Relazione del Progetto di Programmazione Web e Mobile (PWM)

Dipartimento di Informatica Giovanni Degli Antoni

Samuele Manclossi
09882A

10 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

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 disiscriversi. 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.

⚠: Nel momento in cui esco da un gruppo, ogni playlist che avevo condiviso con quel gruppo viene rimossa dal gruppo. Se invece stavo seguendo una playlist, quella playlist rimane seguita, anche se non potrò più accedervi se non rientrando nel gruppo.

⚠: Quando trasferisco la proprietà di una playlist a qualcuno non nel gruppo con cui è condivisa, questa rimane condivisa. Non consiste in una perdita di integrità, ma piuttosto nel rispettare la volontà di chi l'ha trasferita senza prima toglierla dal gruppo.

---

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 C.

Ci sono poi altri packages, come `crypto` o `path`, che venivano già forniti in automatico.
Il file `package.json` risulta quindi composto come segue:

```
{
    "scripts": {
        "start": "nodemon app.js",
        "debug": "nodemon --inspect app.js"
    },
    "dependencies": {
        "cookie-parser": "^1.4.6",
        "cors": "^2.8.5",
        "dotenv": "^16.0.3",
        "express": "^4.18.2",
        "express-mongo-sanitize": "^2.2.0",
        "jsonwebtoken": "^9.0.0",
        "mongodb": "^5.4.0",
        "nodemon": "^1.14.9",
        "swagger-ui-express": "^4.6.2",
        "validator": "^13.9.0"
    },
    "devDependencies": {
        "swagger-autogen": "^2.23.1"
    }
}
```

2.3 Frontend

Le tecnologie usate lato frontend sono HTML5, CSS3 e JavaScript, con l'utilizzo di Bootstrap 5.3. Per le scelte implementative si faccia riferimento all'apposita sezione.

Unica tecnologia significativa usata è stata una libreria: in classe era stato suggerito di poter modificare l'ordine delle canzoni in una playlist. Per farlo, ho usato una libreria per facilitare il drag and drop delle cards. Questa libreria è SortableJS, che consente giusto di ottenere l'effetto di riposizionamento.

Non è stato usato altro codice esterno.

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 C

---

4 Interfaccia utente e front-end

4.1 Pagine e relative funzioni e visibilità

4.1.1 Pagine non richiedenti login

Welcome page - *index.html* La pagina di accoglienza sarà una pagina con pochissime funzionalità, destinata prevalentemente a fare da "vetrina" del servizio offerto. Da essa si potranno trovare i link a tutte le altre funzionalità.

Vi saranno due pulsanti: `login` e `register`, e la navbar per navigare invece la parte pubblica.

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.

Login - *login.html* La pagina di login sarà costituita da un form. Se si è già loggati si viene reindirizzati alla pagina richiesta mediante un parametro, altrimenti ci si può loggare. In ogni caso saranno emessi appositi messaggi per segnalare lo stato.

Ricerca - *search.html* La pagina di ricerca consentirà di effettuare una ricerca per vari campi. Una volta cercato, verranno mostrati i primi risultati di ogni categoria. Se si desidera ottenere maggiori risultati di quella categoria, i parametri vengono ristretti mentre avviene la redirezione a una pagina apposita.

⚠: Al momento non è possibile filtrare i risultati: ho preferito dare la possibilità di selezionare più categorie piuttosto che limitare i risultati. Questo potrebbe diventare uno sviluppo futuro

---

Not Found - *not_found.html* Questa pagina sarà quella mostrata ogniqualvolta sia stata richiesta una pagina non esistente.

4.1.2 Pagine richiedenti login

Playlist - *playlists.html* In questa pagina si potranno cercare le playlist pubbliche o comunque condivise con sé, e crearne di nuove.

Groups - *groups.html* Questa pagina consentirà la creazione, la visualizzazione e il filtraggio dei gruppi.

Spiegazione di una playlist o di un gruppo *explainPlaylist.html*—*explainGroup.html* Come la pagina di `describe`, ma per playlist o gruppi. Dovranno includere l'unirsi e l'uscire dai gruppi, l'unire o il rimuovere una playlist a un gruppo e una canzone ad una playlist.

Profilo - *profile.html* Questa pagina includerà tutte le informazioni legate all'utente, come preferiti, dati personali, playlist seguite o possedute, gruppi in cui si è o posseduti.

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*

Permette di tornare alla pagina `index.html`.

4.2.2 Pulsante *Playlists*

Consente di tornare alla pagina delle playlist pubbliche.

4.2.3 Pulsante *Search*

Consente di tornare alla pagina di ricerca.

4.2.4 Pulsante *Groups*

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

4.2.5 Dropdown *User*

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

Register Visibile solo a chi non è loggato, permette di registrarsi.

Login Visibile solo a chi non è loggato, permette di loggarsi.

