

Ingegneria del software

DOCUMENTO DI SVILUPPO  
VERSIONE: 0.01

WEB MUSIC PLAYER  
*Gruppo T27*

Anno accademico 2022/2023

# Indice

<b>1 Scopo del documento</b>	<b>4</b>
<b>2 UserFlow</b>	<b>5</b>
<b>3 Implementazione e documentazione delle API</b>	<b>6</b>
3.1 Struttura del backend . . . . .	6
3.2 Dipendenze del progetto . . . . .	6
3.3 Modellazione dati nel database . . . . .	7
3.4 Specifica delle risorse . . . . .	8
3.4.1 Estrazione delle risorse . . . . .	8
3.4.2 Modello delle risorse . . . . .	9
<b>4 Sviluppo delle API</b>	<b>19</b>
4.1 Registrazione . . . . .	19
4.2 Accesso . . . . .	20
4.3 Ricerca . . . . .	20
4.4 Elimina account . . . . .	20
4.5 Carica brano . . . . .	21
4.6 Elimina Brano . . . . .	23
4.7 Modifica brano . . . . .	23
4.8 Ottieni brano . . . . .	23
4.9 Ottieni preferiti . . . . .	25
4.10 Modifica preferiti . . . . .	25
<b>5 Documentazione delle API</b>	<b>26</b>
<b>6 Implementazione del frontend</b>	<b>31</b>
<b>7 GitHub repository e deployment</b>	<b>33</b>

---



## 1 Scopo del documento

Il presente documento riporta lo sviluppo di una parte del progetto Web Music Player. Dalla definizione delle API, alla loro implementazione, al deployment del progetto, questo documento descriverà le varie fasi che hanno portato alla creazione del software. Queste fasi sono:

- la specifica delle risorse
- l'implementazione e documentazione delle API
- il testing delle API
- l'implementazione del frontend
- il deployment dell'applicazione

Segue una breve descrizione dell'oggetto dello sviluppo.

Il sito web realizzato permette la **registrazione** e **accesso** alla piattaforma da parte degli utenti. Gli utenti standard possono **ricercare** canzoni tramite il loro titolo e aggiungere i risultati alla loro lista dei **preferiti**. Da qui le canzoni possono essere rimosse. Gli utenti creator possono inoltre **caricare** nuovi brani sulla piattaforma. Quando ricercano un brano, se questo risulta essere stato caricato da loro, possono decidere di **modificarlo** o di **eliminarlo** dalla piattaforma. Entrambe le tipologie di utenti possono far rimuovere il proprio account dalla piattaforma, cancellando tutti i dati a loro associati.

## 2 UserFlow

In questa sezione del documento si riportano gli “User Flows” per il ruolo dell’utente. L’immagine che segue descrive lo user flow relativo alle varie funzioni trattate in questa fase di sviluppo. Sarà anche presente una breve legenda che descrive i simboli utilizzati.

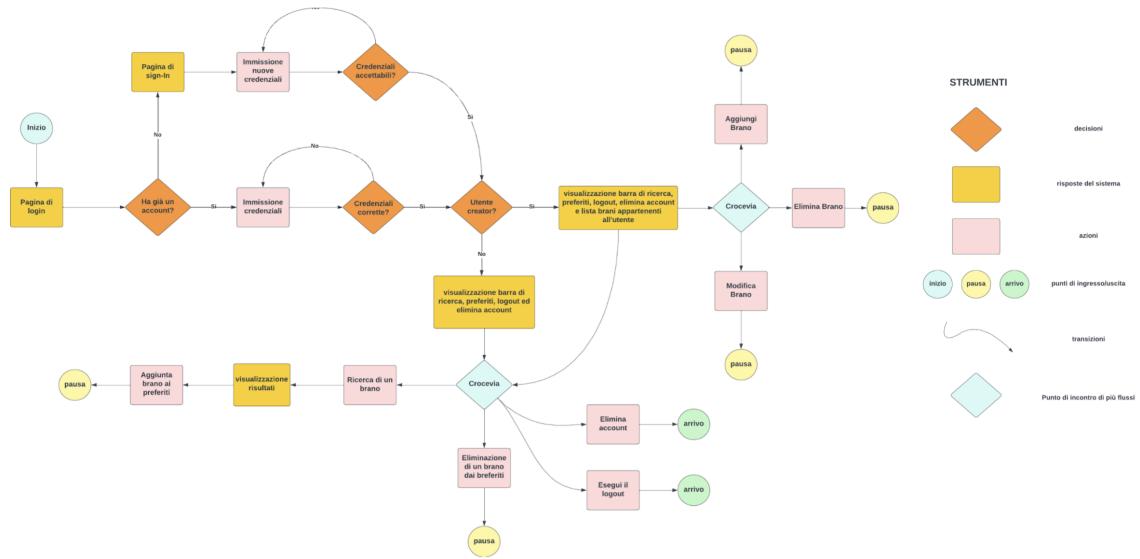


Figura 1: Struttura del backend del progetto

## 3 Implementazione e documentazione delle API

Questa sezione comprende lo sviluppo e la documentazione del backend del progetto, ovvero delle API che abbiamo descritto nelle sezioni precedenti del documento.

### 3.1 Struttura del backend

La struttura del backend è riportata nella figura 2. All'interno della cartella `/src` sono presenti i file che implementano le API e i modelli dei dati che faranno da ponte tra il backend e il database sul quale le informazioni verranno salvate. La cartella `/test` contiene i file per il testing; verrà approfondito nella prossima sezione.

Il file `index.ts` è il file principale del backend, in quanto si occupa di istanziare l'oggetto `app`, di effettuare la connessione al database e di avviare il server. Il file `app.ts` si consiste nell'applicazione stessa, alla quale verranno aggiunte le rotte per le API e per la documentazione. Il file `scripts.ts` contiene funzioni utilizzate in più punti del codice. I restanti file sono di configurazione.

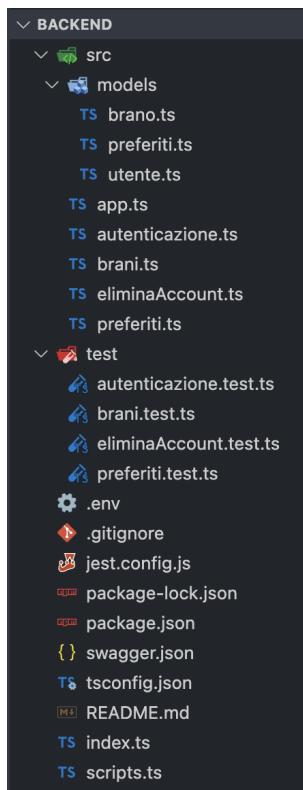


Figura 2: Struttura del backend del progetto

### 3.2 Dipendenze del progetto

Il progetto dipende da diverse librerie NodeJS per il suo funzionamento. Queste sono:

- `cors` per permettere ad un server esterno di accedere alle risorse presenti nel server di backend
- `dotenv` per le variabili d'ambiente
- `express` come framework per la creazione del server
- `jsonwebtoken` per la creazione e validazione degli utenti tramite token
- `mongoose` come ponte tra il backend e il database MongoDB
- `multer` come middleware
- `swagger-ui-express` per la documentazione
- `jest` e `supertest` per il testing
- `typescript` e i vari `@types` come aiuto allo sviluppo

### 3.3 Modellazione dati nel database

Abbiamo utilizzato come base di dati MongoDB, un database non relazionale orientato ai documenti. Abbiamo creato tre *Collections*, gruppi di documenti di tipo diverso, una per tipo di dato che necessita di essere salvato. Questi sono il tipo di dato **Utente**, **Brano** e **Preferiti**.

Ciascuno corrisponde ad un file presente nella cartella `/src/models` del progetto, all'interno dei quali sono stati definiti i corrispettivi *Schema*.

