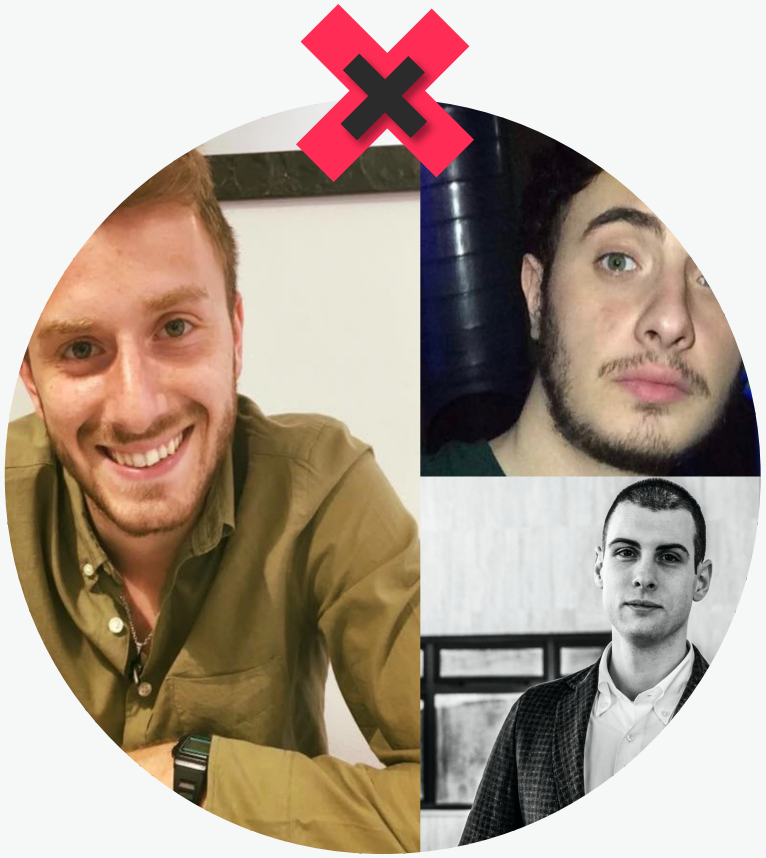
Hoodie
Music Downloader

SAPIENZA
UNIVERSITÀ DI ROMA



-[Describing our Team]-

Our Team

Luca Tomei	1759275
Danilo Marzilli	1878501
Giovanni Trinca	1542534

OUTLINE OF TALK

- Introduction
- Structure
- Design
- Look and Feel
- Conclusion
- Future Works

The app is designed to easily download and listen to songs. No matter if you are with or without internet connection: this is the right choice for you.

Hoodie is the best way to listen to music!

- Search and stream any song without paying a subscription
- Download songs you love for quick access later
- Create unlimited library, reorder and manage your music
- Listen music in portrait and horizontal mode

You can also listen to trending music by genre by selecting one of our favorite charts. Charts is a new, easy way to stay plugged in to what's trending in the world of sound. With unique lists for all the genres, this feature is your destination for the latest and greatest out there.

Introduction.

MAIN FEATURES

-[AN AGILE PROCESS]-

Our Approach

In order to develop the application correctly we decided to use a SCRUM approach, proceeding by sprints.

Each of us developed some small functionalities coming from the user stories realized at the beginning. The sprint was about 2 weeks.

Used Technologies

Backend

Node.js

Asynchronous event-driven Javascript runtime.
We use Node.js to download songs directly from Deezer.

Firebase

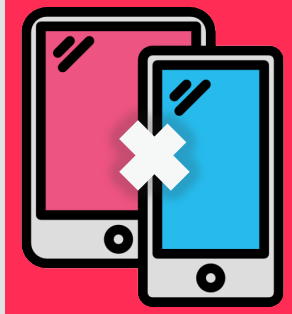
Backend as a Service (Baas).
We store our users, their searches, all the stuff needed by our app.

Frontend

Swift

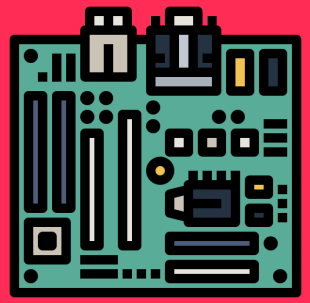
Powerful and intuitive programming language for macOS, iOS, watchOS and tvOS that support both object-oriented and functional programming.

Application Architecture



Mobile Application

The mobile application developed in Swift allows communicationwith the server.



Raspberry Pi

Here is where our backend lies, it handles users and their permissions, together with music download.

Deezer Api

Deezer Simple API provides a nice set of services to build up applications allowing the discovery of music catalogue.

Backend Structure

Firebase Login
Google Signin
Facebook API

We have managed the authentication features with Firebase Authentication, Google OAuth 2 and Facebook API.

Raspberry Pi

Raspberry hosts our server. Through the "express" framework, with the listen method, the application listens on port 8080.

Firebase Realtime Database

This is used to carry on some important data of the user to provide a better UX.