Logout Visibile solo a chi è loggato, permette di effettuare il logout.

My Favorites Visibile solo a chi è loggato, permette di andare al profilo, nella sezione dedicata ai preferiti.

My Playlists Visibile solo a chi è loggato, permette di andare al profilo, nella sezione dedicata alle playlists.

My Groups Visibile solo a chi è loggato, permette di andare al profilo, nella sezione dedicata ai gruppi.

Profile Visibile solo a chi è loggato, permette di andare al profilo.

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.

Struttura dati nel database

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

Users:

```
{
    "_id": {...},
    "name": "First Name",
    "surname": "Surname",
    "userName": "userName",
        UNIQUE
    "email": "a@b.c", UNIQUE
    "birthDate": "2003-10-01",
    "favoriteGenres": [],
    "password": "...",
    "favorites": {
        "album": [],
        "artist": [],
        "audiobook": [],
        "episode": [],
        "show": [],
        "track": []
    },
    "playlistsFollowed": [],
    "playlistsOwned": [],
    "groupsFollowed": [],
    "groupsOwned": []
}
```

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

Playlists:

```
{
    "_id":{...},
    "name": "myList", UNIQUE
    "description": "this is a
        playlist about old
        finnish songs",
    "tags":[
        "finnish",
        "old",
        "42"
    ],
    "visibility": true,
    "owner": "userName"
}
.
```

Dove `owner` referenzia `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.

Groups:

```
{
    "_id":{...},
    "name":"myGroup", UNIQUE
    "decription":"This is a
        group for lovers of
        classical music. Join
        this group to gain
        access to more than
        15 playlists!",
    "playlistsShared":[
        "classicalMusic",
        "musicaClassica",
        ...
    ],
    "owner":"userName",
    "users":[
        "userName",
        "user42",
        "lambda",
        "SophosIoun",
        ...
    ]
}
```

Dove `owner` e `users` referenziano `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:

```
{
    "titolo":"15 secondi di muratori al lavoro",
    "durata":15000,
    "cantante":"Me medesimo",
    "anno_di_pubblicazione":"2023",
}
```

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

Idea: L'uso dei nomi anziché degli _id permette di ridurre al minimo indispensabile le richieste al backend per informazioni parziali. Questo complica le cose (modificare i nomi allungherebbe notevolmente il codice, e pertanto rimarrà uno sviluppo futuro non svolto in questa occasione)

Backend

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:

```
MONGONAME=XXXXXXXXXXX
MONGOPASSWORD=XXXXXXXXXXX
PORT=3000
CLIENT_ID=XXXXXXXXXXX
CLIENT_SECRET=XXXXXXXXXXX
SECRET=XXXXXXXXXXX
```

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`.

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 parlerà 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:

```
async function nomeMetodo(req,res){
    var pwmClient = await new mongoClient(mongoUrl).connect()
    const token = req.cookies.token
    if(token == undefined) res.status(401).json({"reason": `Invalid login`})
    else{
        jwt.verify(token,process.env.SECRET, async (err,decoded) =>{
            if(err){
                res.status(401).json(err)
                pwmClient.close()
            }
            else{
                //vero e proprio codice del metodo
                pwmClient.close()
            }
        })
    }
}
```

: 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

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:

```
var token = {
    value: "none",
    expiration: 42,
    regenAndThen : function(func_to_apply,paramA,paramB){
        fetch(baseUrls.token, {
            method: "POST",
            headers: {
            Authorization: "Basic " + btoa(`${process.env.CLIENT_ID}:${process.env.CLIENT_SECRET}`),
            "Content-Type": "application/x-www-form-urlencoded",
            },
            body: new URLSearchParams({ grant_type: "client_credentials" }),
        })
        .then((response) => response.json())
        .then((tokenResponse) => {
            this.expiration = new Date().getTime(); //ms
            this.value=tokenResponse.access_token
            func_to_apply(paramA,paramB)
        })
    },
    hasExpired : function(){
        if(((new Date().getTime()-this.expiration)/1000/60)>=59){
            return true;
        }
        else return false;
    }
}
```

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

Idea: Non ho usato un setInterval per rigenerarlo ogni ora, ma piuttosto l'applicatore: essendo gli utenti basati al 100% in Italia, sarebbe stato inutile rigenerarlo anche ad orari in cui nessun utente era collegato. Ho pertanto ritenuto più corretta, visti i requisiti, una soluzione del genere.

---

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

```
function perform(questo,paramA,paramB){
    if(token.hasExpired()) token.regenAndThen(questo,paramA,paramB)
    else questo(paramA,paramB)
}
```

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.

