

# Relazione del Progetto di Programmazione Web e Mobile (PWM)

Dipartimento di Informatica Giovanni Degli Antoni

Samuele Manclossi  
09882A

5 luglio 2023

*“Non quia difficilia sunt non audemos, sed quia non  
audemos difficilia sunt”*

— Seneca

## Abstract

*Questo PDF è la relazione del progetto conclusivo del corso di Programmazione Web e Mobile, tenuto dal prof. Valerio Bellandi alla Università Statale di Milano nell’A.A. 2022-2023.*

*Esso consiste nella realizzazione di una applicazione web dal nome SOCIAL NETWORK FOR MUSIC, che si deve prefiggere lo scopo principale di implementare un sistema di gestione delle playlist musicali create da utenti ed (eventualmente) condivise.*



# Contents

<b>1</b>	<b>Analisi delle specifiche</b>	<b>1</b>
1.1	Funzionalità da implementare . . . . .	1
1.1.1	Gestione degli utenti . . . . .	1
1.1.2	Ricerca e visualizzazione dei dati . . . . .	1
1.1.3	Preferiti e Playlist . . . . .	2
1.1.4	Opzionale: creazione di gruppi . . . . .	2
<b>2</b>	<b>Tecnologie utilizzate e interazioni tra esse</b>	<b>3</b>
2.1	Data . . . . .	3
2.2	Backend . . . . .	3
2.3	Frontend . . . . .	4
<b>3</b>	<b>Backend e chiamate alle API</b>	<b>5</b>
3.1	Funzionalità del backend . . . . .	5
3.2	Richieste a Spotify e ottenimento dei dati . . . . .	5
3.3	Gestione degli utenti . . . . .	5
3.4	Gestione delle playlist e dei gruppi di utenti . . . . .	5
<b>4</b>	<b>Interfaccia utente e front-end</b>	<b>6</b>
4.1	Pagine e relative funzioni e visibilità . . . . .	6
4.1.1	Welcome page - <i>index.html</i> . . . . .	6
4.1.2	Registrazione - <i>register.html</i> . . . . .	6
4.1.3	Login - <i>login.html</i> . . . . .	6
4.1.4	Ricerca - <i>search.html</i> . . . . .	6
4.1.5	Playlist - <i>playlists.html</i> . . . . .	6
4.1.6	MyPlaylists - <i>myplaylists.html</i> . . . . .	6
4.2	Navbar condivisa tra tutte le pagine . . . . .	6
4.2.1	Pulsante <i>SNM</i> . . . . .	6
4.2.2	Pulsante <i>Search</i> . . . . .	6
4.2.3	Pulsante <i>Groups</i> . . . . .	6
4.2.4	Dropdown <i>User</i> . . . . .	6
<b>Appendice A:</b>	<b>brani di codice e scelte implementative</b>	<b>a</b>
4.3	Struttura dati nel database . . . . .	a
4.4	Backend . . . . .	b
4.4.1	Accesso alle informazioni riservate . . . . .	b
4.4.2	JWT nei cookies anziché dati dell'utente nel local storage . . . . .	b
4.4.3	Token Object e perform function . . . . .	b
4.4.4	Eliminazione di documenti e integrità delle referenze . . . . .	c
4.5	Frontend . . . . .	f
<b>Appendice B:</b>	<b>swagger</b>	<b>g</b>

# 1 Analisi delle specifiche

## 1.1 Funzionalità da implementare

SOCIAL NETWORK FOR MUSIC (di seguito **SNM**) deve essere in grado di svolgere diverse funzioni, tra cui le seguenti.

Si noti, non saranno riportati i campi dei singoli oggetti, in quanto essi sono già presenti nella apposita sezione riguardante la struttura dati, all'interno delle scelte implementative.

### 1.1.1 Gestione degli utenti

La gestione degli utenti è necessaria, e deve essere affrontata attraverso diverse fasi:

**Registrazione** La registrazione degli utenti avviene attraverso una pagina apposita, a cui può accedere qualsiasi utente. Questa pagina consente (dopo appositi controlli) di effettuare una richiesta POST al backend, che si occuperà di registrare i dati dopo averli nuovamente verificati.

I campi che saranno richiesti in fase di registrazione dovranno essere:

- **Email**, campo unico che verrà usato per autenticarsi.
- **Password**
- **Conferma password**
- **Nome utente**, che sarà il nome mostrato agli altri quando una playlist viene condivisa e per simili funzionalità
- **Preferenze musicali** (genere preferito, da una lista restituita dal backend)
- **Gruppi musicali preferiti**

---

💡 **Idea:** Per ragioni implementative, i gruppi musicali preferiti di default saranno vuoti, come ogni altro preferito. Questi potranno essere infatti aggiunti in maniera più corretta e coerente con il resto dell'applicazione navigando e cercando i gruppi, in modo da salvarne gli ID coerentemente con quelli forniti da Spotify

---

**Login** Mediante il login un utente entra nel proprio profilo, diventando in grado di vedere le proprie playlist e il proprio profilo.

Questo avverrà attraverso la richiesta di due campi:

- Email
- Password

Entrambi i campi saranno modificabili da un apposito sistema.

Nonostante sia stato suggerito di salvarsi le informazioni in locale sarebbe più opportuno usare i token JWT. A questo sarà dedicata una apposita sezione.

**Logout** Questa funzione si spiega da sola, senza bisogno di tanti commenti.

**Cambiare i propri dati** Ogni dato deve essere modificabile tramite apposite richieste al backend.

**Eliminare l'account** Deve essere possibile eliminare l'account, cancellando tutte le informazioni che lo riguardano. Se l'account viene eliminato, vengono rimosse tutte le playlist create da quell'utente. Sarebbe pertanto consigliabile la realizzazione di un sistema che consenta, all'eliminazione dell'account, di stabilire a chi passa la proprietà di quelle playlist, oppure scegliere di eliminarle.