FIREBASE AUTHENTICATION

```swift
import FirebaseAuth

func signUp(email: String, password: String, completionBlock: @escaping
(_ success: Bool) -> Void) {
    Auth.auth().createUser(withEmail: email, password: password)
{(authResult, error) in
        if let user = authResult?.user {
            print(user)
            completionBlock(true)
        } else {
            completionBlock(false)
        }
    }
}
```

```swift
func signIn(email: String, password: String, completion: @escaping (_
error: Error?) -> Void) {
    Auth.auth().signIn(withEmail: email, password: password)
{ authResult, error in
        completion(error)
    }
}
```

```swift
func sendPasswordReset(withEmail email: String, _ callback: ((Error?) -> ())? = nil){
    Auth.auth().sendPasswordReset(withEmail: email) { error in
        callback?(error )
    }
}
```

Structure.

- We use Firebase Authentication to allow users to log into our app using their personal email and password.
- These credentials will be passed to the Firebase Authentication SDK and after getting a response from their servers, the user's data will be saved to Firebase Realtime Database.

GOOGLE AUTHENTICATION

```swift
import GoogleSignIn

func GoogleSign(_ signIn: GIDSignIn!, didSignInFor user: GIDGoogleUser!,
          withError error: Error!) {
    if let error = error {
        if (error as NSError).code == GIDSignInErrorCode.hasNoAuthInKeychain.rawValue {
          print("The user has not signed in before or they have since signed out.")
        } else {
          print("\(error.localizedDescription)")
        }
        return
      }

    // Save User Data
    let userId = user.userID
    let idToken = user.authentication.idToken
    let fullName = user.profile.name
    let givenName = user.profile.givenName
    let familyName = user.profile.familyName
    let email = user.profile.email

    // Store user data ...
    //...
}
```

```swift
func GoogleLogout(){
    do {try GIDSignIn.sharedInstance()?.signOut()}
    catch { print("already logged out") }
}
```

- Google Sign-In manages the OAuth 2.0 flow and token lifecycle, simplifying our integration with Google APIs.

- OAuth 2.0 works with the following four actors:
  - authorization server: responsible for authentication and authorization.
  - resource server: in charge of serving up resources if a valid token is provided.
  - resource owner: the owner of the data, that is, the end user of Hoodie.
  - client: the Hoodie mobile app

FACEBOOK AUTHENTICATION

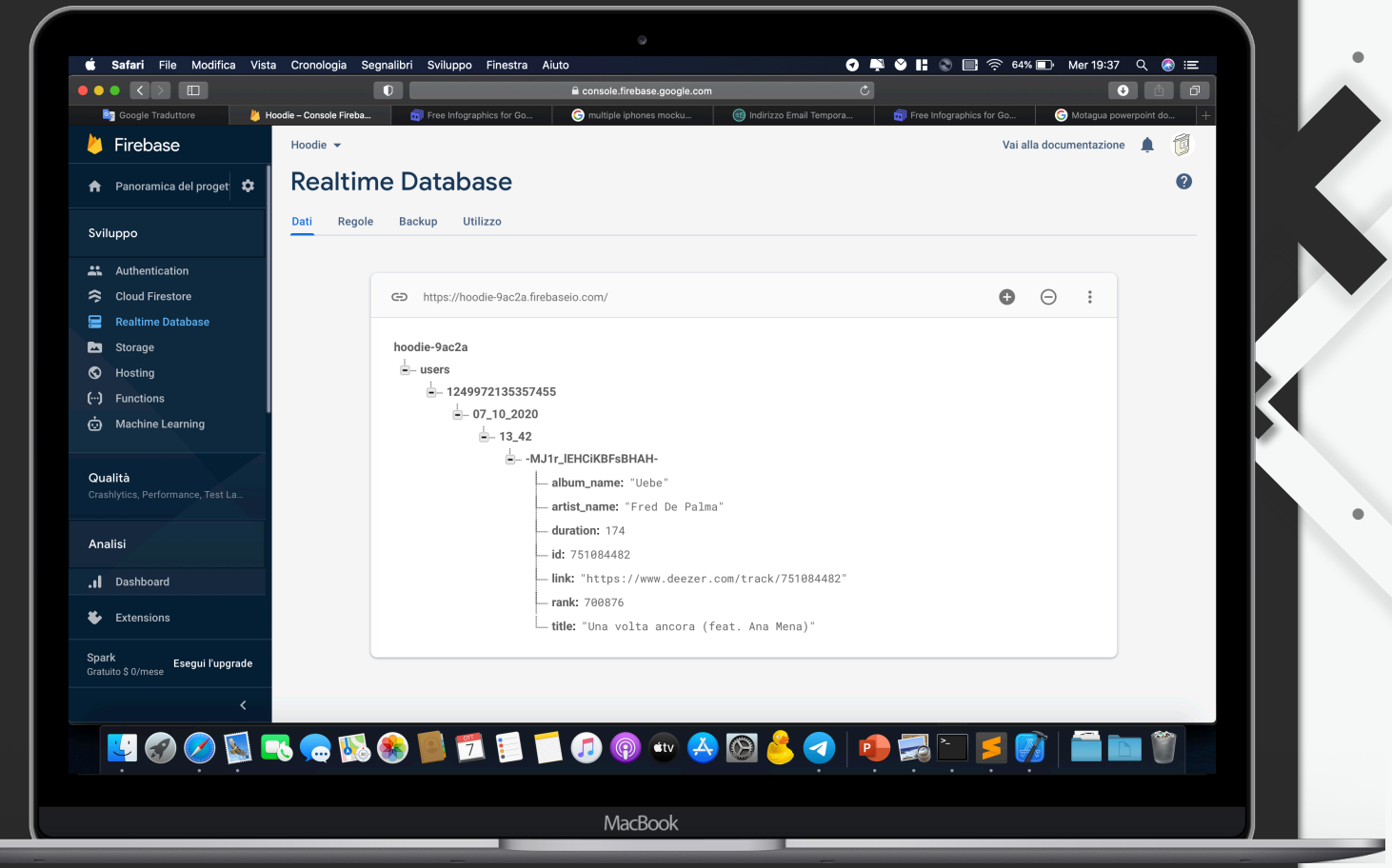
```swift
import FBSDKCoreKit
import FBSDKLoginKit

@IBAction func didPressFacebookSignin(_ sender: Any) {
    let loginManager=LoginManager()
    loginManager.logIn(permissions: ["public_profile", "email"],
viewController : self) { loginResult in
            switch loginResult {
            case .failed(let error):
                print(error)
            case .cancelled:
                print("User cancelled login")
            case .success(let grantedPermissions, let
declinedPermissions, let accessToken):
                print("Logged in")
                self.goToMainView(message: "")
            }
        }
}
```

```swift
func facebookLogout(){
    do {try LoginManager().logOut()}
    catch { print("already logged out") }
}
```

