Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Informatica per il Management

Implementazione e valutazione di Mussida: un social network per amanti della musica

Relatore:
Dott.
Federico Montori

Presentata da:
Youssef Awni Barty
Fahmi Hanna

Sessione III
Anno Accademico 2022/2023

To my family and
all the people I care about.

Sommario

In un contesto in cui i le applicazioni mobili sono parte essenziale delle nostre vite e i *social network* il principale strumento di connessione tra le persone, ho deciso di creare il progetto **Mussida**. *Mussida* non nient'altro che è un *social network* per gli amanti della musica, pensato per essere uno strumento di "socializzazione musicale", tramite la quale, gli utenti possono interagire tra di loro e conoscersi in base ai propri ascolti. L'applicativo permette agli utenti la personalizzazione di un profilo musicale tramite generi, artisti, canzoni e *playlist*, e grazie a un algoritmo di raccomandazione, sarà possibile scoprire altri profili affini ai propri gusti musicali. In questo modo, ogni utente potrà uscire dalla propria *routine* d'ascolto e scoprire nuovi brani e artisti che potrebbero piacergli.

Indice

Elenco delle figure

Listings

Introduzione

All'interno della storia dell'uomo non esiste sicuramente un'innovazione tecnologica che ha mutato così rapidamente e intensamente la sua quotidianità quanto lo *smartphone*. La trasformazione digitale dei telefoni da semplici mezzi di comunicazione a parte integrante della nostra vita è stata rapida e radicale, e l'introduzione dei **social media** non è stata solo un'innovazione, ma un vero e proprio cambiamento dei paradigmi comunicativi e delle dinamiche sociali.

Ad oggi la tecnologia ha trasformato anche il nostro rapporto con la musica, che ha perso fortemente supporti come vinili, *cd* e nastri magnetici e viaggia esclusivamente in forma dematerializzata. L'invenzione forse più impattante sull'arte della musica è la sua trasformazione in una forma liquida con l'introduzione dello **streaming musicale**.

Arrivati a questo punto entra in gioco il progetto **Mussida**, che ha lo scopo di agguantare il primo concetto elencato e sfruttarlo come un potente mezzo di comunicazione per integrare il secondo concetto, ovvero creare un punto di riferimento virtuale nel quale gli utenti possono comunicare e socializzare tramite la musica.

"La musica è forse l'unico esempio di quello che avrebbe potuto essere- se non ci fosse stata l'invenzione del linguaggio, la formazione delle parole, l'analisi delle idee- la comunicazione delle anime"
Ritenevo fosse importante ribadire l'importanza della musica tramite questa famosa citazione dello scrittore francese *Marcel Proust*, che la definisce come una vera e propria alternativa al linguaggio scritto e parlato. In effetti difficilmente nella nostra vita riaffioreranno ricordi che non siano legati a delle melodie o canzoni. La musica diventa uno strumento necessario e indispen-

sabile che funge da colonna sonora a tutta la nostra vita, permettendoci di maturare un sentimento e un rapporto naturale e autentico sin dalla nostra nascita.

Unendo questo ultimo concetto agli altri due, abbiamo le fondamenta e le idee per un progetto che si occuperà di diventare uno strumento di divulgazione musicale, per dare vita a una biblioteca immaginaria piena di libri che si possono ascoltare.

Nel capitolo iniziale, **"Un vita app-dipendente"**, vengono trattati i concetti di base riguardanti le applicazioni e i *social network*, in modo da permettere al lettore di avere un quadro generale sugli argomenti successivi. In seguito, vengono spiegate le scelte implementative che hanno portato lo sviluppo di un applicazione mobile con l'integrazione di un *social*.

Segue il capitolo **"L'idea dietro Mussida"**, in cui viene spiegata l'idea innovativa che porta l'*app* all'interno del settore del *mobile* e i punti di forza rispetto ai *competitors* diretti sul mercato. Infine, vengono anche presentati le motivazioni personali che hanno portato l'inizializzazione del progetto.

Nel terzo capitolo, **"Architettura e flusso di Mussida"**, si analizza l'architettura e la struttura dell'applicazione. In seguito, viene mostrato il principale flusso di utilizzo dell'applicazione tramite *screenshots* permettendo al lettore di comprendere l'esperienza dell'utente e le funzionalità dell'applicazione meglio.

Segue il capitolo **"Tecnologie e implementazione"**, dove viene mostrato tecnicamente il flusso di autorizzazione tramite *Spotify* e il *server*. Dopodiché, si analizzano nello specifico tutte le tecnologie presenti all'interno dell'*app* riportando degli *snippet* di codice per capire al meglio l'implementazione e il contesto di utilizzo.

In seguito, nel capitolo **"Testing"** vengono analizzate le risposte ottenute dai *testers* e individuati i potenziali rafforzamenti da integrare all'interno dell'applicazione.

Infine, nelle **"Conclusioni"**, viene riportato un breve riepilogo del percor-

so effettuato per raggiungere il risultato finale e si spiega l'importanza di quest'esperienza, ancora da finire.

Capitolo 1

Una vita app-dipendente

In questo primo capitolo introduttivo andremo a studiare e analizzare i concetti di base delle *applicazioni mobili* e dei *social network* fornendo al lettore un fondamento e una chiave di lettura per comprendere al meglio le scelte implementative legate all'applicazione.

1.1 *Digital transformation* e applicazioni mobili

A seguito della rivoluzione della macchina a vapore, dell'introduzione della catena di montaggio e il primo computer, l'uomo si trova di fronte alla quarta grande rivoluzione tecnologica e culturale: la *digital transformation.* Si tratta di un processo di cambiamento tecnologico-sociale associato alle applicazioni di tecnologia digitale, che ha portato il settore del mobile a essere al primo posto in questo periodo storico. Il mondo delle applicazioni mobili è in continuo sviluppo ed è uno dei principali motivi per cui ho deciso di costruire questo progetto. Operare in un settore così fiorente potrebbe portare alla creazione di un prodotto alla portata quotidiana di tutti.

Negli ultimi decenni gli *smartphone* hanno invaso le case dell'**86,6%** della popolazione mondiale con **6.92** miliardi di utenti [1]. In figura 1.1 è possibile osservare l'impatto enorme che hanno avuto questi dispositivi dal 2016 fino a oggi e una stima degli utilizzatori per gli anni a venire.

Si può notare come si tratti di un successo inarrestabile che non accenna a diminuire, e che diventerà sempre di più necessario all'interno della vita

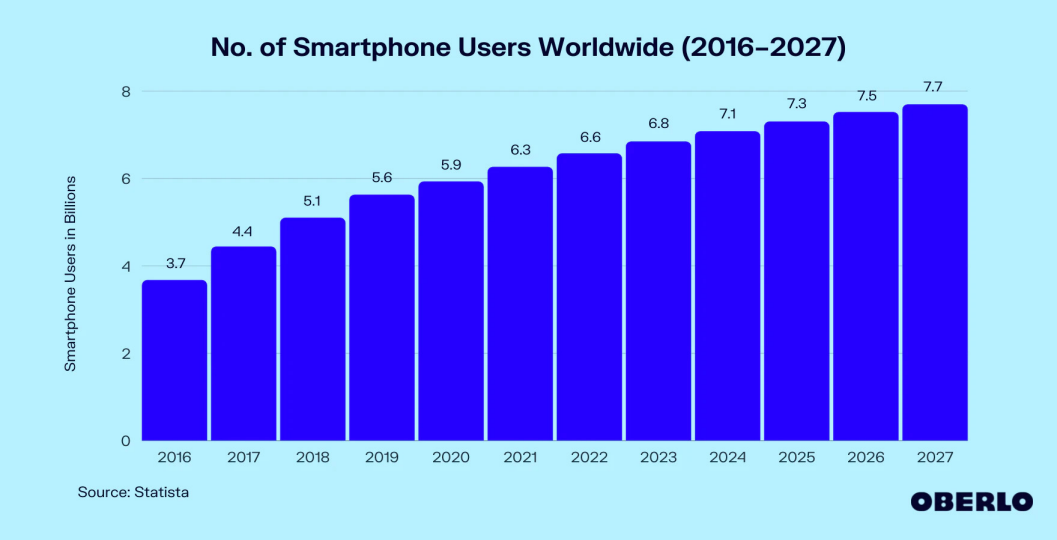

Figura 1.1: Grafico degli utilizzatori di *smartphone* dal 2016 al 2027 [2]

dell'uomo. In seguito a questa breve introduzione sull'impatto degli *smartphone* all'interno della nostra vita, verrà spiegato il concetto di applicazione mobile e la differenza tra varie tipologie.

1.1.1 Concetti base

Un applicazione mobile è un'applicazione *software* progettata per essere eseguita su un dispositivo come uno *smartphone*, un *tablet* oppure uno *smart watch* [3]. Nelle sotto-sezioni a venire, si va a spiegare la differenza tra le varie tipologie di applicazioni mobili e le motivazioni che hanno portato allo sviluppo del progetto per *mobile* e in modalità ibrida.

Distinzione tra applicazione mobile e web

La differenza principale tra un'applicazione mobile e un applicazione *web* risiede principalmente nella piattaforma utilizzata e nella *user experience*. Le applicazioni mobili risultano più performanti perché sono sviluppate per specifici *smartphone* e sfruttano direttamente le risorse presenti su di esso, inoltre possono fare affidamento e dialogare direttamente con le librerie native e avere una memoria locale sul telefono per non dipendere dalla connessione ad *internet* al 100%. La progettazione delle applicazioni *web* avviene per poter essere compatibile fondamentalmente con *computer* e *tablet*, e non possono

accedere a tutte le funzionalità del dispositivo. Dall'altra parte invece le *app* possono sfruttare una serie di proprietà uniche dello *smartphone* come le notifiche, la geolocalizzazione, la fotocamera, il microfono, i sensori e i contatti [4].

Sviluppare un'applicazione mobile permette una potentissima personalizzazione del servizio offerto e un adattamento al dispositivo ottimizzante aumentando la qualità dell'esperienza dell'utente.
Infine, la scelta migliore risiede nelle applicazioni per il *mobile* perché hanno il forte vantaggio di instaurare un rapporto quotidiano e una *routine* di utilizzo per l'utente, che grazie alle notifiche, ma anche semplicemente all'accesso immediato dell'*app* sul telefono, la rendono sempre a portata di mano.

Distinzioni tra applicazioni native e ibride

Effettuato il primo filtro di scelte implementative descritto nella sezione precedente è opportuno considerare un'altra importante distinzione, quella tra applicazioni native e ibride. Le applicazioni native vengono sviluppate esclusivamente per un sistema operativo, quindi possono essere installate solo su un dispositivo *Android* oppure *iOS*, e la loro implementazione sarà in un linguaggio nativo come *Swift* oppure *Java*. Esse garantiscono delle prestazioni elevate in quanto sono progettate per quella specifica piattaforma e garantiscono maggiore velocità e reattività per un'esperienza dell'utente migliore. Inoltre sono ottimizzate per l'accesso alle funzionalità del *device* come l'accelerometro della fotocamera, le notifiche *push*, i sensori e il microfono [5].

Dall'altro lato abbiamo le applicazioni ibride che vengono sviluppate per offrire una compatibilità multipiattaforma. Progettare un applicazione di questo tipo permette di rendere l'applicazione scalabile e flessibile, permettendo l'utilizzo a più sistemi operativi. Tra i linguaggi utilizzati possiamo trovare *Javascript, HTML*, *CSS* e l'implementazione avviene all'interno di contenitori nativi che fungono da *wrapper* al progetto e permettono l'accesso alle funzionalità del dispositivo tramite un solo codice. Si tratta di una metodologia in grado di fare risparmiare tempo agli sviluppatori, però simultaneamente è dispendiosa se si vuole ottenere un'implementazione performante per tutti i dispositivi [6].

Infine, bisogna considerare che esistono molti *framework* utilizzati da applicazioni ibride che permettono di ottenere prestazioni e risultati ai livelli delle *app* native. Questo è proprio quello che viene fatto all'interno di **Mussida** integrando il *framework Javascript* **React Native** [7] creato da *Facebook* nel 2015. Quest'ultimo è un *framework open-source* che permette di ottimizzare i tempi e la flessibilità delle *app*, infatti le discrepanze funzionali e grafiche tra i due sistemi operativi durante lo sviluppo sono state minime e semplici da risolvere.

In seguito a questa prima parte introduttiva sulle applicazioni *mobile* e la spiegazione delle scelte compiute per sviluppare il progetto, viene proposta una sezione in cui vengono spiegati i *social network* più impattanti e l'importanza che hanno acquisito negli anni.

1.2 *Social Network*

Al giorno d'oggi più di **4.9** miliardi di persone, come si può osservare in figura 1.2, accedono ai *social network* ogni giorno e interagiscono tramite esso per intrattenimento, socializzazione, lavoro, informazioni e pubblicità e si potrebbe continuare con un elenco interminabile [8]. Vi invito a pensare all'ultima volta che avete trascorso più di un giorno senza controllare i *social*, si tratta di una vera e propria componente fissa della nostra *routine* a cui è difficile rinunciare. E' anche per quest'ultimo motivo che è stato deciso di iniettare le funzionalità di **Mussida** in un'applicazione *social* e renderla una potenziale candidata per la quotidianità delle persone.

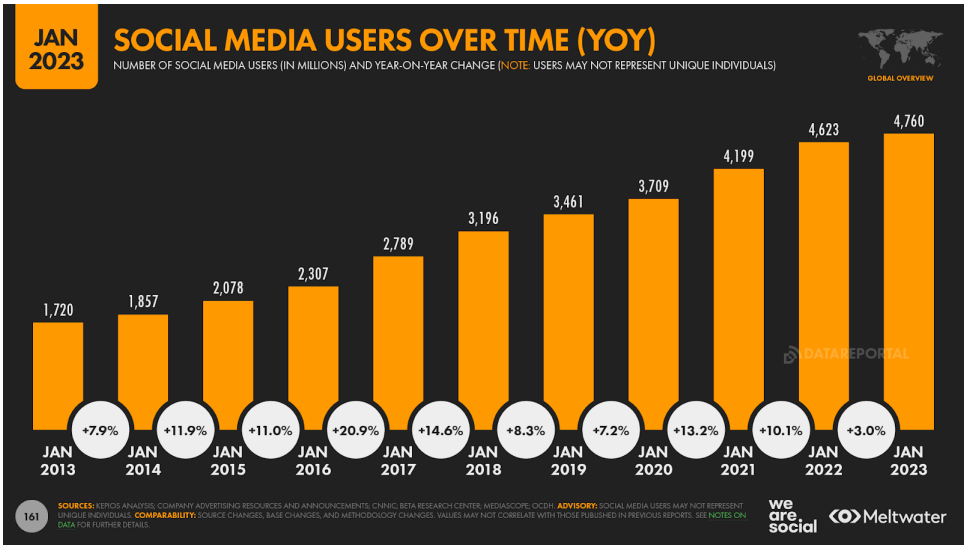

Figura 1.2: Grafico degli utilizzatori mondiali di *social network* [9]

1.2.1 Definizione

I *social network* sono un servizio *online* che permette la realizzazione di reti sociali virtuali. Si tratta di applicazioni, siti online e piattaforme nate con lo scopo di mettere in comunicazione tra loro gli utenti e permetterli di interagire e condividere contenuti come immagini, messaggi, video e audio. [10].

In altre parole sono reti di persone connesse tra di loro tramite legami di amicizia, conoscenza o lavoro, ma allo stesso tempo si può anche entrare in contatto con persone non realmente conosciute nel mondo fisico e interagire con esse.

Per far comprendere al lettore l'odierno successo dei *social network* e l'importanza che hanno acquisito negli ultimi anni ho ritenuto fosse necessario mostrare nella prossima sezione la storia e il percorso dei *social* più influenti.

1.2.2 Da SixDegrees a Mussida

I *social media* nascono intorno agli anni 2000 e negli ultimi decenni hanno avuto un considerevole impatto sociale sulla vita dell'uomo. In un mondo dotato di un cambiamento tecnologico così veloce, l'uomo ha adattato gran parte della sua essenzialità a essi.

Il primo *social network* fu creato, senza l'intenzione di essere un *social network*, nel 1997. Si tratta di **SixDegrees**, un sito nato dall'idea dell'avvocato *Andrew Weinreich* con il seguente motto:
"Trova le persone che vuoi conoscere attraverso le persone che conosci già"
Infatti fu creato per mettere in contatto persone che condividessero lo stesso ceto sociale, le stessa fascia di età e il concetto di sei gradi di separazione [11]. Quest'ultima teoria spiega come una persona poteva conoscere una qualsiasi altra persona nel mondo attraverso al massimo 5 persone, che poi si rivelarono 3. Nell'arco di 6 mesi il sito raggiunse i 600 mila iscritti, dopo 3 anni oltre 1 milione e poi venne venduta alla *Youthstream Media Networks* e chiuse per mancanza di fondi qualche anno dopo.

MySpace è stato il primo *social network* a raggiungere milioni di utenti attivi in un mese, nasce nel novembre del 2003 da due studenti di nome *Tom Anderson* e *Chris DeWolf* ed è considerato il padre dei *social*. La piattaforma permetteva di condividere messaggi, video e foto con i propri contatti e inoltre aveva un funzionalità aggiuntiva che permetteva ai musicisti di pubblicare contenuti e di avere un riscontro dagli altri utenti. Molti artisti che a voi potrebbero essere familiari come gli ***Arctic Monkeys, Nicki Minaj, Adele e Mika*** hanno avuto un forte successo e dei veri e propri gruppi di *fan* su *MySpace*. E' molto considerevole l'impatto di questa piattaforma perché oltre ad aver permesso la condivisione di contenuti di ogni genere, gli utenti potevano anche autopromuoversi e farsi conoscere.