### 1.1.2 Ricerca e visualizzazione dei dati

Questo deve essere fatto attraverso due pagine apposite, una che si occupi della ricerca e ne mostri i risultati e una che mostri le informazioni sul singolo brano, permettendo l'inserimento di questo nelle playlist (eventualmente la creazione di una nuova playlist in caso non dovesse già esistere).

Vista l'ampia varietà di campi per cui è possibile cercare, una soluzione sarebbe prima mostrare alcuni risultati per ogni campo e poi restringere la ricerca su un campo specifico.

### 1.1.3 Preferiti e Playlist

**Preferiti** Un utente può decidere di selezionare un numero imprecisato di brani come suoi brani preferiti. Questo li aggiunge alle informazioni (private) del suo account. Non solo, egli potrà inserire potenzialmente ogni categoria di dati restituiti da Spotify.

**Playlist** Una playlist è una collezione di brani denotata da alcune informazioni, ritrovabili nella sezione apposita.

Una playlist privata può essere vista solamente dal creatore, una playlist pubblica può essere vista liberamente da chiunque mediante una apposita pagina, mentre una playlist condivisa con un gruppo può essere visibile a chiunque sia all'interno di quella

Si noti che le playlist devono implementare le seguenti azioni:

- **Cancellazione**
- **Rendere privata**
- **Rendere pubblica**
- **Condividere con un gruppo**
- **Rimuovere condivisione con un gruppo**
- **Follow/Unfollow**
- **Trasferimento della proprietà ad un nuovo owner**
- **Aggiunta o Rimozione di canzoni**
- **Recuperare le informazioni**

### 1.1.4 Opzionale: creazione di gruppi

Si potranno creare delle comunità di utenti, di qui in avanti **gruppi**, a cui gli utenti potranno iscriversi e disisciversi. Quando un utente è iscritto, esso è in grado di vedere tutte le playlist condivise con quella comunità specifica, e risulta anche in grado di parteciparvi condividendo altre playlist.

Il creatore del gruppo non può essere in grado di escludere qualcuno dal gruppo. Questo perché i gruppi nascono come comunità aperte. Il creatore del gruppo, però, ipoteticamente, potrebbe essere in grado di annullare le condivisioni di playlist verso quel gruppo da parte di altri, cosa che agli utenti normali non è consentita.

## 2 Tecnologie utilizzate e interazioni tra esse

Le tecnologie utilizzate saranno divise a seconda della tipologia.

### 2.1 Data

Per i dati si è utilizzato, come da istruzioni ricevute, MongoDB. La struttura dati è approfondita nelle scelte progettuali, come anche l'uso delle informazioni di accesso, pertanto non vi sono altre informazioni rilevanti da specificare in questa sede.

### 2.2 Backend

Per il backend si è utilizzato NodeJS, con l'utilizzo del framework `express`[4].

L'utilizzo di NPM ha consentito l'uso dei package, in particolare sono stati usati:

- **cookie-parser**: un package che consente di parsare l'header `Cookie` e popolare `req.cookies` con un oggetto avente per attributi i nomi dei cookie[1].
- **cors**: un middleware per eliminare i fastidiosi problemi con i cors[2].
- **dotenv**: un package per la gestione del `.env`, con un comodo comando `require('path/to/file')` e di seguito `.configure()[3]`.
- **express**: il framework già citato sopra.
- **express-mongo-sanitize**: un package che fornisce un middleware per la sanitizzazione di molti campi delle richieste. Si noti che l'ho usato con la configurazione `allowDots: true` per evitare che l'email venisse modificata. Per il resto consente un notevole miglioramento della sicurezza con riduzione del rischio di injections[5].
- **jsonwebtoken**: il package che consente una autenticazione un po' più sicura rispetto al minimo richiesto mediante l'uso, la firma e la verifica dei JWT tramite comodi metodi[6].
- **mongodb**: il package ufficiale per gestire MongoDB da NodeJS[7].
- **nodemon**: un package che consente di riavviare il server nel momento in cui rileva qualsiasi cambiamento[8].
- **swagger-ui-express**: un package per fornire uno swagger[10].
- **validator**: un package molto utile per sanitizzare e validare stringhe[11].
- \* **swagger-autogen**: utilizzato solo come `devDependency`, esso è utile per generare automaticamente lo swagger[9]. L'export di questo swagger lo trovate nell'Appendice B.

Ci sono poi altri packages, come `crypto` o `path`, che venivano già forniti in automatico.

Il file `package.json` risulta quindi composto come segue:

```
1 {
2   "scripts": {
3     "start": "nodemon app.js",
4     "debug": "nodemon --inspect app.js"
5   },
6   "dependencies": {
7     "cookie-parser": "^1.4.6",
8     "cors": "^2.8.5",
9     "dotenv": "^16.0.3",
10    "express": "^4.18.2",
11    "express-mongo-sanitize": "^2.2.0",
12    "jsonwebtoken": "^9.0.0",
13    "mongodb": "^5.4.0",
14    "nodemon": "^1.14.9",
15    "swagger-ui-express": "^4.6.2",
16    "validator": "^13.9.0"
17  },
18  "devDependencies": {
19    "swagger-autogen": "^2.23.1"
20  }
21 }
```

## 2.3 Frontend

## 3 Backend e chiamate alle API

### 3.1 Funzionalità del backend

Al backend vengono delegate funzionalità relative fondamentalmente a quattro gruppi di operazioni:

- Richieste a Spotify e ottenimento dei dati
- Gestione degli utenti
- Gestione delle playlist
- Gestione dei gruppi

Queste funzionalità saranno spiegate meglio nelle seguenti sezioni.