- With the rise of social media, Facebook login integration has become one of the must have features in mobile apps. Despite the fact that every developer is integrating Facebook login into their apps, Facebook is doing a very poor job on updating their documentations.

- Using the Facebook authentication token we get from Swift, we can access the Facebook API and get the user's name and email to create an account.

REALTIME DATABASE



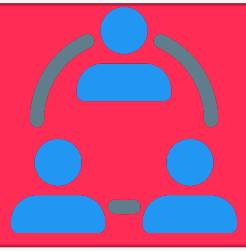
- The Firebase Realtime Database is a NoSQL cloud-hosted database. Data is stored as JSON and synchronized in real time to every connected client.

- All of our clients share one Realtime Database instance and automatically receive updates with the newest data.

iOS Application

Multi-Platform iOS Design

Responsive Layout

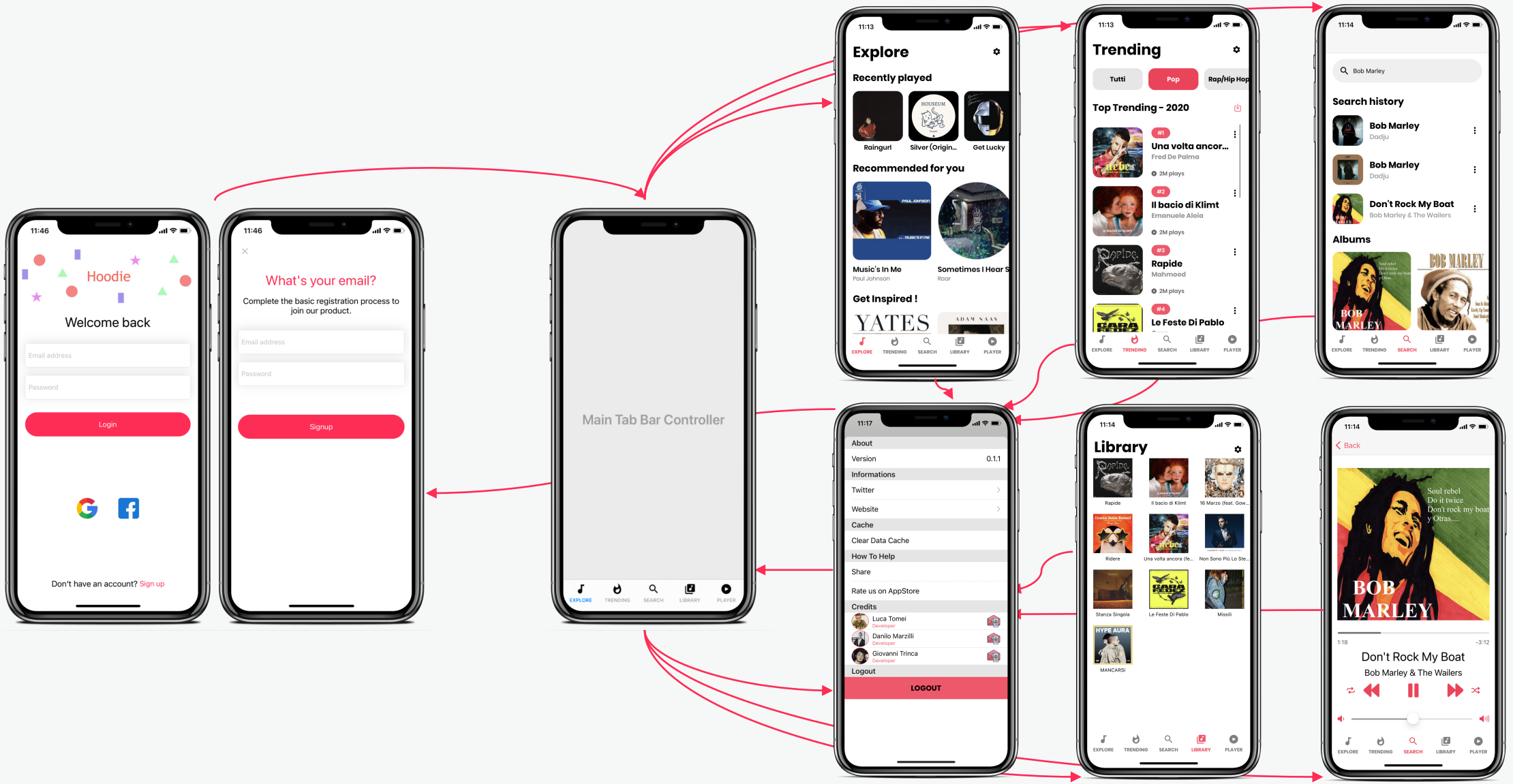


Multithreading

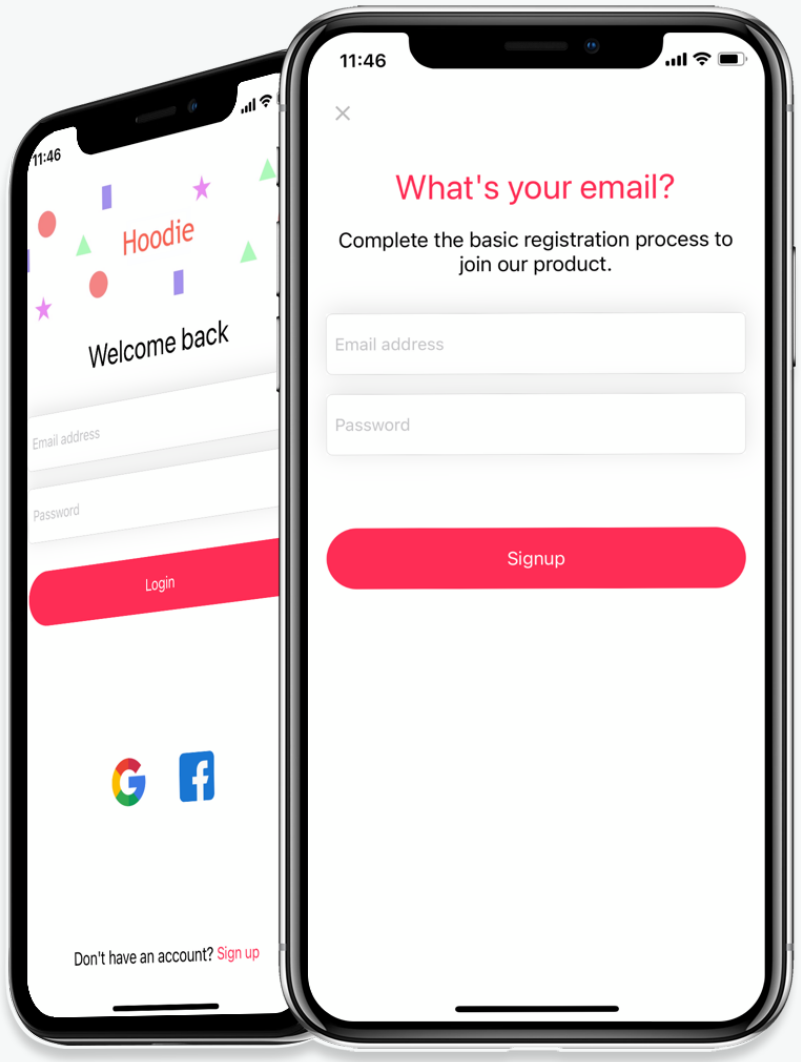
DispatchQueue + URLSession for https requests on separated thread
(Codable protocol used to convert JSON to native Swift struct)