In seguito, sempre nel 2003 viene data vita a quello che viene definito il *social network*, **Facebook**. *Mark Zuckerberg*, ai tempi un normale studente dell'università di *Harvard* sviluppò insieme a dei suoi colleghi universitari una piattaforma che permetteva di creare un album interattivo con tutte le immagini degli studenti universitari; In pratica, all'utente apparivano due immagini casuali e lui doveva scegliere la sua preferita. La crescita esponenziale e il successo del sito si propagandò dagli studenti di *Harvard* a tutti le persone con almeno 13 anni nel mondo. Al giorno d'oggi *Facebook* possiede 2,9 miliardi di utenti attivi al mese e 1,9 miliardi di utenti attivi al giorno.

E' importante aver menzionato la storia di questi ultimi due *social*, perché da qui in poi nascono altri migliaia di applicativi con le stesse funzionalità.

Il loro enorme successo è dovuto a una componente fondamentale che forse al giorno d'oggi sembra scontata, ovvero quella di permettere a due utenti distanti fisicamente di condividere contenuti multimediali tra loro. Questo aspetto ha letteralmente rivoluzionato il mondo tecnologico e sociale, la gente poteva "teletrasportare" la propria vita in pacchetti e arrivare dall'altra parte del mondo, le persone potevano viaggiare attraverso il telefono e visitare luoghi, abitudini e socializzare dentro le mura della propria casa.

Youtube nasce nel 2005 e diventa la più grande piattaforma di condivisione video al mondo, nata dall'idea di *Jawed Karim, Chad Hurley e Steve Chan*. Nasce con lo scopo di condividere esperienze e documentazioni multimediali di viaggi e diventa un potentissimo servizio di *videosharing* di tutte le categorie, da video scientifici a *videoclip* musicali. Non a caso nel 2006 cattura l'attenzione di *Google* che decide di acquistarla per 1,65 miliardi di dollari. Al giorno d'oggi *Youtube* beneficia di 2 miliardi di utenti attivi al mese.

Nel 2010 viene implementata da *Mike Kryger e Kevin Systrom* **Instagram**, come un'applicazione per la condivisione di immagini e diventa al giorno d'oggi il *social* più usato dai giovani con 1 miliardo di utenti attivi al mese. Seguono moltissimi *social* tra cui **Twitter** nato nel 2006 da *Jack Dorsey*, **Snapchat** nel 2011 da *Evan Spiegel, Bobby Murphy e Reggie Brown* [12].

L'idea di partire analizzando *SixDegrees* è dovuta al fatto che **Mussida** prende spunto dal motto citato precedentemente per crearne un altro:

"Trova la musica che vuoi conoscere attraverso la musica che conosci già"
Si tratta proprio dell'idea che guida il nostro algoritmo, grazie al quale grazie alla musica che ascoltiamo ne vogliamo conoscere di nuova. A questo si unisce l'innovazione che ha portato *MySpace* per la condivisione musicale tramite *social* e l'importanza che dal 2003 *Facebook* attribuisce alla parola *"Social Network"*.

Capitolo 2

L'idea dietro Mussida

L'applicazione è nata per permettere agli utenti di condividere musica in più contesti. All'interno del *social* non sono importanti i propri contatti o conoscenti, ma solo le persone più affini musicalmente all'utente. Questa caratteristica è stata studiata come un punto di forza per creare una connessione oltre la propria cerchia quotidiana di persone, attraverso la musica.
Ogni utente può trovare persone che condividono i suoi gusti musicali e interagire con loro conoscendo nuovi artisti e canzoni.
Mussida viene anche creata per incentivare la musica dal vivo, dando la possibilità agli utenti di pubblicare un evento e agli altri di partecipare.

2.1 Il perché

Come noteranno tutti gli utenti che proveranno l'applicazione, la maggior parte delle funzionalità di **Mussida** è inglobata all'interno di altri *social network*. Pubblicare canzoni, eventi e avere un profilo sono tutte *features* già esistenti.
Risulterà naturale domandarsi il motivo dello sviluppare un progetto che a livello funzionale sussiste già, un servizio che non porta quasi nessun tipo di novità. Il motivo in realtà è molto semplice e allo stesso tempo è il punto di forza principale dell'applicativo: **l'esclusività della musica.** Il flusso dell'applicazione e le funzionalità sono state studiate in modo che agli utenti, ai quali non interessa condividere e ascoltare musica, risulterà molto difficile interagire con l'applicativo per più minuti. Infatti chiunque deciderà di utilizzare **Mussida** avrà la sicurezza di essere un punto di interesse per il suo

"pubblico", e l'introduzione di un algoritmo di raccomandazione permetterà a tutti gli utenti di confrontarsi con persone che condividono gli stessi interessi e gusti musicali.
E' lo stesso motivo per cui **LinkedIn** è l'applicazione più utilizzata da *recruiter*, datori di lavoro e dipendenti, perché ha uno scopo principale per essere installata, un *focus* che interessa tutti gli utenti: il lavoro. Si tratta di una piattaforma con più di 660 milioni di utenti in 200 paesi diversi, e utilizzata da tutti per le stesse cause: assunzione e ricerca del lavoro, *marketing*, e *networking* professionale. Il nostro *software* finale mira proprio a questo esempio, non vuole eccedere di funzionalità differenti e non correlate tra di loro, ma vuole porsi come obiettivo quello di essere un punto di riferimento dove poter creare legami musicali di ogni tipo.

2.2 Socializzazione tramite musica

L'applicazione permette a utenti sconosciuti, ma con gusti musicali affini, di conoscersi. L'importanza che la musica ha nella vite umane è di forte valore, si tratta uno degli attributi che più caratterizza il nostro lato intimo, e introdurla come unico parametro per conoscere una persona è una sfida più che ragionevole. La musica, infatti, gioca un ruolo molto importante per l'autenticità di un profilo all'interno dell'*app*, i gusti musicali sono un sentimento che maturiamo sin da piccoli e sul quale risulta difficile mentire.

You can't copy anybody and end with anything. If you copy, it means you're working without any real feeling. No two people on earth are alike, and it's got to be that way in music or it isn't music.
The whole basis of my singing is feeling. Unless I feel something, I can't sing. When you sing, always tell the truth" [13]

Come dice Billie Holiday, cantante statunitense, fra le più grandi di tutti i tempi nei generi jazz e blues, la musica è una delle forme d'arte più grande che caratterizza il sentimento e l'ideologia di una persona. Non c'è nessun motivo per copiare la musica, perché come gli esseri umani sono unici lo saranno anche i loro gusti musicali. Cosi come l'ultima frase riportata della citazione di *Billie Holiday*:

"When you sing, always tell the truth"

Io credo vivamente che quando si ha a che fare con la musica non si può non offrire un lato naturale e autentico di noi. Per questo sarebbe stato inconcludente caratterizzare il profilo di un utente con altre caratteristiche poco rilevanti e trascurabili per conoscere una persona.

2.3 *Competitors* diretti

In questa sezione andremo a riportare e analizzare le principali applicazioni musicali che potrebbero essere dei potenziali *competitors* sul mercato.

1. **TikTok**[14]: è una delle piattaforme di *video-sharing* più in utilizzo al giorno d'oggi. Principalmente permette agli utenti di condividere e vedere video di 15 secondi circa con musica o effetti sonori su un *feed* personalizzato. Permette anche a *influencers* e personaggi famosi di compiere video promozionali e pubblicità di ogni tipo. Un aspetto molto importante che disaccoppia *Mussida* dalla seguente *app* è sicuramente la *user-experience*, l'utilizzo di *TikTok* è progettato affinché l'utente "non faccia nulla" a livello interattivo e rimanga attaccato al feed grazie alla breve durata dei video e la velocità con cui scorrono automaticamente. *Mussida* intenzionalmente vuole stancare l'utente, ha l'obiettivo di fare socializzare gli utenti permettendoli di scambiare musica per poter ampliare il repertorio musicale sulla piattaforma e arricchire i *feed* di canzoni. Infine, i video su *TikTok* solitamente hanno il *focus* puntato sul video che raramente ha un legame diretto con la canzone, anzi la musica è messa in secondo piano proprio come una *mini* colonna sonora del video.

2. **Last.fm**[15] è un'applicazione che permette agli utenti di collegare un *account Spotify, Apple Music, Google Play ecc.* e in seguito esegue quello che viene chiamato *"scrobbling"*. Quest'ultima funzionalità non è nient'altro che un sistema di monitoraggio dei nostri ascolti che viene utilizzato per l'implementazione di un algoritmo di raccomandazione che ha lo scopo di offrire una radio personalizzata in base ai gusti dell'utente. Si tratta di una logica di funzionamento molto simile a *Mussida*, ma che non integra al suo interno nessun meccanismo di socializzazione tra gli utenti e un luogo di incontro virtuale per conoscersi.

Inoltre, l'applicativo è pensato per essere un sito *web* e non un'applicazione *mobile*, di conseguenza la fluidità nel flusso di utilizzo su telefono risulta meccanica e complicata, dovuta a un grande quantitativo di reindirizzamenti a pagine *web* per effettuare determinate operazioni.

3. **Bandcamp**[16] è una piattaforma *online* che permette di caricare musica, ascoltarla e venderla. Ogni utente può crearsi le proprie *playlist* e navigare all'interno di essa per conoscere nuovi artisti e connettersi a quelli che preferisce. È un'applicazione che nasce per offrire l'opportunità ad artisti emergenti di caricare la propria musica e autopromuoversi, quindi non sono presenti al suo interno canzoni di artisti che non sono direttamente iscritti all'*app*. Questo fa si che le librerie musicali di *Bandcamp* non contengano tanti brani di artisti "già conosciuti" e in questo modo l'utente non avrà un ampio reperto come *Spotify* con cui poter creare le proprie *playlist*. Si tratta di una singola funzionalità che si vuole integrare all'interno di *Mussida*, ma che non è vincolante per la caratterizzazione del profilo e degli ascolti come in questo caso.

Uno degli aspetti più rilevanti e importante dell'applicazione, come si può osservare da quanto riportato, è che non ha dei *competitors* diretti sul mercato. Sono state trovate molte applicazioni che riguardassero la musica, ma solo una piccola percentuale è incentrata solo su quest'ultima ed è allo stesso tempo un *social network*.

2.4 Motivazioni

L'idea dell'applicazione è nata da un forte bisogno personale di costruire un ambiente creativo virtuale dove le persone iscritte potessero "comunicare" attraverso la musica. **Mussida** è un'applicazione gratis che permette agli utenti di scoprire nuova musica, apre le porte a un pozzo di suggerimenti e aggiornamenti su nuove canzoni, concerti ed eventi. Uno dei traguardi principali dell'*app* è quello di diventare un forte servizio di ampliamento della cultura musicale di ciascun utente. Ogni *user* potrà uscire dalla propria *routine* musicale e scoprire nuove canzoni e anche persone.

Capitolo 3

Architettura e flusso di Mussida

3.1 Architettura

La nostra architettura si compone di 4 entità che interagiscono tra di loro. Queste sono:

- *frontend*, un'applicazione *mobile* sviluppata in *React Native* tramite il *framework Expo*.
- *backend*, un *server* sviluppato in *Node.js* che utilizza il *framework Express* e *Trpc*.
- *database*, sviluppato con *MongoDB*.
- *SpotifyWebApi*, un servizio messo a disposizione da *Spotify* per interagire con le sue *api*.

L'applicazione interagisce sia con *Spotify* che con il *backend* per la visualizzazione e la modifica di dati. Il *backend* interagisce con le *api* di *Spotify* e con il *database* per salvare i dati.

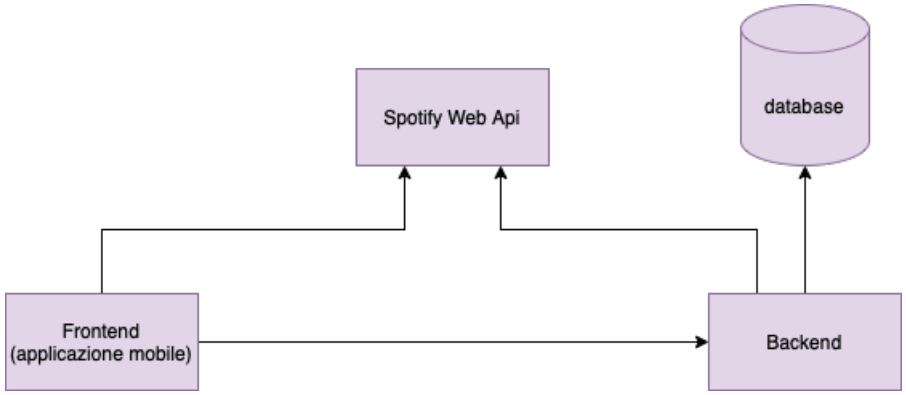

Figura 3.1: Diagramma dell'architettura dell'applicazione

Per comprendere al meglio l'interazione tra questi componenti all'interno del flusso dell'applicazione, sono stati creati dei **diagrammi di sequenza** per le operazioni principali. Cosi il lettore potrà comprendere al meglio come comunicano queste 4 entità.

3.2 Flusso di utilizzo

In questa sezione verrà mostrato il principale flusso di utilizzo dell'applicativo con i relativi *screenshots* per mostrare l'esperienza dell'utente e diagrammi di sequenza per spiegare come avvengono le operazioni.

3.2.1 Autenticazione

Al primo avvio dell'applicazione si richiede all'utente di effettuare l'accesso tramite *Spotify*.

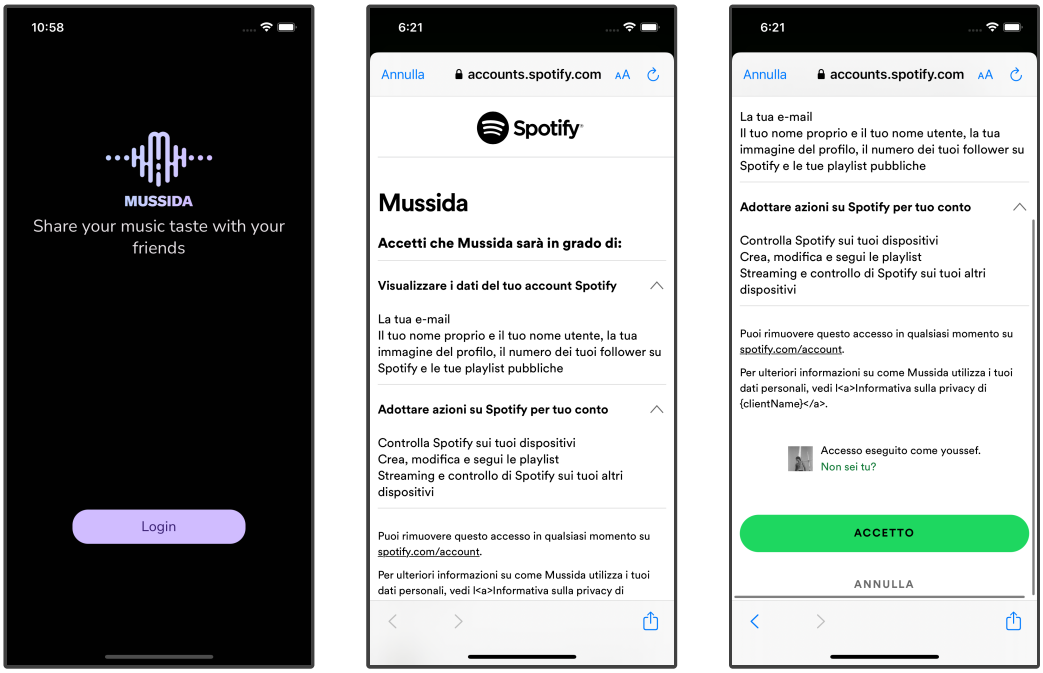

Figura 3.2: Flusso di autenticazione tramite *Spotify*

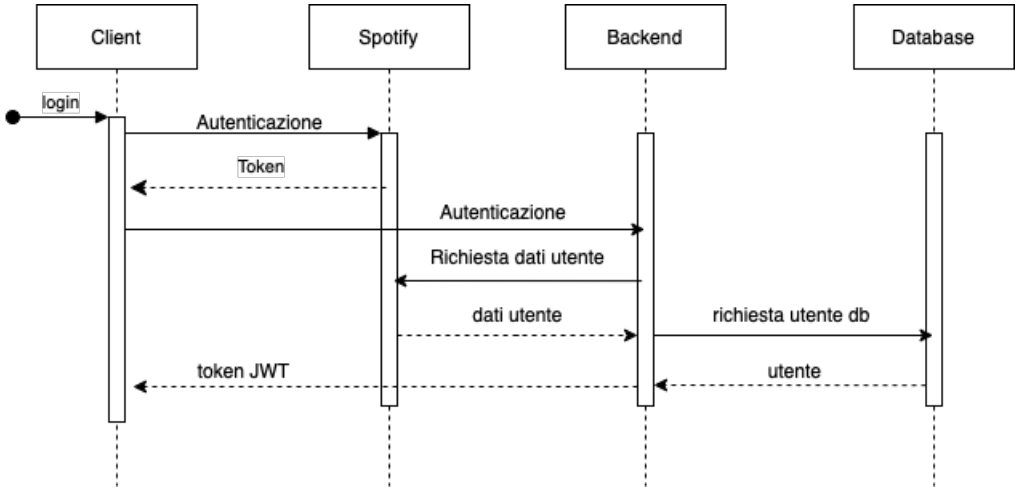

Figura 3.3: Diagramma di sequenza del *login*. Si può notare come l'utente effettui prima l'autenticazione su *Spotify* e, in seguito, sul *server*.