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 C, 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 erroneo 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.

```
async function deleteUser(req,res){
    ...
    let user = await pwmClient.db("pwm_project").collection('users').findOne({"email": decoded.email})
    //-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
    let allGroups = await pwmClient.db("pwm_project").collection("groups").find({}).toArray()
    for(let index in allGroups){
        let group = allGroups[index]
        if(group.users.some(element => element == user.userName)){
            group.users.splice(group.users.indexOf(user.userName),1)
            await pwmClient.db("pwm_project").collection('groups')
                .updateOne({"name":group.name},{$set:{"users":group.users}})
        }
    }
    //-3 bis elimino ogni gruppo posseduto dall'utente
    await pwmClient.db("pwm_project").collection("groups").deleteMany({"owner":user.userName})
    //-2 elimino ogni playlist dell'utente da ogni lista di playlist seguite altrui
    allGroups = await pwmClient.db("pwm_project").collection("groups").find({}).toArray()
    let allPlaylists =
        pwmClient.db("pwm_project").collection("playlists").find({"owner":user.userName}).toArray()
    let allUsers = pwmClient.db("pwm_project").collection("playlists")
        .find({"email":{$ne:decoded.email}}).toArray()
    for(let index1 in allPlaylists){
        let playlist = allPlaylists[index1]
        for(let index2 in allUsers){
            let utente = allUsers[index2]
            if(utente.playlistsFollowed.some(element => element == playlist.name)){
                utente.playlistsFollowed.splice(utente.playlistsFollowed.indexOf(playlist.name),1)
                await pwmClient.db("pwm_project").collection("users").updateOne({"email":utente.email},
                    {$set:{"playlistsFollowed":utente.playlistsFollowed}})
            }
        }
```

```
        for(let index in allGroups){
            let group = allGroups[index]
            if(group.playlistsShared.some(element => element == playlist.name)){
                group.playlistsShared.splice(group.playlistsShared.indexOf(playlist.name),1)
                await pwmClient.db("pwm_project").collection("groups").updateOne({"name":group.name},
                    {$set:{"playlistsShared":group.playlistsShared}})
            }
        }
    }
    //-2 bis elimino ogni playlist owned dall'utente
    await pwmClient.db("pwm_project").collection("playlists").deleteMany({"owner":user.userName})
    //-1 elimino l'account dell'utente
    await pwmClient.db('pwm_project').collection('users').deleteOne({"email":decoded.email})
    //0. forse e' andato tutto liscio, e spero di non aver lasciato riferimenti pending da qualche
        parte
    pwmClient.close()
    res.status(200).json({"reason":"ok"})
    ...
}
```

Mentre per le playlist:

```
async function deletePlaylist(req,res){
    ...
    let a = await pwmClient.db("pwm_project").collection('playlists').findOne({"name":
        validator.escape(req.params.name)})
    let userToUpdate = await
        pwmClient.db("pwm_project").collection('users').findOne({"email":decoded.email});
    if(a != null && a != undefined && isOwner(a,userToUpdate.userName)){
        //-3. per ogni utente, diverso dall'owner, elimino la playlist da quelle seguite, laddove
            presente.
        let allUsers = pwmClient.db("pwm_project").collection("users").find({"email":{$ne:
            decoded.email}}).toArray()
        try{
            for(let index in allUsers){
                let user = allUsers[index]
                if(user.playlistsFollowed.some(element => element == a.name)){
                    user.playlistsFollowed.splice(user.playlistsFollowed.indexOf(a.name),1)
                    await
                        pwmClient.db("pwm_project").collection('users').updateOne({"email":user.email},
                        {$set:{"playlistsFollowed":user.playlistsFollowed}})
                }
            }
        }catch(e){log(...);}
        //-2.5. per ogni gruppo elimino la playlist da quelle seguite, laddove presente
        let allGroups = await pwmClient.db("pwm_project").collection("groups").find({}).toArray()
        try{
            for(let index in allGroups){
                let group = allGroups[index]
                if(group.playlistsShared.some(element => element == a.name)){
                    group.playlistsShared.splice(group.playlistsShared.indexOf(a.name),1)
                    await
                        pwmClient.db("pwm_project").collection('groups').updateOne({"name":group.name},
                        {$set:{"playlistsShared":group.playlistsShared}})
                }
            }
        }catch(e){log(...);}
        //-2. elimino, dall'utente che la possedeva, la playlist, sia da quelle seguite che da quelle
            totali (ossia cerco di assicurare l'integrita' referenziale)
        userToUpdate.playlistsOwned.splice(await userToUpdate.playlistsOwned.indexOf(a.name),1)
        userToUpdate.playlistsFollowed.splice(await userToUpdate.playlistsFollowed.indexOf(a.name),1)
        await pwmClient.db("pwm_project").collection('users').updateOne({"email":decoded.email},
            {$set:{"playlistsFollowed":userToUpdate.playlistsFollowed,
                "playlistsOwned":userToUpdate.playlistsOwned}});
        //-1. elimino la playlist
        await pwmClient.db("pwm_project").collection('playlists').deleteOne(a)
```

```
        //se non e' esploso niente allora e' tutto okay, spero
        res.status(200).json({"reason":"done correctly"})
    }
    else res.status(400).json({"reason":"Probably you haven't specified the right params"})
    ...
}
```