### 3.2 Richieste a Spotify e ottenimento dei dati

---

💡 **Idea:** Ho deciso di delegare la funzionalità di ricerca su Spotify al backend per una ragione di sicurezza: non è desiderabile che le chiavi di accesso a Spotify vengano condivise con gli utenti, anche solo inserendole nel frontend. Più avanti sarà spiegata la modalità di accesso a queste informazioni da parte del backend

---

**Ottenere informazioni** Per effettuare una ricerca o comunque ottenere informazioni si deve effettuare una richiesta a Spotify specificando il proprio token. Per l'utilizzo del token, si confronti la sezione `Token Object` nelle scelte implementative.

Esistono due percorsi appositi per fare una ricerca o per ottenere informazioni riguardo a un oggetto specifico. Per queste si consulti l'allegato B, ossia lo swagger.

### 3.3 Gestione degli utenti

Per ogni utente sono possibili diverse operazioni. In particolare, le principali sono `register`, `login`, `checkLogin` (verificare se l'utente è loggato), `delete`.

### 3.4 Gestione delle playlist e dei gruppi di utenti

Le playlist devono prevedere le diverse funzioni trovate nell'analisi delle specifiche, così come i gruppi.

---

💡 **Idea:** Per le singole funzionalità si prega di guardare lo swagger all'Appendice B

---

## 4 Interfaccia utente e front-end

### 4.1 Pagine e relative funzioni e visibilità

#### 4.1.1 Welcome page - *index.html*

La pagina di accoglienza sarà una pagina con pochissime funzionalità, destinata prevalentemente a fare da “vetrina” del servizio offerto.

Vi saranno due pulsanti: **login** e **register**, e la navbar per navigare invece la parte pubblica. Se si è già loggati, nel load si verrà reindirizzati alla pagina dove sono mostrate le playlist pubbliche o a quella dove è possibile effettuare ricerche.

#### 4.1.2 Registrazione - *register.html*

La pagina di registrazione sarà costituita da un form. Se si è loggati, si verrà reindirizzati automaticamente alla pagina di login. Altrimenti, ci si potrà registrare. Se la registrazione va a buon fine si viene reindirizzati alla pagina di login, altrimenti viene mostrato il messaggio d'errore restituito dal backend.

#### 4.1.3 Login - *login.html*

La pagina di login sarà costituita da un form. Se si è già loggati si viene reindirizzati alla pagina di visualizzazione dati pubblici, altrimenti ci si può loggare. In ogni caso saranno emessi appositi messaggi per segnalare lo stato.

#### 4.1.4 Ricerca - *search.html*

La pagina di ricerca consentirà di effettuare una ricerca per vari campi. Il meccanismo è ancora da studiare nei dettagli.

#### 4.1.5 Playlist - *playlists.html*

In questa pagina si potranno vedere le playlist pubbliche.

#### 4.1.6 MyPlaylists - *myplaylists.html*

In questa pagina si potranno vedere le proprie playlist, divise tra private, pubbliche e quelle altrui che vengono seguite.

### 4.2 Navbar condivisa tra tutte le pagine

Tutte le pagine avranno accesso a una navbar condivisa, costituita dai seguenti elementi.

#### 4.2.1 Pulsante *SNM*

Consente di tornare alla pagina delle playlist pubbliche.

#### 4.2.2 Pulsante *Search*

Consente di tornare alla pagina di ricerca.

#### 4.2.3 Pulsante *Groups*

Consente di tornare alla pagina dove sono mostrati i gruppi, ed eventualmente visualizzare le informazioni su di essi.

#### 4.2.4 Dropdown *User*

Esso conterrà diverse operazioni sul profilo, tra cui le seguenti:

Nome	Cosa fa	Condizioni di visualizzazione
<i>login</i>	Redirige alla pagina di login	Non essere loggati
<i>register</i>	Redirige alla pagina di registrazione	Sempre visibile
<i>profile</i>	Redirige alla pagina del profilo	Essere loggati
<i>logout</i>	Effettua il logout e reindirige alla “vetrina”	Essere loggati
<i>my playlists</i>	Redirige alla pagina privata delle playlist personali	Essere loggati



## Appendice A: brani di codice e scelte implementative

Di seguito saranno spiegate alcune scelte implementative relative a varie sezioni, iniziando da quelle riguardanti la struttura dati nel database.

### 4.3 Struttura dati nel database

Il database prevede tre collezioni, ciascuna costituita da documenti con una data struttura:

**Users:**

```

1 {
2   "_id": {...},
3   "name": "First Name",
4   "surname": "Surname",
5   "userName": "userName",
6     UNIQUE
7   "email": "a@b.c", UNIQUE
8   "birthDate": "2003-10-01",
9   "favoriteGenres": [],
10  "password": "...",
11  "favorites": {
12    "album": [],
13    "artist": [],
14    "audiobook": [],
15    "episode": [],
16    "show": [],
17    "track": []
18  },
19  "playlistsFollowed": [],
20  "playlistsOwned": [],
21  "groupsFollowed": [],
22  "groupsOwned": []
23 }
```

**Playlists:**

```

1 {
2   "_id": {...},
3   "name": "myList", UNIQUE
4   "description": "this is a
5     playlist about old
6     finnish songs",
7   "tags": [
8     "finnish",
9     "old",
10    "42"
11  ],
12  "visibility": true,
13  "owner": "userName"
14 }
15 .
16
```

**Groups:**

```

1 {
2   "_id": {...},
3   "name": "myGroup", UNIQUE
4   "decription": "This is a
5     group for lovers of
6     classical music. Join
7     this group to gain
8     access to more than
9     15 playlists!",
10  "playlistsShared": [
11    "classicalMusic",
12    "musicaClassica",
13    ...
14  ],
15  "owner": "userName",
16  "users": [
17    "userName",
18    "user42",
19    "lambda",
20    "SophosIoun",
21    ...
22  ]
23 }
```

Dove `playlistsFollowed`, `playlistsOwned`, `groupsFollowed`, `groupsOwned` riferenziano gruppi o playlist nelle altre collezioni.