In seguito all'autenticazione tramite *Spotify*, l'utente potrà accedere alla *home* dell'*app* dove potrà navigare tra le due pagine principali che sono:

1. *Feed*, dove l'utente potrà interagire con *post* consigliati tramite algoritmo di raccomandazione e altri dai profili che segue.

2. Profilo, dove l'utente potrà rappresentarsi e descriversi tramite le sue preferenze sui generi, le canzoni e gli artisti che più lo rispecchiano.

3.2.2 Personalizzazione del profilo

Come riportato nella sezione precedente gli attributi che raffigurano il profilo di un utente sono:

- I suoi generi musicali preferiti.
- Le sue canzoni preferite.
- I suoi artisti preferiti.

Nella pagina del profilo dell'utente vengono anche impostati come predefiniti il suo nome e la sua immagine del profilo di *Spotify*. Inoltre, sotto gli attributi musicali di un utente viene riportata una *history* dei *post* di quell'utente.

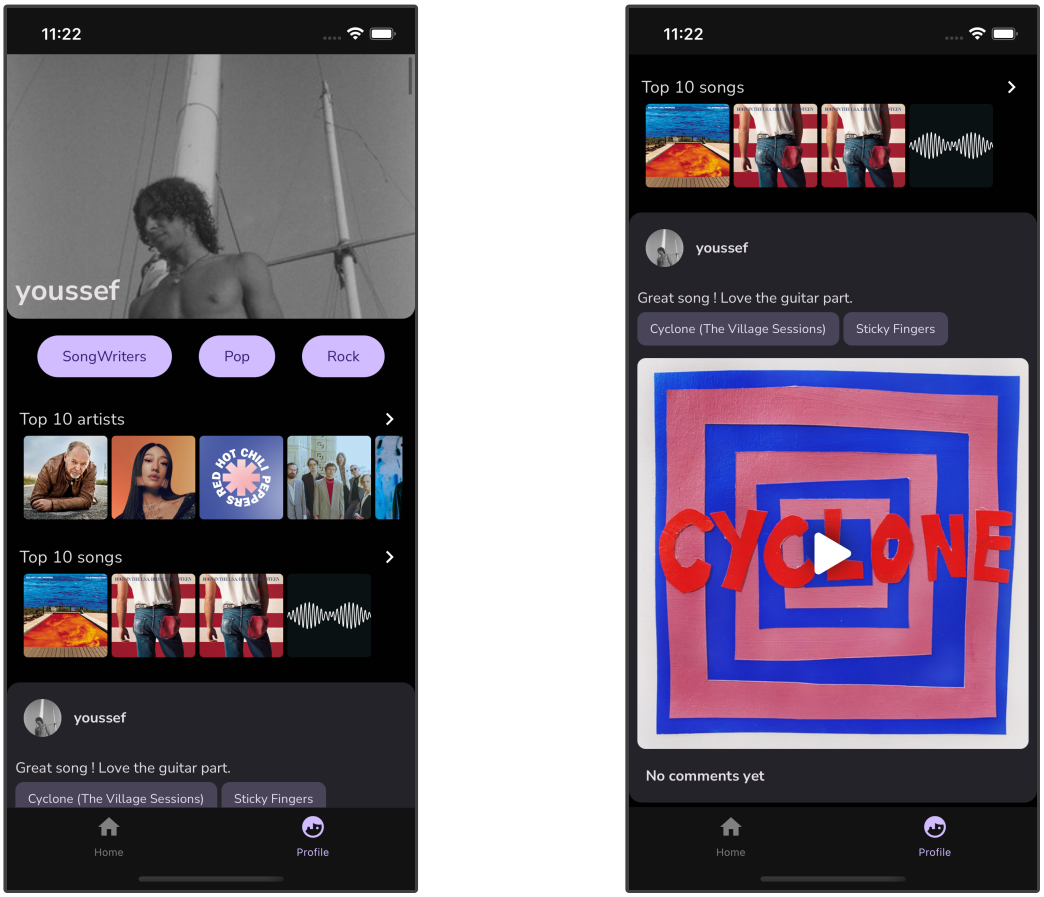

Figura 3.4: Pagina del profilo dell'utente.

I generi dell'utente si trovano all'interno di tre bottoni colorati sotto l'immagine del profilo ed è la prima caratterizzazione che si vede immediatamente e che risalta agli occhi. Per modificare i propri generi basterà cliccare sul bottone che si vuole aggiornare, successivamente l'utente accederà alla lista globale dei generi presenti su *Spotify* e potrà selezionare quello che preferisce.

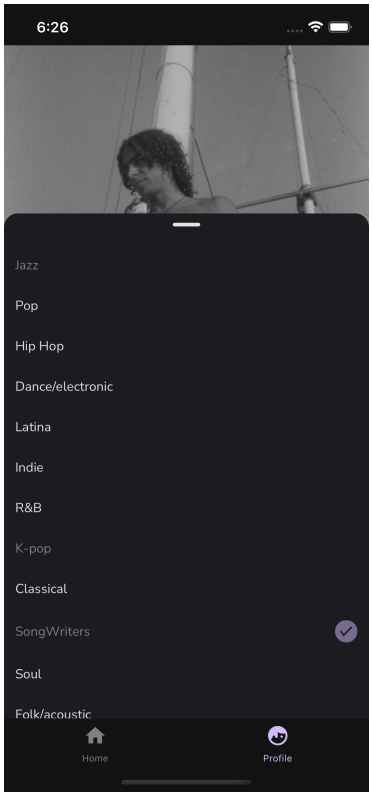

Figura 3.5: Scelta dei generi preferiti dell'utente.

Per modificare la lista delle canzoni o degli artisti, l'utente dovrà'accedere alle corrispettive liste, e in seguito, tramite un bottone in basso a destra, potrà espandere un *bottomsheet* contenente la barra di ricerca per le canzoni o gli artisti su Spotify. Una volta che l'utente ha selezionato l'entità che si vuole aggiungere alla propria lista, il *bottomsheet* si chiuderà e si aggiornerà la lista dei preferiti.

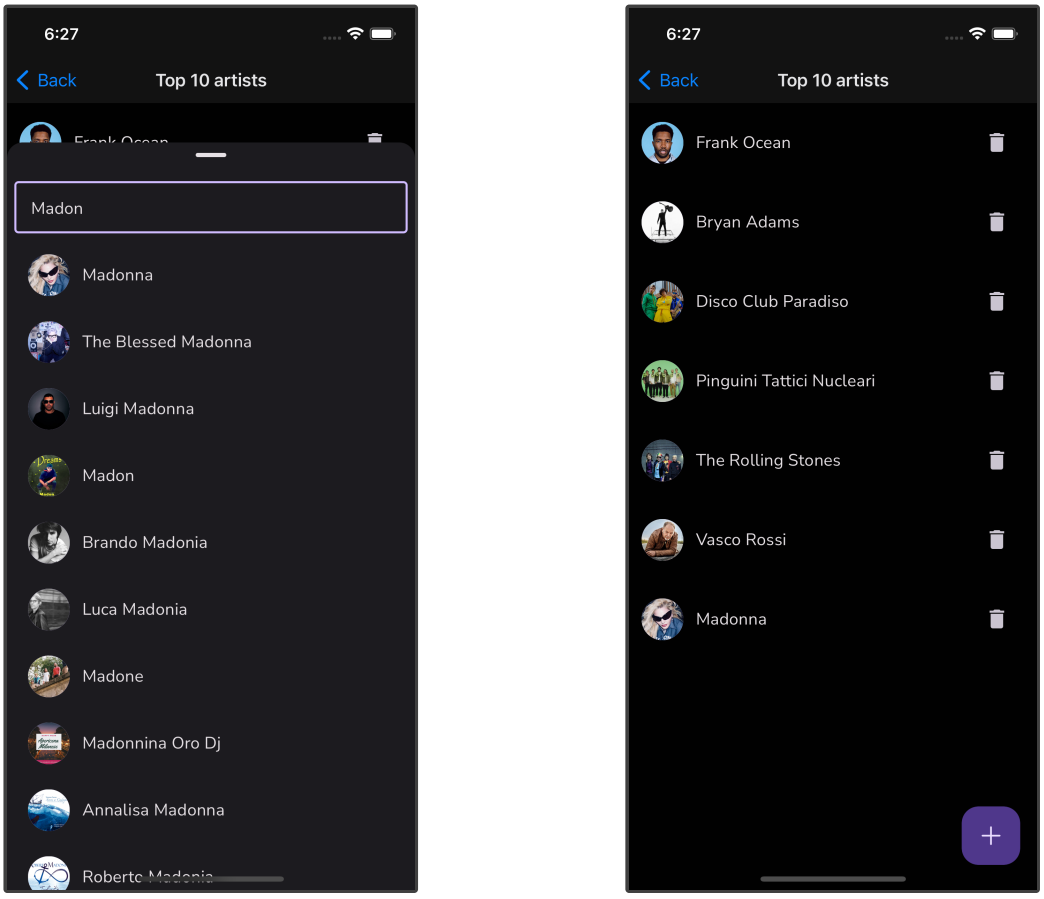

Figura 3.6: Aggiunta di un artista nella *top 10*.

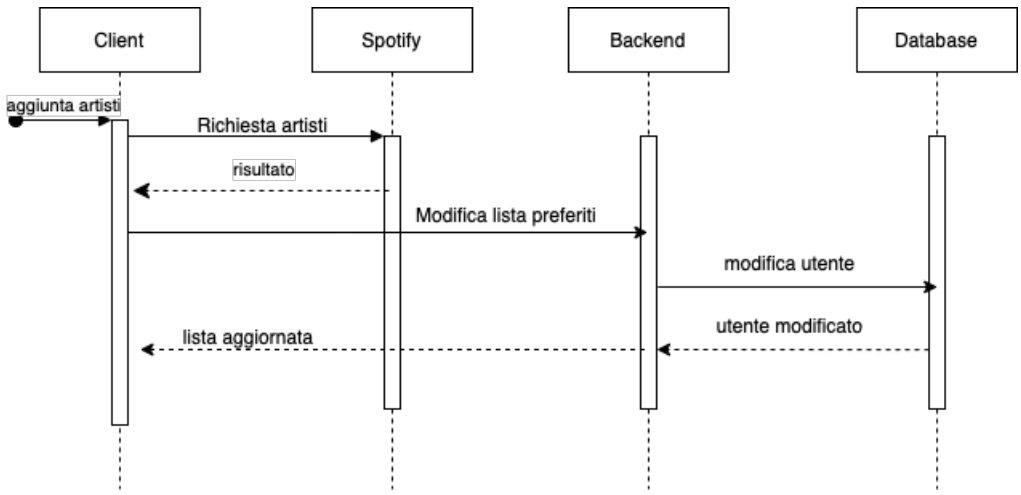

Figura 3.7: L'utente per aggiungere un nuovo *post* effettua un ricerca su *Spotify* e sceglie l'artista da aggiungere. In seguito, l'utente restituito da *Spotify*, viene mandato al *backend* che lo salva nel *database*.

L'utente potrà eliminare dalla lista gli artisti o le canzoni premendo il bottone presente sulla destra con l'icona del cestino e confermando l'operazione tramite un *bottomsheet*.

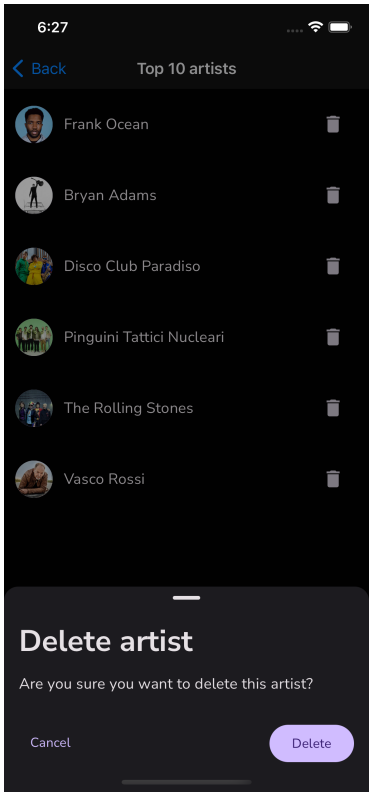

Figura 3.8: Eliminazione di un artista.

3.2.3 Feed

La seguente sezione dell'applicativo rappresenta il principale punto di forza di **Mussida**, grazie a essa infatti l'utente potrà ascoltare tramite un *player* la canzone associata al singolo *post*. È come se ogni giorno tramite il *feed* l'utente abbia accesso a una *playlist* dinamica e nuova da ascoltare. All'interno della pagina del *Feed* l'utente potrà visualizzare due tipologie di *post*:

- I *post* calcolati tramite l'algoritmo di raccomandazione.
- I *post* degli utenti seguiti.

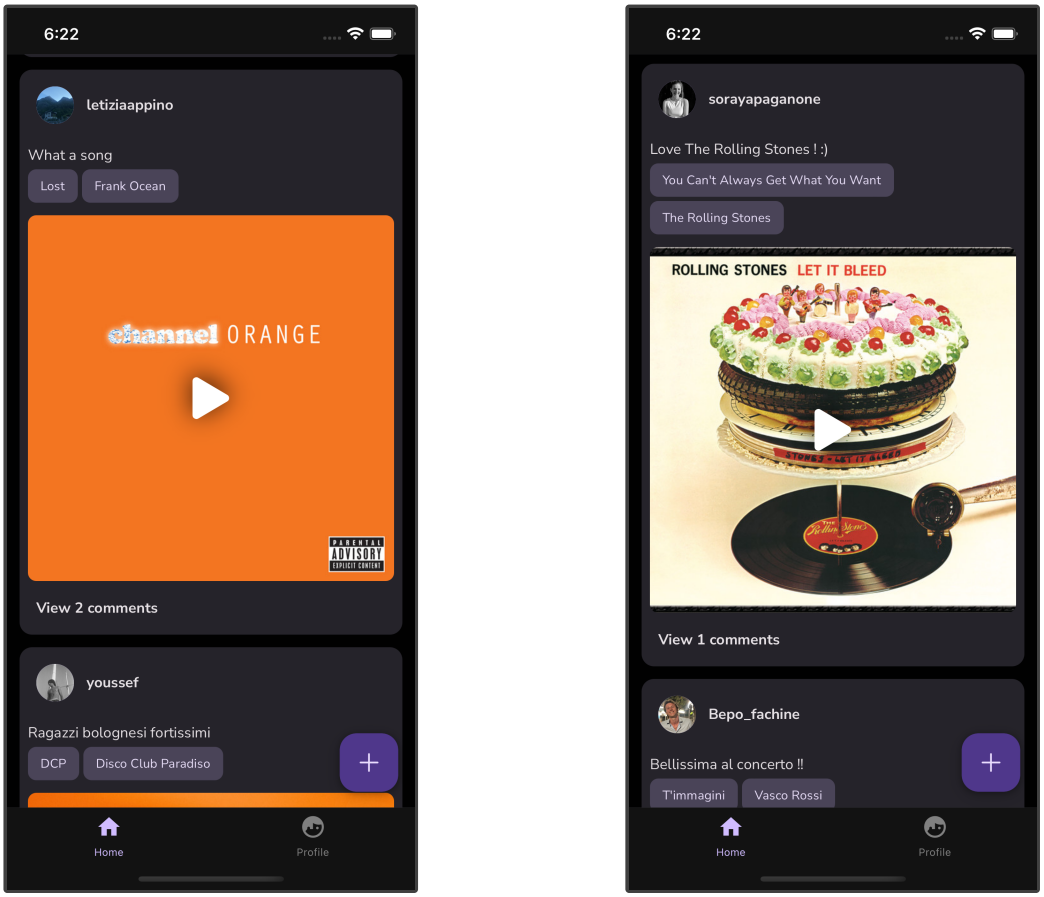

Figura 3.9: Esempio di *feed*.

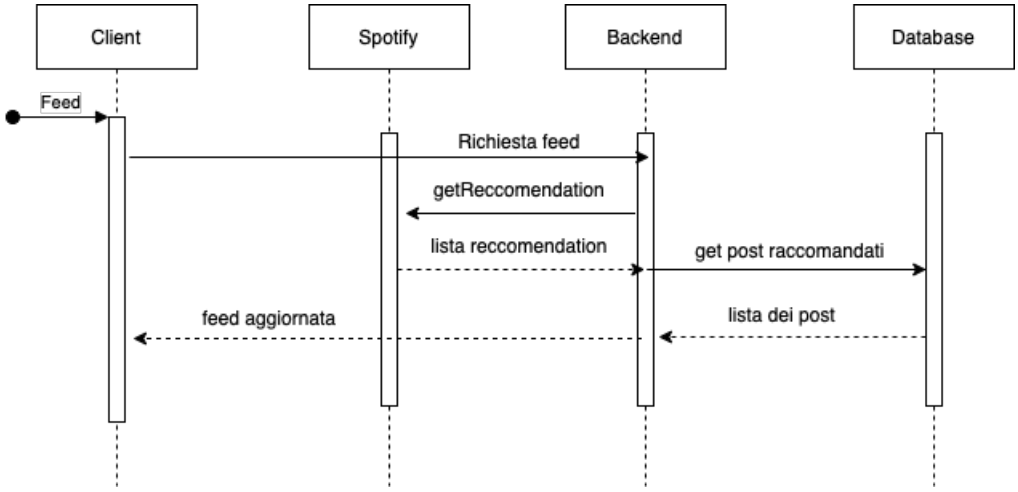

Figura 3.10: Diagramma di sequenza per il *feed*. In questo caso, l'utente effettua prima una richiesta al *backend* per avere la lista dei *post* raccomandati. In seguito, il *backend* prenderà da *Spotify* la lista delle canzoni "potenzialmente consigliabili" per quell'utente e la inserirà nel *database*. Infine aggiornerà il *feed*.