E per i gruppi:

```
async function deleteGroup(req,res){
    ...
    let a = await pwmClient.db("pwm_project").collection('groups').findOne({"name":
        validator.escape(req.params.name)})
    //-3. elimino dagli utenti non owner ogni riferimento a quel gruppo
    let allUsers = await pwmClient.db("pwm_project").collection("users").find({"email":{$ne:
        decoded.email}}).toArray()
    for(let index in allUsers){
        let user = allUsers[index]
        if(user.groupsFollowed.some(element => element == a.name)){
            user.groupsFollowed.splice(user.groupsFollowed.indexOf(a.name),1)
            await pwmClient.db("pwm_project").collection('users').updateOne({"email":user.email},
                {$set:{"groupsFollowed":user.groupsFollowed}})
        }
    }
    //-2. elimino dall'owner il gruppo, sia owned che followed
    let userToUpdate = await
        pwmClient.db("pwm_project").collection("users").findOne({"email":decoded.email})
    userToUpdate.groupsFollowed.splice(await userToUpdate.groupsFollowed.indexOf(a.name),1)
    userToUpdate.groupsOwned.splice(await userToUpdate.groupsOwned.indexOf(a.name),1)
    await pwmClient.db("pwm_project").collection('users').updateOne({"email":decoded.email},
        {$set:{"groupsFollowed":userToUpdate.groupsFollowed,
            "groupsOwned":userToUpdate.groupsOwned}});
    //-1 elimino il gruppo
    await pwmClient.db("pwm_project").collection('groups').deleteOne(a)
    //0. forse e' andato tutto liscio
    res.status(200).json({"reason":"ok"})
    ...
}
```

⚠: 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

---

Frontend

Le scelte implementative effettuate riguardanti il frontend sono molteplici e molto ampie, pertanto verranno trattate separatamente. In questa sede saranno riportate solo le scelte che ritengo significative, per evitare di riportare nozioni a mio giudizio inutili.

Redirect e rischio di loop

In svariate situazioni un redirect avrebbe potuto essere una ottima idea. Quando ad esempio qualcuno, senza essere loggato (magari usando un vecchio link o avendo salvato nei `bookmarks` l'indirizzo) cerca di accedere al proprio profilo, egli viene reindirizzato alla login, e successivamente al proprio profilo.

Una scelta simile, pensavo inizialmente, si potrebbe applicare anche nel caso di playlist per le quali non ci fosse la possibilità, causa permessi mancanti, di visualizzarle. Alla fine ho optato invece per non farlo perché il rischio era che un utente, loggato, non avesse accesso alla playlist in quanto privata, venisse quindi reindirizzato alla login. Constatando che è già loggato, la pagina di login reindirizzerebbe verso la pagina richiesta inizialmente, pur non essendo di fatto cambiate le credenziali, e pertanto la pagina delle playlist avrebbe nuovamente rediretto alla login, generando un ciclo.

⚠: Si noti che anche nel caso peggiore il ciclo non avrebbe potuto durare più di un'ora causa scadenza dei token JWT. Il tempo sarebbe comunque stato decisamente eccessivo.

⚠: Un ulteriore punto di criticità è il fatto che se un utente cambia password senza cambiare email, chiunque sia loggato mediante quella email rimane loggato, sempre per un'ora.

Eventi

In molte diverse componenti del Javascript frontend ho avuto necessità di creare degli eventi, per ragioni fondamentalmente riassumibili in tre tipi. Queste sono spiegate nei paragrafi seguenti, a partire da uno dei casi concreti e dall'esempio di codice.

Responsiveness La pagina che mostra le informazioni ottenute da Spotify deve essere responsive. Questo normalmente, usando le classi di Bootstrap, si risolverebbe senza bisogno di particolari menzioni. In questo caso, tuttavia, il mio appoggiarmi a https://flagcdn.com/ rende necessaria una operazione diversa: le flag vengono richieste con una dimensione fissa, pertanto all'aumentare dello schermo oltre i 2k rischiano di diventare microscopiche.