Dove `owner` riferenzia `userName`, che è lo `userName` di uno `user`. Si noti che una playlist non sa da quali utenti è seguita o in quali gruppi sia inserita.

Dove `owner` e `users` riferenziano `userName` nella collezione `users`, mentre `playlistsShared` è un array di referenze ai nomi delle `playlists` condivise con quel gruppo.

In tutti e tre i casi i campi indicati come `UNIQUE` permettono di fare riferimento da altri. Laddove l'`email` è solo usata per la login, tutti e tre i campi `name` (`user.userName`, `playlist.name` e `group.name`) hanno lo scopo di essere i riferimenti per documenti nelle altre collezioni.

**💡 Idea:** So bene che alcune informazioni sono duplicate (in quanto sarebbe possibile avere le informazioni complete anche senza riportare sia gli `users` nei gruppi che i gruppi negli `users`, ad esempio) tuttavia queste informazioni sono presenti in entrambi i casi per una questione di facilità di uso delle informazioni e per ridurre il numero di chiamate al backend o al database necessarie per svolgere ogni operazione, al modico prezzo di un po' di spazio aggiuntivo occupato.

Si noti poi che `playlists.songs` è un array di oggetti costituiti come segue:

```

1 {
2   "titolo": "15 secondi di muratori al lavoro",
3   "durata": 15000,
4   "cantante": "Me medesimo",
5   "anno_di_publicazione": "2023",
6 }
```

Mentre ogni preferito è salvato come un oggetto contenente titolo e id, tutto questo nell'apposito array.

## 4.4 Backend

### 4.4.1 Accesso alle informazioni riservate

Alcune informazioni non dovevano diventare pubbliche, ossia venire direttamente a contatto (per esempio venendovi inserite) con il frontend.

A questo scopo ho optato per l'utilizzo di un file `.env`, prontamente inserito nel `.gitignore` e che sarà consegnato a fianco della repo, contenente queste informazioni secondo il seguente formato:

```
1 MONGONAME=XXXXXXXXXXXX
2 MONGOPASSWORD=XXXXXXXXXXXX
3 PORT=3000
4 CLIENT_ID=XXXXXXXXXXXX
5 CLIENT_SECRET=XXXXXXXXXXXX
6 SECRET=XXXXXXXXXXXX
```

Dove `SECRET` è la chiave per firmare i JWT, `MONGONAME` e `MONGOPASSWORD` sono relativi all'utilizzo di MongoDB, `CLIENT_ID` e `CLIENT_SECRET` servono per Spotify e `PORT` può essere usato per cambiare il numero della porta su cui viene esposto il servizio.

Per accedervi viene usato il package `DotEnv`, di cui parlo più approfonditamente nella sezione **tecnologie**.

### 4.4.2 JWT nei cookies anziché dati dell'utente nel local storage

---

**❗ Idea:** L'utilizzo dei JWT nei cookies permette di eseguire più facilmente e in modo più sicuro i controlli se l'utente è loggato o meno

---

Voglio realizzare una autenticazione lievemente più sicura di quella richiesta. Per ottenere questo scopo utilizzo i JWT (con un apposito package di cui parlerò nella sezione **tecnologie**), firmati con una chiave presente nel file `.env`, e li salvo nei cookie in modo che vengano passati in automatico ad ogni richiesta.

Si avrà quindi il logout nella forma di un `res.clearCookie('token')` e un controllo sull'identità dell'utente mediante un codice come quello che segue:

```
1 async function nomeMetodo(req,res){
2   var pwmClient = await new MongoClient(mongoUrl).connect()
3   const token = req.cookies.token
4   if(token == undefined) res.status(401).json({"reason": 'Invalid login'})
5   else{
6     jwt.verify(token,process.env.SECRET, async (err,decoded) =>{
7       if(err){
8         res.status(401).json(err)
9         pwmClient.close()
10      }
11      else{
12        //vero e proprio codice del metodo
13        pwmClient.close()
14      }
15    })
16  }
17 }
```

---

**⚠:** Questo sistema è sempre vulnerabile ad un possibile “furto” dei cookie: qualora questi venissero rubati è possibile per un utente inserirsi al posto di un altro, cambiare email e password e rendere impossibile all'utente originario l'accesso. Questo è un problema che non ho risolto in quanto ho ritenuto fuori dalle finalità di questo progetto, tuttavia sono consapevole del problema

---

### 4.4.3 Token Object e perform function

La richiesta (per quasi ogni informazione) di un token valido che scade ogni ora per operare con Spotify mi ha fatto optare per la creazione di un oggetto `Token`, come nel seguente brano di codice:

```

1  var token = {
2    value: "none",
3    expiration: 42,
4    regenAndThen : function(func_to_apply,paramA,paramB){
5      fetch(baseUrls.token, {
6        method: "POST",
7        headers: {
8          Authorization: "Basic " + btoa(`${process.env.CLIENT_ID}:${process.env.CLIENT_SECRET}`),
9          "Content-Type": "application/x-www-form-urlencoded",
10       },
11       body: new URLSearchParams({ grant_type: "client_credentials" }),
12     })
13     .then((response) => response.json())
14     .then((tokenResponse) => {
15       this.expiration = new Date().getTime(); //ms
16       this.value=tokenResponse.access_token
17       func_to_apply(paramA,paramB)
18     })
19   },
20   hasExpired : function(){
21     if(((new Date().getTime()-this.expiration)/1000/60)>=59){
22       return true;
23     }
24     else return false;
25   }
26 }

```

Questo oggetto presenta una serie di caratteristiche: esso ha due attributi, **value** che rappresenta il valore corrente del token, all'inizio un valore fasullo, e **expiration** che permette di sapere il momento in cui è stato ottenuto, all'inizio un valore spurio (42) così da essere sicuri che al primo uso esso venga ricaricato.

