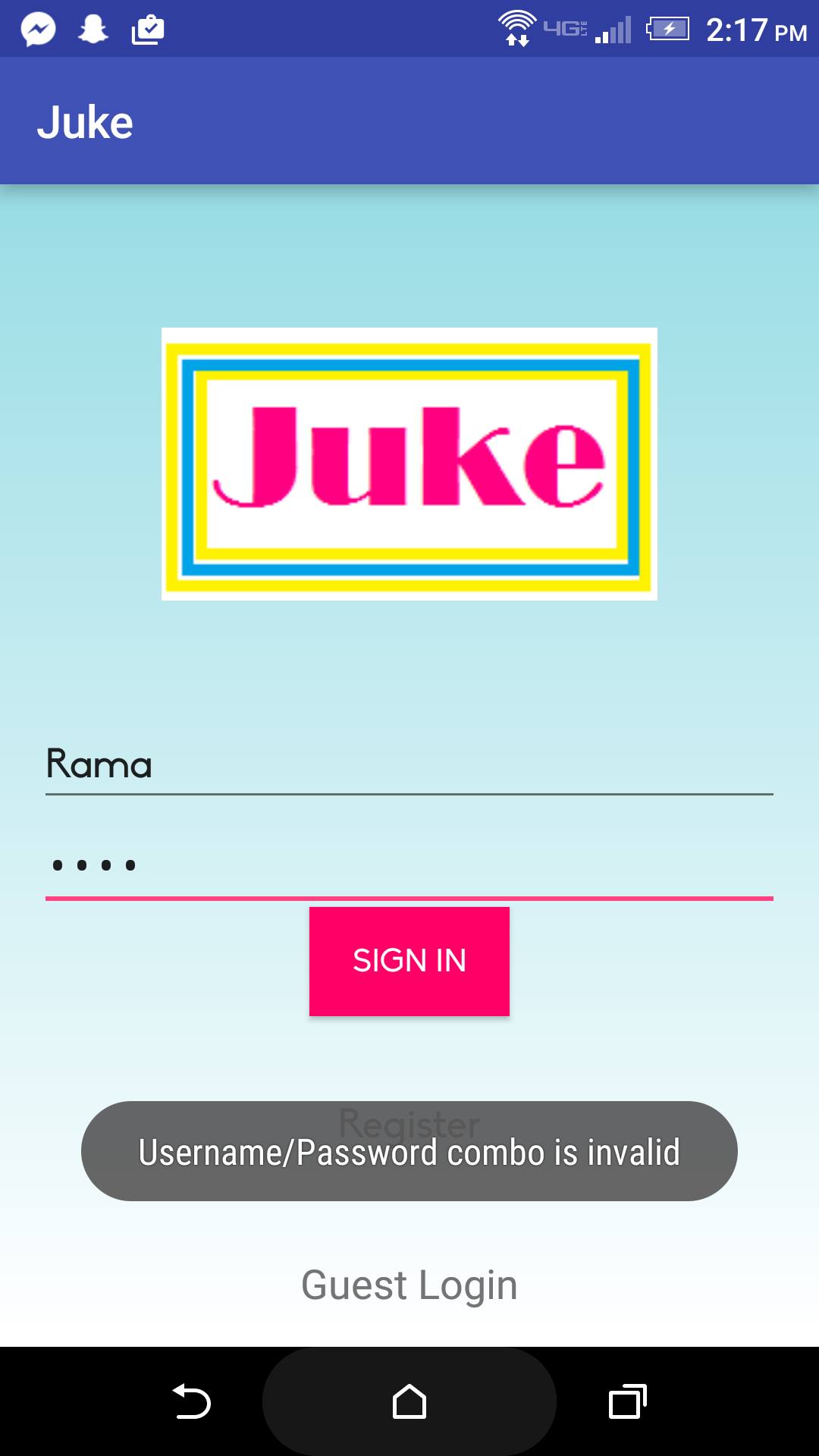
Login Window

*Main Menu Login for Registered Users*

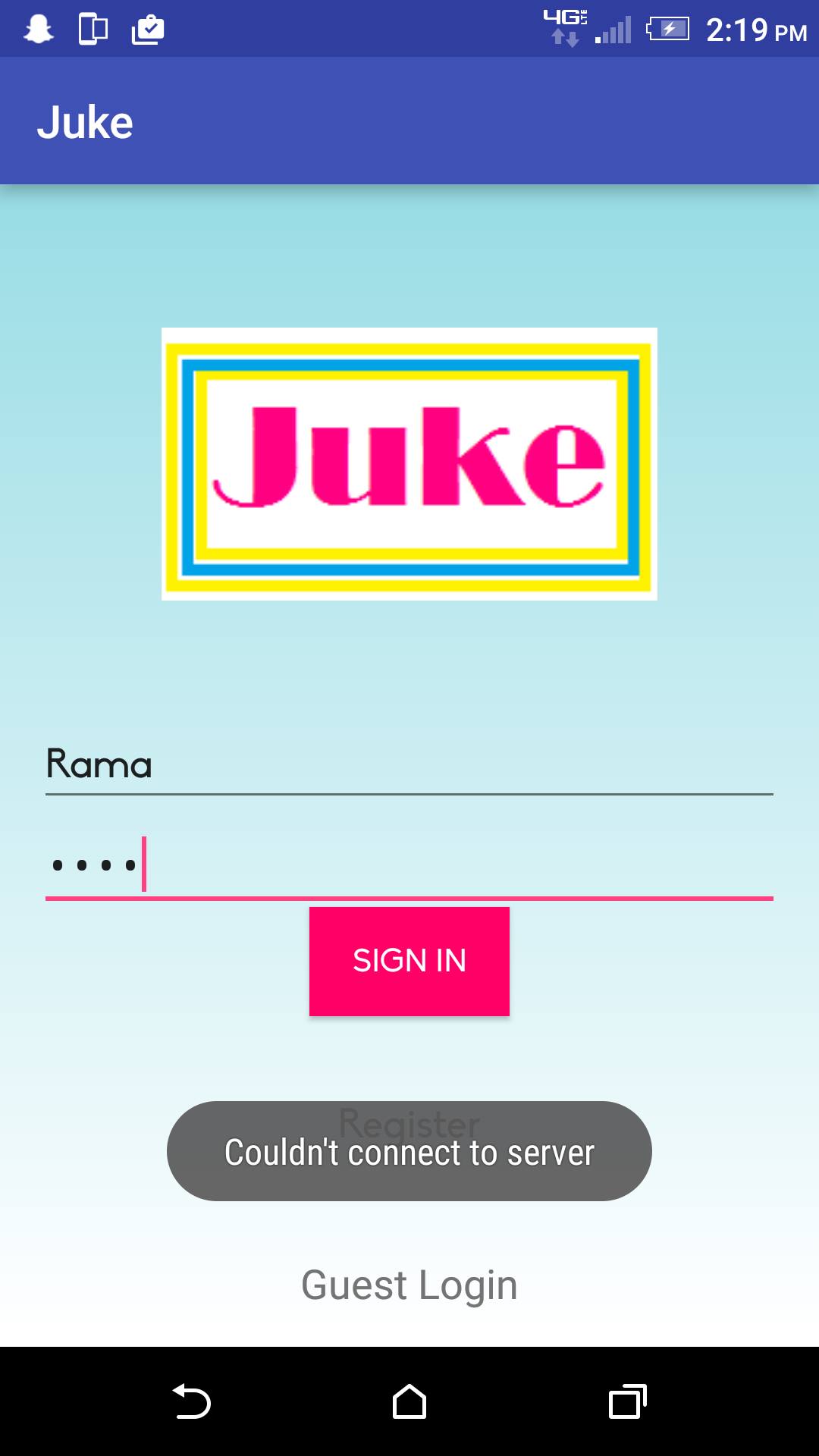
The First page that pops up when opening Juke is the main menu login for registered users. This displays a GUI that allows registered users to enter their username and password combination. Once entering this information, the database will be checked to see if the username actually is registered with Juke. Here is a sample GUI for the Main Menu:



When the sign in button is clicked if the user is not registered, Juke will display an error message saying, “Username/Password combo is invalid”. Here is a sample GUI showing the error message:

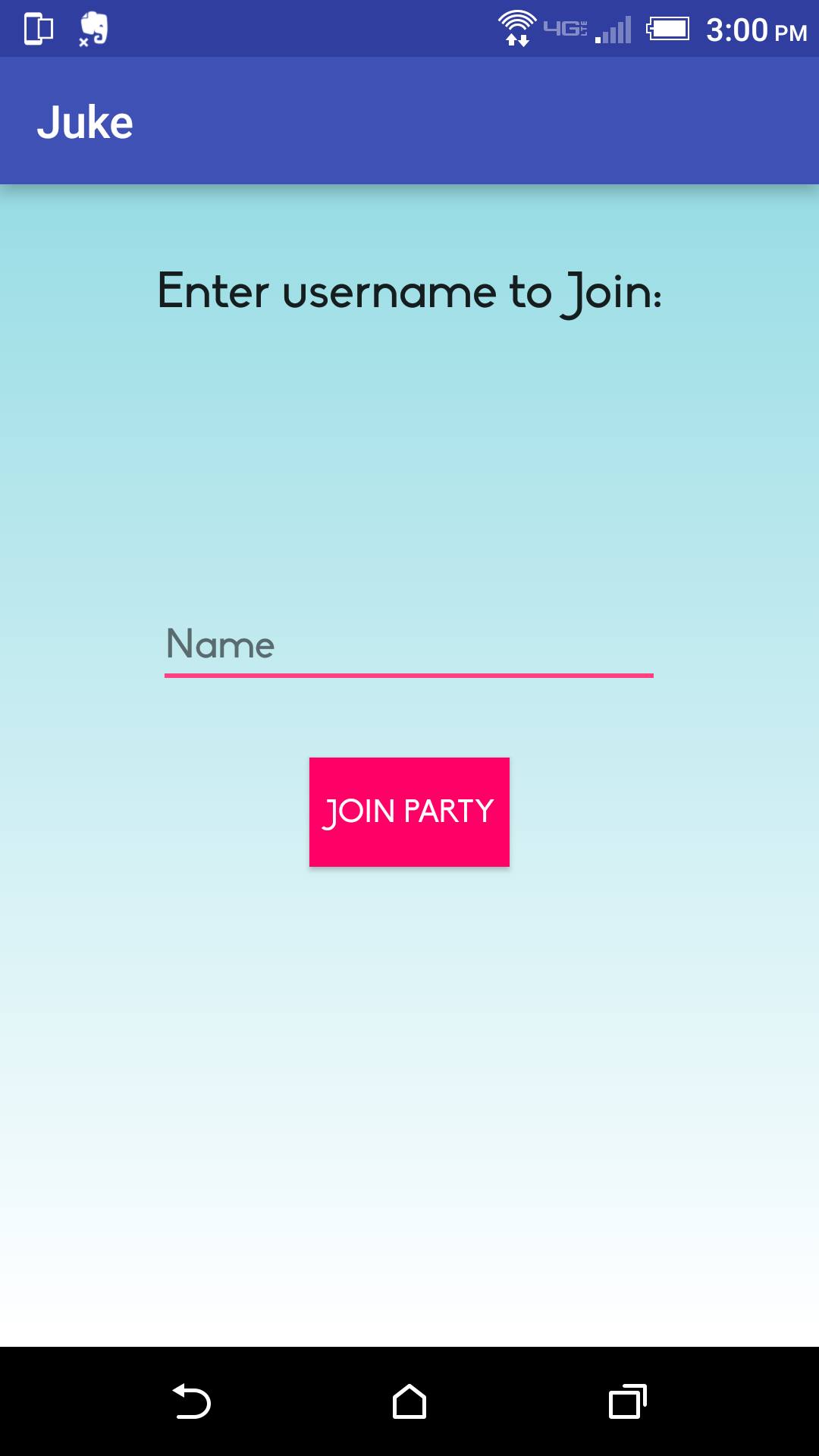


When the sign in button is clicked and the device is not able to connect to the server, Juke will display an error message saying, “Couldn’t connect to server”. Here is a sample GUI showing the error message:



*Main Menu for Guest Users*

In the main menu there is button for Guest Login at the bottom of the page in the middle. When that is clicked, it will take the user to a page that allows guest users to log in. This displays a GUI that allows guest users to simply enter a username to be able to use Juke as a guest user. This allows the guest user access to other active playlists.