Il singolo *post* del *feed* contiene una *caption*, un *player* e l'immagine della canzone. Tramite il *player*, l'utente può riprodurre una *preview* di circa 30 secondi della canzone. L'ordine dei *post* è filtrato tramite l'algoritmo di raccomandazione e l'ordine cronologico di pubblicazione per determinare quelli più recenti e allo stesso tempo quelli più interessanti secondo i gusti dell'utente.

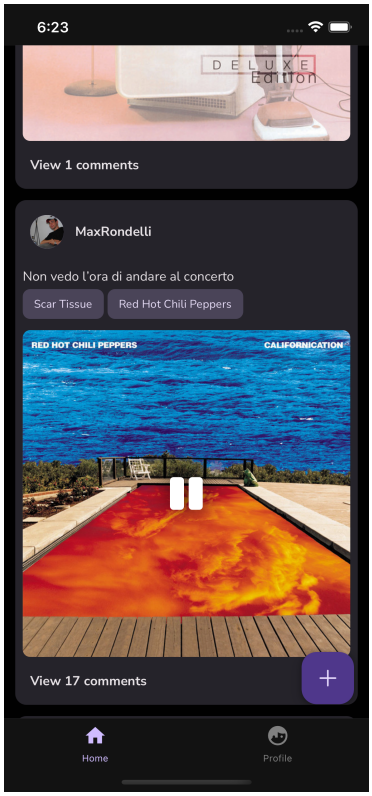

Figura 3.11: Riproduzione di una canzone tramite *player*.

Per visualizzare il singolo profilo di un altro utente, basterà cliccare sull'immagine di profilo associata all'autore che ha pubblicato un determinato *post* e si verrà indirizzati alla pagina.
Infine, gli utenti possono anche interagire con i commenti associati ai *post* cliccando sulla la scritta *"view comments"* e scrivendo un giudizio oppure un apprezzamento sulla canzone o semplicemente leggendo gli altri commenti.

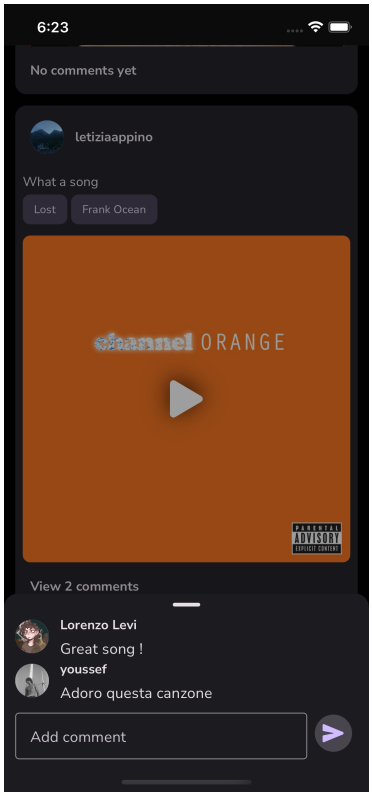

Figura 3.12: Commenti associati a un *post*.

3.2.4 Profilo altri utenti

Come specificato precedentemente, si può accedere al profilo di un altro utente se si clicca l'icona che rappresenta l'autore/creatore di un post dal *feed*. All'interno della pagina di un altro *account* è possibile osservare tutti i suoi gusti musicali e in fondo alla pagina viene mostrata una *history* delle pubblicazioni associati all'utente. Infine, è possibile decidere di "seguire" l'utente tramite il bottone *"follow"* presente sotto l'immagine del suo profilo, in questo modo si popolerà il *feed* aggiungendo i *post* relativi a quell'*account*.

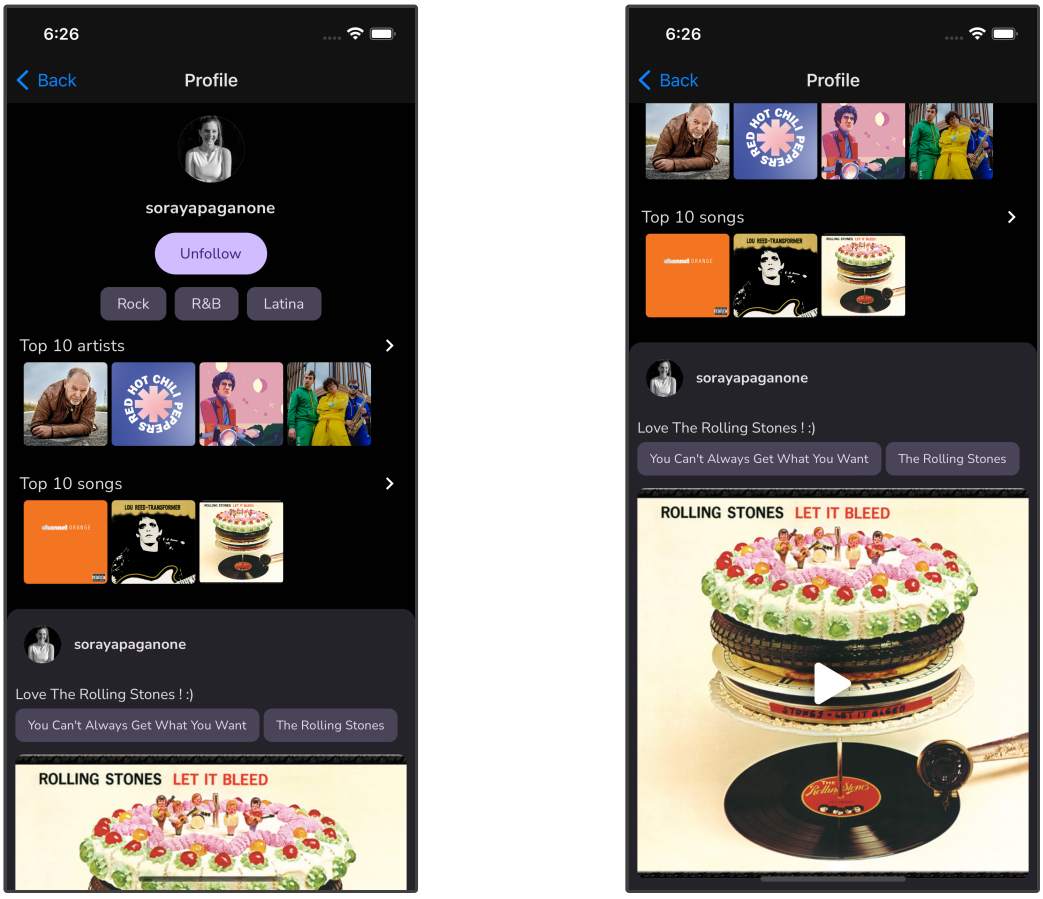

Figura 3.13: *Click* sull'immagine un altro utente.

3.2.5 Post

L'utente potrà creare un nuovo *post* dalla sezione *Home* tramite il bottone in basso a sinistra con l'icona del `"+"`.

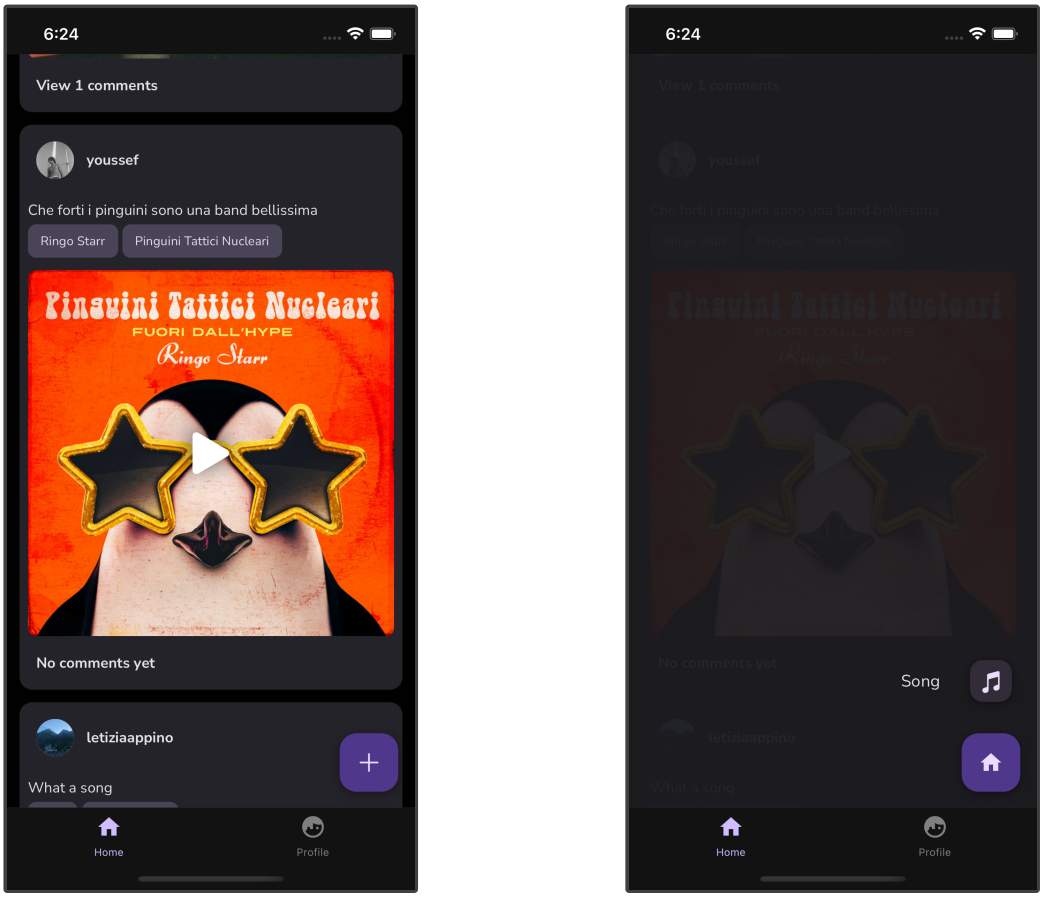

Figura 3.14: *Click* sul bottone in basso a destra per aggiungere un *post*.

In seguito potrà scegliere una canzone da *Spotify* e allegarci una *caption*. Successivamente l'applicativo si occuperà di renderizzare il *post* con l'immagine corretta relativa alla canzone da *Spotify* e di generare i *tag* corretti.

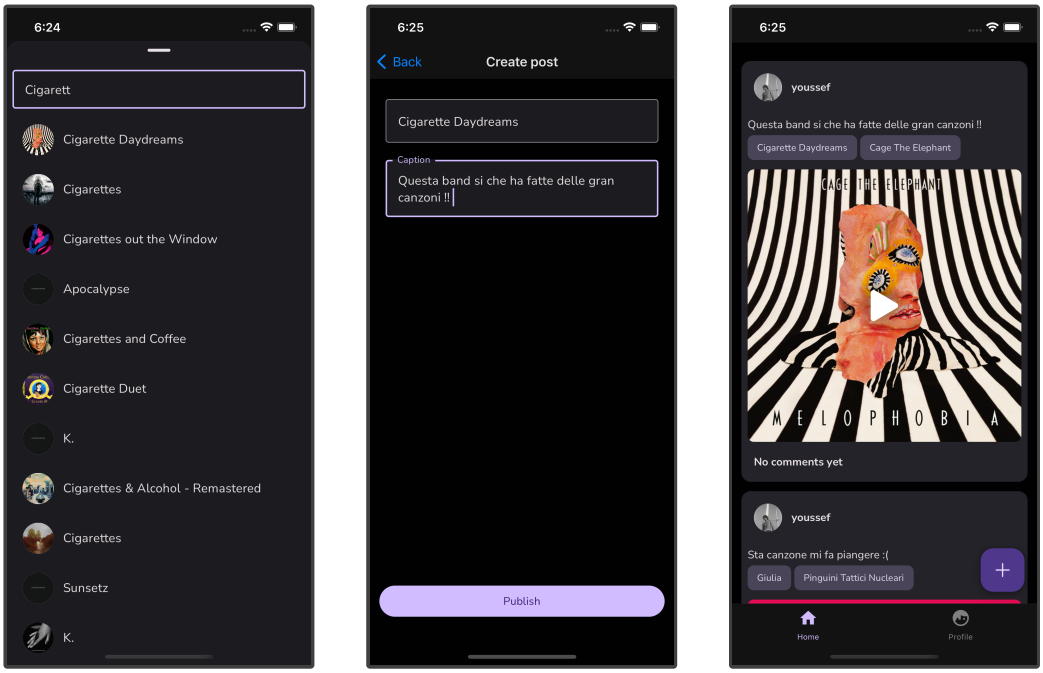

Figura 3.15: Ricerca di una canzone, *posting* e pubblicazione sul *feed*.

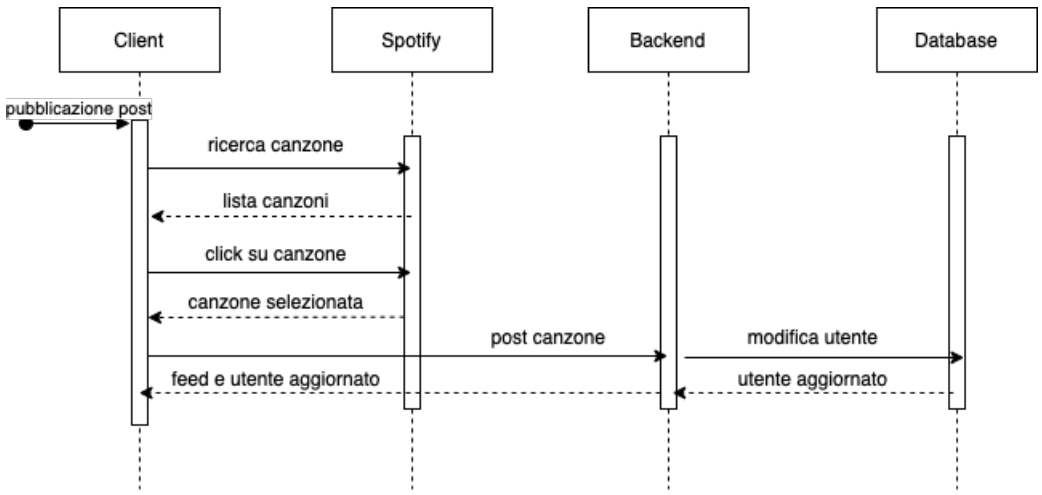

Figura 3.16: Diagramma di sequenza per pubblicazione di un *post*. Per pubblicare una canzone, l'utente effettua la richiesta su *Spotify* e sceglie quella che preferisce. Dopodiché, viene inviata al *backend* che andrà a salvarla sul *database* e ad aggiornare il *feed*.

Capitolo 4

Tecnologie e implementazione

In questo capitolo verranno mostrate e spiegate le decisioni riguardanti le tecnologie utilizzate per l'implementazione di **Mussida**. L'applicazione è stato sviluppata utilizzando il framework ***React Native*** [7] e il codice è stato implementato in ***Typescript*** [17]. Si è cercato di implementare l'applicazione secondo le *best practices* dello sviluppo *software*, in modo tale che l'aggiunta di nuove *features* e la manutenzione del codice risultino più semplici possibile. Il principale vantaggio di *React Native* è la creazione dell'interfaccia grafica tramite componenti *standalone*, ovvero indipendenti tra di loro. Quest'ultimo dettaglio permette un disaccoppiamento tra i tutti *file* del progetto e l'implementazione di componenti riutilizzabili. Grazie a questo criterio viene rispettato il *design pattern open-closed principle* di *Bertrand Meyer* [18], secondo il quale l'inserimento di nuove funzionalità deve avvenire senza modificare il codice esistente. Infine, questo meccanismo permette di individuare efficientemente gli errori e agire direttamente su di essi senza modificare file di gerarchia più bassa.

Prima di analizzare le tecnologie usate viene mostrato tecnicamente il flusso che parte dal *frontend* per effettuare le richieste alle *api* di *Spotify*[19] e quelle del server [20].

4.1 Autorizzazione tramite *Spotify*

L'applicativo è stato inizializzato con l'idea di appoggiarsi all'applicazione di *Spotify* e alle sue *api*. Infatti, al momento dell'autorizzazione l'utente

viene indirizzato all'applicazione di *Spotify*, dove, se si effettuano correttamente l'autenticazione e si accettano i termini e i servizi, viene rispedito all'applicazione e può accedere a tutte le funzionalità.