Esso ha inoltre due metodi. Abbiamo infatti **hasExpired** permette di sapere se sono passati più di (o esattamente) 59 minuti dalla scadenza.

---

💡 **Idea:** Ho impostato come tempo limite 59 minuti e non un'ora perché le operazioni richiedono tempo, pertanto un'ora non sarebbe stato utile in quanto avrebbe potuto scadere prima del completamento della richiesta. In questo modo, invece, dovrebbe essere sempre possibile avere dei token validi.

---

Vi è poi **regenAndThen** che consiste in un applicatore: esso riceve tre parametri (**func\_to\_apply**, funzione; **paramA** e **paramB** parametri da passare a quella funzione) ed effettua le seguenti operazioni:

1. Rigenera il token
2. Sostituisce il valore del token e il momento in cui è stato generato a quelli precedenti
3. Chiama la funzione con i parametri passati

**perform** Il token object risulta di gran lunga più pratico nel momento in cui è utilizzato insieme alla funzione **perform**, definita come segue:

```

1  function perform(questo,paramA,paramB){
2    if(token.hasExpired()) token.regenAndThen(questo,paramA,paramB)
3    else questo(paramA,paramB)
4  }

```

Questa funzione è ancora una volta un applicatore, in particolare nei confronti del parametro-funzione **questo**. Esso verifica se il token è scaduto sfruttando gli appositi metodi dell'oggetto **token**. In caso sia scaduto chiama l'applicatore **regenAndThen**, altrimenti esegue direttamente, così da non rigenerare il token ad ogni chiamata, ma solo quando necessario.

#### 4.4.4 Eliminazione di documenti e integrità delle referenze

---

💡 **Idea:** Non voglio dovermi ritrovare con dati inconsistenti o con riferimenti pendenti all'interno del database, perciò preferisco realizzare del codice molto inefficiente piuttosto che poi dover interpretare dati incompleti, parziali o superflui

Ci sono tre richieste di tipo `delete` che si possono fare all'API. Queste richieste, visibili anche nello swagger di cui all'Appendice B, sono `app.delete('/playlist/:name')`, `app.delete('/group/:name')` e `app.delete('/user')`.

Quando avviene una chiamata a questi endpoint sarebbe facile risolverla con un banale `deleteOne`. Questo, però, genererebbe problemi: dato che i documenti si referenziano tra loro creare dei riferimenti pending sarebbe impossibile.

Per eliminare quelli ho optato per una strategia divisa in più fasi, cercando sempre di assicurarmi che lo svolgimento erraneo di una fase non potesse compromettere quelle successive, ma piuttosto le fermasse. Questo perché è meglio, a mio giudizio, eliminare dei riferimenti senza riuscire a eliminare l'oggetto in questione piuttosto che eliminare l'oggetto lasciando dei riferimenti pending.

Le soluzioni sono le seguenti: iniziamo dagli utenti.

```

1  async function deleteUser(req,res){
2      ...
3      let user = await pwmClient.db("pwm_project").collection('users').findOne({"email": decoded.email})
4      //-3 elimino l'utente da ogni gruppo in cui e', ed elimino ogni gruppo che sia owned,
        rimuovendolo prima da ogni utente che sia in quel gruppo
5      let allGroups = pwmClient.db("pwm_project").collection("groups").find({})
6      for(let group in allGroups){
7          if(group.users.some(user.userName)){
8              group.users.splice(group.users.indexOf(user.userName),1)
9              await pwmClient.db("pwm_project").collection('groups').updateOne({"name":group.name},group)
10         }
11     }
12     //-3 bis elimino ogni gruppo posseduto dall'utente
13     await pwmClient.db("pwm_project").collection("groups").deleteMany({"owner":user.userName})
14     //-2 elimino ogni playlist dell'utente da ogni lista di playlist seguite altrui
15     allGroups = pwmClient.db("pwm_project").collection("groups").find({})
16     let allPlaylists =
17         pwmClient.db("pwm_project").collection("playlists").find({"owner":user.userName})
18     let allUsers =
19         pwmClient.db("pwm_project").collection("playlists").find({"email":{"$not:decoded.email"}})
20     for(let playlist in allPlaylists){
21         for(let utente in allUsers){
22             if(utente.playlistsFollowed.some(playlist.name)){
23                 utente.playlistsFollowed.splice(utente.playlistsFollowed.indexOf(playlist.name),1)
24                 await pwmClient.db("pwm_project").collection("users")
25                     .updateOne({"email":utente.email},utente)
26             }
27         }
28         for(let group in allGroups){
29             if(group.playlistsShared.some(playlist.name)){
30                 group.playlistsShared.splice(group.playlistsShared.indexOf(playlist.name),1)
31                 await pwmClient.db("pwm_project").collection("groups").updateOne({"name":group.name})
32             }
33         }
34     }
35     //-2 bis elimino ogni playlist owned dall'utente
36     await pwmClient.db("pwm_project").collection("playlists").deleteMany({"owner":utente.userName})
37     //-1 elimino l'account dell'utente
38     await pwmClient.db('pwm_project').collection('users').deleteOne({"email":decoded.email})
39     //-0. forse e' andato tutto liscio, e spero di non aver lasciato riferimenti pending da qualche
        parte
40     res.status(200).json({"reason":"ok"})
41     ...
42 }

```

Mentre per le playlist:

```

1  async function deletePlaylist(req,res){
2      ...