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
Brano	18	1.97kB	115B	36KB	2	72KB	36KB
Preferiti	47	3.31kB	73B	36KB	1	36KB	36KB
Utente	8	913B	115B	36KB	1	36KB	36KB

Figura 3: Collezioni presenti nel database

```

1   _id: ObjectId('63ae416baa85ec273e32fcf9')          ObjectId
2   email: "silvanus.bordignon@studenti.unitn.it"        String
3   password: "PasswordValida&%"                      String
4   tipoAccount: "standard"                            String
5   __v: 0                                              Int32

```

Figura 4: Modello tipo di dato Utente

```

1   _id: ObjectId('63af7d91f09d4c0b5980cd75')          ObjectId
2   nome: "Il tempo vola"                                String
3   artista: 63ae4cefc675dce04d60dfd0                  ObjectId
4   durata: 120                                         Int32
5   > tags: Array                                         Array
6   __v: 0                                              Int32

```

Figura 5: Modello tipo di dato Brano

```

1  _id: ObjectId('63af31beb12248ea498185cf')          ObjectId
2  utente: 63af31beb12248ea498185c9                  ObjectId
3  ▼ listaBrani: Array                                    Array
4    0: 63af3139d6da1d1f4118bbbb                         ObjectId
5    1: 63af3151d6da1d1f4118bbbd                         ObjectId
6    2: 63af3175d6da1d1f4118bbbbf                        ObjectId
7  __v: 0                                               Int32

```

Figura 6: Modello tipo di dato Preferiti

### 3.4 Specifica delle risorse

Questa sezione comprende due fasi distinte, entrambe antecedenti e necessarie allo sviluppo delle API.

#### 3.4.1 Estrazione delle risorse

In questo paragrafo vengono descritte le risorse estratte dal Class Diagram. Per ogni sottoparagrafo avrò una risorsa principale e tutte le API che questa risorsa può andare ad utilizzare.

##### «resource» Utente

La risorsa Utente ha come attributi:

- email
- password
- tipoAccount, che può essere Standard oppure Creator
- idUtente
- tokenUtente

##### «resource» Brano

Questa risorsa identifica un brano ed ha come attributi:

- idBrano
- nomeBrano
- nomeArtista
- durata
- tags

Le API contrassegnate da doppio asterisco (\*\*) interagiscono sia con la risorsa **Utente** che con **Brano**.

## API

«resource» signUp: Permette al nuovo utente di registrarsi. Viene svolta una POST e i parametri in ingresso sono email, password e tipoAccount. E' un API del backEnd.

«resource» accesso: Permette all'utente di accedere al servizio. Viene svolta una GET per il prelievo dell'utente dal database e i parametri in ingresso sono l'email e la password dell'utente. E' un API del BackEnd.

«resource» eliminaAccount: Permette ad un utente di eliminare il proprio account e di conseguenza tutti i suoi dati dal servizio. Viene svolta una DELETE e il parametro in ingresso è l'idUtente. E' un API del BackEnd.

«resource» ottieniPreferiti: Permette all'utente di visualizzare la playlist dei preferiti. Viene svolta una GET e il parametro in ingresso è l'idUtente. E' un API del FrontEnd.

«resource» modificaPreferiti: Permette all'utente di eliminare oppure aggiungere un brano dalla playlist dei preferiti. Viene svolta una PUT e i parametri in ingresso sono: idUtente, idBrano, azione (parametro di tipo String che identifica l'aggiunta oppure la rimozione). E' un API del BackEnd.

«resource» carica Brano\*\*: Permette ad un utente di tipo Creator di caricare un brano. Viene svolta una POST e i parametri in ingresso sono: nomeBrano, idUtente, durata e tags (lista dei tag). E' un API del BackEnd.

«resource» modificaBrano\*\*: Permette ad un utente di tipo Creator di modificare il nome oppure i tags di un suo brano. Viene svolta una PUT e i parametri in ingresso sono idBrano, nomeBrano, idUtente e tags. E' un API del BackEnd

«resource» eliminaBrano\*\*: Permette ad un utente di tipo Creator di eliminare un suo brano. Viene svolta una DELETE e i parametri in ingresso sono idBrano e idUtente. E' un API del BackEnd.

«resource» ottieniBrano: Questa API permette di ottenere una risorsa di tipo Brano a partire dal suo id. Viene svolta una GET e il parametro in ingresso è l'idBrano. E' un API del BackEnd.

«resource» ricerca: Permette all'utente di cercare un brano all'interno del database. Viene svolta una GET e il parametro in ingresso è il nomeBrano. E' un API del FrontEnd.

### 3.4.2 Modello delle risorse

Seguono i modelli delle risorse, una descrizione più accurata di ciascuna API, che comprende URI e metodo HTTP, descrizione della richiesta e delle varie risposte che quell'endpoint può restituire.

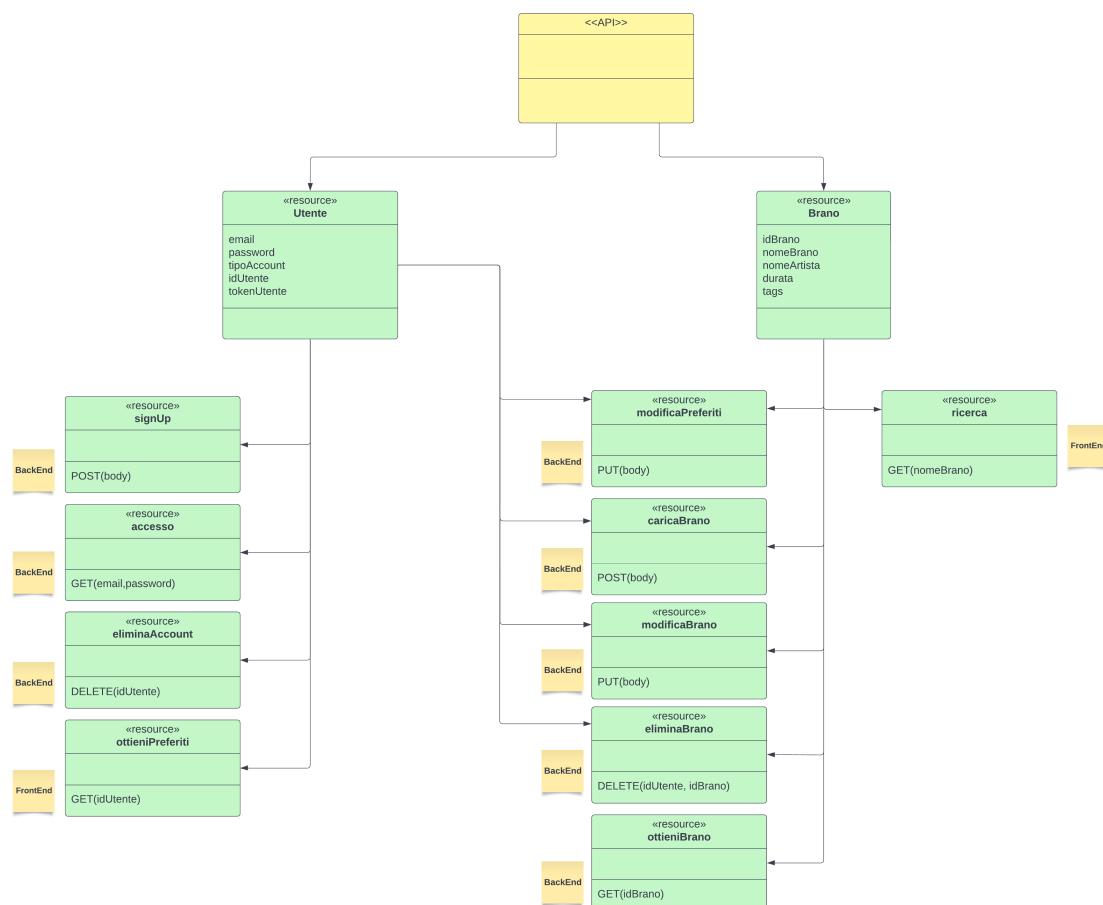


Figura 7: Diagramma di estrazione delle risorse

## Autenticazione

### Registrazione

Questa API, all'indirizzo `/api/auth/registrazione`, ha un metodo POST e viene utilizzata per memorizzare un nuovo utente nel database.

La request body è formata dalla risorsa Utente che ha come attributi i campi `email(String)`, `password(String)` e `tipoAccount(tipoAccount)`.