Ho pertanto optato per la creazione dell'evento sotto, in grado di recuperare le informazioni sullo stato di cui occorre recuperare la bandiera (mediante una apposita regular expression) e aumentare o ridurre la dimensione in base alle necessità.

```
window.addEventListener('resize', () =>{
    const re = /\/.....\//
    let a = document.getElementsByClassName('flag')
    for (let i=0;i<a.length;i++){
        let element = a[i].src
        let code = element.split(re)[1].split(".png")[0]
        a[i].src = `https://flagcdn.com/${window.screen.availWidth<2000?"16x12":"64x48"}/${code}.png`
    }
});
```

Selezione dei gruppi Mediante la proprietà `oninput` di un `input` di tipo `text` sono in grado di filtrare dinamicamente i gruppi in modo che i titoli contengano come sottostringa il testo inserito. Si può notare questa cosa in azione nella pagina `groups.html`.

```
function changing(){
    let query = document.getElementById('search').value.toLowerCase()
    for(let i =0;i<groupList.length;i++){
        let card = document.getElementById('card-groups-followed-'+i)
        if(!card.getElementsByClassName(`card-title`)[0].innerHTML.toLowerCase().includes(query))
            card.classList.add('d-none')
        else card.classList.remove('d-none')

    }
}
```

Cambio della voce selezionata in un select Per verificare che una canzone appartenga o meno ad una playlist, e quindi se essa possa o meno essere aggiunta o rimossa, ho creato un evento, come segue:

```
document.querySelector('#floatingSelect').addEventListener('change', () =>{
    let playlist = document.querySelector('#floatingSelect').value
    let id = params.get('value')
    fetch(`/playlist/info/${playlist}`).then(async a => {
        if(a.ok){
            response = await a.json()
            if(!response.songs.some(element => element.id == id)){
                //button to remove it
                console.log(document.getElementById('the-mystic-button').onclick)
```

```
                document.getElementById('the-mystic-button').innerHTML='Add it!'
                document.getElementById('the-mystic-button').classList.add('text-bg-success')
            }
            else{
                //button to add it
                console.log(document.getElementById('the-mystic-button').onclick)
                document.getElementById('the-mystic-button').innerHTML='Remove it!'
                document.getElementById('the-mystic-button').classList.add('text-bg-danger')
            }
        }
    })
});
```

Sortable e riordinamento delle canzoni

Per riordinare le canzoni in una playlist ho fatto uso, come spiegato nelle `tecnologie utilizzate`, della libreria `SortableJS`. Questa consente di istanziare un oggetto Sortable su un nodo `HTML`, in modo tale che i suoi nodi figli siano riordinabili.

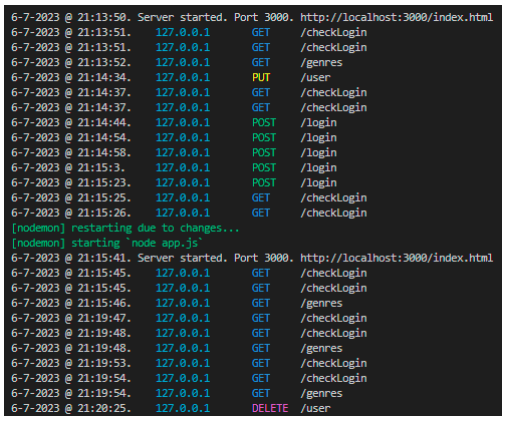
Quella di renderli riordinabili è una delle parti meno significative del processo di riordino effettivo, pertanto sarà soltanto qui riportato come si può istanziare tale oggetto.

```
new Sortable(document.getElementById('anche-questo-songs'),{
        animation: 150,
        ghostClass: 'blue-background-class'
    });
```

Per ulteriore documentazione si consulti la loro repo su GitHub.

Appendice B: schermate di funzionamento

Sistema di logging del backend



Appendice C: 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

Swagger
Supported by SMARTBEAR

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 /grouplist Get a list of all groups
- 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
- PUT /group/owner Updates a group status: changes the owner
- PUT /group/description Updates a group status: changes the description
- PUT /group/join/{name} Updates a group status: joins a group
- PUT /group/leave/{name} Updates a group status: leaves a group
- PUT /playlist/sort/{name} Updates a playlist status: adds 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/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
- 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 /group/playlist Updates a group status: adds a playlist
- POST /playlist Creates a group
- POST /playlist/song Updates a playlist status: adds a song
- POST /playlist/tag Updates a playlist status: adds a tag

DELETE All DELETE requests

- DELETE /user Deletes an user
- DELETE /group/{name} Delete a group
- DELETE /group/playlist Updates a group status: removes a playlist
- DELETE /playlist/song Updates a playlist status: removes a song
- DELETE /playlist/tag Updates a playlist status: removes a tag
- 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

Code	Description
200	The result of the query is sent back **Example Value** | Model

```
{
  "status": 200,
  "results": [
    "acoustic",
    "afrobeat",
    "alt-rock",
    "alternative"
  ]
}
```

GET /types Gets a list of types

Parameters Try it out

No parameters

Responses Response content type application/json

Code	Description
200	The known types are sent back **Example Value** | Model

```
[
  "album",
  "artist",
  "episode",
  "show",
  "track"
]
```

User Everything related to users

POST /register Creates a new user

Parameters Try it out

Name	Description
obj	User data.