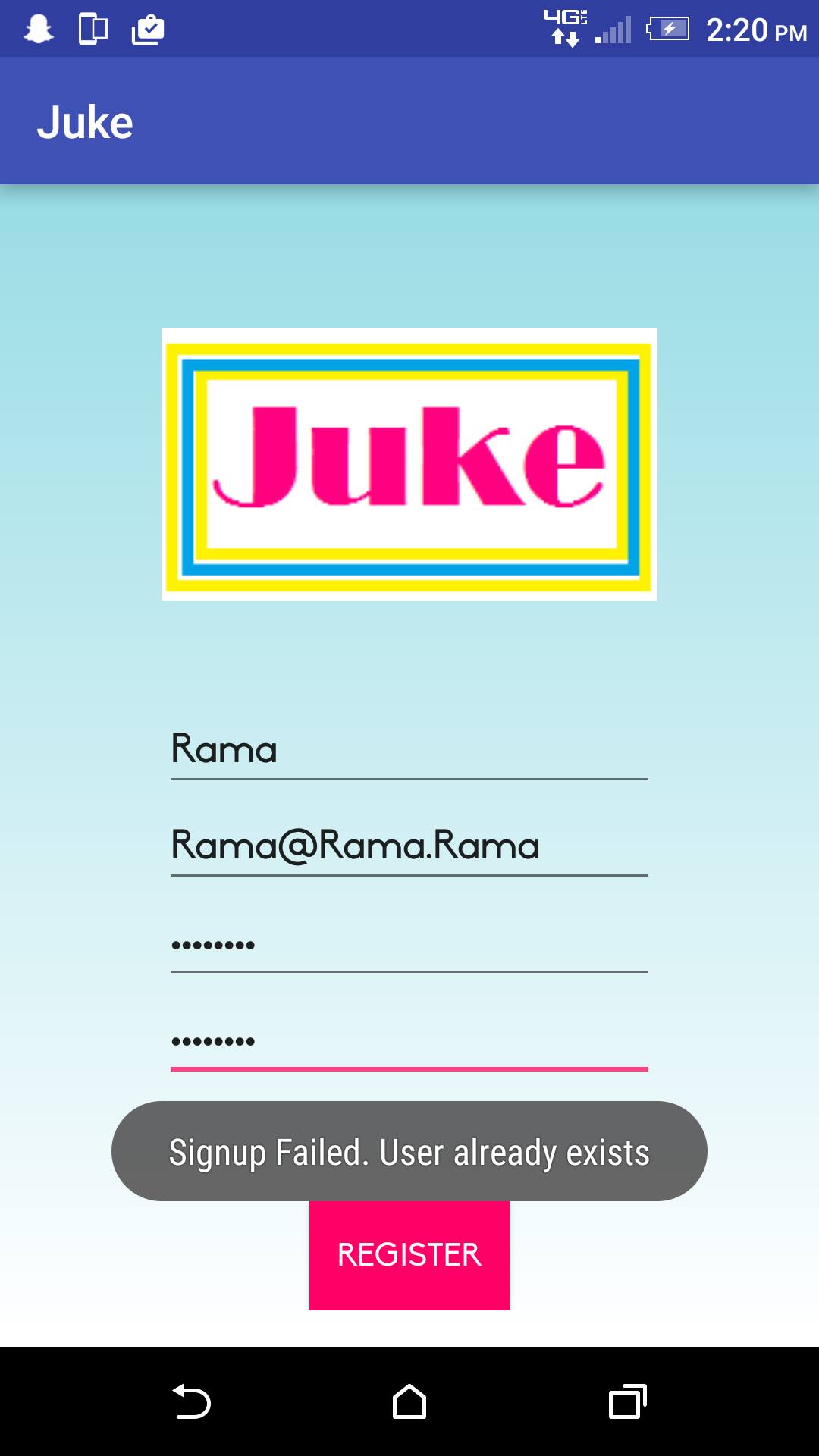


*Register page for New Users*

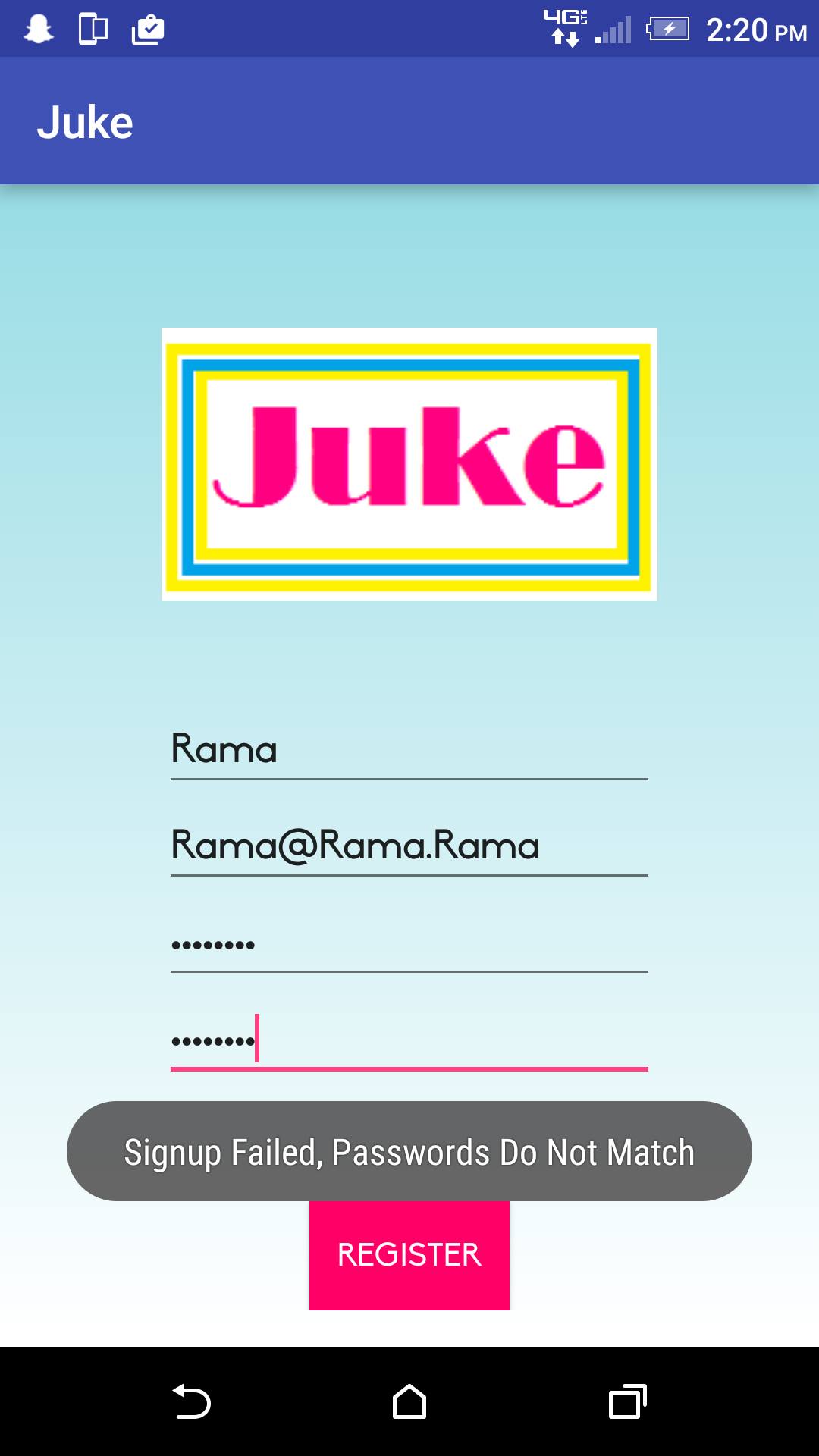
On the main menu page, there is also a button called “Register” just under the Sign In button. When this button is clicked, the user is taken to a page that allows new users to sign up for Juke. This displays a GUI that allows new users to enter a new username, email, password, and the same password once again for verification. Here is a sample GUI showing this page:



Upon pressing on the “SIGN UP” button, Juke will check the database and check if a previous user has taken the username. If the username is already taken, Juke will display an error message that says “Signup Failed. User already exists”. Here is a sample GUI showing this error:



Upon pressing on the “SIGN UP” button, there will be check to see of the two passwords entered are the same. If they are not, Juke will display an error message that says, “Signup Failed, Passwords do not match”. Here is a sample GUI showing this error:



*Main Menu Page once logged in*

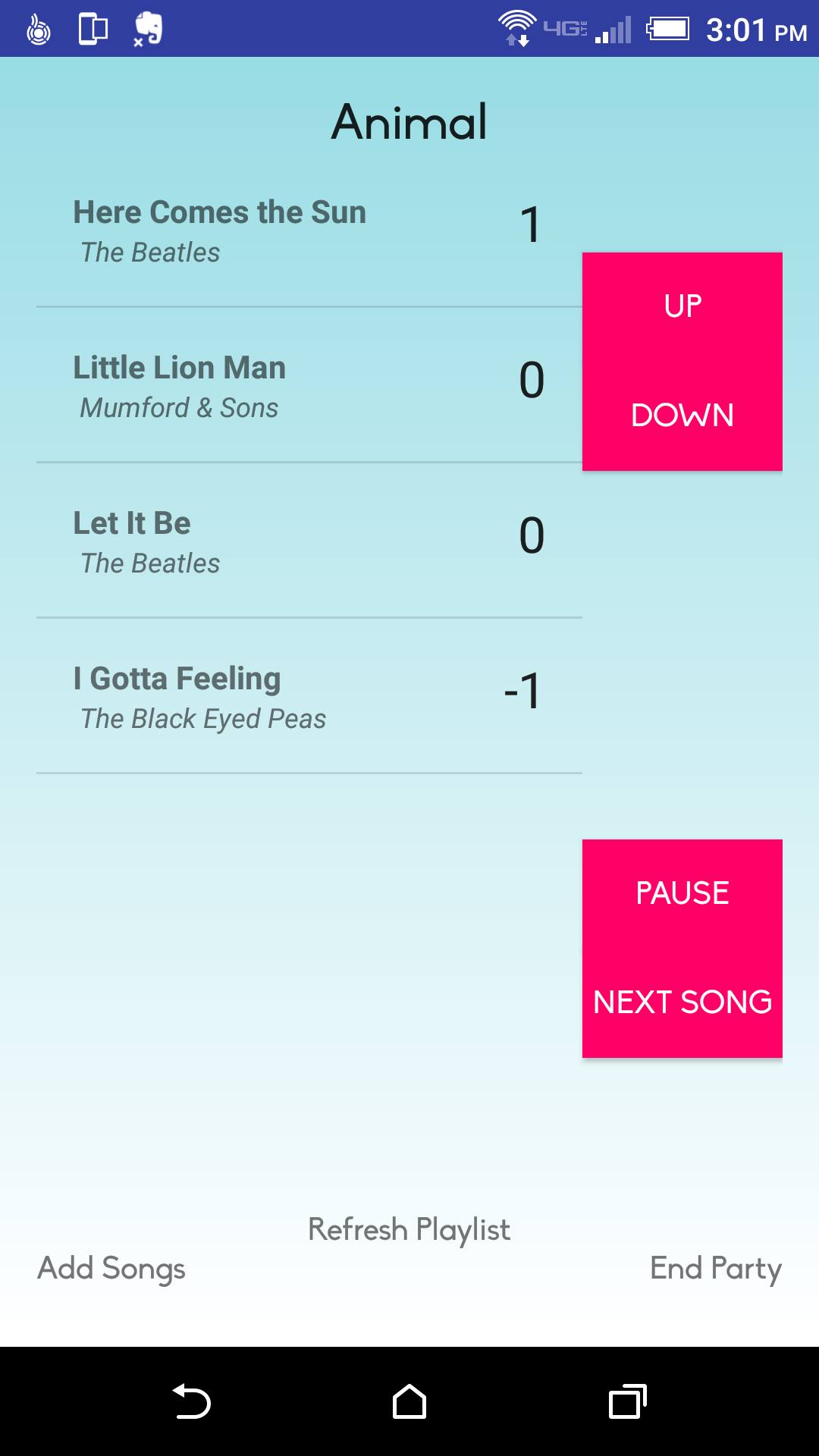
Once a registered user has logged in, the user will be taken to the main menu page. This page will display a GUI that has two buttons, “Start Party” and “Join Party”. Here is a sample GUI showing this page.



If a Registered user presses on the “Start party” button, the user will be take to a page with a single button saying “Let’s get Funky!” Upon pressing on this button, the user will create a new playlist that other users can connect to. Here is a sample GUI showing this page:

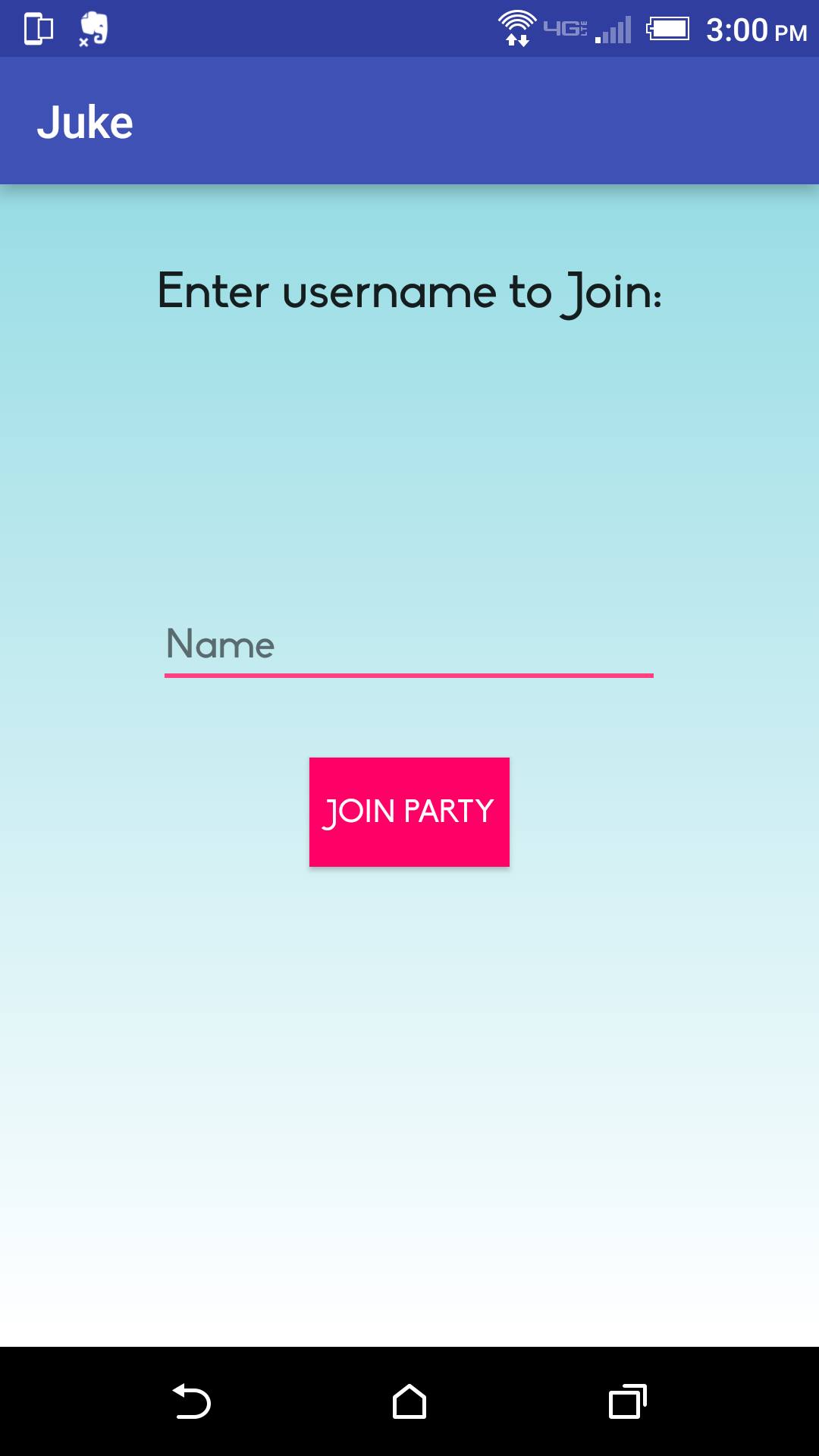


If a registered user presses on the “Let’s get Funky” button, the user has now started a new playlist that they can add and vote on songs on and is the ‘host’ of this playlist. Other users can connect to this playlist that the now host has just started. Here is a sample GUI showing this page:

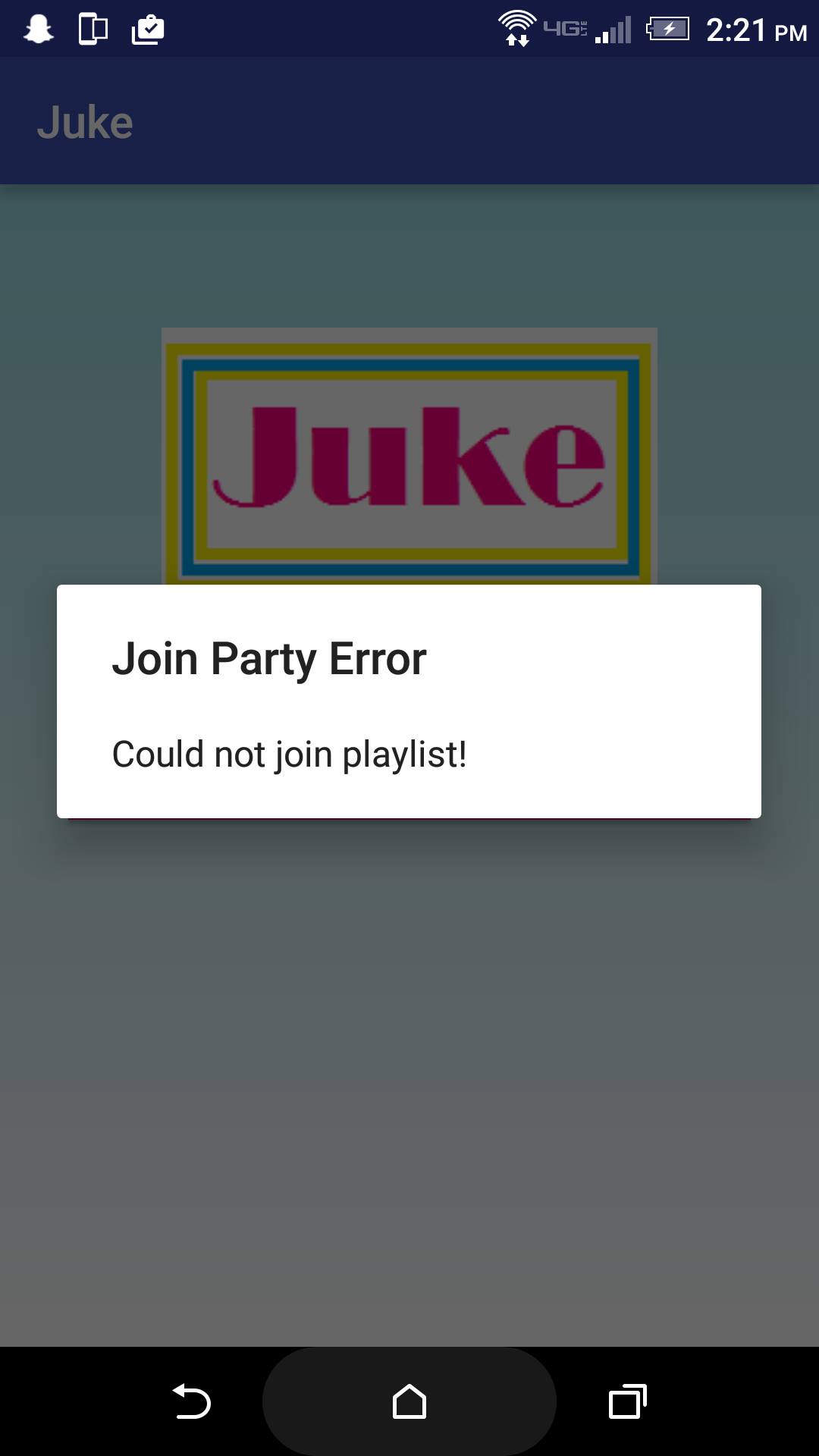


*Join Playlist*

When a user presses on the Join Playlist button, they are taken a page where they can input the username of the host of the playlist that they wish to join. This page displays a GUI that allows the user to enter the name of the host and button saying “Join Party”. Here is a sample GUI of this page:



Once the Join Part button is pressed there will be check to see if the party is active and can be connected to. If not, Juke will display an error message stating that “Could not join playlist!” Here is a sample GUI of the error:



*Main Menu for Guest Users*

The Main Menu is very different from the main menu for registered users. Once a guest user has logged in there is a single button that says Join Party. Upon clicking on Join Party, all other functions are the same for guest users as registered users. Guest users are only able to Join Playlists, not create their own.