Può avere 3 tipologie di response body a seconda della riuscita dell'operazione.

Se la response body è formata da `code = 201` e `message = "Created"` allora l'operazione è riuscita e viene restituito l'`idUtente(String)`, il `tokenUtente(String)`, l'`email(String)` e il `tipoAccount(String)`.

Nel caso in cui `code = 409` e `message = "Conflict"` allora l'operazione non è riuscita in quanto l'email inserita è già presente nel database.

Se invece `code = 400` e `message = "Bad request"` allora l'operazione non è riuscita in quanto l'email o la password inseriti non sono corretti o non conformi con le specifiche del sito.

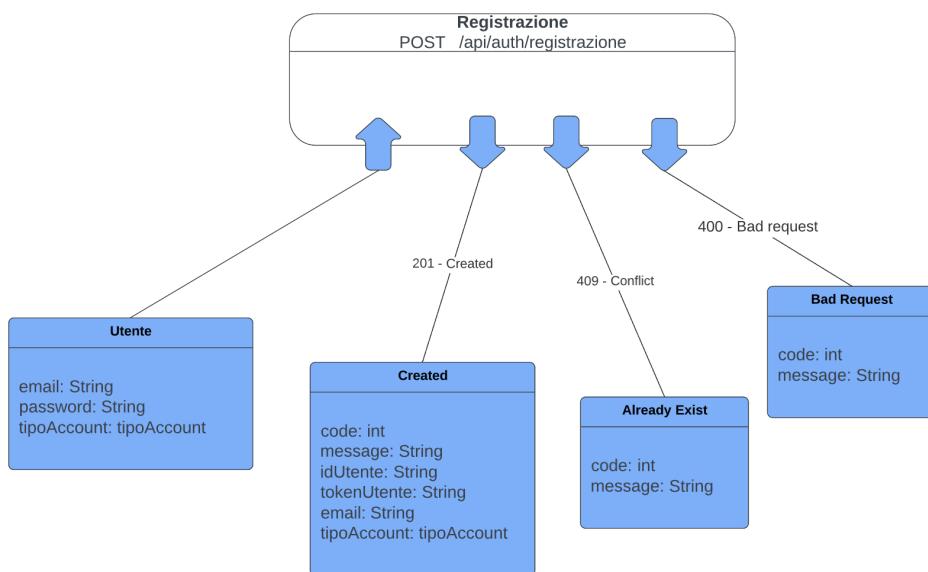


Figura 8: Modello della risorsa Registrazione

## Accesso

Questa API, all'indirizzo `/api/auth/accesso`, ha un metodo GET e viene utilizzata per fare il login di un utente. La request body è la risorsa Utente, la quale ha come attributi il campo `email (String)` e `password (String)`.

Può avere 3 tipologie di response body a seconda della riuscita dell'operazione.

Se la response body è formata da `code = 200` e `message = "OK"` allora l'operazione è riuscita e nel corpo della risorsa viene restituito l'`idUtente(String)`, il `tokenUtente(String)`, l'`email(String)` e il `tipoAccount(String)`.

Nel caso in cui `code = 404` e `message = "Not Found"` allora l'operazione non è riuscita in quanto l'email inserita non è presente nel database.

Se invece `code = 403` e `message = "Forbidden"` allora l'operazione non è riuscita in quanto la password inserita non è corretta.

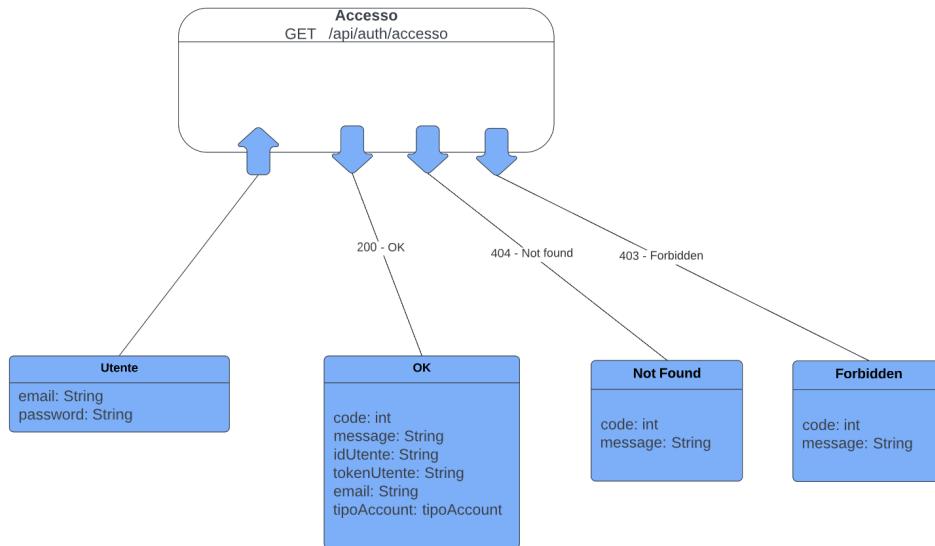


Figura 9: Modello della risorsa Accesso

## Ricerca

### Ricerca

Questa API, permette di ricercare all'interno del database un determinato brano e restituisce uno o più brani a seconda del testo digitato dall'utente.

All'indirizzo `/api/ricerca`, ha un metodo GET e prende in ingresso una stringa di testo `nomeBrano`. La response body è formata da `code(int)` e `message(String)` che saranno rispettivamente uguali a 200 e "OK" e dal campo `brani(Brano[*])` formato da una lista di oggetti di tipo Brano.

## Eliminazione account

### Elimina Account

Questa API all'uri `/api/eliminaAccount` permette all'utente la cancellazione del suo account e di conseguenza di tutti i suoi dati all'interno del sito web. Viene utilizzato il metodo **DELETE**.

La request body è formata da un solo campo: `idUtente` (String). Può avere 3 tipologie di response body a seconda della riuscita dell'operazione.

Se la response body è formata da `code = 204` e `message = "No content "` allora l'azione è stata eseguita e non devono essere fornite ulteriori informazioni.

Nel caso in cui `code = 404` e `message = "Not Found"` allora l'operazione non è riuscita in quanto l'`idUtente` inserito non è presente nel database.

Se invece `code = 400` e `message = "Bad Request"` allora l'operazione non è riuscita in quanto l'`idUtente` passato non è valido.