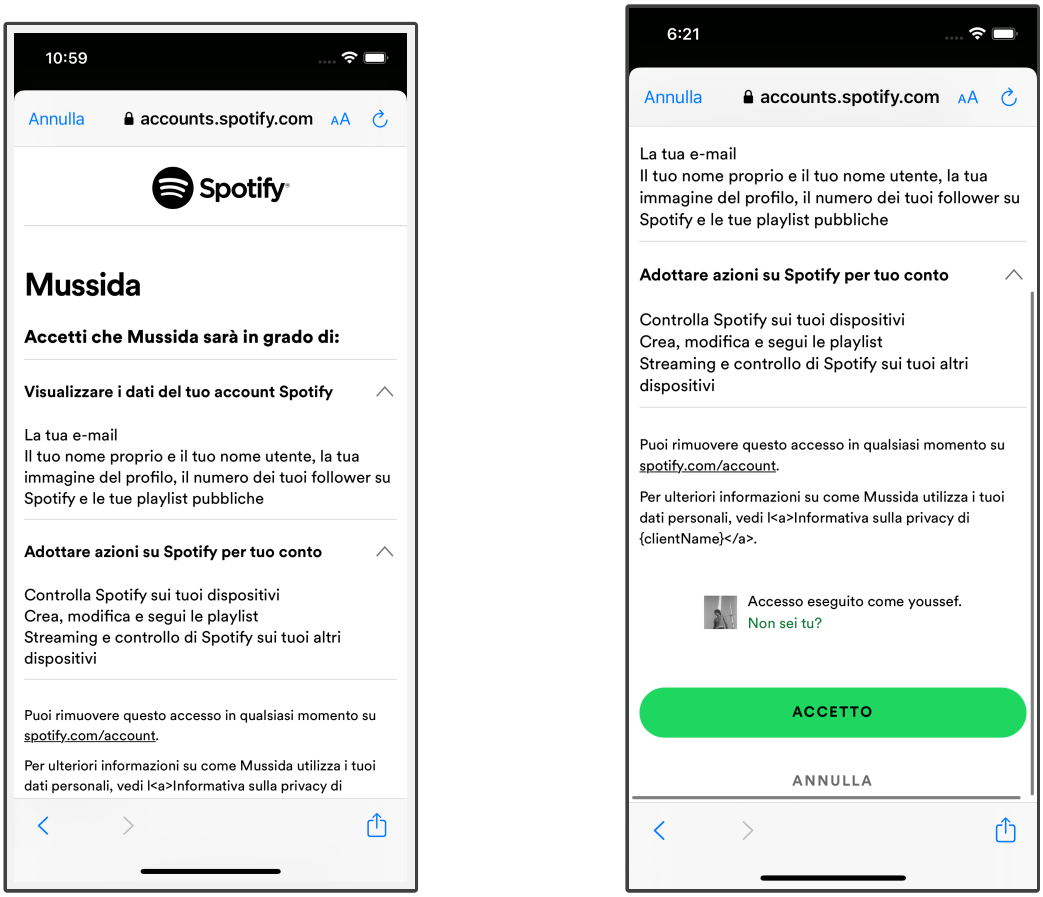

Figura 4.1: *Scopes* da accettare effettuando l'autenticazione tramite *Spotify*. Come si può osservare tramite queste due immagini all'utente vengono mostrati tutti i dati ai quali **Mussida** potrà accedere e le azioni che potrà effettuare.

4.1.1 Autorizzazione

L'autorizzazione si riferisce a quel processo mirato a garantire all'utente i permessi per i dati che verranno utilizzati e le *features* di *Spotify*. *Spotify* implementa il *framework* di autorizzazione OAuth 2.0 [21].

Figura 4.2: Flusso di autorizzazione dal *client* a *Spotify* per tramite l'applicazione per richiedere l'autorizzazione.

L'accesso alle risorse private è determinato da uno o più *scopes* [22]. Gli *scopes* permettono all' applicazione di accedere a una o più funzionalità (e.g. leggere una playlist, modificare la tua libreria oppure la riproduzione di brani) per conto dell'utente. Questi *scopes* hanno lo scopo, dunque, di determinare al momento dell'accesso i permessi per le operazioni che l'utente potrà eseguire in seguito.
Una volta ottenuta l'autorizzazione, il server genererà un *access token*, che verrà usato in seguito per effettuare le chiamate alle *api* di *Spotify*.
Lo standard OAuth 2.0. definisce quattro tipi di flussi per richiedere l'*access token*:

1. Authorization code
2. Authorization code with PKCE (utilizzato da **Mussida**)
3. Client credentials
4. Implicit grant

4.1.2 *The authorization code flow*

L'*authorization code flow* è adatta per applicazioni *long-running(e.g. web e mobile)*.
Se si sta utilizzando l'*authorization code flow* in un'applicazione mobile, o una qualsiasi altra applicazione dove non si può salvare in maniera sicura il *client secret* bisognerebbe usare l'estensione della *PKCE*. Nella figura che segue

vengono mostrate le 4 richieste principali che si devono effettuare all'interno del flusso.

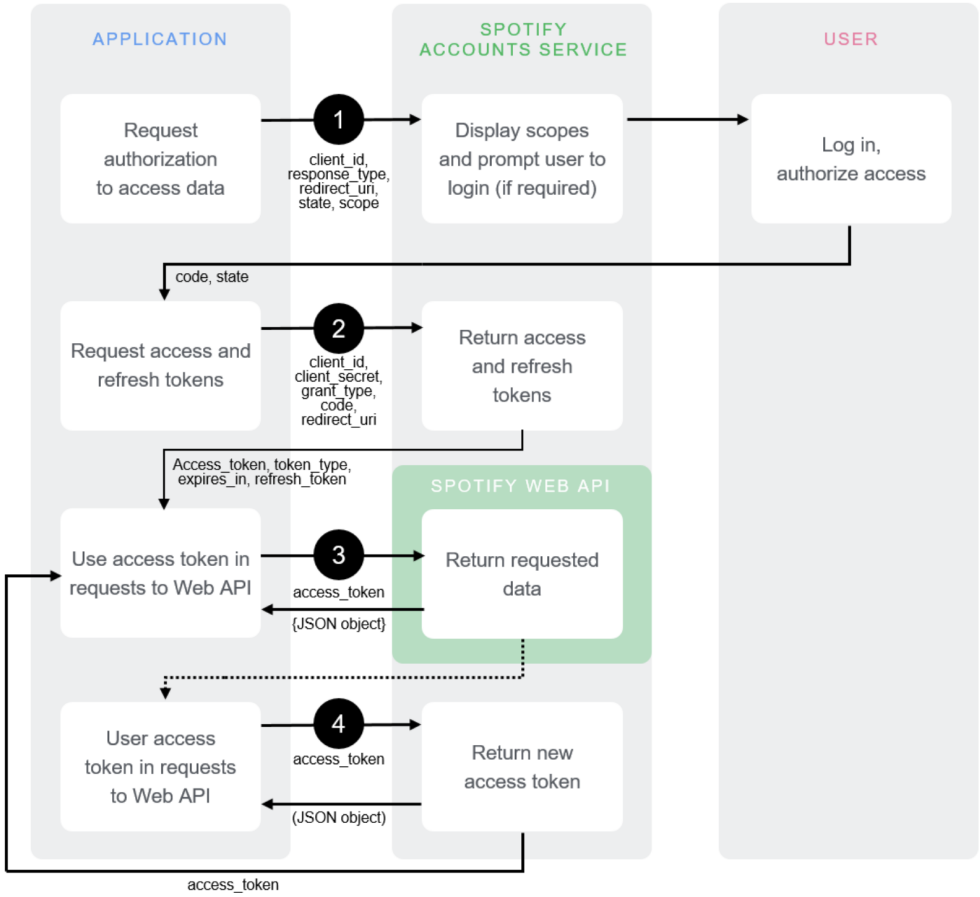

Figura 4.3: La figura mostra i passaggi per effettuare l'autorizzazione su *Spotify*. Come si può osservare avviene una richiesta tramite l'utente per l'autorizzazione, *Spotify* mostra una pagina in cui specifica *gli scopes* ai quali l'applicativo potrà accedere. Se l'autorizzazione avviene con successo, l'utente ottiene in risposta un codice che verrà scambiato con un *access token* e uno stato inerente alla richiesta. Infine verrà fatta un'altra richiesta per ottenere l'*access* e il *refresh token*. In tal modo, l'utente potrà effettuare le chiamate alle *api* di *Spotify* utilizzando il *token*. [23]

Richiesta di autorizzazione all'utente

Il primo passo da effettuare è la richiesta di autorizzazione all'utente, in questo modo la nostra applicazione deve costruire e mandare una richiesta

GET all'**/authorize** *endpoint* con i seguenti parametri:

QUERY PARAMETER	VALUE
client_id	*Required* The Client ID generated after registering your application.
response_type	*Required* Set to `code`.
redirect_uri	*Required* The URI to redirect to after the user grants or denies permission. This URI needs to have been entered in the Redirect URI allowlist that you specified when you registered your application (See the app settings guide). The value of `redirect_uri` here must exactly match one of the values you entered when you registered your application, including upper or lowercase, terminating slashes, and such.
state	*Optional, but strongly recommended* This provides protection against attacks such as cross-site request forgery. See RFC-6749.
scope	*Optional* A space-separated list of scopes.If no scopes are specified, authorization will be granted only to access publicly available information: that is, only information normally visible in the Spotify desktop, web, and mobile players.
show_dialog	*Optional* Whether or not to force the user to approve the app again if they've already done so. If `false` (default), a user who has already approved the application may be automatically redirected to the URI specified by `redirect_uri`. If `true`, the user will not be automatically redirected and will have to approve the app again.

Figura 4.4: Parametri per la richiesta di autorizzazione a *Spotify*.

Di seguito viene riportato un frammento di codice presente nel *hook "useSpotifyLogin.ts"* del *frontend* per effettuare la chiamata riportata sopra e autenticare l'utente.

```
33    const redirectUrl = makeRedirectUri({ useProxy: false });
34
35      const authRequestOptions: AuthRequestConfig = {
36        usePKCE: true,
37        responseType: ResponseType.Code,
38        clientId: clientId,
39        scopes: [
40          "user-read-email",
41          "playlist-modify-public",
42          "user-modify-playback-state",
43          "streaming",
```

```
        ],
        redirectUri: redirectUrl,
        state: state,
      };

    const authRequest = new AuthRequest(authRequestOptions);
    const authorizeResult = await authRequest.promptAsync(
  discovery, {
                useProxy: false,
    });
```

Listing 4.1: Snippet autorizzazione tramite Spotify

La risposta ottenuta dal lato server è un *token*, che verrà trasformato, in seguito, in un *access token* e un *refresh token*. In questo modo alla prima occasione in cui l'*access token* scade non ci sarà bisogno che l'utente rifaccia il *login*, ma questo sarà sostituito dal *refresh token*.

Response

Se l'utente accetta la richiesta, viene rispedito indietro all'applicazione e riceve una *callback* contenente due parametri *query*:

1. Codice, questo può essere scambiato con un *access token*.
2. Stato, il suo valore rappresenta il valore del parametro *state* della richiesta.

Se l'utente non accetta la richiesta oppure riscontriamo un errore, la risposta *query* contiene i seguenti parametri:

1. Error, la ragione per cui l'autorizzazione ha fallito(e.g. *"access denied"*).
2. State, il suo valore rappresenta il valore del parametro *state* della richiesta.

4.1.3 Richiesta dell'*access token*

Se l'utente accetta la nostra richiesta, in seguito l'applicazione sarà pronta a scambiare l'*authorization code* con l'*access token*. Se la richiesta ha successo, si ottiene come stato della risposta `200` e il JSON seguente nella *response body*:

KEY	VALUE TYPE	VALUE DESCRIPTION
access_token	string	An Access Token that can be provided in subsequent calls, for example to Spotify Web API services.
token_type	string	How the Access Token may be used: always “Bearer”.
scope	string	A space-separated list of scopes which have been granted for this `access_token`
expires_in	int	The time period (in seconds) for which the Access Token is valid.
refresh_token	string	A token that can be sent to the Spotify Accounts service in place of an authorization code. (When the access code expires, send a POST request to the Accounts service `/api/token` endpoint, but use this code in place of an authorization code. A new Access Token will be returned. A new refresh token might be returned too.)

Figura 4.5: *Response body* con risposta `200 OK` per ottenere *access* e *refresh token* da *Spotify*.

Viene riportato uno *snippet* di codice all’interno del *hook* *useSpotifyLogin.ts* che effettua la chiamata per ottenere i due tipi di *token*. Nello specifico la funzione che si occupa dello scambio è la `exchangeCodeAsync`.

```
50    const redirectUrl = makeRedirectUri({ useProxy: false });
51          const { accessToken, refreshToken } =
52              await exchangeCodeAsync(
53              {
54                  code: authorizeResult.params.code,
55                  clientId: clientId,
56                  redirectUri: redirectUrl,
57                  extraParams: {
58                      code_verifier: authRequest.codeVerifier || "
    ",
59                  },
60              },
61              discovery
62          );
```

Listing 4.2: Snippet richiesta per l'access e il refresh token

4.1.4 Richiesta di un *refresh token.*

L'*access token* è deliberatamente impostato per scadere dopo un breve periodo, dopodiché viene sostituito dal *refresh token* ottenuto dallo scambio dell'*authorization code*.

4.2 Autenticazione su Mussida

4.2.1 Chiamate

All'interno di **Mussida** abbiamo principalmente a che fare con due tipi di chiamate:

1. Chiamate dirette alle *api* di *Spotify*.
   - Login con il metodo *PKCE* sulle *api* di *Spotify* e viene usato il *token* che mi viene restituito per le chiamate su *Spotify* e l'autenticazione al mio *server*.
2. Chiamate al nostro *server*, che possono essere di due tipi.
   - Chiamate pubbliche, non sono autenticato in nessun modo.
   - Chiamate private, richiedono autenticazione e autorizzazione.

4.2.2 *Login* al *server* dell'applicativo

(a) L'applicativo richiede l'*access token* con il metodo *PKCE* da *Spotify*.

(b) L'applicativo chiama l'*endpoint* `/users/login` del nostro server passando nel body della chiamata il *token* di *Spotify*.

```
66    usersApi
67        .queryUserRouterLogin({
68            spotifyToken: accessToken,
69         })
```

```
70          .then((res) => {
71               setBackendToken(res.data.token);
72          })
73          .catch((err) => {
74              console.log(err);
75          });
```

Listing 4.3: Snippet autenticazione sul server dell'applicativo con il token di Spotify.

(c) La chiamata arriva al *server* e `/users/login` opera nel seguente modo:

i. Si prende l'*access token* che ha nel *body*.

ii. Effettua una richiesta a `/me` delle *api* di *Spotify* per ottenere i dettagli relativi all'utente corrispondenti al *token*.

iii. Se la chiamata fallisce, restituisce un errore. Altrimenti verifica se esiste un utente nel nostro *database* corrispondente, se non esiste lo crea e lo inserisce.

```
50 export const userRouter = createTRPCRouter({
51   login: publicProcedure
52     .meta({
53       openapi: {
54         method: "POST",
55         path: "/users/login",
56         tags: ["users"],
57       },
58     })
59     .input(
60       z.object({
61         spotifyToken: z.string(),
62       })
63     )
64     .output(z.object({ token: z.string() }))
65     .query(async ({ input }) => {
66       const spotifyToken = input.spotifyToken;
67       try {
68         const spotifyUser = await fetch("https://api.spotify.com
     /v1/me", {
69           headers: {
70             Authorization: `Bearer ${spotifyToken}`,
71             "Content-Type": "application/json",
72           },
```

```
73          })
74            .then((res) => res.json())
75            .then((res) => {
76              if (res.error) {
77                throw new Error("spotify error");
78              }
79              return res as SpotifyUser;
80            });
81
82          let user = await prisma.user.findUnique({
83            where: {
84              id: spotifyUser.id,
85            },
86          });
87
88          if (!user) {
89            user = await prisma.user.create({
90              data: { id: spotifyUser.id },
91            });
92          }
93
94          const accessToken = generateAccessToken(user.id);
95
96          return {
97            token: accessToken,
98          };
99        } catch (e) {
100         console.log(e);
101         throw new TRPCError({
102           code: "INTERNAL_SERVER_ERROR",
103           message: "Unable to fetch user from spotify",
104           cause: e,
105         });
106       }
107     }),
```

Listing 4.4: Chiamata lato backend che controlla token di Spotify e con esso genera token per le chiamate alle api del server dell'applicazione

iv. Una volta che abbiamo l'utente nel nostro *database*, il *server* genera un *JWT* contente l'*id* dell'utente a *database* nel seguente modo:

A. Deve esserci un *secret* salvato lato *server* che non è nient'altro che una stringa *random*.

B. Si mischia il *secret* con il contenuto che vogliamo inserire nel *JWT*(in questo caso l'id dell'utente) e si *"hasha"* (nel nostro caso usiamo *sha256*).

```
50 const jwt = require("jsonwebtoken");
51
52 export function generateAccessToken(userId: string) {
53     return jwt.sign({ userId }, process.env.TOKEN_SECRET, {
       expiresIn: "24h" });
54 }
```

Listing 4.5: Snippet generazione del token lato backend

C. Con questa firma si crea un oggetto che contiene l'*hash* e il *body* effettivo del *JWT*(`({userId: string})`).

D. Si offusca il *JWT* tramite *encoding base64*.

v. La chiamata risponde all'utente con il *token* creato che sarà valido, nel nostro caso per 24h.

4.2.3 Autenticazione delle chiamate al nostro *database*

. Tutte le chiamate autenticate avranno bisogno del *token* creato precedentemente da inserire nell'*header "authorization"* con metodo *"bearer"*. Per verificare che l'utente sia *"loggato"* il *server* procede nel seguente modo:

1. Si assicura che gli sia arrivato l'*header* di *authorization* e ne prende il *token*. Se non c'è il *token* torna `401`.

2. Verifica che il *token* non sia stato manipolato tramite la funzione *verify* di *JWT* [24] come riportato nel codice seguente:

```
50 const validateAuthToken = () => {
51         const token = req.headers.authorization;
52         if (!token)
53             throw new TRPCError({
54                 code: "UNAUTHORIZED",
55                 message: "Missing auth token",
56             });
57
58         try {
59             const actualToken = token.replace("Bearer ", "");
60             return jwt.verify(
61                 actualToken,
```

```
62                  process.env.TOKEN_SECRET as string
63              ) as {
64                  userId: string;
65                  iat: number;
66                  exp: number;
67              };
68          } catch (e) {
69              throw new TRPCError({
70                  code: "UNAUTHORIZED",
71                  message: "Invalid auth token",
72              });
73          }
74      };
75
```

Listing 4.6: Snippet funzione di validazione del token lato backend.