Detailed Design

Hardware Requirements

*Android:*

6.0 Marshmallow

RAM: 2GB

Processor: ARM Cortext-A57

Software Requirements

Android Studio

Java 8

Eclipse IDE for Java EE Developers (Luna/Mars)

Server

*Class Diagram*

The Server Class connects to a port with a serversocket, and upon accepting a connection, creates a ServerThread to server it. It reads the port from a configuration file. It also manages the hosts currently running playlists on the server.

|  |
| --- |
| Server |
| -ServerSocket : ss  -int : port  -Map<String,Host> : activeHosts |
| +Server(int)  +shutDownServer() : void  +removeServerThread(ServerThread) : void  +addHost(String,String,int) : void  +removeHost(String) : void  +run() : void  +main() : void |

The ServerThread has an inputstream and output stream and loops continuously serving commands from one user. It receives ServerCommands and sends back the appropriate ServerReplies

|  |
| --- |
| ServerThread |
| -ObjectInputStream : ois  -ObjectOutputStream : oos  -Server : server  -DatabaseConnector : dc |
| +ServerThread(Socket,Server)  +run() : void  -sendMessage(Object) : void |

**Webserver**

For the webserver interactions, the client connects directly to the server through jdbc and queries the database through the container class:

|  |
| --- |
| DatabaseConnector |
| -Connection : conn |
| +DatabaseConnector()  +userExists(String) : boolean  +createUser(String,String,String) : boolean  +login(String,String) : boolean  +getFriends(String) : List<String>  +followFriend(String,String) : boolean  +playlistExists(String,String) : boolean  +createPlaylist(String,String,List<PlaylistSong>) : boolean  +copyPlaylist(String,String,String) : boolean  +removePlaylist(String,String) : boolean  +getPlaylist(String,String) : List<Song>  +getPlaylists(String) : List<String>  +getFollowing(String) : List<String>  -hashPassword(String) : String |

Client

*Class Diagrams*

The client class is a wrapper class created for convenience. It handles all interactions of the front end with the backend, stores the current user info, and playlist state if currently hosting a playlist.

|  |
| --- |
| Client |
| -String : username  -ServerConnector : sc  -CommandThread : commandThread  -boolean : isHost  -HostServer : hs  -Playlist : playlist |
| +getAvailableSongs() : List<Song>  +refreshPlaylist() : Playlist  +upvoteSong(Song) : boolean  +downvoteSong(Song) : boolean  +removeSong(Song) : boolean  +addSong(Song) : boolean  +advanceSong() : Song  +joinPlaylist(String) : boolean  +exitPlaylist() : boolean  +stopPlaylist() : boolean  +startPlaylist(String,int,List<Song>) : boolean  +getPlaylist(String,String) : List<Song>  +getPlaylists(String) : List<String>  +removePlaylist(String) : boolean  +followUser(String) : boolean  +getFollowing() : List<String>  +createPlaylist(String,Playlist) : boolean  +copyPlaylist(String,String) : boolean  +isHost() : boolean  +getUsername() : String |

The ServerConnector connects to the webserver and sends and receives data. It is normally accessed through the Client class, with the exception of the login method which is called directly.

|  |
| --- |
| ServerConnector |
| -Socket : server  -ObjectInputStream : ois  -ObjectOutputStream : oos |
| +ServerConnector(String,String) > IOException  +Login(String,String) : boolean  +addHost(String,String) : boolean  +connect(String) : Host  +removeHost(String) : boolean  +getPlaylist(String,String) : List<Song>  +getPlaylists(String) : List<String>  +followUser(String,String) : boolean  +getFollowing(String) : List<String>  +createPlaylist(String,String,Playlist) : boolean  +copyPlaylist(String,String,String) : boolean  -sendMessage(Object) : void |

The running playlist is stored in a playlist object. The playlist object contains all the state information about the playlist and has limited function to perform all the actions such as sorting, upvoting, and advancing the song that the host would need.

|  |
| --- |
| Playlist |
| -List<PlaylistSong> : allSongs  -List<PlaylistSong> : playedSongs  -List<PlaylistSong> : toPlay  -PlaylistSong : current |
| +Playlist()  +addSong(Song) : boolean  +deleteSong(Song) : boolean  +upvote(Song) : boolean  +downvote(Song) : boolean  +sortToPlay() : void  +advanceSong() : Song  +songIndex(Song) : int  +getAllSongs() : List<PlaylistSong>  +getPlayedSongs() : List<PlaylistSong>  +getToPlay() : List<PlaylistSong>  +getCurrentSong() : PlaylistSong |

Playlist songs are objects to represent a song within a playlist. It stores data about which song tt represents, the score of the song and whether or not it has been played.

|  |
| --- |
| PlaylistSong |
| +long : servialVersionUID  -Song : song  -Playlist : owner  -int : voteScore  -boolean : played |
| +PlaylistSong(Song)  +PlaylistSong(PlaylistSong)  +upvote() : void  +downvote() : void  +setVoteScore(int) : void  +setOwner(Playlist) : void  +setPlayed(boolean) : void  +getSong() : Song  +getScore() : int  +getOwner() : Playlist  +hasPlayed() : boolean  +toString() : String |
|  |

A Song stores a song name, artist and uri. It represents a song.

|  |
| --- |
| Song |
| +long : servialVersionUID  -String : uri  -String : songName  -String : songArtist |
| +Song(String,String,String)  +equals(Song) : boolean  +getURI() : String  +getSongName() : String  +getSongArtist() : String |

Songs are stored in a wrapper object:

The command Thread is a thread that the client invokes to send commands to the host. It stores a queue of commands to send, and on its run method sends them all.

|  |
| --- |
| CommandThread |
| -int : TIMEOUTATTEMPTS  -int : CONNECTIONSLEEP  -ConcurrentLinkedQueue<HostCommand> : commands  -Socket : socket  -String : hostname  -int : port  -ObjectInputStream : ois  -ObjectOutputStream : oos  -List<Song> cachedSongs |
| +CommandThread(Socket,String,int) > IOException  +addCommand(HostCommand) : void  +executeImmediate(ImmediateCommand) : ServerReply  +executeCommand(HostCommand) : ServerReply  +exitPlaylist() : void  +run() : void  -reconnect() : void |

Host

*Class Diagram*

The HostServer is created to serve clients that want to connect to the host’s playlist.