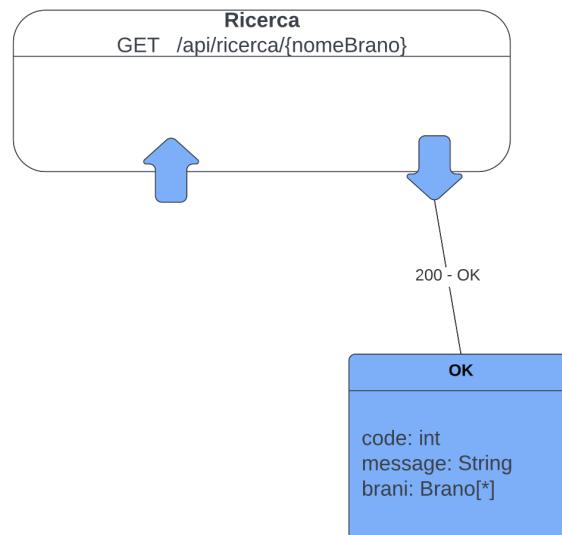


Figura 10: Modello della risorsa Ricerca

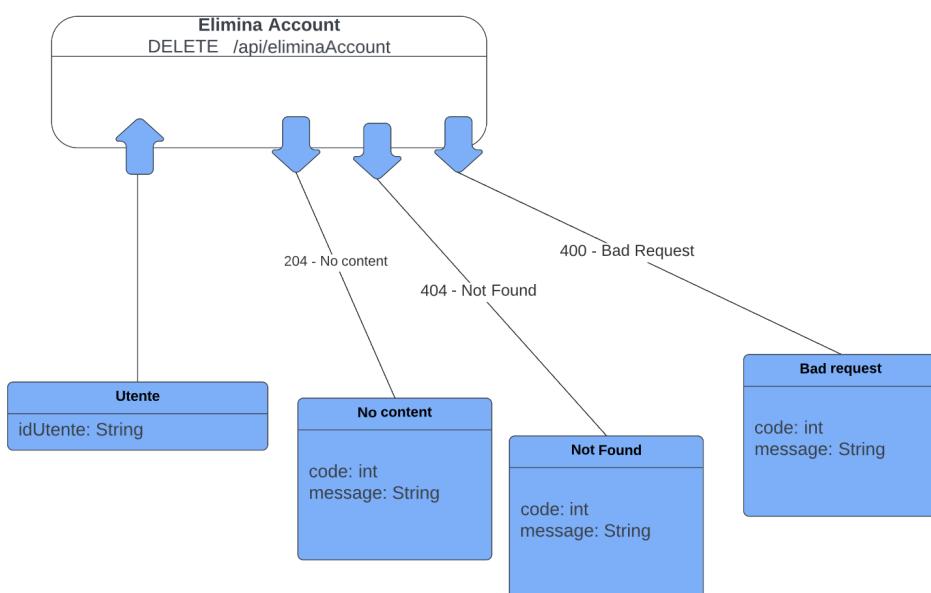


Figura 11: Modello della risorsa Eliminazione account

## Operazioni creator

### Carica Brano

Questa API consente ad un utente Creator di caricare un nuovo brano. Viene fatta una POST all'indirizzo: /api/brano.

La request body (Brano e Utente) è formata da *nomeBrano*(String), *idUtente*(String), *durata*(int) e *tags*(Tag[\*]).

Può avere 4 tipologie di response body a seconda della riuscita dell'operazione.

Se la response body è formata da *code* = 201 e *message* = "Created" allora l'azione è stata eseguita e il brano è stato caricato correttamente.

Nel caso in *code* = 404 e *message* = "Not Found" allora l'operazione non è riuscita in quanto l'*idUtente* non corrisponde ad un utente registrato.

Nel caso in cui *code* = 409 e *message* = "Conflict" allora l'operazione non è riuscita in quanto l'utente Creator ha già caricato un brano con quel nome.

Se invece *code* = 400 e *message* = "Bad Request" allora l'operazione non è riuscita in quanto almeno uno dei parametri passati non è valido oppure *idUtente* non corrisponde ad un utente Creator.

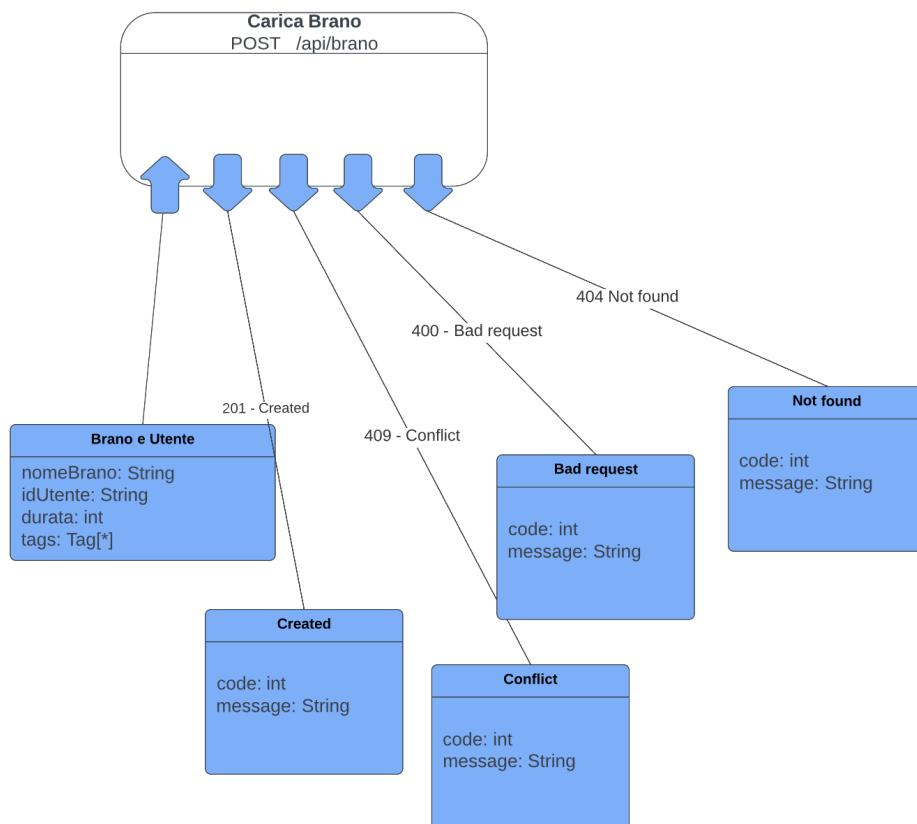


Figura 12: Modello della risorsa Carica brano

### Elimina Brano

Questa API consente ad un utente Creator di eliminare un brano. Viene fatta una DELETE all'uri: /api/brano. La request body (Brano e Utente) è formata da *idBrano*(String) e *idUtente*(String).

Può avere 3 tipologie di response body a seconda della riuscita dell'operazione.  
Se la response body è formata da *code* = 204 e *message* = "No Content" allora l'azione è stata eseguita e non devono essere fornite ulteriori informazioni.  
Nel caso in cui *code* = 404 e *message* = "Not Found" allora l'operazione non è riuscita in quanto idBrano o idUtente passati non sono presenti nel database.  
Se invece *code* = 400 e *message* = "Bad Request" allora l'operazione non è riuscita in quanto almeno uno dei parametri passati non è valido.

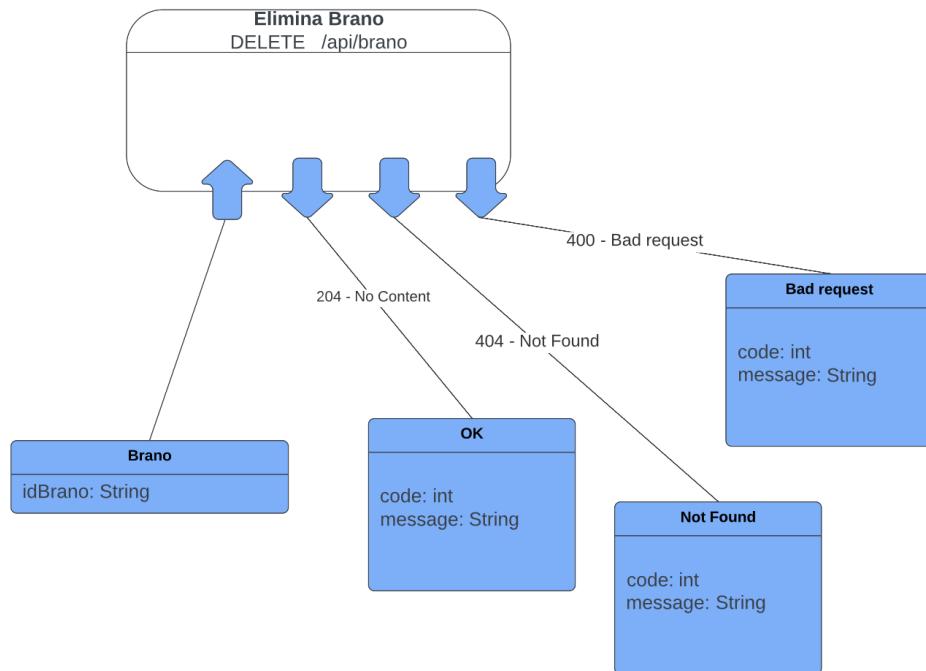


Figura 13: Modello della risorsa Elimina Brano

## Modifica Brano

Questa API consente ad un utente Creator di modificare un suo brano andando a cambiare il nome del brano oppure i tags. Viene fatta una PATCH all'indirizzo **/api/brano**. La request body (Brano e Utente) è formata da *idBrano(String)*, *idUtente(String)*, *nomeBrano(String)*, e *tags(Tag[\*])*.

Può avere 4 tipologie di response body a seconda della riuscita dell'operazione.  
Se la response body è formata da *code* = 200 e *message* = "OK" allora l'azione è stata eseguita e il brano è stato modificato correttamente.