La funzione opera come segue:

(a) Si prende l'*hash* contenuto nel *JWT*.

(b) Si crea un nuovo *hash*, sempre con lo stesso algoritmo, mischiando il nostro *secret* lato *server* con il *body* del *JWT*.

(c) Si comparano l'*hash* appena creato con quello contenuto nel *JWT*. Se sono diversi il *token* è stato manomesso.

(d) Se il *token* è stato manomesso la chiamata risponde con `401`, altrimenti va avanti nella sua esecuzione.

Tutte le chiamate autenticate passano per il *middleware*, che va a eseguire la funzione di validazione del *token* e va a popolare il *context* disponibile alle chiamate con il valore del *body* del nostro *JWT*.

4.3 *Frontend*

Il *frontend* è stato sviluppato con l'intento di rendere l'architettura dei *file* ordinata e ben strutturata, cercando di separare nettamente logica e grafica. Il *directory tree* a partire dalla cartella *src* è il seguente:

- **Pages**, all'interno della quale sono presenti tutte le rotte dell'applicazione e la logica di funzionamento dei componenti. All'interno delle pagine l'aggregazione dei componenti forma i *layouts* dell'*app*.

- **Components**, all'interno di questa cartella risiedono i singoli componenti dell'applicazione. Sono implementati senza nessuna logica al loro interno, ma questa li viene "iniettata" [1] dalla pagina che li utilizza.
- **Hooks**, questa cartella racchiude le funzionalità che sono utilizzate più di una volta all'interno dell'applicazione, in modo tale che siano riutilizzabili senza la duplicazione di codice.
- **Utils**, dentro la seguente cartella sono presenti tutte le variabili globali e le costanti che vengono utilizzati all interno di **Mussida**.

4.3.1 Navigazione

La navigazione tra le pagine è stata sviluppata usando la libreria ***React Native Navigation*** [25], in particolare è stato adottato come navigatore principale lo ***Stack Navigator***. Quest'ultima è una funzione che ritorna un oggetto contenente due proprietà: *Screen* e *Navigator*. Queste due componenti sono entrambi usate per configurare il navigatore. Il *Navigator* deve contenere gli elementi dello *Screen* e i suoi figli per definire la configurazione delle *routes*[2]. Il nome della funzione è `createNativeStackNavigator`. L'altra componente, che si occupa di impostare e gestire l'albero della navigazione, è il *Navigation Container*. Questo componente "avvolge" tutta la struttura di navigazione. Solitamente viene renderizzato alla radice dell' applicazione, come si può osservare nel *file* App.tsx. L'esempio che segue mostra come vengono implementati il *container* e lo *stack navigator*.

[1] Con il termine "iniettare" si intende la pratica della programmazione *software* attraverso la quale si introduce un dipendenza a *runtime* tra due componenti. In questo caso viene passato come *input* al componente grafico la logica delle *callback* da eseguire per rispondere a determinati eventi.

[2] Con il termine *routes* si intendono le rotte di destinazione della navigazione, grazie a esse è possibile manovrare il flusso di un'applicazione in esecuzione attraverso la richiesta di specifici path

```
<NavigationContainer theme={DarkTheme}>
    <Stack.Navigator>
      {token == null ? (
        <Stack.Screen
          options={{
            header: () => null,
          }}
          name="Home"
          component={Login}
        />
      ) : (
        <>
          <Stack.Screen
            name="RootPage"
            component={RootPage}
            options={{
              header: () => null,
            }}
          />
          <Stack.Screen
            name="CreatePost"
            options={{ title: "Create post" }}
            component={CreatePost}
          />
        </>
      )}
    </Stack.Navigator>
</NavigationContainer>
```

Listing 4.7: Snippet del navigation container.

Come si può osservare nel frammento di codice precedente, lo *stack navigator* può contenere della logica, nello specifico, come mostrato nell'esempio precedente, si effettua un controllo sul *token* dell'utente per potergli assegnare il permesso o meno di accedere a determinate sezioni dell'*app*. Per specificare le rotte di navigazione all'interno dei componenti, serve importare l'*hook* `useNavigation`, il quale offre l'accesso all'oggetto `navigation`. Questo espone l'*api navigate*, la quale dato in *input* il *path* di una rotta si occupa ti reindirizzarti a quella.

```
import { useNavigation } from "@react-navigation/native";
    ...
   const navigation = useNavigation();
```

```
    ...
    onPress: () =>
     navigation.navigate("Home");
```

Listing 4.8: Snippet esempio di navigazione tra due pagine.

La funzione `navigate` viene implementata come *callback* di un determinato evento, in questo caso il *click* di un per arrivare alla *home*.

4.3.2 Notifiche

L'applicativo utilizza il sistema di notifiche offerto dal *package expo-notifications* [26], che offre un *api* per inviare e ricevere notifiche all'interno dell'applicazione. Quando un utente si autentica per la prima volta gli viene richiesto il permesso di ricevere notifiche, in caso accetti riceverà una notifica ogni volta che un utente pubblicherà un *post*. Per permettere l'invio delle notifiche viene salvato un *expo-token* lato *backend* associato a un determinato utente, lato *frontend* invece, quando un utente pubblica un post viene inviata la notifica a tutti gli *user* con quel *token*. Il codice che gestisce questo processo utilizza un *hook* riportato nel frammento seguente.

```
export function useExpoNotification() {

  const usersApi = useApi(UsersApi)
  const token = useAtomValue(backendTokenAtom)
  const hasRegisteredTokenRef = useRef(false)

  useEffect(() => {
    if(hasRegisteredTokenRef.current || !token) return
    registerForPushNotificationsAsync().then(async (token) => {
      hasRegisteredTokenRef.current = true
      if(token){
        await usersApi.queryUserRouterUpdateExpoToken({expoToken
    : token})
      }
    }).catch((err) => {
      Toast.show({
        type: "error",
      })
    })
  }, [token]);
}
```

Listing 4.9: Hook di registrazione delle notifiche

Come si può osservare nello *snippet* la funzione `registerForPushNotificationAsync` chiede all'utente il permesso di inviare le notifiche. In seguito date le politiche del sistema operativo una volta che l'utente ha scelto se accettare o meno i permessi, in automatico *expo* non li chiede più sino a quando non c'è un ripristino delle impostazioni delle notifiche. Infine, una volta ottenuto il *token* da quest'ultima funzione si fa la chiamata al *backend* per *settare* l'*expoToken* che servirà per comprendere a quali utenti inviare la notifica. Qui di seguito viene riportata la funzione che si occupa di renderizzare l'*alert* all'utente per richiedere il permesso all'utente e ritorna un *token* se quest'ultimo accetta.

```
async function registerForPushNotificationsAsync() {
  let token;
  if (true) {
    const { status: existingStatus } =
      await Notifications.getPermissionsAsync();
    let finalStatus = existingStatus;
    if (existingStatus !== "granted") {
      const { status } = await Notifications.
   requestPermissionsAsync();
      finalStatus = status;
    }
    if (finalStatus !== "granted") {
      alert("Failed to get push token for push notification!");
      return;
    }
    token = (await Notifications.getExpoPushTokenAsync()).data;
  } else {
    alert("Must use physical device for Push Notifications");
  }
  return token;
}
```

Listing 4.10: Snippet richiesta di permesso per le notifiche.

4.3.3 *React Hooks*

All'interno dell'applicazione vengono usati gli *hooks*, nonché una novità introdotta da *React 16.8* [27]. Essi permettono di usare determinate funzionalità

di *React* come lo *state*, il *context* e l'*effect* senza dover riscrivere una classe e sono retrocompatibili. Si tratta di una componente funzionale necessario per il codice e per la logica del progetto perché non contengono alcun comportamento che possa rompere le funzionalità già esistenti, e aggiungono praticità e indipendenza tra i componenti riutilizzabili. I due hook più utilizzati all'interno di **Mussida** sono `useState()` e `useEffect()`:

`useState()`

La seguente funzione permette all'utente di assegnare un valore a una variabile e allo stesso tempo dichiarare il metodo per modificarla. L'unico parametro di questo di questa funzione è lo stato iniziale, e a differenza delle classi lo *state* deve essere un oggetto. Ritorna una coppia di valori: lo stato corrente e una funzione che lo aggiorna.

`useEffect()`

Con lo `useEffect` è possibile determinare una funzione e in base a quale evento deve essere eseguita. La seguente funzione consente di gestire il ciclo di vita dei componenti. Il primo argomento è una *callback*, che viene eseguita di default dopo ogni *rendering*. Il secondo parametro è un vettore di dipendenze che indica all'*hook* di essere eseguito solo se queste dipendenze cambiano stato (valore).

4.3.4 *Query* e *Mutation*

Tanstack query[28] è una libreria utilizzata per il *data-fetching* all'interno di **Mussida**. Si occupa di effettuare tutte le operazioni di *fetching, caching, synchronizing, updating server state* all'interno dell'applicativo. La libreria, viene utilizzata per effettuare tutte le chiamate necessarie per interagire con *Spotify* e il *backend* dell'applicativo.

Query

All'interno dell'applicazione la *query* è una dipendenza dichiarativa su una risorsa asincrona di dati legati a una chiave univoca. La query viene usata con qualsiasi metodo di "promessa" per *fetching* di dati dal server.
Per comporre una query si chiama la funzione `useQuery` con in input:

1. Una chiave univoca per la *query*.
2. Una funzione che ritorna una promessa che:
   - risolve i dati, oppure
   - ritorna l'errore

Di seguito verrà riportato un esempio di *query* in cui dato l'*id* di una canzone o di un'artista da *Spotify*, abbiamo accesso alla foto della canzone o dell'artista nel *body*.

```
75 const { data } = useQuery({
76     queryKey: ["favourite", imageId],
77     queryFn: () => {
78       if (variant === "artists") {
79         return spotifyApi.getArtist(imageId).then((res) => {
80           return res.body.images?.[0];
81         });
82       } else {
83         return spotifyApi.getTrack(imageId).then((res) => {
84           return res.body.album.images?.[0];
85         });
86       }
87     },
88   });
```

Listing 4.11: Snippet esempio query per *fetch* di immagini da *Spotify*.

Mutation

La *mutation*, al contrario della *query*, è una dipendenza imperativa, che restituisce il dato risultante da una promessa, ritorna una funzione che, se invocata, esegue una funzione asincrona tramite il metodo `mutate`. Viene riportato un frammento di codice per comprendere a livello programmatico l'implementazione.

```
75 return useMutation({
76     mutationFn: () => {
77         return isFollowing
78             ? usersApi.queryUserRouterUnfollowUser({ id: userId
    })
79             : usersApi.queryUserRouterFollowUser({ id: userId })
    ;
```

```
80      },
81      onSuccess: () => {
82          queryClient.setQueryData(["isFollowing", userId], !
    isFollowing);
83          queryClient.invalidateQueries(["isFollowing", userId]);
84          queryClient.invalidateQueries(getRecommendPostsQueryKey)
    ;
85      },
86      onError: () => {
87          Toast.show({
88              type: "error",
89          });
90      },
91  });
```

Listing 4.12: Snippet esempio mutation per ”seguire” un utente.

Come si può notare nella seguente *mutation*, si effettua un controllo per verificare che l'utente sia già seguito o meno, e si va a eseguire la chiamata necessaria.

4.4 Backend

4.4.1 Tecnologie

Il *backend* dell'applicativo è stato implementato utilizzando il *framework* **Express** [29], idoneo alla creazione di *api* di routing e all'impostazione di *middleware* per rispondere alle richieste *HTTP*, fornendo semplici meccanismi di *debugging* e una rapida integrazione con vari motori di *templating*. Il *server* si mette in ascolto su una determinata porta e tutte le richieste su quella porta vengono gestite da esso.
All'interno del *backend* viene sfruttata la libreria **Trpc** [30] che permette di costruire *api typesafe* [3] senza l'uso di schemi o generazione di codice. Permette la condivisione di ”tipi” tra il *client* e il *server*, infatti all'interno di **Mussida** il *frontend* comunica direttamente con i *router* messi a disposizione da *Trpc*. È inoltre presente un *adapter* che si occupa di reindirizzare le chiamate ad *Express* per poi inviarle a un *handler trpc* per un funzionamento tale a quello di un *server* tradizionale.

[3] In informatica la type safety (sicurezza rispetto ai tipi) è la misura con cui un linguaggio di programmazione previene o avvisa rispetto agli errori di tipo.

4.4.2 Database

Per l'implementazione dell'applicazione è stato utilizzato **MongoDB** [31] come *database*. È stato scelto un *database* non relazionale, per garantire scalabilità e flessibilità. L'architettura del progetto è stata infatti studiata per poter sempre introdurre nuove funzionalità, proprietà e attributi. *MonogoDB* permette di salvare i dati in documenti flessibili *JSON-like*, e permette di scrivere *query*, indicizzare e aggregare i dati.

All'interno viene integrata Prisma, una libreria *server-side* che serve a leggere e scrivere dati nel *database* in maniera efficiente, intuitiva e sicura. Si tratta di un *ORM (Object Relational Mapping)*, ovvero un *layer* che si pone sopra il *database* per permetterci di manipolarlo, e allo stesso tempo ci permette di cambiarlo nel momento in cui vogliamo perché è compatibile con più *db*. Nello specifico con *Prisma*[32] puoi definire i tuoi modelli nel *Prisma schema* che funge da singola fonte di verità per il nostro *database schema* e i modelli che usiamo nel linguaggio di programmazione. Nel codice dell'applicativo, si può usare *Prisma Client* per leggere e scrivere dati nel *database* in maniera *type-safe* senza il problema di interagire con modelli complessi.

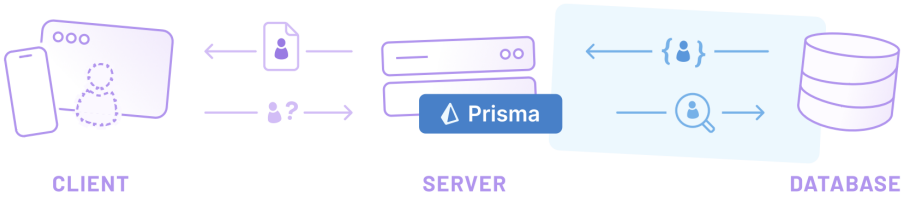

Figura 4.6: Lo schema di prisma

A livello programmatico per creare i modelli sul *database*, vengono prima definiti all'interno di un *file* nominato *schema.prisma* nel *backend*. In seguito, sono generati con il comando: `npx prisma generate`. Nella figura A.1 dell'appendice viene mostrato il digramma *ER* del *database* generato con il pacchetto di *Prisma prisma-dbml-generator*[33].

4.4.3 Deployment del backend

Il *deployment* dell'applicativo è stato effettuato su **Cloud Run** [34], una piattaforma di *computing gestita* che ha il forte vantaggio di avere un architettura *serverless* che permette agli sviluppatori la creazione di *server* senza la loro gestione. Questo significa che le attività di *provisioning*, manutenzione e scalabilità dell'infrastruttura server vengono gestite da un *provider* di servizi *cloud*. Le *app serverless* vengono distribuite in *container* che vengono avviati al momento della chiamata, quindi *cloud run* riceve la chiamata e solo in quel momento avvia il processo per effettuare le chiamate *backend*.

4.4.4 Openapi

All'interno del nostro applicativo si usa lo *standard* **OpenApi** [35] che funge da intermediario tra il *server* e il *front-end*. È un interfaccia per interagire con le *api http*, di cui il principale vantaggio è quello di avere una documentazione completa e rapida da realizzare. Considerato comunque un numero non ridotto di *api*, con *OpenApi* abbiamo una descrizione dell'*endpoint* munita di:

- metodo di invocazione
- parametri che accetta in ingresso
- esempio di ciò che bisogna fornirgli
- descrizione del comportamento
- risposta

All'interno del progetto i parametri riportati sopra vengono generati tramite il pacchetto *trpc-openapi* [36] che dalla definizione delle nostre rotte genera il *file yaml* per definire la struttura del *backend*. Questo file poi viene messo a disposizione in determinato *endpoint* nel backend, nello specifico *openapi-documents.ts*, e riporta il *json* rappresentante lo *yaml*.

```
0 import { generateOpenApiDocument } from "trpc-openapi";
1 import { appRouter } from "../routers/root";
2
3 // Generate OpenAPI schema document
4 export const openApiDocument = generateOpenApiDocument(appRouter
    , {
```

```
  title: "tRPC OpenAPI",
  version: "1.0.0",
  baseUrl: `http://localhost:${process.env.PORT || 8080}/api`,
});
```

Listing 4.13: Snippet backend per generazione json dell'openapi.

Lato *frontend*, tramite un altro pacchetto *openapitools/openapi-generator-cli*[37] si leggere quest'ultimo *file json* e si generano le classi delle *api* in *Typescript*.

4.4.5 Struttura

La struttura del *backend* viene strutturata a partire dal *file server.ts*, all'interno del quale il *server* si mette in ascolto per ricevere le chiamate. Qui di seguito viene riportato un frammento di codice della *file* per comprendere meglio l'implementazione.

```
async function main() {
    // express implementation
    await prisma.$connect();

    const app = express();
    const port = process.env.PORT || 8080;
    app.use((req, _res, next) => {
        // request logger
        console.log("       ", req.method, req.path, req.body ??
    req.query);

        next();
    });

    app.use(
        "/api/trpc",
        createExpressMiddleware({
            router: appRouter,
            createContext,
        })
    );
    app.use(
        "/api",
        createOpenApiExpressMiddleware({ router: appRouter,
   createContext })
    );
```

```
24
25      // Serve Swagger UI with our OpenAPI schema
26      app.use("/", swaggerUi.serve);
27      app.get("/", swaggerUi.setup(openApiDocument));
28      app.get("/api-docs", (_req, res) => res.json(openApiDocument
        ));
29
30      // app.get("/", (_req, res) => res.json({ status: 200,
        message: "ok" }));
31      app.listen(port, () => {
32          console.log("listening on port 2021");
33      });
34  }
```

Listing 4.14: Snippet server express.