```

```

3 let a = await pwmClient.db("pwm_project").collection('playlists').findOne({"name":
  validator.escape(req.params.name)})
4 let userToUpdate = await
  pwmClient.db("pwm_project").collection('users').findOne({"email":decoded.email});
5 if(a != null && a != undefined && isOwner(a,userToUpdate.userName)){
6   //-3. per ogni utente, diverso dall'owner, elimino la playlist da quelle seguite, laddove
     presente.
7   let allUsers = pwmClient.db("pwm_project").collection("users").find({"email":{$not:
     decoded.email}})
8   for(let user in allUsers){
9     if(user.playlistsFollowed.some(a.name)){
10       user.playlistsFollowed.splice(user.playlistsFollowed.indexOf(a.name),1)
11       await pwmClient.db("pwm_project").collection('users')
12         .updateOne({"email":user.email},user)
13     }
14   }
15   //-2.5. per ogni gruppo elimino la playlist da quelle seguite, laddove presente
16   let allGroups = pwmClient.db("pwm_project").collection("groups").find({})
17   for(let group in allGroups){
18     if(group.playlistsShared.some(a.name)){
19       group.playlistsShared.splice(group.playlistsShared.indexOf(a.name),1)
20       await pwmClient.db("pwm_project").collection('groups')
21         .updateOne({"name":group.name},group)
22     }
23   }
24   //-2. elimino, dall'utente che la possedeva, la playlist, sia da quelle seguite che da quelle
     totali (aka assicuro integrita' referenziale)
25   userToUpdate.playlistsOwned.splice(await userToUpdate.playlistsOwned.indexOf(a.name),1)
26   userToUpdate.playlistsFollowed.splice(await userToUpdate.playlistsFollowed.indexOf(a.name),1)
27   await pwmClient.db("pwm_project").collection('users')
28     .updateOne({"email":decoded.email},userToUpdate);
29   //-1. elimino la playlist
30   await pwmClient.db("pwm_project").collection('playlists').deleteOne(a)
31   //-se non e' esploso niente allora e' tutto okay, spero
32   res.status(200).json({"reason":"done correctly"})
33 }
34 else res.status(400).json({"reason":"Probably you haven't specified the right params"})
35 ...
36 }

```

E per i gruppi:

```

1 async function deleteGroup(req,res){
2   ...
3   let a = await pwmClient.db("pwm_project").collection('groups').findOne({"name":
     validator.escape(req.params.name)})
4   //-3. elimino dagli utenti non owner ogni riferimento a quel gruppo
5   let allUsers = pwmClient.db("pwm_project").collection("users").find({"email":{$not:
     decoded.email}})
6   for(let user in allUsers){
7     if(user.groupsFollowed.some(a.name)){
8       user.groupsFollowed.splice(user.groupsFollowed.indexOf(a.name),1)
9       await pwmClient.db("pwm_project").collection('users').updateOne({"email":user.email},user)
10    }
11  }
12  //-2. elimino dall'owner il gruppo, sia owned che followed
13  userToUpdate.groupsFollowed.splice(await userToUpdate.groupsFollowed.indexOf(a.name),1)
14  userToUpdate.groupsOwned.splice(await userToUpdate.groupsOwned.indexOf(a.name),1)
15  await pwmClient.db("pwm_project").collection('users')
16    .updateOne({"email":decoded.email},userToUpdate);
17  //-1 elimino il gruppo
18  await pwmClient.db("pwm_project").collection('groups').deleteOne(a)
19  //-0. forse e' andato tutto liscio
20  res.status(200).json({"reason":"ok"})
21  ...
22 }

```

---

**⚠:** Un problema simile si avrebbe anche in caso qualcuno desiderasse modificare lo username. Per risolverlo, ho reso impossibile modificare lo username, consentendo invece di modificare le credenziali usate per il login ossia solamente email e password

---

## 4.5 Frontend

## Appendice B: swagger

Di seguito lo swagger.

---

**⚠:** La mancanza degli status codes è dovuta all'uso dei wrapper e degli applicatori, oltre in generale all'uso di funzioni chiamate all'interno di essi, una issue ancora aperta su GitHub, vedasi <https://github.com/davibaltar/swagger-autogen/issues>

---

---

**⚠:** Per un uso più consono dello swagger, si guardi al path `/api-docs`

---

# Social Network for Music: Backend 1.0.0

[ Base URL: localhost:3000/ ]

The backend for SNM project

Schemes

HTTP

## GET All GET requests

GET /genres Gets a list of genres

GET /types Gets a list of types

GET /requireInfo/{kind}/{id} Gets infos about a specific item

GET /logout Performs logout

GET /checkLogin Checks if the user is logged in

GET /group/{name} Gets infos about a group

GET /playlist/info/{name} Gets infos about a playlist

GET /playlist/search/name/{name} Gets playlist with that name as a substring of theirs

GET /playlist/search/tag/{tag} Gets playlist with that tag in their tags

## PUT All PUT requests

PUT /user Updates a user

POST /isStarred Checks if something is in the favorites

POST /register Creates a new user

POST /login Logs in

POST /group Creates a group

POST /playlist Creates a group

## DELETE All DELETE requests

DELETE /user Deletes an user

DELETE /group/{name} Delete a group

DELETE /playlist/{name} Delete a playlist

## General Basic

GET /genres Gets a list of genres

### Parameters

Try it out

No parameters

### Responses

Response content type

application/json

CodeDescription

GET /types Gets a list of types

### Parameters

Try it out

PUT /group/owner Updates a group status: changes the owner

PUT /group/description Updates a group status: changes the description

PUT /group/playlists/add Updates a group status: adds a playlist

PUT /group/playlists/remove Updates a group status: removes a playlist

PUT /group/join/{name} Updates a group status: joins a group

PUT /group/leave/{name} Updates a group status: leaves a group

PUT /playlist/songs/add Updates a playlist status: adds a song

PUT /playlist/songs/remove Updates a playlist status: removes a song

PUT /playlist/owner Updates a playlist status: changes the owner

PUT /playlist/description Updates a playlist status: changes the description

PUT /playlist/follow/{name} Updates a playlist status: start following

PUT /playlist/unfollow/{name} Updates a playlist status: stop following

PUT /playlist/tags/add Updates a playlist status: adds a tag

PUT /playlist/tags/remove Updates a playlist status: removes a tag

PUT /playlist/publish/{name} Updates a playlist status: publishes it

PUT /playlist/private/{name} Updates a playlist status: makes it private

## POST All POST requests

POST /search Performs a search on Spotify

POST /addOrRemoveFavorite Adds or removes favorites

No parameters

### Responses

Response content type

application/json

Code Description

200 OK

## User Everything related to users

POST /register Creates a new user

### Parameters

Try it out

Name Description

obj  
object  
(body)