Nel caso in cui *code* = 409 e *message* = "Conflict" allora l'operazione non è riuscita in quanto è possibile che il nuovo nome dato al brano sia già il nome di un altro brano del Creator.

Se *code* = 400 e *message* = "Bad Request" allora l'operazione non è riuscita in quanto almeno uno dei parametri passati non è valido.

Nel caso in cui *code* = 404 e *message* = "Not Found" allora l'operazione non è riuscita in quanto idBrano o idUtente (oppure entrambi) non sono presenti nel database.

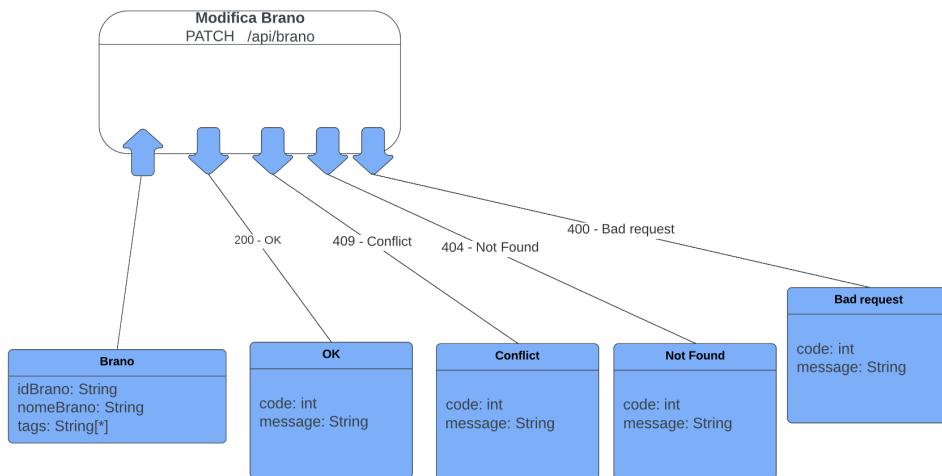


Figura 14: Modello della risorsa Modifica brano

## Preferiti e ottenimento brano

### Ottieni Brano

Questa API consente di ottenere un oggetto di tipo Brano a partire da un idBrano. Viene utilizzato il metodo GET all'indirizzo /api/brano e la request body è formata dall'*idBrano*(String).

Può avere 3 tipologie di response body a seconda della riuscita dell'operazione.

Se la response body è formata da *code* = 200 e *message* = "OK" allora l'azione è stata eseguita correttamente e nel corpo della response troviamo anche l'attributo *brano*(Brano). Nel caso in cui *code* = 400 e *message* = "Bad Request" allora l'operazione non è riuscita in quanto l'idBrano passato alla funzione non è valido.

Nel caso in cui *code* = 404 e *message* = "Not Found" allora l'operazione non è riuscita in quanto l'idBrano passato all'API non è presente nel database.

### Ottieni Preferiti

Questa API permette di ottenere e visualizzare la lista dei preferiti. Viene utilizzato il metodo GET all'indirizzo /api/preferiti.

La request body è formata da un solo parametro: *idUtente*.

Può avere 3 tipologie di response body a seconda della riuscita dell'operazione.

Se la response body è formata da *code* = 200 e *message* = "OK" allora l'azione è stata eseguita correttamente e nel corpo della response troviamo anche l'oggetto *idBrani*(String[]), ovvero una lista degli id dei brani appartenenti alla playlist Preferiti di quel particolare utente.

Nel caso in cui *code* = 400 e *message* = "Bad Request" allora l'operazione non è riuscita in quanto l'idUtente passato alla funzione non è valido.

Nel caso in cui *code* = 404 e *message* = "Not Found" allora l'operazione non è riuscita in quanto l'idUtente passato all'API non è presente nel database.

### Modifica Preferiti

Questa API permette all'utente di modificare la propria playlist di preferiti. Grazie ad

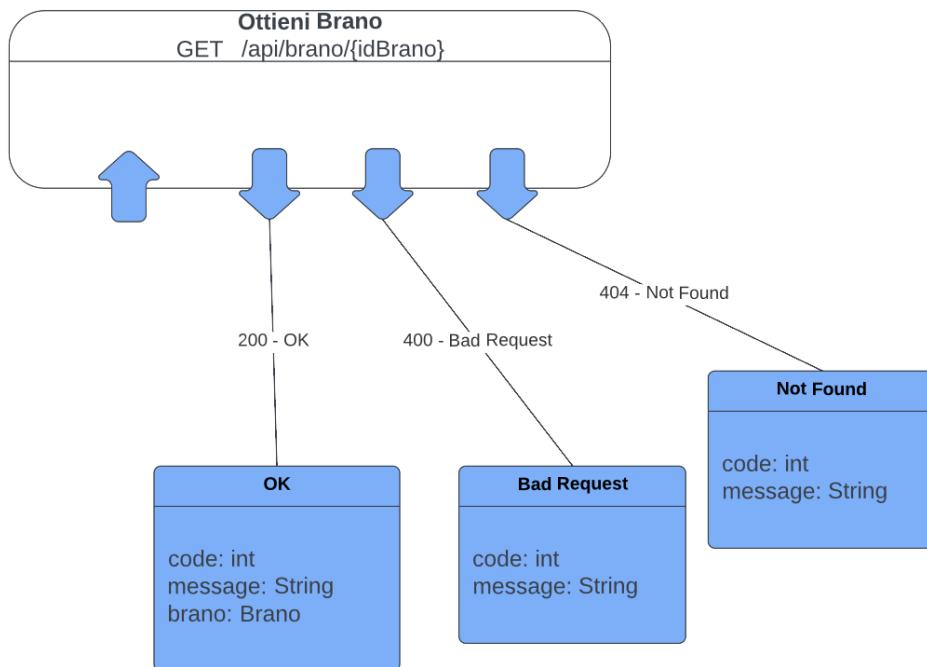


Figura 15: Modello della risorsa Ottieni Brano

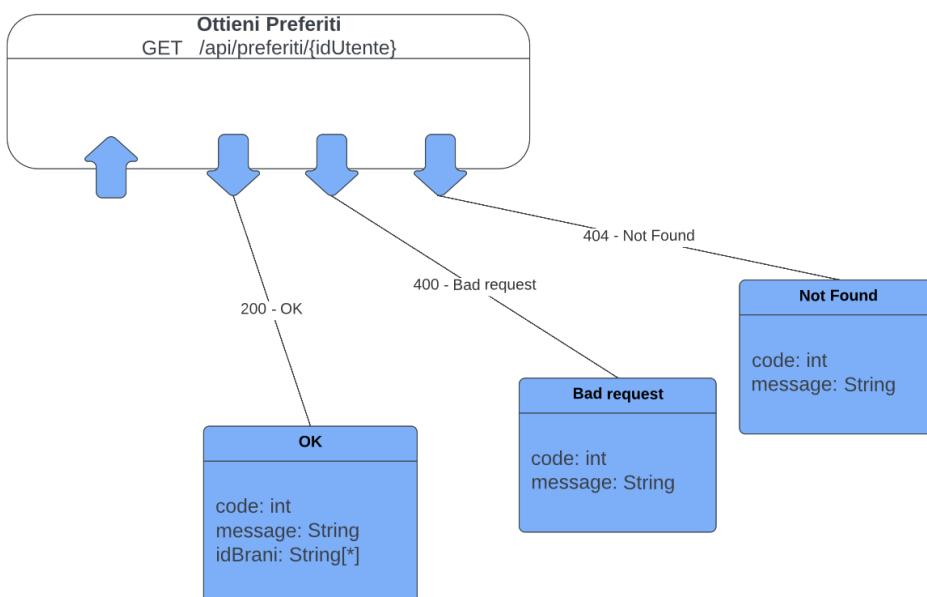


Figura 16: Modello della risorsa Ottieni preferiti

essere l'utente può aggiungere o eliminare un brano dai preferiti. Viene utilizzato il metodo PATCH all'indirizzo /api/preferiti/modifica.

La request body è formata da 3 parametri: *idUtente*(String), *idBrano*(String), e *azione*(String); quest'ultimo attributo può assumere solo i valori “aggiungi” ed “elimina” in relazione ad un determinato brano.