Model View ViewModel Pattern Design

Navigation Path



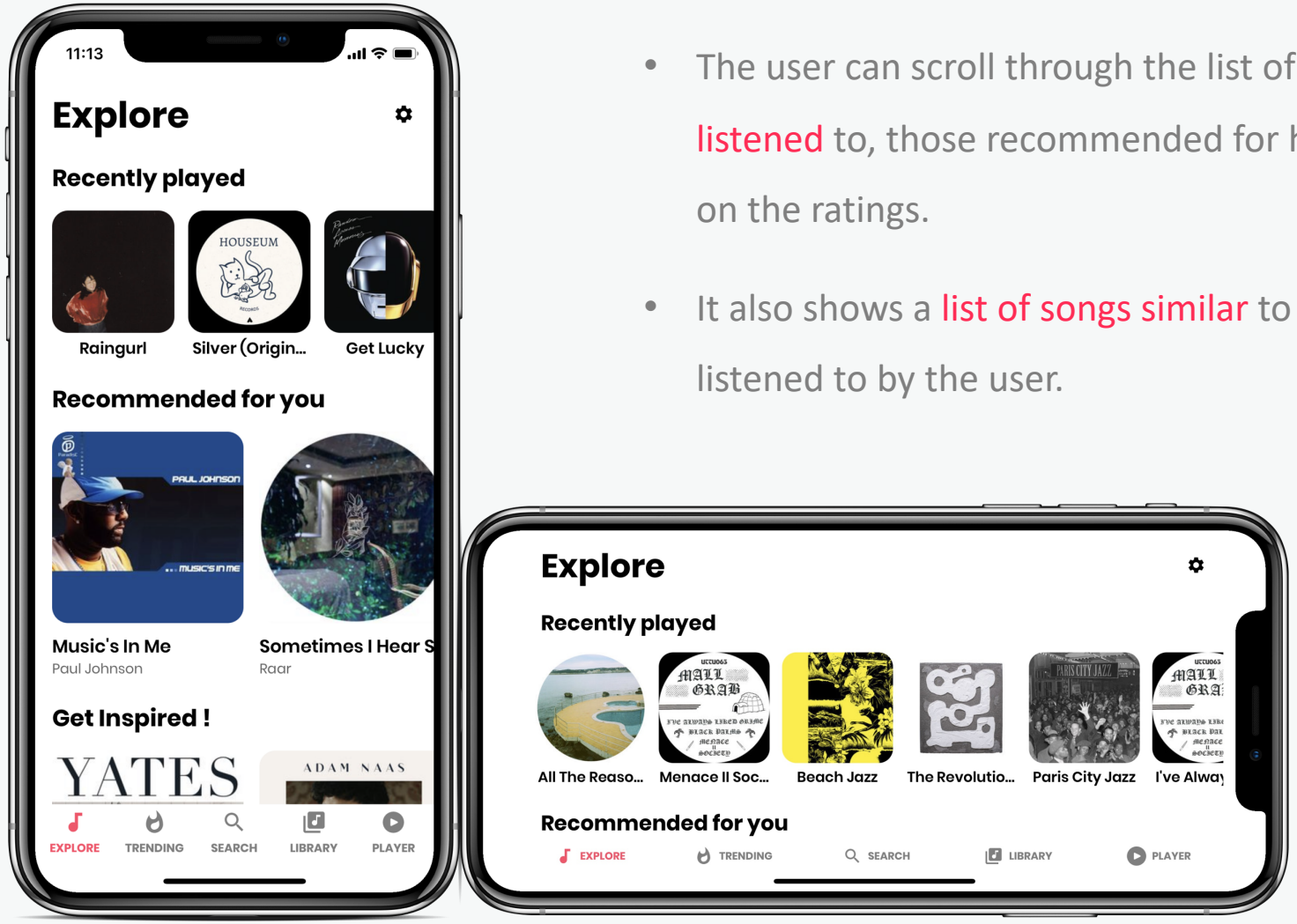
Login & Signup



- Login methods supported:
  - Email and Password (Firebase Auth)
  - Google OAuth
  - Facebook