Come si può osservare alla riga 2, viene effettuata una connessione al *database*, in seguito viene creato il *server express* alla riga 4. Nel codice vengono dichiarati anche i *middleware*, questi hanno la funzionalità di fungere da *handler* per tutte le chiamate. Se i *middleware* non possiedono nessun *path*, allora vengono effettuati per tutte le richieste. Tramite la funzione `next` chiamano, se presente, il *middleware* successivo. Nel codice è presente un *middleware* alla riga 6 che ha lo scopo di stampare a *console* le richieste e le informazioni correlate per aiutare lo sviluppatore nel *debugging*. Il secondo *middleware* riga 13, è responsabile della creazione del collegamento tra il *server* e *trpc*.

A livello del *file server.ts* si trova la cartella *src* all'interno del quale si trovano i *router trpc*.

4.4.6 *Router*

I *router* gestiscono l'implementazione degli *handler* per gli *endpoints* chiamati dal *frontend*. All'interno della cartella *routers* del *backend*, è presente il router principale dove vengono dichiarati gli altri differenziati per entità. Nello specifico questo si può osservare nel *file roots.ts* che sono presenti due *router*: uno riguardante le chiamate inerenti all'utente e l'altro ai *post*.

```
export const appRouter = createTRPCRouter({
  userRouter: userRouter,
  postsRouter: postsRouter,
});
```

Listing 4.15: Snippet creazione dei router

Le *procedure* poi all'interno dei due *router* possono essere di due tipi:

- *publicProcedure*, sono gli *endpoints* per i quali non bisogna essere autenticati in nessun modo.
- *authProcedure*, sono gli *endpoints* per i quali bisogna essere autenticati. Per queste è presente un *middleware* che controlla l'esistenza dell'utente all'interno del *database*, e con una funzione controlla la validità del *token*. Infine restituisce l'utente all'*handler* della *procedure*. Qui di seguito viene riportato un frammento di codice dove si associa il *middleware* a tutte le *authProcedure*.

```
const isAuthMiddleware = t.middleware(({ next, ctx }) => {
    const accessToken = ctx.validateAuthToken();

    return next({
        ctx: {
            ...ctx,
            accessToken,
            },
        });
});
export const authProcedure = t.procedure.use(isAuthMiddleware);

```

Listing 4.16: Snippet middleware per le richieste con autenticazione

Come si può osservare al suo interno nella riga 1 viene richiamata la funzione `validateToken`, che si occupa tramite la funzione `verify` della libreria *jwt* di validare il *token*. Viene riportata la funzione nel frammento di codice che segue.

```
const validateAuthToken = () => {
    const token = req.headers.authorization;
    if (!token)
        throw new TRPCError({
            code: "UNAUTHORIZED",
            message: "Missing auth token",
        });

    try {
        const actualToken = token.replace("Bearer ", "");
        return jwt.verify(
            actualToken,
            process.env.TOKEN_SECRET as string
        ) as {
            userId: string;
            iat: number;
            exp: number;
        };
    } catch (e) {
        throw new TRPCError({
            code: "UNAUTHORIZED",
            message: "Invalid auth token",
        });
    }
};

```

Listing 4.17: Snippet funzione di validazione del token.

Tutte le *procedure* sono caratterizzate dai seguenti *tag*:

- `meta`, che si occupa di creare e generare la tipizzazione *openapi*.
- `input`, che definisce i parametri in *input* e li valida tramite la libreria di *zod* [38].
- `output`, che definisce il tipo di *output* e lo valida *zod*.
- query, all'interno del quale viene definita la funzione quando avviene una richiesta a quell'*endpoint*.

Vengono riportati in seguito due frammenti di codice rappresentati una *publicProcedure* e una *authProcedure*.

```
login: publicProcedure
        .meta({
            openapi: {
                method: "POST",
                path: "/users/login",
                tags: ["users"],
            },
        })
        .input(
            z.object({
                spotifyToken: z.string(),
            })
        )
        .output(z.object({ token: z.string() }))
        .query(async ({ input }) => {
            const spotifyToken = input.spotifyToken;
            try {
                const spotifyUser = await fetch(
                    "https://api.spotify.com/v1/me",
                    {
                        headers: {
                            Authorization: `Bearer ${
   spotifyToken}`,
                            "Content-Type": "application/json",
                        },
                    }
                )
                    .then((res) => res.json())
                    .then((res) => {
                        if (res.error) {
                            throw new Error("spotify error");
                        }
                        return res as SpotifyUser;
                    });

                let user = await prisma.user.findUnique({
                    where: {
                        id: spotifyUser.id,
                    },
                });

                if (!user) {
                    user = await prisma.user.create({
                        data: { id: spotifyUser.id },
```

```
43                      });
44                  }
45
46                  const accessToken = generateAccessToken(user.id)
    ;
47                  return {
48                      token: accessToken,
49                  };
50              } catch (e) {
51                  throw new TRPCError({
52                      code: "INTERNAL_SERVER_ERROR",
53                  });
54              }
55          }),
56
```

Listing 4.18: Snippet di una *publicProcedure*

Come si può osservare nel frammento di codice precedente si nota che, come definito nell'attributo `meta`, la seguente funzione appartiene all'*api* dello *user* di cui il metodo *http* [39] è *post*. L'*endpoint* si aspetta in *input* il *token* di *Spotify*, e restituisce in *output* quello per autenticarsi sul *server* dell'*app*. Per quanto riguarda le *authProcedeure* la sintassi è la medesima, ma l'istanza con cui viene dichiarata non è più *publicProcedure*, ma viene usata *authProcedeure* come si può leggere nello *snippet* che segue:

```
0 followUser: authProcedure
1 .meta({
2               openapi: {
3                   method: "PUT",
4                   path: "/user/follow/:id",
5                   protect: true,
6                   tags: ["users"],
7               },
8           })
9   .......
```

Listing 4.19: Snippet dichiarazione di un *authProcedure*

4.4.7 Algoritmo di raccomandazione

Per la creazione dell'algoritmo di raccomandazione ci si basa sull'utilizzo di un *api* di *Spotify* [40] che, dati in *input* le preferenze impostate dall'utente

restituisce un *range* di canzoni affini ai suoi gusti musicali.

Con il risultato ottenuto da *Spotify* andiamo a cercare nel *database* tutti i *post* che contengono un riferimento a una delle canzoni presenti nella lista.

Oltre alle canzoni raccomandate, vengono utilizzati come criterio di filtraggio anche:

- artisti preferiti
- canzoni preferite
- utenti seguiti

Nel caso in cui l'utente fosse nuovo alla piattaforma vengono mostrati nel *feed* gli ultimi 100 *post*. Il *feed* è sempre popolato in ordine cronologico, e per fare si che non cambi troppo rapidamente i suggerimenti di *Spotify* vengono generati al più una volta al giorno e salvati all'interno di una lista nel *database* che funge da *cache*.

L'utilizzo dell'*api* di *Spotify* si è reso necessario in quanto, secondo i termini e le condizioni d'uso [41], non è consentito utilizzare i dati per eseguire il *training* di un modello di *machine learning*. L'implementazione attuale non è funzionalmente ottimale, in quanto non si disponeva dei dati e delle risorse necessarie per sperimentare un algoritmo più personalizzato. In futuro, si potrà andare a creare un modello di *machine learning* che tenga conto delle interazioni dell'utente con la piattaforma.

Capitolo 5

Testing della versione 1.0.0.

In questo capitolo andremo a descrivere il ciclo di *testing* compiuto facendo provare l'applicativo a numerose persone e raccogliendo tramite moduli e statistiche le risposte. E' stato creato un *Google Form* per sviluppare un questionario inerente all'esperienza dell'utente dopo l'utilizzo, inoltre è stato anche un buon mezzo per popolare gli utenti e avere un *dataset* di supporto.

5.1 Tempi d'implementazione

La seguente versione è stata sviluppata in un periodo complessivo di 6 mesi, da febbraio 2023 a luglio 2023.
In questo arco di tempo è stato importante studiare le tecnologie più appropriate da utilizzare e impostare l'architettura dell'applicativo. E' stato creato un piano indicante tutte le possibili funzionalità che l'applicazione avrebbe potuto avere in futuro e tutti i possibili dati da immagazzinare in un *database*. A partire dalla scelta di queste è iniziata l'implementazione su due fronti contemporaneamente: il *frontend* e il *backend*. L'applicativo in questo modo a ogni nuova *feature* sarebbe potuto essere *testato* e validato immediatamente e non in un secondo momento.

5.2 *L'esperienza Mussida*

Il *testing* della prima versione di *Mussida* è stato senza alcun dubbio uno dei componenti più rafforzativi per le potenzialità dell'applicazione. Grazie alla

criticità delle persone è stato possibile avere un *feedback* di forte impatto per eventualmente migliorare/cambiare:

- La *user-experience*.
- La grafica.
- Le scelte implementative e le funzionalità.

5.2.1 Scelte implementative e funzionalità

Sono state poste ai *tester* domande inerenti alle scelte implementative che influenzano le funzionalità dell'applicazione.

Una delle domande più rilevanti riguardava il numero degli artisti e delle canzoni scelte per rappresentare il profilo di un utente, l'80% delle persone hanno espresso un voto a favore dell'aumento sia delle canzoni che degli artisti e il quantitativo doveva anche avere un minimo per poter calcolare correttamente le preferenze del singolo utente.

Per quanto riguarda invece il numero dei generi, il 86.7% degli utenti lo ha giudicato come un numero corretto e hanno aggiunto che nel caso ci sarebbe dovuto essere un aumento di massimo 1/2 generi.

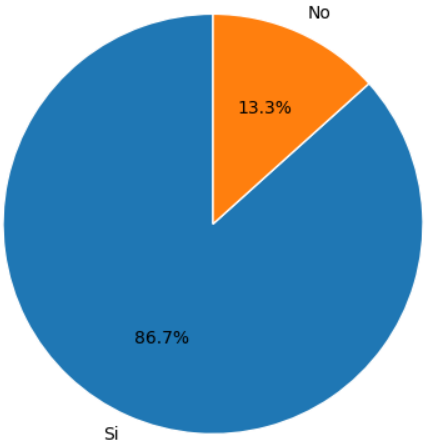

Figura 5.1: Piechart dei generi

All'interno del *form* è stata creata una sezione dedicata ai consigli per implementare **nuove funzionalità** per la prossima versione. Sicuramente è stata una componente che ha acceso la creatività degli utenti permettendoli di scrivere un elenco di *features* fortemente valide per il prossimo *sprint*. Qui di seguito vengono riportate le più valide e implementabili:

- Molti hanno inserito la possibilità di poter creare una *chat* con le persone affini musicalmente e scambiare messaggi, canzoni ed eventi.
- Alcuni hanno suggerito una collegamento con gli amici di *Spotify*, ovvero una pagina in cui ti vengono consigliati.
- Molti hanno richiesto l'eventuale inserimento dei *like*.
- Tanti hanno richiesto una schermata di settings, per tutte le funzionalità dell'utente ma anche per esempio per il cambio colore dell'applicativo.
- Atri utenti hanno proposto l'inserimento di una lista di consigliati per caratterizzare il proprio profilo.
- La possibilità di aggiungere una canzone ai preferiti dal profilo di altri utenti oppure direttamente dalla *home*.

- La possibilità di modificare post e commenti, e cercare contenuti in base a filtri come ad esempio il genere.

E' stato anche effettuato una sezione simile per inserire nuovi attributi lato **profilo** e anche in questo caso le richieste sono state multiple per:

- aggiungere statistiche personali sugli ascolti più frequenti e quindi quelli che ti piacciono di più.
- aggiungere la "*top 3*" nelle liste dei preferiti per esaltare l'importanza di alcuni artisti o canzoni.
- richiedere l'introduzione di *playlist* all'interno del profilo.
- creare *link* ai *social*.
- mostrare la lista dei seguiti e seguaci.
- vedere nella pagina del profilo i *post* pubblicati da noi.

Infine, l'utente ha espresso anche un parere riguardo alla modalità di ***posting*** **e ai** ***post***, ed è emerso che un'altissima percentuale ha fortemente gradito le pubblicazioni così senza modifiche, altri hanno dato dei consigli per migliorarne la funzionalità. Vengono riportate alcune delle risposte degli utenti:

- poter mettere un frammento di testo insieme o al *post* della copertina dell'album.
- la possibilità di votare i post.
- la possibilità di salvare le canzoni condivise dagli altri per poter creare una sezione di "preferiti".
- aggiungere i *post* legati ai concerti e ai *tour* dei cantanti, compresa la possibilità di vederne le date.
- oltre ai commenti aggiungere i *like* e la possibilità di inserire il *mood* che una persona ha nel momento in cui ascolta quella determinata canzone.
- renderli più interattivi, magari permettendo di richiedere consigli sulle canzoni da ascoltare con/in un determinato *mood*, o chiedere una canzone del "giorno" in base alle preferenze comuni dei *follower* in tal modo l'interazione con gli amici sarebbe maggiore.

5.2.2 La grafica

Per quel che concerne la **grafica** il 21.1% degli utenti ha espresso il voto "eccellente", il 52.6% "sopra la media" e il 21.1% "in media". Sono state poste delle domande specifiche per comprendere al meglio le caratteristiche grafiche da cambiare e quelle da tenere.

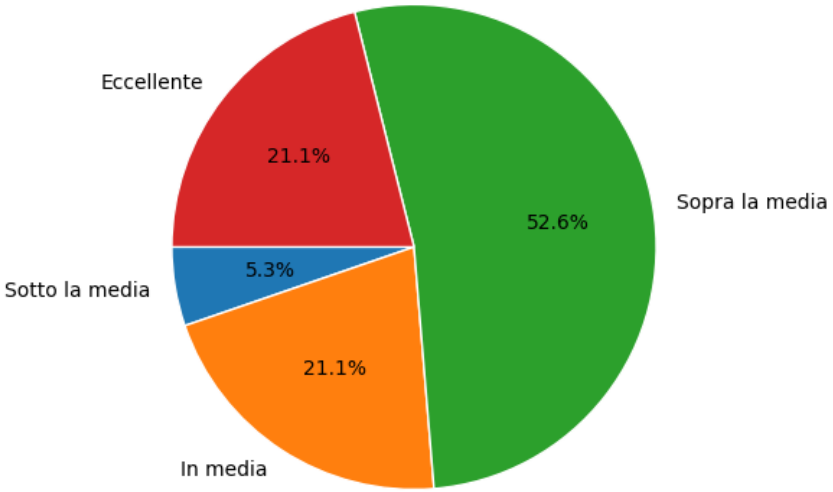

Figura 5.2: Piechart grafica

Per iniziare è stato richiesto agli utenti di esprimere un voto riguardo alla **card** dei *post* e il 37,5% dei voti è stato 5/5, il 56,3% 4/5 e infine il restante 6,3% 3/5. Un risultato soddisfacente, che può essere ottimizzato inserendo e arricchendo il *post* con più componenti grafiche come ad esempio: i *link* diretti a Spotify, un altro attributo interattivo come i commenti (es. i *like*) oppure la possibilità di salvare la canzone.

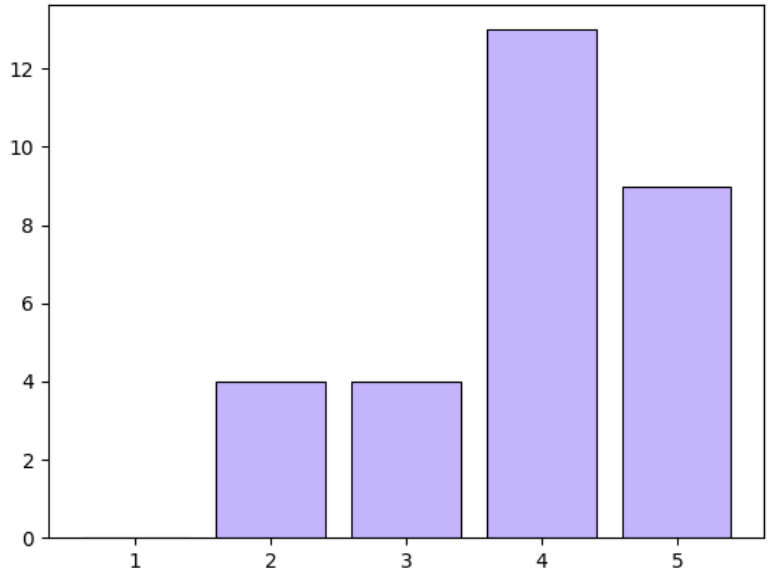

Figura 5.3: Istogramma Card

Per quanto riguarda l'influenza che ha avuto la grafica sul **flusso di utilizzo** è stata fortemente positiva con il 60% che ha votato 5/5 e il 40% 4/5. Gli utenti hanno commentato che grazie a componenti e "disegni" già visti all'interno di altre *app* come le icone, il *bottomsheet* e la *navbar* in basso l'utilizzo è stato molto intuitivo.