Name	Description
object *(body)*	**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

Code	Description
200	The user has been inseted into the database
400	The user was already registered, or at least his email or username were, or some other invalid data was inserted **Example Value** | Model

```
{
  "code": -1,
  "reason": "You are missing some fields..."
}
```

500	The database refused the insertion, more details are provided in the response **Example Value** | Model

```
{
  "code": 6,
  "reason": "Generic error: explanation"
}
```

POST `/login` Logs in

Parameters Try it out

Name	Description
obj	User data.

Name	Description
object *(body)*	**Example Value** | Model

```
{
  "email": "my@email.com",
  "password": "Asup3rs3cur3P4ssw0rd!!!"
}
```

Parameter content type: application/json

Responses Response content type: application/json

Code	Description
200	The user has correctly logged in **Example Value** | Model

```
{
  "code": 4,
  "reason": "Logged successfully!"
}
```

400	The email was not an email **Example Value** | Model

```
{
  "code": 2,
  "reason": "This isn't really an email, is it?"
}
```

401	A user with such credentials does not exist **Example Value** | Model

```
{
  "code": 3,
  "reason": "This user does not exist or its password is not the one you inserted."
}
```

GET `/logout` Performs logout

Parameters Try it out

No parameters

Responses Response content type: application/json

Code	Description
200	The logout was successful **Example Value** | Model

```
{
  "success": true
}
```

400	The logout failed - probably the user was not logged in **Example Value** | Model

```
{
  "success": false
}
```

GET `/checkLogin` Checks if the user is logged in

Parameters Try it out

No parameters

Responses Response content type: application/json

Code	Description
200	The user is logged in **Example Value** | Model

```
{
  "name": "Name",
  "surname": "Surname",
  "userName": "User Name",
  "email": "email@email.email",
  "birthDate": "YYYY-MM-DD",
  "favoriteGenres": [],
  "favorites": {
    "album": [],
    "artist": [],
    "audiobook": [],
    "episode": [],
    "show": [],
    "track": []
  },
  "playlistsFollowed": [],
  "playlistsOwned": [],
  "groupsOwned": [],
```


Code	Description

```
  "groupsFollowed": []
}
```

401	The user is not logged in **Example Value** | Model

```
{
  "reason": "Invalid login"
}
```

PUT /user Updates a user

Parameters Try it out

Name	Description

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

Responses Response content type application/json

Code	Description

200	The user successfully updates his data
	Example Value | Model
	{
	"code": 0,
	"explanation": "you will now be logged out. Please re-login with your new crede
	ntials"
	}

400	A user with such an email is already present. The email needs to be unique.
	Example Value | Model
	{
	"code": 5,
	"reason": "Already present user: please choose a different username or email"
	}

401	The user is not logged in
	Example Value | Model
	{
	"reason": "Invalid login"
	}

Code	Description

500	The database refused the operation. More details are available in the response
	Example Value | Model
	{
	"code": 6,
	"reason": "Generic error: Explanation"
	}

DELETE /user Deletes an user

Parameters Try it out

No parameters

Responses Response content type application/json

Code	Description

200	The user was successfully deleted
	Example Value | Model
	{
	"reason": "ok"
	}

400	There was an error. More details in the response
	Example Value | Model
	{
	"reason": "Generic error: Explanation"
	}

401	The user is not logged in
	Example Value | Model
	{
	"reason": "Invalid login"
	}

Data Can be used to interact with data

POST /search Performs a search on Spotify

Parameters Try it out

Name	Description

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

Responses Response content type application/json

Code	Description
200	The result is sent back

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

Parameters Try it out

Name	Description
kind * required string (path)	kind
id * required string (path)	id

Responses Response content type application/json

Code	Description
200	The result is sent back

Favorites Can be used to interact with Favorites

POST /addOrRemoveFavorite Adds or removes favorites

Parameters Try it out

Name	Description

obj
object
(body)
	User data.
	Example Value | Model
	{
	"category": "album",
	"id": "1kCHru7uhxBUdzkm4gzRQc",
	"name": "Hamilton (Original Broadway Cast Recording)"
	}
	Parameter content type
	application/json

Responses Response content type application/json

Code	Description

200	The query was executed correctly
	Example Value | Model
	{
	"removed": true
	}

401	The user was not logged in
	Example Value | Model
	{
	"reason": "Invalid login"
	}

Code Description

POST /isStarred Checks if something is in the favorites

Parameters Try it out

Name	Description

obj
object
(body)
	User data.
	Example Value | Model
	{
	"category": "album",
	"id": "1kCHru7uhxBUdzkm4gzRQc"
	}
	Parameter content type
	application/json

Responses Response content type application/json

Code	Description

200	A boolean value is sent back to indicate if the element is starred or not
	Example Value | Model
	{
	"favorite": true
	}