Può avere 4 tipologie di response body a seconda della riuscita dell'operazione.

Se la response body è formata da *code* = 200 e *message* = “OK” allora l'azione è stata eseguita correttamente e nel response body ottengo l'oggetto *idBrani*(String[\*]).

Nel caso in cui *code* = 409 e *message* = “Conflict” allora l'operazione non è riuscita.

Se *code* = 400 e *message* = “Bad Request” allora l'operazione non è riuscita in quanto almeno uno dei parametri passati non è valido.

Nel caso in cui *code* = 404 e *message* = “Not Found” allora l'operazione non è riuscita in quanto *idBrano* o *idUtente* (oppure entrambi) non sono presenti nel database.

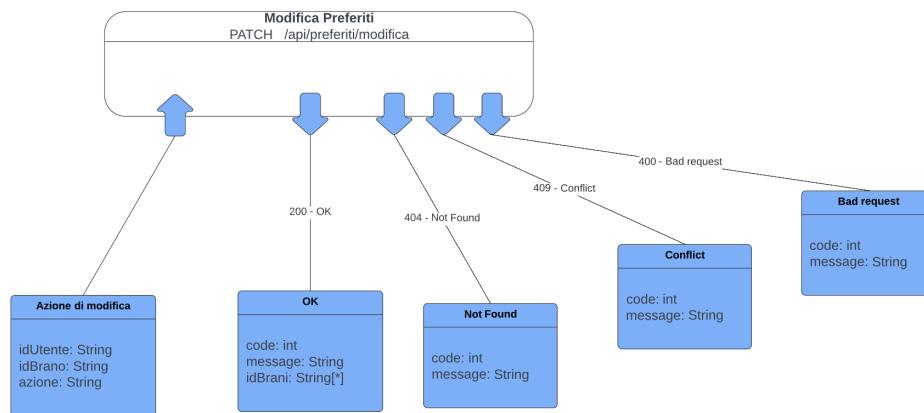


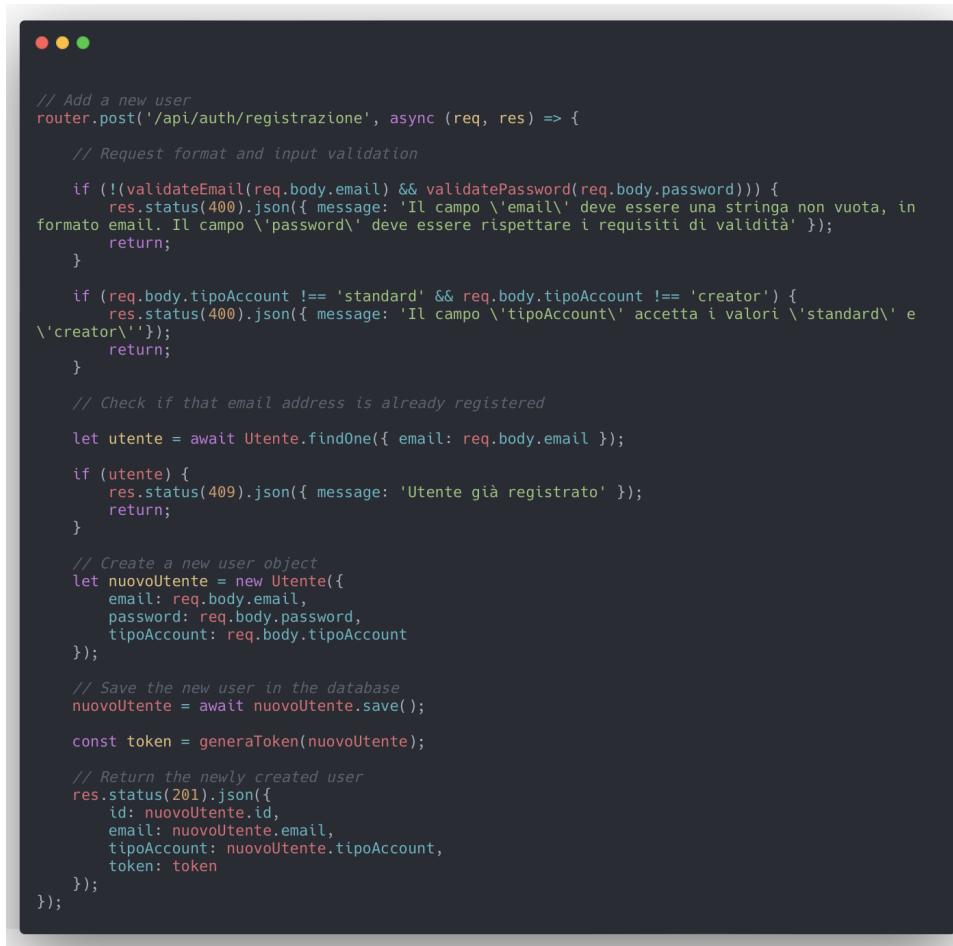
Figura 17: Modello della risorsa Modifica preferiti

## 4 Sviluppo delle API

Abbiamo sviluppato 10 API a partire dai **modelli delle risorse** visti precedentemente. Queste consistono nelle API per la registrazione e l'accesso, per la ricerca e l'eliminazione account, per caricare, modificare ed eliminare un brano, per ottenere un brano a partire dall'ID e per ottenere o modificare una lista preferiti.

### 4.1 Registrazione

Questa API viene utilizzata per inserire un nuovo Utente al database. Una volta ricevuta la richiesta, avviene una validazione dell'input per assicurare l'inserimento di dati corretti. Se l'utente non è già registrato, viene creato un nuovo oggetto basato sullo schema Utente al quale vengono assegnati i dati inseriti. Questo viene inserito nel database, e alcuni suoi campi assieme ad un token generato al momento vengono restituiti all'utente.



```
// Add a new user
router.post('/api/auth/registrazione', async (req, res) => {
    // Request format and input validation
    if (!(validateEmail(req.body.email) && validatePassword(req.body.password))) {
        res.status(400).json({ message: 'Il campo \'email\' deve essere una stringa non vuota, in formato email. Il campo \'password\' deve essere rispettare i requisiti di validità' });
        return;
    }
    if (req.body.tipoAccount !== 'standard' && req.body.tipoAccount !== 'creator') {
        res.status(400).json({ message: 'Il campo \'tipoAccount\' accetta i valori \'standard\' e \'creator\''});
        return;
    }

    // Check if that email address is already registered
    let utente = await Utente.findOne({ email: req.body.email });

    if (utente) {
        res.status(409).json({ message: 'Utente già registrato' });
        return;
    }

    // Create a new user object
    let nuovoUtente = new Utente({
        email: req.body.email,
        password: req.body.password,
        tipoAccount: req.body.tipoAccount
    });

    // Save the new user in the database
    nuovoUtente = await nuovoUtente.save();

    const token = generaToken(nuovoUtente);

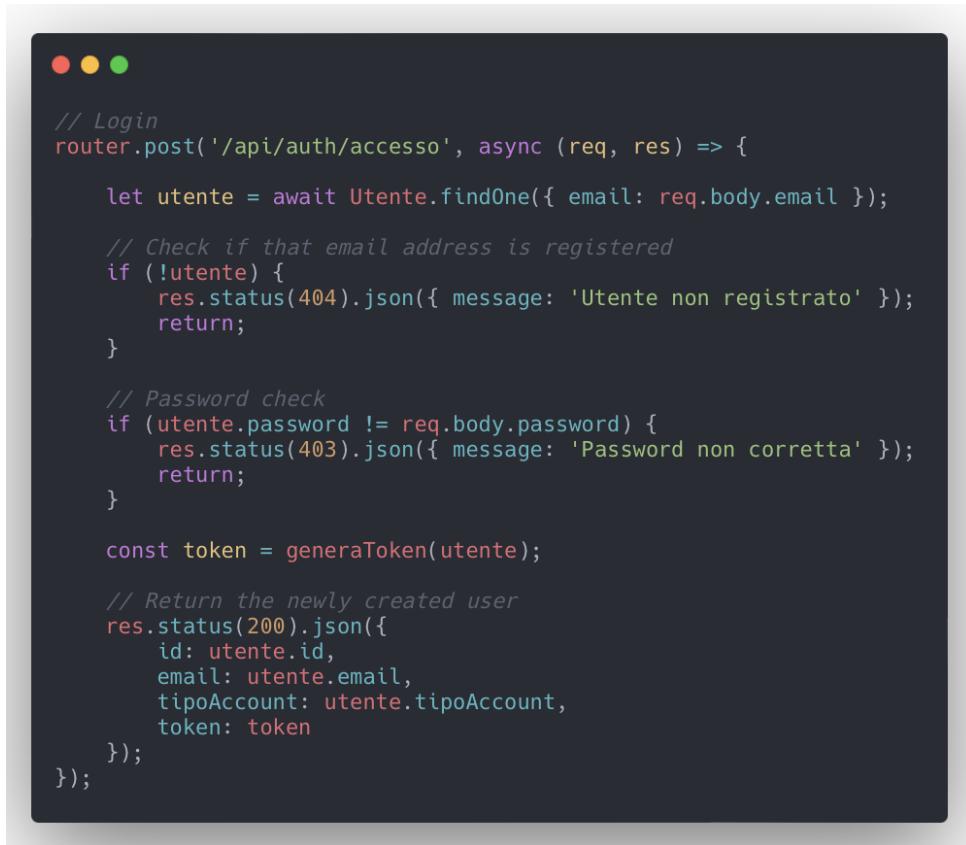
    // Return the newly created user
    res.status(201).json({
        id: nuovoUtente.id,
        email: nuovoUtente.email,
        tipoAccount: nuovoUtente.tipoAccount,
        token: token
    });
});
```