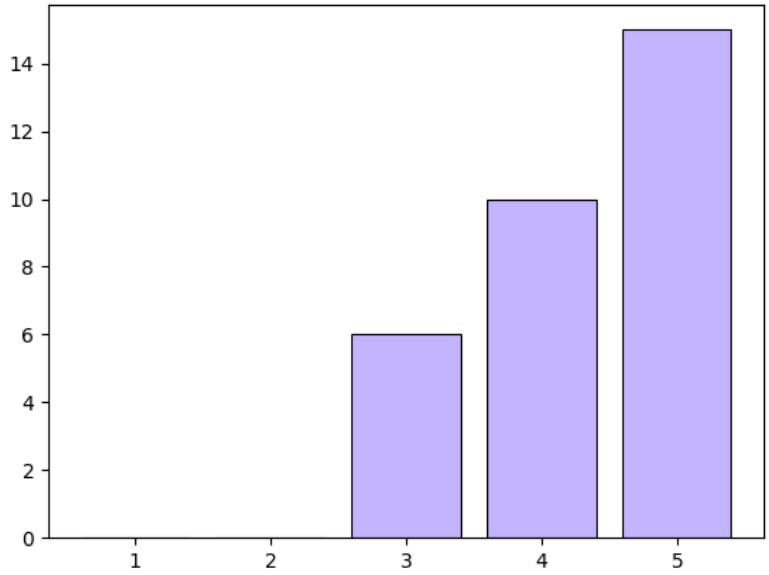

Figura 5.4: Istogramma Flusso di Utilizzo

Infine l'utente ha avuto la possibilità di scrivere quello che cambierebbe all'interno dell'applicativo nella sezione **home** e le risposte più appropriate e implementabili sono state riportate qua sotto:

- inserire la modalità *light mode*.
- togliere la *textbox* delle ricerche e dei commenti.
- cambiare colore di sfondo.
- arricchire grafica e *design*, troppo *minimal*.
- mettere nella *home* uno *switch* per rimbalzare da un *feed* globale a uno solo con i *followed*.
- potrebbe essere utile uno spazio nel quale puoi vedere gli ultimi brani ascoltati o cercati e vedere anche quante persone ascoltano la canzone che pubblichi nel post.

Nel complessivo l'esperienza dell'utente è stata di forte apprezzamento nella reattività e intuività dell'applicazione. Infine è stato richiesto un parere

generico su punti di forza della grafica di **Mussida** e alcuni dei commenti sono stati inseriti nella lista che segue:

- *"La semplicità"*
- *"Il punto di forza dell'applicazione è la pagina dove si possono vedere tutti i post pubblicati, è semplice e con una grafica molto bella"*
- *"Le cover grandi dell'album, l'anteprima della canzone e i colori "*
- *"Il punto di forza è la semplicità e l'intuitività di utilizzo. Si potrebbe aggiungere un tema chiaro e un tema scuro che gli utenti possono cambiare dalle impostazioni"*
- *"La grafica è semplice, pulita ma d'impatto"*
- *"Semplicità, intuitività, collegamenti per salvare canzoni direttamente su spoty"*
- *"sicuramente la semplicità e l'immediatezza sono dominanti in questa grafica. è molto intuitiva e quindi facile da utilizzare anche se si è al primo accesso"*

5.2.3 *User-experience*

Per avere un riscontro sul flusso di utilizzo e l'esperienza dell'utente è stato proposto agli utenti di descrivere e votare dei frammenti di sessione durante l'utilizzo.

Per cominciare il 68% degli utenti si è trovato bene durante la fase di **autenticazione** tramite *Spotify* descrivendolo come un processo semplice e intuitivo. In seguito per quanto riguarda la **personalizzazione del profilo**, l'utente ha riscontrato qualche difficoltà immediata, ma come riportato è stato comunque semplice specificare i generi e popolare le liste delle canzoni e artisti. **L'interazione con il *feed*** è stata ottima, veloce e reattiva. E' stato gradito un po' meno, invece, la fase di **creazione del *post*** per il *click* del bottone in basso a destra, ma allo stesso tempo dopo qualche minuto di utilizzo, come riportato, la sua funzionalità è stata chiara.

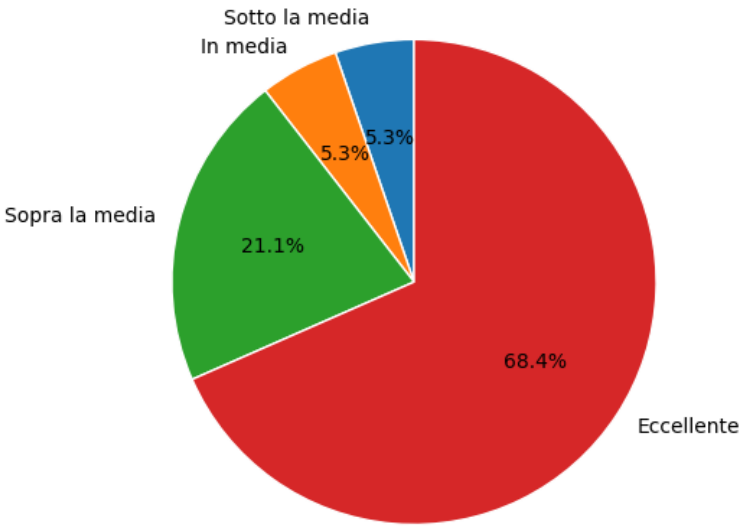

Figura 5.5: Piechart autenticazione

L'esperienza dell'utente è stata apprezzata dalla maggior parte degli utenti e uno dei prossimi attributi che verrà‘ aggiunto per migliorarla ‘e sicuramente una guida introduttiva per descriverne alcuni passaggi che potrebbero non essere immediatamente comprensibili.

5.3 Il prossimo *Sprint*

Il *testing* dell'applicativo è stato sicuramente uno dei passi più fondamentali e costruttivi per rendere completo il progetto. Bisogna trascendere tutto il processo tecnologico-implementativo e soffermarsi sull'aspetto più grande, ovvero che il fine del prodotto è proprio quello di diventare un utilità per le persone e magari un'abitudine. Quindi uno dei contributi più preziosi è stato senza dubbio il *testing*, comprendere cosa piace e cosa vuole la gente deve essere inserito della lista di "cose da fare" per la versione successiva.

Quello che è stato compiuto è prendere tutte le modifiche, le funzionalità, i problemi e consigli degli utenti e creare uno **sprint backlog** in ordine gerarchico per la prossima versione. È stato quindi effettuato uno studio su quello

che può veramente contribuire in maniera impattante al successo di **Mussida** e integrarlo all'interno del prossimo ciclo d'implementazione. Qui di seguito vengono riportati in ordine gerarchico i cambiamenti su cui si andrà a lavorare:

1. l'introduzione di una **chat** e una componente messaggistica dove gli utenti possono conoscersi e condividere musica e pareri a riguardo.
2. l'inserimento dei **settings** per permettere all'utente la visualizzazione di dati personali del proprio *account* e la loro eventuale modifica.
3. La possibilità di effettuare un *post* scegliendo una località sulla mappa e l'inserimento di un **evento**, e le persone possono interagire con questo **post** decidendo di partecipare o meno.
4. L'eventuale **modifica/eliminazione** di commenti e *post*.
5. L'introduzione delle **playlist** per la caratterizzazione dei profili.
6. La **divisione del *feed*** in due parti una contenenti gli utenti seguiti e l'altra globale calcolata con anche l'algoritmo di raccomandazione.
7. L'aggiunta di un attributo simile ai ***like*** ma che introduce altri significati come "non conoscevo questa canzone" oppure la reazione tramite un emozione.
8. Il calcolo di **ascolti** di un determiato *post*.

Capitolo 6

Conclusioni

In questa tesi abbiamo affrontato tutti i passaggi necessari per spiegare le scelte implementative e tecnologiche dell'applicazione.

Siamo partiti dando al lettore un fondamento tramite concetti base riguardanti le applicazioni mobili e i *social network* per permettergli di comprendere al meglio la parte tecnica che seguiva.

In seguito abbiamo spiegato le motivazioni inerenti al contesto di sviluppo dell'*app*, specificando le potenzialità del mercato e i *competitors* diretti nel settore. Dopo aver introdotto le motivazioni personali è iniziata la spiegazione della componente tecnica del progetto con l'architettura e l'implementazione.

Nel capitolo "Architettura e flusso di Mussida" è stato spiegata la struttura e il *flow* di utilizzo dell'*app* per mostrare al lettore l'esperienza dell'utente e le funzionalità offerte.

Invece, nel capitolo "Tecnologie e implementazione", è stato analizzato nel dettaglio il flusso di autenticazione tramite *Spotify* e il *server* per spiegare la modalità in cui avvenivano tutte le chiamate e richieste sul *software*. Sono state anche trattate le tecnologie usate lato *backend* e *frontend* con frammenti di codice per spiegare in modo programmatico le funzioni e la struttura del progetto.

Infine, è stata riportata nel capitolo "Testing" l'esperienza costruttiva ri-

cavata dal parere delle persone per migliorare l'applicazione introducendo nuove funzionalità e contenuti.

Sicuramente è stato un percorso lungo, partito a febbraio 2023, che mi ha portato ad avere una valigia piena di esperienza del processo di studio, analisi, inizializzazione, implementazione, produzione e pubblicazione di un *software*. Ho imparato a creare da zero un applicazione mobile e avere una visione di 360 gradi su tutto il procedimento.

La componente che mi ha arricchito di più durante questo viaggio è senza alcun dubbio l'aver compreso che percorso voglio intraprendere all'interno della mia vita, è stato di fondamentale importanza questo progetto perché ho avuto la conferma che sto adoperando nel settore che più mi appartiene e soddisfa.

Ringraziamenti

Ringrazio il professore Federico Montori per avermi accettato come tesista e ad avermi offerto tutto il supporto e i consigli di cui avevo bisogno.

Ringrazio Michele per avermi accompagnato in questo percorso altalenante di studio e avermi offerto una bellissima amicizia.

Ringrazio la mia famiglia per avermi dato tutto il sostegno e l'appoggio che mi serviva.

Bibliografia

[1] Statistiche sugli utilizzatori di *Smartphone*, https://wearesocial.com/it/blog/2023/01/digital-2023-i-dati-globali/#:~:text=Digital%202023%3A%20i%20principali%20highlights&text=5%2C44%20miliardi%20di%20persone,utenti%20negli%20ultimi%2012%20mesi..

[2] Statistiche utilizzari mondiali *smartphone*, https://www.oberlo.com/statistics/how-many-people-have-smartphones.

[3] Definizione *smartphone*, https://www.nextre.it/mobile-app/.

[4] Distinzione applicazioni mobile e *web*, https://www.nextre.it/web-app/.

[5] Distinzione applicazione nativa e ibrida, https://corael.it/qual-e-la-differenza-tra-app-nativa-e-app-ibrida/.

[6] Distinzione *app native* ed ibride, https://vitolavecchia.altervista.org/caratteristiche-e-differenza-tra-app-mobile-native-ibride-e-web-app/.

[7] Documentazione *React Native*, https://reactnative.dev.

[8] Dati sugli utilizzatori di *social network* nel mondo, https://datareportal.com/social-media-users.

[9] Statistiche mondiali *social network*, https://datareportal.com/reports/digital-2023-deep-dive-the-worlds-top-social-media-platforms.

[10] Definizione *social network*, https://www.studiosamo.it/social-network-piu-famosi/.

[11] Speigazione di *Six Degrees*, https://www.fc1492.com/la-nascita-dei-social-network/.

[12] Breve storia dei *social network*, https://www.postpickr.com/storia-dei-social-network-come-sono-nati/.

[13] B. Holiday, Citazione di Billie Holiday sulla musica, https://quotepark.com/quotes/682450-billie-holiday-you-cant-copy-anybody-and-end-with-anything-if-y/.

[14] TikTok, https://play.google.com/store/apps/details?id=com.zhiliaoapp.musically&hl=it.

[15] Last.fm, https://play.google.com/store/apps/details?id=fm.last.android&hl=it.

[16] Bandacamp, https://play.google.com/store/apps/details?id=com.bandcamp.android&hl=it.

[17] Typescript, https://www.typescriptlang.org.

[18] B. Meyer, *Object-Oriented Software Construction*, **1988**.

[19] Spotify api, https://developer.spotify.com/documentation/web-api.

[20] Mussida api, https://backendmussida-fwaxwt562a-oc.a.run.app.

[21] Framework OAuth2.0, https://datatracker.ietf.org/doc/html/rfc6749.

[22] Scopes di Spotify, https://developer.spotify.com/documentation/web-api/concepts/scopes#user-read-playback-state.

[23] Flusso di autorizzazione su Spotify, https://developer.spotify.com/documentation/web-api/tutorials/code-flow.

[24] Json Web Token (jwt), https://datatracker.ietf.org/doc/html/rfc9068.

[25] React Native navigation, https://reactnavigation.org.

[26] Expo notifications, https://docs.expo.dev/versions/latest/sdk/notifications/.

[27] React hooks, https://legacy.reactjs.org/docs/hooks-intro.html.

[28] Libreria Tanstack Query, https://tanstack.com/query/latest.

[29] Framework Express, https://expressjs.com.

[30] Libreria Trpc, https://trpc.io.

[31] Database Mongodb, https://www.mongodb.com.

[32] Libreria Prisma, https://www.prisma.io.

[33] Pacchetto prisma per generare diagramma ER, https://github.com/notiz-dev/prisma-dbml-generator.

[34] Deployment Cloud Run, https://cloud.google.com/run?hl=it.

[35] Openapi Standard, https://swagger.io/specification/.

[36] Pacchetto Trpc Openapi, https://github.com/jlalmes/trpc-openapi.

[37] Pacchetto Openapi Generator, https://github.com/OpenAPITools/openapi-generator.

[38] Zod Validator, https://zod.dev.

[39] Http methods, https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

[40] Spotify reccomandation, https://developer.spotify.com/documentation/web-api/reference/get-recommendations.

[41] Termini e condizioni imposti da Spotify, https://developer.spotify.com/terms#section-iv-restrictions.

Appendice A

Diagramma E-R

Viene riportato il **diagramma entità relazione** per comprendere al meglio la struttura delle tabelle presenti nel *database* e le relazioni presenti tra gli attributi.

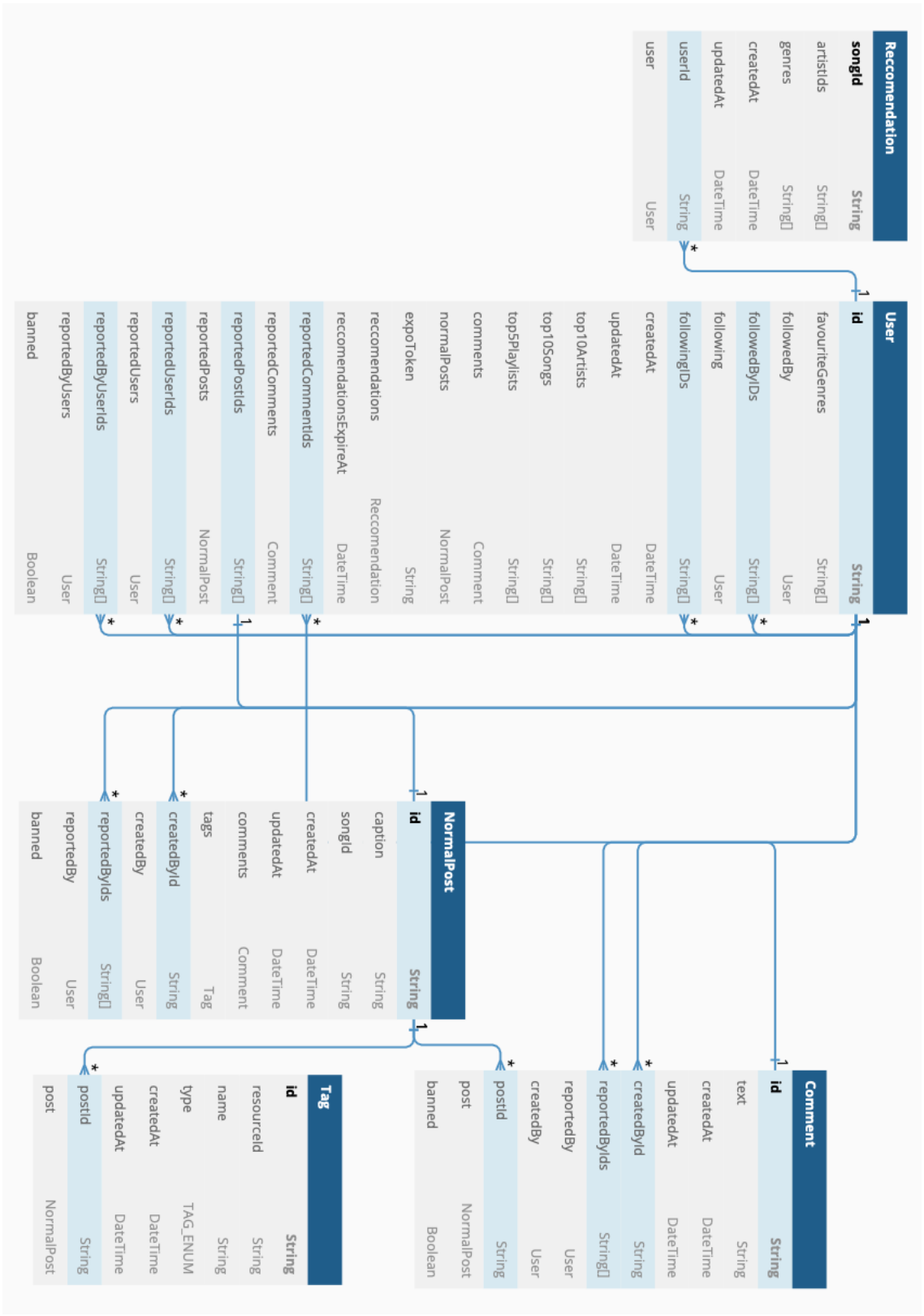

Figura A.1: Diagramma *entity relationship*