It creates a serversocket that upon accepting a client connection, creates a receiverthread .

|  |
| --- |
| HostServer |
| -UpdateThread : updater  -ServerSocket : ss  -int : port  -List<Song> : availableSongs |
| +HostServer(int,UpdateThread,List<Song>)  +stopHost() : void  +run() : void  -readConfigurationFile() : void |

On run(), ReceiverThread has while(true) loop that reads ServerCommands or ImmediateCommands.If it receives an ImmediateCommand, it will read which type it is and execute it immediately, sending the data back to the client in a ServerReply.

If it recieves a ServerComand, it passes it to the UpdateThread through addCommand();

|  |
| --- |
| ReceiverThread |
| -UpdateThread : updater  -ObjectInputStream : ois  -ObjectOutputStream : oos  -Socket : socket  -List<Song> : availableSongs |
| +run() : void |

These threads only run on the Host or Jukebox machine. They run during the playlists existence

The Update thread receives commands. On run, it has an endless loop that executes all of the ServerCommands in the Queue on toPlayBuffer, sends the proper ServerReply, and then sorts the playlists’s ToPlayBuffer list (since PlaylistSongs will have changed score, or have been added or deleted), and copies it to toPlay;

While executing the commands in the queue, updating is set to true, to prevent access of the playlist in mid-operation.

|  |
| --- |
| UpdateThread |
| -ConcurrentLinkedQueue<HostCommand> : commands  -Playlist : playlist  -boolean : running |
| +UpdateThread(Playlist)  +addCommand(HostCommand) : void  +stopUpdating() : void  +getRefreshedPlaylist() : Playlist  +run() : void  -sendMessage(Object,ObjectOutputStream) : void |

Database:

The database consists of 4 tables:

User

|  |  |  |  |
| --- | --- | --- | --- |
| userID | username | email | hash |
| Auto generated, primary key, not null | Not null, unique | Not null | Not null  \*hashed password\* |

Friend

|  |  |  |
| --- | --- | --- |
| friendshipID | followerID | followedID |
| Auto generated, primary key, not null | fk(User.userID),  \*represents which user is following\* | Not null, fk(User.userID),  \*represents which user is being followed\* |

Note, the friendship feature works more like twitter’s follow system.

Playlist

|  |  |  |  |
| --- | --- | --- | --- |
| playlistID | userID | playlistname | filepath |
| Auto generated,Primary key, not null | fk(User.userID),  \*represents owner of playlist\* | Notnull, \*playlist name\* | Not null \* path where the playlist is stored on the server’s file system\* |

PlaylistContent

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| playlistSongID | playlistID | songIndex | songName | songArtist |
| Auto generated,Primary key, not null | fk(Playlist.playlistID),  \*represents specific playlist\* | Not null  \* Song’s position in the playlist \* | Not null  \* Song name\* | Not null  \* Artist name\* |

Testing

Test - Host(jukebox)-Client Connection

* Start Server
  + Steps to Run Test:
    - Run the function to start the server
    - Try to connect to the server
  + Expected Result
    - Different threads can connect to the server
* UpdateThread updates playlist
  + Steps to Run Test:
    - Create a Playlist
    - Create an UpdateThread
    - Populate it with ad-hoc ServerCommands
    - Run the thread
  + Expected Result:
    - The ad hoc ServerCommands are all executed on the playlist associated with the update thread
* UpdateThread gives proper replies
  + Steps to Run Test:
    - Create a Playlist
    - Create an UpdateThread
    - Create three different ad-hoc Threads that access UpdateThreads
    - Call addCommand() multiple times from each UpdateThread, with many different ServerCommands.
  + Expected Result
    - Each ServerCommand receives a reply
    - Each reply corresponds to the ServerCommand that was sent
    - Each reply is given to the correct thread.
    - The playlist is updated by each servercommand correctly
    - Each reply is received in the proper order
* ReceiverThread properly handles commands
  + Steps to Run Test:
    - Create a Playlist
    - Create an updateThread
    - Create a receiver Thread
    - Create mulitple Ad hoc threads that have both the receiver thread’s inputstream and updateThread’s outputstream.
    - Send ServerCommands and ImmediateCommands to the receiver thread via stream
  + Expected Result:
    - Each ServerCommand Receives a Reply
    - ImmediateCommands Receive a Reply
    - Each Reply corresponds to the servercommand sent
    - Each Reply is sent to the proper thread
    - Each Reply is received in the proper order
    - The playlist is updated correctly by each serverCommand
* \*Client’s CommandThread sends commands and receives replies
  + Steps to run test:
    - Connect client and host via socket
    - Create Command Thread
    - Create Playlist
    - Create Update Thread
    - Create Receiver Thread
    - Call AddCommand on updatethread
  + Expected Result:
    - The command is received by the receiver thread
    - The command is either executed by the receiver or passed to the update thread
    - If the command is passed to the update thread it is executed properly
    - The Command Thread receives a proper reply
* Client can send multiple commands
  + Steps to Run Test:
    - \*See Above\*
    - Instead of adding one command, add multiple
  + Expected Result:
    - Each command is executed correctly
    - Each receives a reply
    - The reply fits the command
    - Each receives a reply in the proper order
* Multiple Clients can send commands
  + Steps to Run Test:
    - \*See Above\*
    - Instead of one Command thread, create many
    - Send addCommand() multiple times from each
  + Expected Result:
    - Each command is executed
    - Each command is executed correctly (though maybe not in the exact same order sent)
    - Each command receives the proper reply
    - Each thread receives replies in the order they sent commands