Figura 18: API per la registrazione

## 4.2 Accesso

Questa API permette l'accesso di un utente già registrato alla piattaforma. Viene verificato si tratti di un utente esistente, e in caso affermativo si procede al confronto tra le password. Se il confronto è positivo viene generato il token e viene restituita la risposta corrispondente.

**Nota:** da questo momento in poi le richieste devono essere accompagnate da un **token** di accesso. Questo deve essere presente nell'header della richiesta e viene verificato prima che la richiesta raggiunga la rispettiva API.



```
// Login
router.post('/api/auth/accesso', async (req, res) => {
    let utente = await Utente.findOne({ email: req.body.email });

    // Check if that email address is registered
    if (!utente) {
        res.status(404).json({ message: 'Utente non registrato' });
        return;
    }

    // Password check
    if (utente.password != req.body.password) {
        res.status(403).json({ message: 'Password non corretta' });
        return;
    }

    const token = generaToken(utente);

    // Return the newly created user
    res.status(200).json({
        id: utente.id,
        email: utente.email,
        tipoAccount: utente.tipoAccount,
        token: token
    });
});
```

Figura 19: API per l'accesso

## 4.3 Ricerca

La ricerca di un brano tramite il titolo fa uso di una funzione di ricerca del database: `$search`. Per abilitarla è stato creato un indice per il campo `titolo` dello schema Brano. Vengono restituiti i primi 10 risultati della ricerca.

## 4.4 Elimina account

Questa API si occupa della cancellazione di un account dalla piattaforma. Una volta ricevuta la richiesta, si verifica che l'utente in questione sia presente nel database. In caso



```
// Search for a song in the database
router.get('/api/ricerca/:testo', async (req, res) => {
  let risultati = await Brano.find({$text: {$search: req.params.testo}}).limit(10);
  res.status(200).json(risultati);
});
```

Figura 20: API per la ricerca

affermativo, vengono rimosso dal sistema l'utente, la sua lista preferiti ed eventuali brani da lui caricati.



```
// Remove a user
router.delete('/api/eliminaAccount', async (req, res) => {
  // Input validation
  if (!Types.ObjectId.isValid(req.body.idUtente)) {
    res.status(400).json({ message: 'Il valore inserito per l\'utente non è un ID valido' });
    return;
  }

  // Check if the user exists
  let utente = await Utente.findById(req.body.idUtente).exec();
  if (!utente) {
    res.status(404).json({ message: 'Utente non trovato' });
    return;
  }

  // Delete the user
  await utente.deleteOne();

  // Rimuovi la lista preferiti dell'utente
  let preferiti = await Preferiti.find({ utente: req.body.idUtente });

  if (preferiti.length !== 0) {
    await Preferiti.deleteOne({ id: preferiti[0].id });
  }

  // Rimuovi tutti i brani caricati da quell'utente (anche da liste preferiti)
  let brani = await Brano.find({ artista: utente.id });

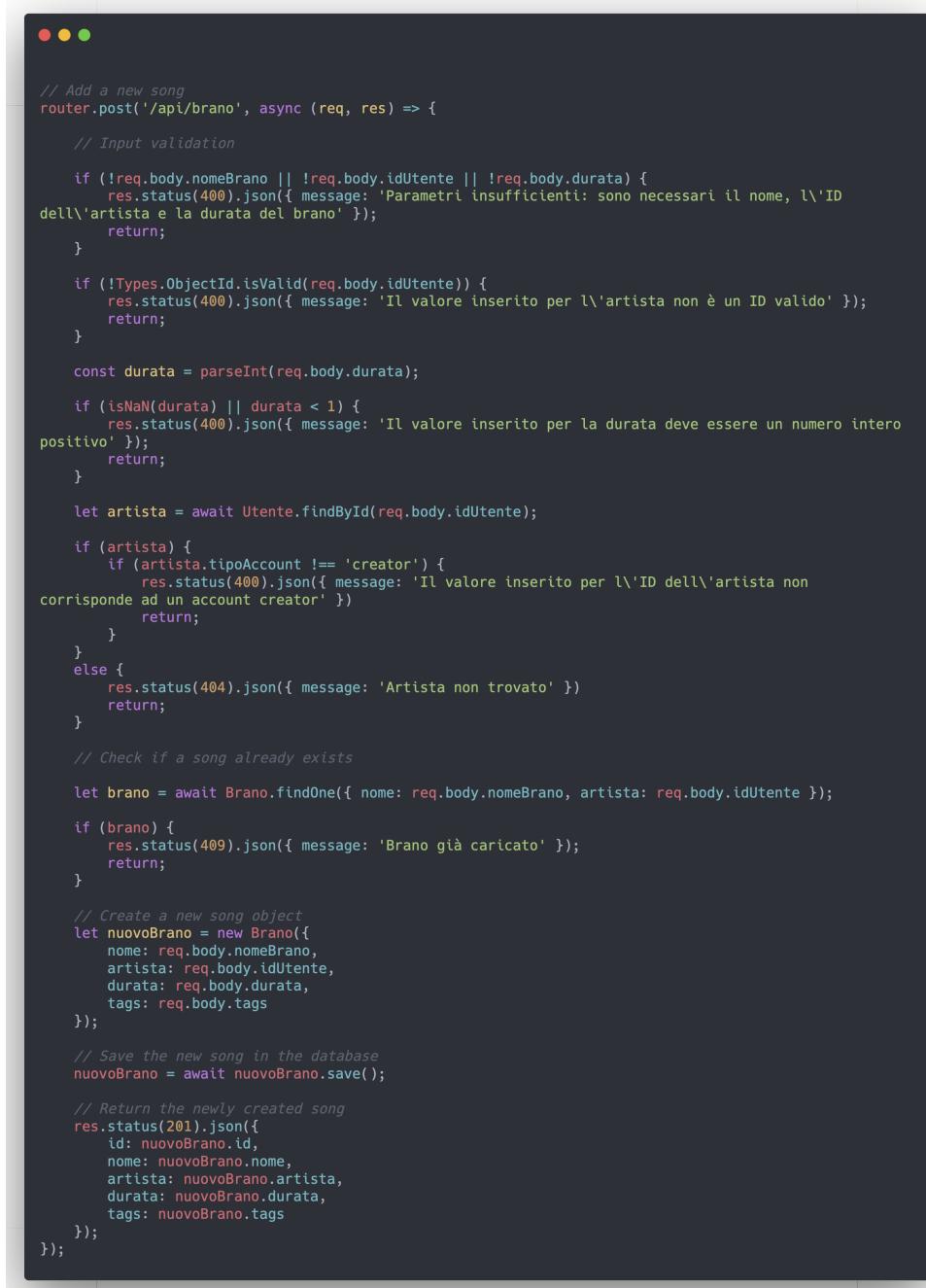
  brani.forEach(async (brano: any) => {
    await Preferiti.updateMany(
      {},
      { $pull: { listaBrani: brano.id } }
    );
  });

  res.sendStatus(204);
});
```