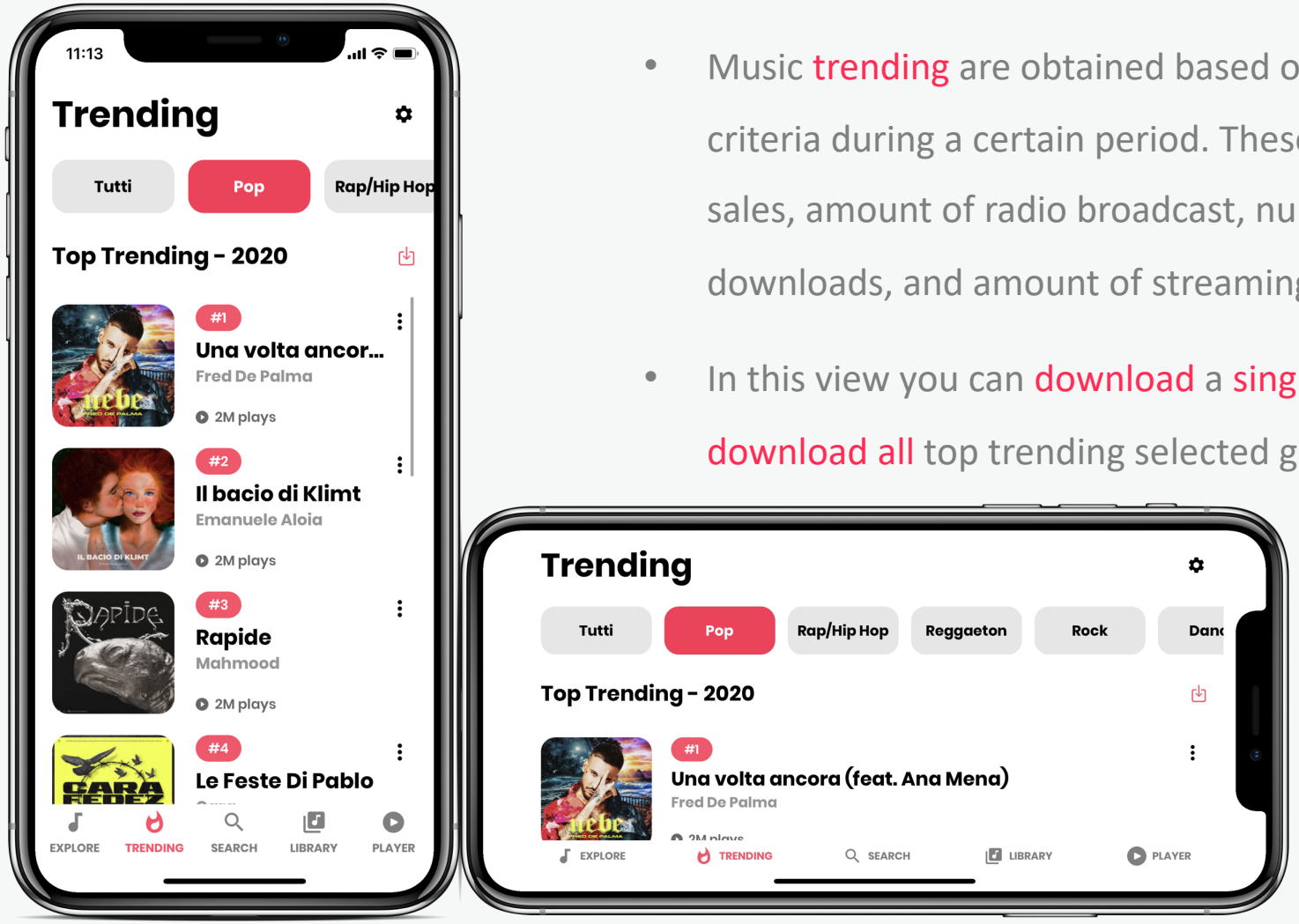
- After login or registration, the user is redirected to the main page of the application.

Explore View



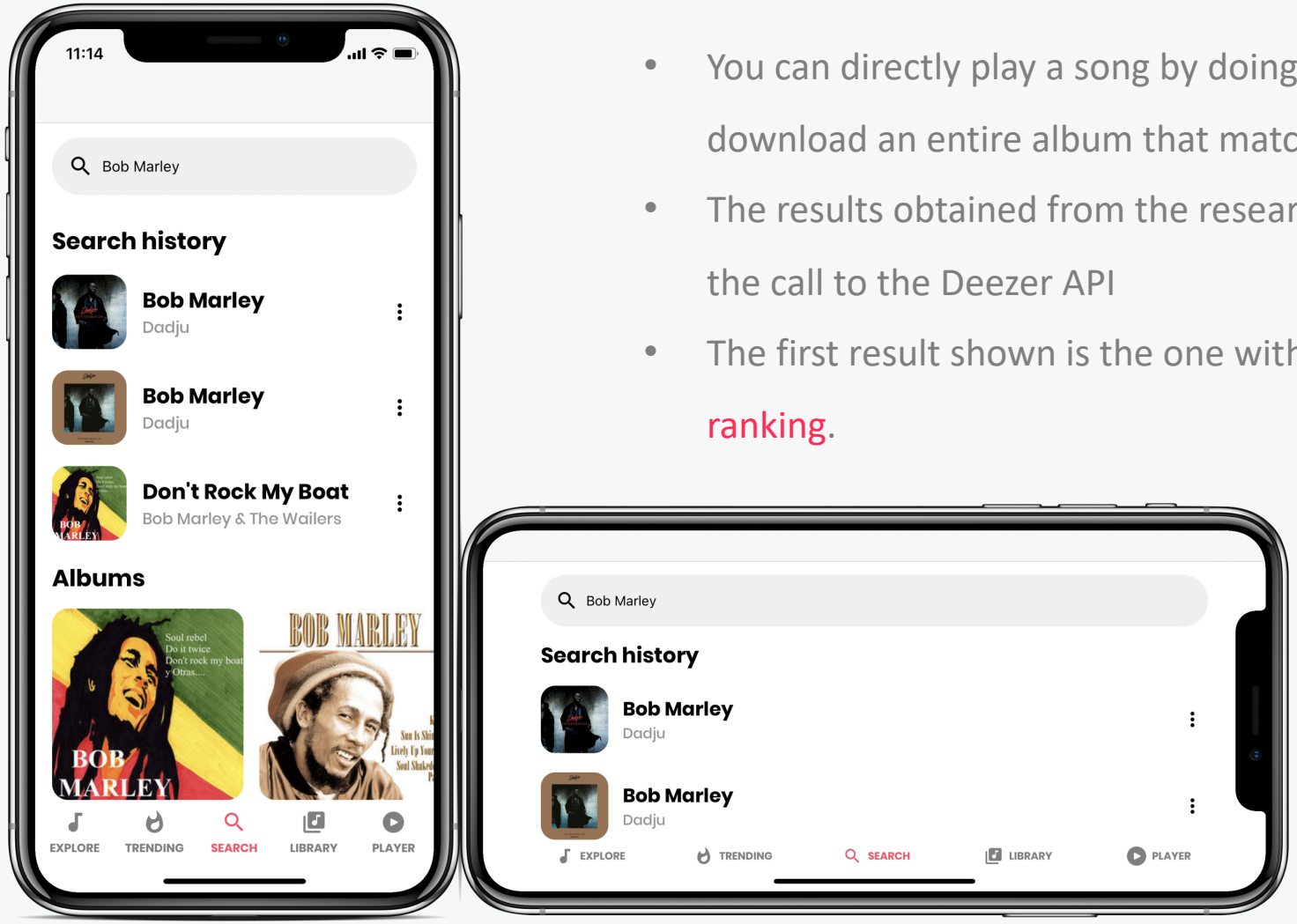
- The user can scroll through the list of songs recently listened to, those recommended for him/her based on the ratings.
- It also shows a list of songs similar to the type most listened to by the user.