401	The user was not logged in
	Example Value | Model
	{
	"reason": "Invalid login"
	}

Playlists Can be used to interact with playlists

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

Parameters Try it out

Name	Description
name * required string (path)	name

Responses Response content type application/json

Code	Description

200	The response contains the requested informations
	Example Value | Model
	{
	"name": "myList",
	"description": "this is a playlist about old finnish songs",
	"tags": [
	"finnish",
	"old",
	"42"
	],
	"visibility": true,
	"owner": "userName",
	"doIOwnIt": false,
	"following": true
	}

400	Something failed. Refer to the response for more details
	Example Value | Model
	{
	"reason": "Generic error: Explanation"
	}

401	The user is not logged in
	Example Value | Model
	{
	"reason": "Invalid login"
	}

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

Parameters Try it out

Name	Description
name * required string (path)	name

Responses Response content type application/json

Code	Description

200	The response contains the requested informations
	Example Value | Model
	[
	{
	"name": "myList",
	"description": "this is a playlist about old finnish songs",
	"tags": [
	"finnish",
	"old",
	"42"
	],
	"visibility": true,
	"owner": "userName"
	}
	]

400	Something failed. Refer to the response for more details
	Example Value | Model
	{
	"reason": "Generic error: Explanation"
	}

401	The user is not logged in
	Example Value | Model
	{
	"reason": "Invalid login"
	}

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

Parameters Try it out

Name	Description
tag * required string (path)	tag

Responses Response content type application/json

Code	Description

200	The response contains the requested informations
	Example Value | Model
	[
	{
	"name": "myList",
	"description": "this is a playlist about old finnish songs",
	"tags": [
	"finnish",
	"old",
	"42"
	],
	"visibility": true,
	"owner": "userName"
	}
	]

400	Something failed. Refer to the response for more details
	Example Value | Model
	{
	"reason": "Generic error: Explanation"
	}

401	The user is not logged in
	Example Value | Model
	{
	"reason": "Invalid login"
	}

POST /playlist Creates a group

Parameters Try it out

Name	Description
obj object *(body)*	Playlist data. **Example Value** | Model `{ "nome": "The name of your new playlist", "descrizione": "The description of your new playlist" }` **Parameter content type** application/json

Responses — Response content type: application/json

Code	Description
200	A playlist was created successfully **Example Value** | Model `{ "reason": "inserted correctly" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Probably you haven't specified the right params" }`
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

PUT `/playlist/sort/{name}` Updates a playlist status: adds a song

Parameters — Try it out

Name	Description
name * required string *(path)*	name
obj object *(body)*	Playlist data. **Example Value** | Model `{ "order": ["7m9XR7FquXLP1FewdAcNS9", "6oF8ueLn5hIl4PRp17sxW6", "6p7jXaTJdpzGWnOJoK2jYr", "3lXyAQ0kekAvY5LodpWmUs", "27MB0qHaYAZiTlwg25js1Y", "7EqpEBPOohgk7NnKvBGFWo", "1CzeuSrm71wHP9qsjg7p3F", "3dP0pLbg9OfVwssDjp9aT0", "54Sc7mZQ1RM03STpk4SfaA", "2yBMVrq96wb9OHbMdBs0lF", "3nJYcY9yvKP8Oi2M18brXt", "6OG1S805gIrH5nAQbEOPY3", "2G91ekfCh83S0lt2yfffBz", "71X7bPDljJHrmEGYCe7kQ8", "0NJWhm3hUwIZSy5s0TGJ8q", "4cxvludVmQxryrnx1m9FqL", "6dr7ekfhlbquvsVY8D7gyk", "4TTV7EcfroSLWzXRY6gLv6"] }` **Parameter content type** application/json

Responses — Response content type: application/json

Code	Description
200	The order of the songs was changed successfully **Example Value** | Model `{ "reason": "everything is fine" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

POST `/playlist/song` Updates a playlist status: adds a song

Parameters — Try it out

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

Responses — Response content type: application/json

Code	Description
200	The song was added successfully **Example Value** | Model `{ "reason": "done" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

DELETE `/playlist/song` Updates a playlist status: removes a song

Parameters — Try it out

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

Responses — Response content type: application/json

Code	Description
200	The song was removed successfully **Example Value** | Model `{ "reason": "done" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

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

Parameters Try it out

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

Responses Response content type application/json

Code	Description
200	The owner changed, as requested **Example Value** | Model `{ "reason": "ok" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

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

Parameters Try it out

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

Responses Response content type application/json

Code	Description
200	The playlist changed, as requested **Example Value** | Model `{ "reason": "ok" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

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

Parameters Try it out

Name	Description
name * required `string` *(path)*	name

Responses Response content type application/json

Code	Description
200	The user is now following the playlist **Example Value** | Model `{ "reason": "ok" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

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

Parameters Try it out

Name	Description
name * required `string` *(path)*	name

Responses Response content type application/json

Code	Description
200	The user has ceased following the playlist **Example Value** | Model `{ "reason": "ok" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