Unit Test - Playlist Class

* Add song to playlist
  + Steps to Run Test:
    - Create playlist object
    - Call addSong() on playlist
  + Expected Result:
    - That song is added to the playlist
* Upvote/Downvote Song
  + Steps to Run Test:
    - Create Playlist Object
    - Add 3 different songs to the playlist
    - Upvote the third song, then downvote the second
  + Expected Result:
    - The third song is upvoted, second is downvoted, first is unchanged.
* Add song already in playlist
  + Steps to run test:
    - Create a playlist object
    - Add multiple songs
    - Add one of the songs a second time
  + Expected result:
    - The song that was added twice has an upvote
* Upvote a song not in the playlist
  + Steps to run Test:
    - Create and add songs to playlist object
    - Upvote, but pass in a song not in the playlist
  + Expected result:
    - No change
* Sorting the to-play list
  + Steps to run test:
    - Create playlist and add songs
    - Upvote and downvote songs so that there is a range of positive and negative values. Make sure there are duplicate values, and multiple unvoted songs.
    - Call sortToPlay()
  + Expected Result:
    - The to-play list is sorted in descending order. Any songs with the same number of votes are in the same relative order as before
* Advance Song
  + Steps to Run Test:
    - Create playlist
    - Add songs
    - upvote/downvote different songs
    - Sort playlist
    - Call advanceSong()
  + Expected Result:
    - The song will be appended to ‘played’
    - The current song is the highest song on the to-play list
    - The song is removed from the to-play list
    - The correct song was returned;
* Advance song empty
  + Steps to run test:
    - Create playlist, but don’t add any songs
    - Call advanceSong()
  + Expected Result
    - No change, return null

Test - Login

* Login from existing user with correct username and password
  + Steps to Run Test:
    - Enter a username/password correctly from the database
  + Expected Result:
    - Normal login, with screen transitioning to the Main Menu
* Login from non-existing user with a username/password combo not present in the database
  + Steps to Run Test:
    - Enter an incorrect username/password
  + Expected Results:
    - Error message visible to user, detailing that either password or username is incorrect
    - User is able to re-enter password
* Login with correct username but incorrect password
  + Steps to Run Test:
    - Enter a correct username from the database but incorrect password
  + Expected Results:
    - Error message visible to user, detailing that either password or username is incorrect
    - User is able to re-enter password
* Login with incorrect username but correct password
  + Steps to Run Test:
    - Enter an incorrect username and a correct password from the database
  + Expected Results:
    - Error message visible to user, detailing that either password or username is incorrect
    - User is able to re-enter password
* SQL code entered as part of username/password
  + Steps to Run Test:
    - Enter “Drop Database” or another ‘sneaky’ command
  + Expected Results:
    - Error message visible to user, detailing that either password or username is incorrect
    - User is able to re-enter password

Test - Sign Up

* Sign Up with valid email, username, and password
  + Steps to Run Test:
    - Enter a valid email, username, and password (2x)
  + Expected Results:
    - Confirmation email sent to user
    - User brought to correct “waiting for confirmation” page
* Sign Up with invalid email, but valid username and password
  + Steps to Run Test:
    - Enter an invalid email, valid username, and valid password
  + Expected Results:
    - Error message visible to user, detailing that some part of information was incorrect
    - User is able to re-enter information
* Sign Up with invalid username, but valid email and password
  + Steps to Run Test:
    - Enter an invalid username, valid email, and valid password
  + Expected Results:
    - Error message visible to user, detailing that some part of information was incorrect
    - User is able to re-enter information
* Sign Up with invalid password, but valid email and username
  + Steps to Run Test:
    - Enter an invalid password, valid email, and valid username
  + Expected Results:
    - Error message visible to user, detailing that some part of information was incorrect
    - User is able to re-enter information
* Sign Up with missing information (user doesn’t enter information into all fields)
  + Steps to Run Test:
    - Enter partial information into Sign Up
  + Expected Results:
    - Error message detailing that all information was not provided
    - User is able to re-enter information
* Sign Up with non-matching password and password confirmation fields
  + Steps to Run Test:
    - Enter valid information but differing passwords into the password and password confirmation fields
  + Expected Results:
    - Error message detailing that passwords differed
    - User is able to re-enter information
* SQL code entered as part of new user info
  + Steps to Run Test:
    - Enter “Drop Database” or another ‘sneaky’ command
  + Expected Results:
    - Error message visible to user, detailing that either password or username is incorrect
    - User is able to re-enter information

Test - Start Party

* User attempts to Start Party with missing information
  + Steps to Run Test:
    - Enter partial information
  + Expected Results:
    - Error message visible to user detailing that information is
* User attempts to Start Party with correct network connection
  + Steps to Run Test:
    - Click Start Party while connected to the internet
  + Expected Results:
    - Correct transition to Playlist Main Menu
* User attempts to Start Party with incorrect network connection
  + Steps to Run Test:
    - Click Start Party while not connected to the internet
  + Expected Results:
    - Error message detailing that there is no network connection
    - User remains on same screen
* User enters invalid password
  + Steps to Run Test:
    - Enter invalid password (all other information correct)
  + Expected Results:
    - Error message detailing invalid password
    - User is able to re-enter information
* SQL code entered as part of playlist info
  + Steps to Run Test:
    - Enter “Drop Database” or another ‘sneaky’ command
  + Expected Results:
    - Error message visible to user, detailing that either password or username is incorrect
    - User is able to re-enter information

Test - Playlist Generating

* User wants to generate the current playlist
  + Steps to Run Test:
    - Update the playlist by adding new songs
  + Expected Results:
    - When the refresh button is pressed, the list will be generated sorted by upvotes
* User loses connection
  + Steps to Run Test:
    - Turn off wifi with the playlist open on the phone
  + Expected Results:
    - When the refresh button is pressed, a pop up will show indicating lack of network connectivity

Test - Join Party

* User is able to successfully join a party
  + Steps to run test:
    - Create a playlist
    - create a new user/login
    - Try to join a party/playlist
  + Expected Results:
    - new user is able to join playlist for a party
* User attempts to join a party that no longer exists
  + Steps to run test:
    - User tries to join a party that has just been deleted
  + Expected Results:
    - An error message pops up letting the user know that the party no longer exists
* User attempts to join a party (is unsuccessful)
  + Steps to run test:
    - Create a Party
    - Create an account, user attempts to join a party
  + Expected Results:
    - Error message telling the user that they were unsuccessful in joining the party

Test - My Playlists/Parties

* User has host control of playlist
  + steps to run test:
    - Create a Party
    - Check that the user can add/remove songs in addition to voting
  + Expected Results:
    - host can remove songs whereas guest/non-hosts can only add and vote on songs
* User tries to create a playlist as a guest
  + steps to run test:
    - Go on the app as a guest
    - Try to create a playlist
  + Expected Results:
    - User is not able to create a playlist without a username/password/account

Deployment

To deploy this application on an android device, download the app from android studio. This should create a copy of the application called “Juke” on the android device.

To execute Juke, click on the application icon, and the application will run.