Trending View



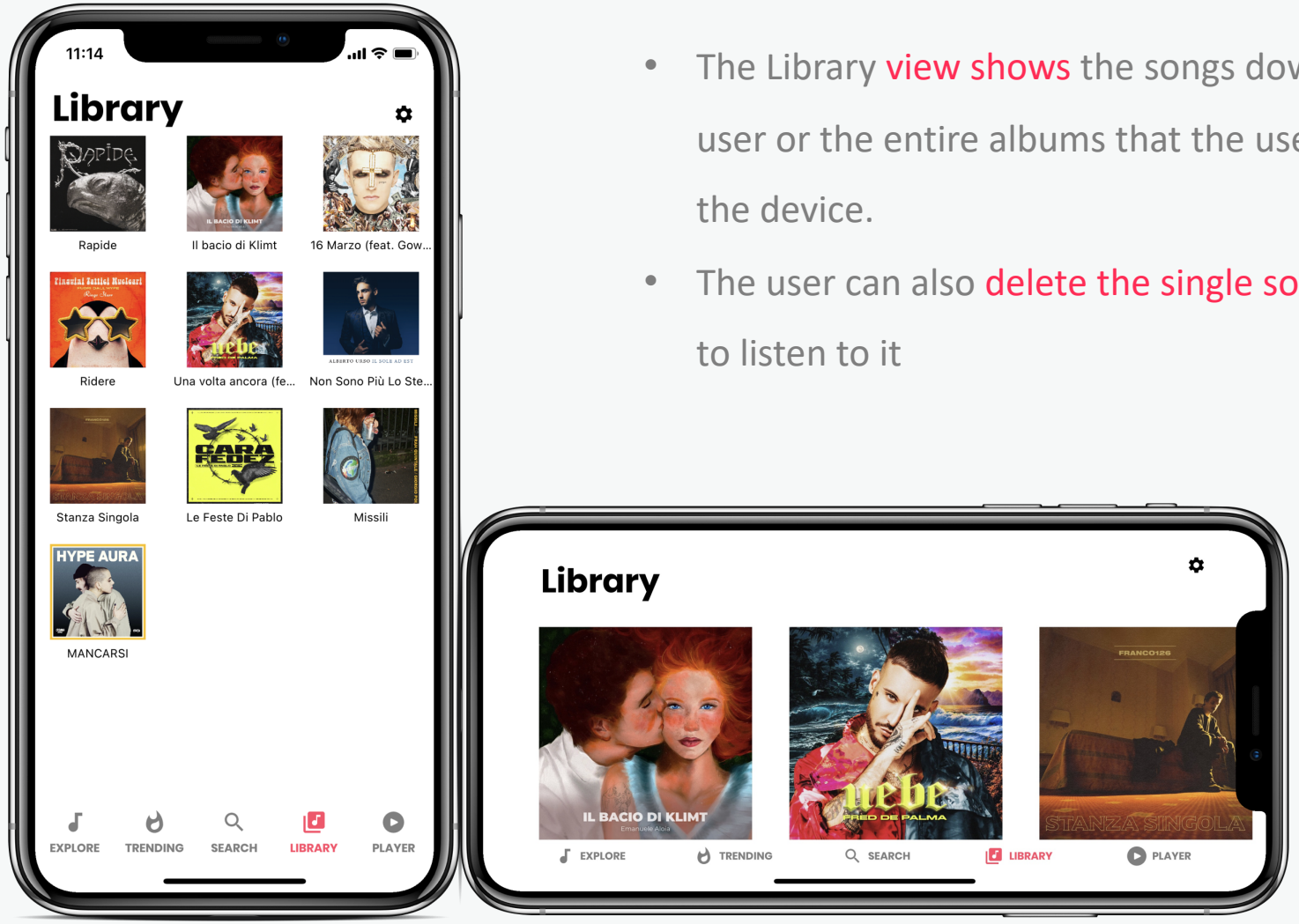
- Music trending are obtained based on certain criteria during a certain period. These include record sales, amount of radio broadcast, number of downloads, and amount of streaming activity.
- In this view you can download a single song or download all top trending selected genre.