User data.  
Example Value | Model

```
{  "email": "The@new_email",  "name": "Mario",  "surname": "Rossi",  "userName": "rossimario42",  "birthDate": "01-01-2000",  "favoriteGenres": [    "classical",    "broadway"  ],  "password": "Asup3rs3cur3P4ssw0rd!!!"}
```

Parameter content type

application/json

### Responses

Response content type

application/json

CodeDescription

POST /login Logs in



Parameters

Try it out

Name	Description
obj object <i>(body)</i>	User data. Example Value   Model <pre>{   "email": "my@email.com",   "password": "Asup3rs3cur3P4ssw0rd!!!" }</pre>
Parameter content type application/json	

Responses

Response content type application/json

CodeDescription

GET

/logout Performs logout

^

Parameters

Try it out

No parameters

Responses

Response content type application/json

Code	Description
200	OK
400	Bad Request

GET

/checkLogin Checks if the user is logged in

^

Parameters

Try it out

No parameters

Responses

Response content type application/json

CodeDescription

PUT

/user Updates a user

^

Parameters

Try it out

Name	Description
obj object <i>(body)</i>	User data. Example Value   Model <pre>{   "email": "The@new.email",   "name": "Mario",   "surname": "Rossi",   "birthDate": "01-01-2000",   "favoriteGenres": [     "classical",     "broadway"   ],   "password": "Asup3rs3cur3P4ssw0rd!!!" }</pre>
Parameter content type application/json	

Responses

Response content type application/json

CodeDescription

DELETE

/user Deletes an user

^

Parameters

Try it out

No parameters

Responses

Response content type application/json

CodeDescription

Data

Can be used to interact with data

^

POST

/search Performs a search on Spotify

^

Parameters

Try it out

Name	Description
obj object <i>(body)</i>	User data. Example Value   Model <pre>{   "string": "the string to search for",   "type": [     "album",     "track"   ],   "limit": 0,   "offset": 18 }</pre>
Parameter content type application/json	

Responses

Response content type application/json

CodeDescription

GET

/requireInfo/{kind}/{id} Gets infos about a specific item

^

Parameters

Try it out

Name	Description
kind * required string <i>(path)</i>	kind
id * required string <i>(path)</i>	id

Responses

Response content type application/json

CodeDescription

Favorites

Can be used to interact with Favorites

^

POST

/addOrRemoveFavorite Adds or removes favorites

^

Parameters

Try it out

Name	Description
obj object <i>(body)</i>	User data. Example Value   Model <pre>{   "category": "album",   "id": "1kCHru7uhx8Udzkm4gzRQc",   "name": "Hamilton (Original Broadway Cast Recording)" }</pre>
Parameter content type application/json	

Responses

Response content type application/json

CodeDescription

POST

/isStarred Checks if something is in the favorites

Try it out

Parameters

Name	Description
obj object (body)	User data. Example Value   Model <pre>{   "category": "album",   "id": "1kCHru7uhxBUdzm4gzRQc" }</pre>
Parameter content type application/json	

Responses

Response content type  
application/json

CodeDescription

## Playlists

Can be used to interact with playlists

GET

/playlist/info/{name} Gets info about a playlist

Try it out

Parameters

Name	Description
name * required string (path)	<input type="text" value="name"/>

Responses

Response content type  
application/json

CodeDescription

CodeDescription

GET

/playlist/search/name/{name} Gets playlist with that name as a substring of theirs

Try it out

Parameters

Name	Description
name * required string (path)	<input type="text" value="name"/>

Responses

Response content type  
application/json

CodeDescription

GET

/playlist/search/tag/{tag} Gets playlist with that tag in their tags

Try it out

Parameters

Name	Description
tag * required string (path)	<input type="text" value="tag"/>

Responses

Response content type  
application/json

CodeDescription

POST

/playlist Creates a group

Try it out

Parameters

Name	Description
------	-------------

Responses

Response content type  
application/json

CodeDescription

Name	Description
obj object (body)	Playlist data. Example Value   Model <pre>{   "name": "The name of your new playlist",   "descripcion": "The description of your new playlist" }</pre>
Parameter content type application/json	

Responses

Response content type  
application/json

CodeDescription

PUT

/playlist/songs/add Updates a playlist status: adds a song

Try it out

Parameters

Name	Description
obj object (body)	Playlist data. Example Value   Model <pre>{   "name": "The name of the playlist",   "song_id": "The id of the song to add" }</pre>
Parameter content type application/json	

Responses

Response content type  
application/json

CodeDescription

PUT

/playlist/songs/remove Updates a playlist status: removes a song

Try it out

Parameters

Name	Description
obj object (body)	Playlist data. Example Value   Model <pre>{   "name": "The name of the playlist",   "song_id": "The id of the song to add" }</pre>
Parameter content type application/json	

Responses

Response content type  
application/json

CodeDescription

PUT

/playlist/owner Updates a playlist status: changes the owner

Try it out

Parameters

Name	Description
obj object (body)	Playlist data. Example Value   Model <pre>{   "name": "The name of the playlist",   "new_owner": "The username of the new owner" }</pre>
Parameter content type application/json	