POST `/playlist/tag` Updates a playlist status: adds a tag

Parameters Try it out

Name	Description
obj `object` *(body)*	Playlist data. **Example Value** | Model `{ "name": "The name of the playlist", "tag": "The new tag" }` **Parameter content type** application/json

Responses Response content type application/json

Code	Description
200	A tag was added to the playlist successfully
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

Code	Description
	[redacted]

DELETE `/playlist/tag` Updates a playlist status: removes a tag

Parameters Try it out

Name	Description
obj `object` *(body)*	Playlist data. **Example Value** | Model `{ "name": "The name of the playlist", "tag": "The old tag" }` **Parameter content type** application/json

Responses Response content type application/json

Code	Description
200	A tag was removed from the playlist successfully **Example Value** | Model `{ "reason": "ok" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

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

Parameters Try it out

Name	Description
name * required `string` *(path)*	name

Responses Response content type application/json

Code	Description
200	The playlist is now public **Example Value** | Model `{ "reason": "ok" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

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

Parameters Try it out

Name	Description
name * required `string` *(path)*	name

Responses Response content type application/json

Code	Description
200	The playlist is now private **Example Value** | Model `{ "reason": "ok" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

DELETE `/playlist/{name}` Delete a playlist

Parameters Try it out

Name	Description
name * required `string` *(path)*	name

Responses Response content type application/json

Code	Description
200	The playlist was successfully deleted **Example Value** | Model `{ "reason": "done correctly" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

Groups Can be used to interact with groups

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

Parameters Try it out

Name	Description
name * required `string` *(path)*	name

Responses Response content type application/json

Code	Description
200	The response contains the data requested **Example Value** | Model `{ "name": "myGroup", "decription": "This is a group for ...", "playlistsShared": [ "classicalMusic", "musicaClassica" ], "owner": "userName", "users": [ "userName", "user42", "lambda", "SophosIoun" ], "doIOwnIt": false, "following": true }`
400	Invalid data was providen

Code	Description
	Example Value | Model `{ "reason": "bad data" }`
401	The user was not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

DELETE `/group/{name}` Delete a group

Parameters Try it out

Name	Description
name * required `string` *(path)*	name

Responses Response content type application/json

Code	Description
200	The group was deleted **Example Value** | Model `{ "reason": "ok" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`
401	The user is not logged in

Code	Description
	Example Value | Model `{ "reason": "Invalid login" }`

GET `/grouplist` Get a list of all groups

Parameters Try it out

No parameters

Responses Response content type application/json

Code	Description
200	The response contains the requested data **Example Value** | Model `[ { "name": "myGroup", "decription": "This is a group for ...", "playlistsShared": [ "classicalMusic", "musicaClassica" ], "owner": "userName", "users": [ "userName", "user42", "lambda", "SophosIoun" ] } ]`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Generic error: Explanation" }`
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

Code	Description
	`}`

POST `/group` Creates a group

Parameters Try it out

Name	Description
obj `object` *(body)*	Group data. **Example Value** | Model `{ "nome": "The name of your new group", "descrizione": "The description of your new group" }` Parameter content type: application/json

Responses Response content type: application/json

Code	Description
200	A new group was created **Example Value** | Model `{ "reason": "inserted correctly" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "Probably you haven't specified the right params" }`
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

Code	Description

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 the group", "new_owner": "The username of the new owner" }` Parameter content type: application/json

Responses Response content type: application/json

Code	Description
200	The owner was changed, according to the request **Example Value** | Model `{ "reason": "ok" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "you do not own this group, or some other data you inserted is not valid. Stop trying to hack me, please" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

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

Code	Description
200	The description was changed, according to the request **Example Value** | Model `{ "reason": "ok" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "you do not own this group, or some other data you inserted is not valid. Stop trying to hack me, please" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

POST `/group/playlist` 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

Code	Description
200	The playlist was added, according to the request **Example Value** | Model `{ "reason": "done successfully" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "invalid data or wrong permissions" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

DELETE `/group/playlist` 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

Code	Description
200	The playlist was removed, according to the request **Example Value** | Model `{ "reason": "done successfully" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "invalid data or wrong permissions" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

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

Code	Description
200	The user joined the group, according to the request **Example Value** | Model `{ "reason": "ok" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "bad data or already in this group" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

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

Code	Description
200	The user left the group, according to the request **Example Value** | Model `{ "reason": "ok" }`
400	Something failed. Refer to the response for more details **Example Value** | Model `{ "reason": "bad data or not in this group" }`

Code	Description
401	The user is not logged in **Example Value** | Model `{ "reason": "Invalid login" }`

Test

GET `/coffee`

Parameters Try it out

No parameters

Responses Response content type application/json

Code	Description
418	I'm a teapot **Example Value** | Model `{ "answer": "I'm not a teapot, but I cannot brew 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).