Search View



- You can directly play a song by doing a search or you can download an entire album that matches that search.
- The results obtained from the research are the result of the call to the Deezer API
- The first result shown is the one with the highest ranking.

Library View



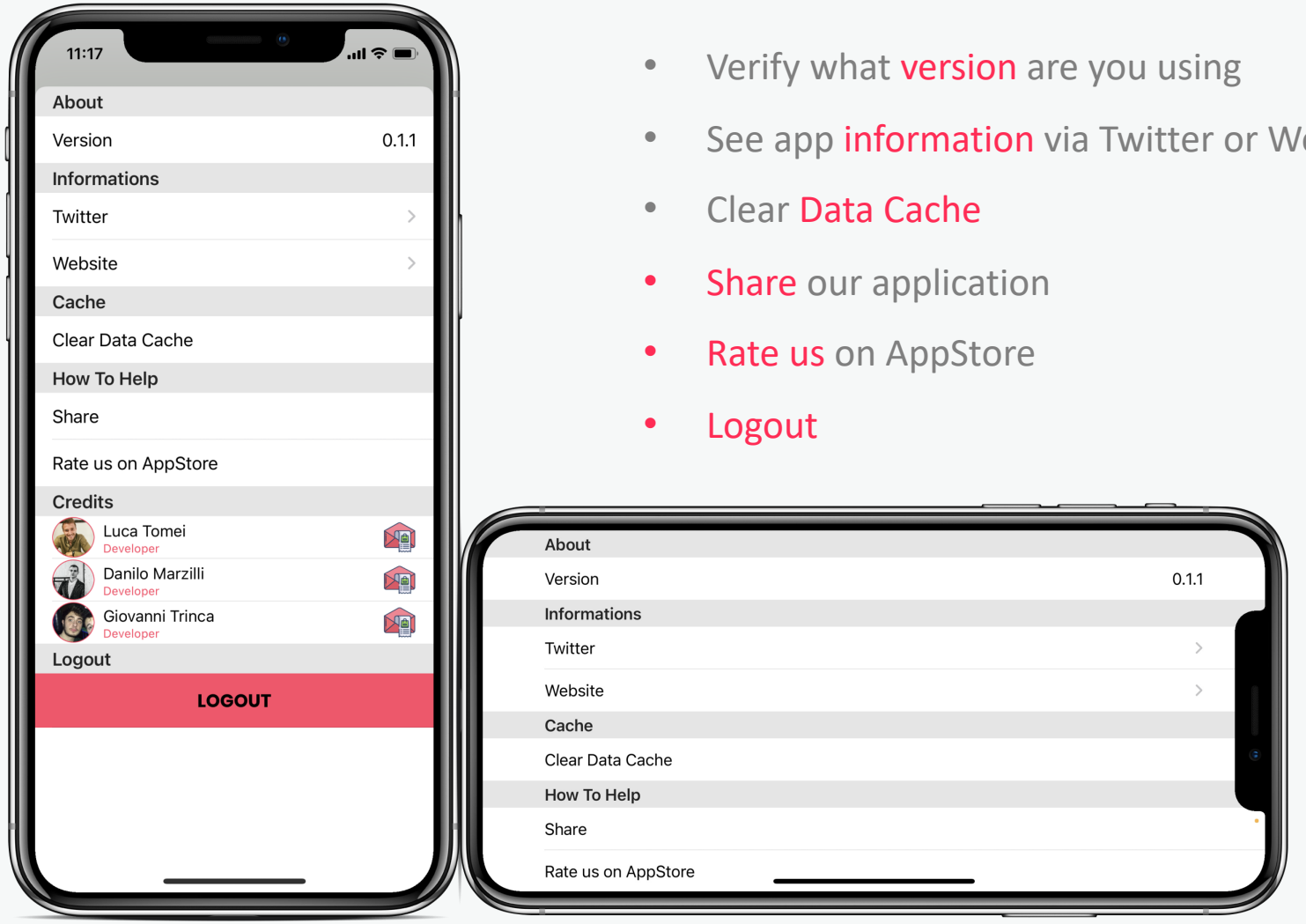
- The Library view shows the songs downloaded by the user or the entire albums that the user has stored in the device.
- The user can also delete the single song or select one to listen to it

Player View

Our player shows the music currently playing accompanying the listening showing:

- The name of the album, the song, the author
- Cover picture
- play/pause, next, previous, shuffle, repeat one, repeat all, volume slider buttons

Settings View



- Verify what version are you using
- See app information via Twitter or Website or email
- Clear Data Cache
- Share our application
- Rate us on AppStore
- Logout

Requirements

What we achieved

Responsive Design.

Auth. Service.

Multi Thread.

External WebAPI.

Storage Service.

Personal Backend.

-[WHAT NEXT?]-

Future Works.

A possible extension of the application is to support dynamic playlists and self-created suggestions based on the musical preferences of the individual user through the use of the data saved on the Firebase Realtime Database.

-[For your attention]-

Thank You