Responses

Response content type  
application/json

CodeDescription

CodeDescription

PUT

/playlist/description

Updates a playlist status: changes the description

Parameters

Try it out

Name	Description
obj	Playlist data.
object	Example Value   Model
(body)	<pre>{   "name": "The name of the playlist",   "new_description": "The new description" }</pre>
Parameter content type	
application/json	

Responses

Response content type

application/json

CodeDescription

PUT

/playlist/follow/{name}

Updates a playlist status: start following

Parameters

Try it out

Name	Description
name	
string	
(path)	

Responses

Response content type

application/json

CodeDescription

PUT

/playlist/unfollow/{name}

Updates a playlist status: stop following

Parameters

Try it out

Name	Description
name	
string	
(path)	

Responses

Response content type

application/json

CodeDescription

PUT

/playlist/tags/add

Updates a playlist status: adds a tag

Parameters

Try it out

No parameters

Responses

Response content type

application/json

CodeDescription

PUT

/playlist/tags/remove

Updates a playlist status: removes a tag

Parameters

Try it out

No parameters

Responses

Response content type

application/json

CodeDescription

PUT

/playlist/publish/{name}

Updates a playlist status: publishes it

Parameters

Try it out

Name	Description
name	
string	
(path)	

Responses

Response content type

application/json

CodeDescription

PUT

/playlist/private/{name}

Updates a playlist status: makes it private

Parameters

Try it out

Name	Description
name	
string	
(path)	

Responses

Response content type

application/json

CodeDescription

DELETE

/playlist/{name}

Delete a playlist

Parameters

Try it out

Name	Description
name	
string	
(path)	

Responses

Response content type

application/json

CodeDescription

Groups

Can be used to interact with groups

GET

/group/{name}

Gets infos about a group

Parameters

Try it out

Name	Description
name	
string	
(path)	

Responses

Response content type

application/json

CodeDescription

DELETE

/group/{name}

Delete a group

Parameters

Try it out

Name

Description

name ★ required

string

(path)

name

Responses

Response content type

application/json

CodeDescription

POST

/group

Creates a group

Parameters

Try it out

Name

Description

obj

object

(body)

Group data.

Example Value | Model

```
{
  "name": "The name of your new group",
  "descrizione": "The description of your new group"
}
```

Parameter content type

application/json

Responses

Response content type

application/json

CodeDescription

PUT

/group/owner

Updates a group status: changes the owner

Parameters

Try it out

Name

Description

obj

object

(body)

Group data.

Example Value | Model

```
{
  "name": "The name of your new group",
  "descrizione": "The description of your new group"
}
```

Parameter content type

application/json

Responses

Response content type

application/json

CodeDescription

Name

Description

obj

object

(body)

Group data.

Example Value | Model

```
{
  "name": "The name of the group",
  "new_owner": "The username of the new owner"
}
```

Parameter content type

application/json

Responses

Response content type

application/json

CodeDescription

PUT

/group/description

Updates a group status: changes the description

Parameters

Try it out

Name

Description

obj

object

(body)

Group data.

Example Value | Model

```
{
  "name": "The name of the group",
  "new_description": "The new description"
}
```

Parameter content type

application/json

Responses

Response content type

application/json

CodeDescription

PUT

/group/playlists/add

Updates a group status: adds a playlist

Parameters

Try it out

Name

Description

obj

object

(body)

Group data.

Example Value | Model

```
{
  "group_name": "The name of the group",
  "playlist_name": "The name of the playlist to add"
}
```

Parameter content type

application/json

Responses

Response content type

application/json

CodeDescription

PUT

/group/playlists/remove

Updates a group status: removes a playlist

Parameters

Try it out

Name

Description

obj

object

(body)

Group data.

Example Value | Model

```
{
  "group_name": "The name of the group",
  "playlist_name": "The name of the playlist to remove"
}
```

Parameter content type

application/json

Responses

Response content type

application/json

CodeDescription

CodeDescription

PUT

/group/join/{name}

Updates a group status: joins a group

Parameters

Try it out

Name

Description

name ★ required

string

(path)

name

Responses

Response content type

application/json

CodeDescription

PUT

/group/leave/{name}

Updates a group status: leaves a group

Parameters

Try it out

Name

Description

name ★ required

string

(path)

name

Responses

Response content type

application/json

CodeDescription

Test

GET

/coffee

## Bibliografia

- [1] *cookie-parser* - *npm*. URL: <https://www.npmjs.com/package/cookie-parser>. (accessed: 03.07.2023).
- [2] *cors* - *npm*. URL: <https://www.npmjs.com/package/cors>. (accessed: 03.07.2023).
- [3] *dotenv* - *npm*. URL: <https://www.npmjs.com/package/dotenv>. (accessed: 03.07.2023).
- [4] *express* - *npm*. URL: <https://www.npmjs.com/package/express>. (accessed: 03.07.2023).
- [5] *express-mongo-sanitize* - *npm*. URL: <https://www.npmjs.com/package/express-mongo-sanitize>. (accessed: 03.07.2023).
- [6] *jsonwebtoken* - *npm*. URL: <https://www.npmjs.com/package/jsonwebtoken>. (accessed: 03.07.2023).
- [7] *mongodb* - *npm*. URL: <https://www.npmjs.com/package/mongodb>. (accessed: 03.07.2023).
- [8] *nodemon* - *npm*. URL: <https://www.npmjs.com/package/nodemon>. (accessed: 03.07.2023).
- [9] *swagger-autogen* - *npm*. URL: <https://www.npmjs.com/package/swagger-autogen>. (accessed: 03.07.2023).
- [10] *swagger-ui-express* - *npm*. URL: <https://www.npmjs.com/package/swagger-ui-express>. (accessed: 03.07.2023).
- [11] *validator* - *npm*. URL: <https://www.npmjs.com/package/validator>. (accessed: 03.07.2023).