Figura 21: API per l'eliminazione dell'account

## 4.5 Carica brano

Il caricamento di un nuovo brano viene gestito da questa API. I dati in input vengono validati (campi non vuoti, artista effettivamente registrato come creator, etc.), e viene verificato che non esista un altro brano dello stesso artista con quel titolo. Se tutti i controlli vengono passati, si crea un nuovo oggetto a partire dallo schema Brano e viene inserito nel database.



```
// Add a new song
router.post('/api/brano', async (req, res) => {
    // Input validation

    if (!req.body.nomeBrano || !req.body.idUtente || !req.body.durata) {
        res.status(400).json({ message: 'Parametri insufficienti: sono necessari il nome, l\'ID dell\'artista e la durata del brano' });
        return;
    }

    if (!Types.ObjectId.isValid(req.body.idUtente)) {
        res.status(400).json({ message: 'Il valore inserito per l\'artista non è un ID valido' });
        return;
    }

    const durata = parseInt(req.body.durata);

    if (isNaN(durata) || durata < 1) {
        res.status(400).json({ message: 'Il valore inserito per la durata deve essere un numero intero positivo' });
        return;
    }

    let artista = await Utente.findById(req.body.idUtente);

    if (artista) {
        if (artista.tipoAccount !== 'creator') {
            res.status(400).json({ message: 'Il valore inserito per l\'ID dell\'artista non corrisponde ad un account creator' });
            return;
        }
    } else {
        res.status(404).json({ message: 'Artista non trovato' });
        return;
    }

    // Check if a song already exists

    let brano = await Brano.findOne({ nome: req.body.nomeBrano, artista: req.body.idUtente });

    if (brano) {
        res.status(409).json({ message: 'Brano già caricato' });
        return;
    }

    // Create a new song object
    let nuovoBrano = new Brano({
        nome: req.body.nomeBrano,
        artista: req.body.idUtente,
        durata: req.body.durata,
        tags: req.body.tags
    });

    // Save the new song in the database
    nuovoBrano = await nuovoBrano.save();

    // Return the newly created song
    res.status(201).json({
        id: nuovoBrano.id,
        nome: nuovoBrano.nome,
        artista: nuovoBrano.artista,
        durata: nuovoBrano.durata,
        tags: nuovoBrano.tags
    });
});
```

Figura 22: API per caricare un brano

## 4.6 Elimina Brano

Questa API si occupa della rimozione di un brano dalla piattaforma. L'ID ricevuto viene validato, si verifica che il brano sia presente sulla piattaforma, e lo si rimuove dal database. Viene inoltre rimosso dalle varie liste preferiti che lo includevano.



```
// Remove a song
router.delete('/api/brano', async (req, res) => {
    // Input validation
    if (!Types.ObjectId.isValid(req.body.idBrano)) {
        res.status(400).json({ message: 'Il valore inserito per il brano non è un ID valido' });
        return;
    }

    // Check if the song exists
    let brano = await Brano.findById(req.body.idBrano).exec();
    if (!brano) {
        res.status(404).json({ message: 'Brano non trovato' });
        return;
    }

    // Delete the song
    await brano.deleteOne();

    // Delete the song from any favorite list that could have it
    await Preferiti.updateMany(
        {},
        { $pull: { listaBrani: brano.id } }
    );

    res.sendStatus(204);
});
```

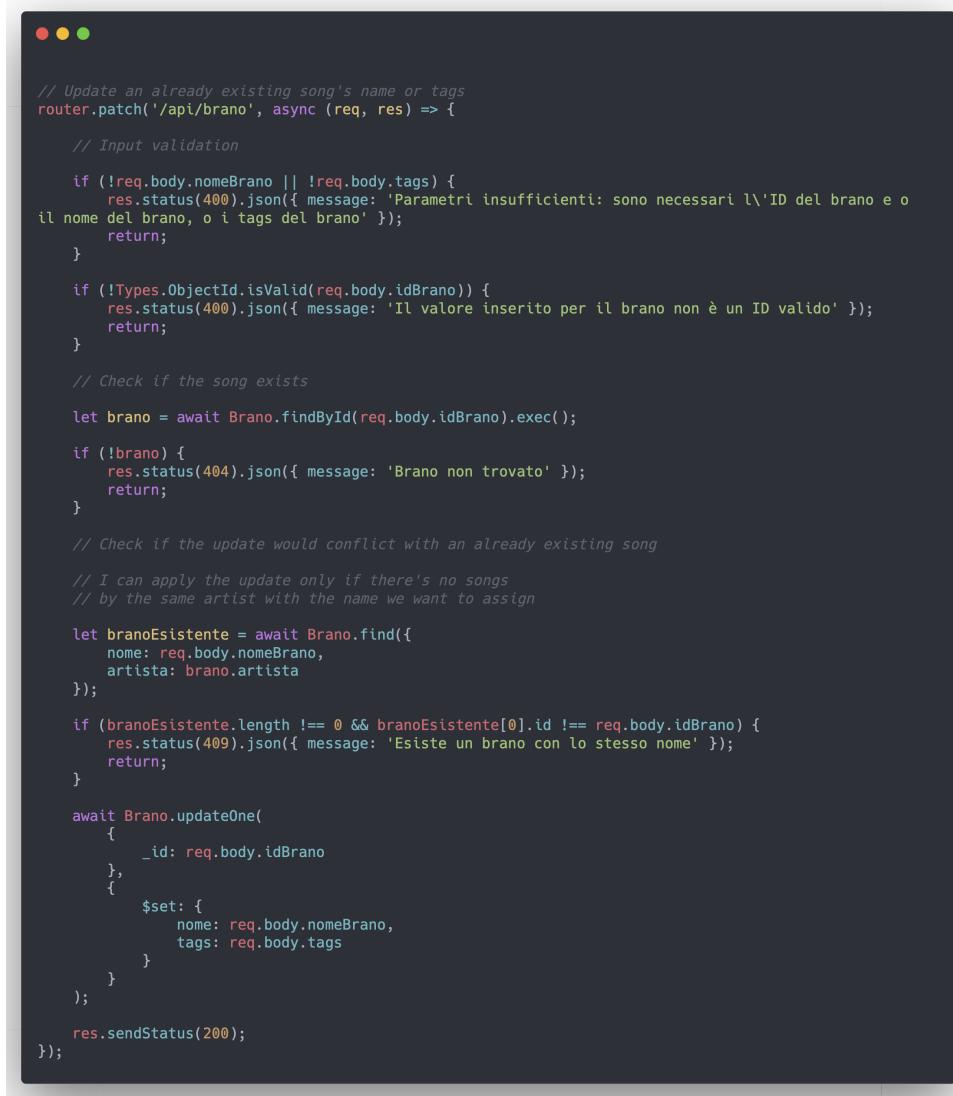
Figura 23: API per eliminare un brano

## 4.7 Modifica brano

La modifica di un brano avviene tramite un'API di tipo PATCH. Vengono infatti forniti solamente i campi che si possono modificare, quindi il titolo e i tags del brano. Dopo una validazione dell'input si verifica che non esista un altro brano dello stesso artista con il titolo che si intende assegnare al brano corrente. In caso negativo, si procede al salvataggio della modifica nel database.

## 4.8 Ottieni brano

Questa API si occupa di ottenere un brano a partire dall'ID. Quando un'altra API restituisce una lista di ID brani, questa API viene chiamata per ottenere a partire da quegli ID tutti i dati dei rispettivi brani. Validazione dell'input, verifica dell'esistenza del brano, e in caso affermativo il brano viene restituito.



```
// Update an already existing song's name or tags
router.patch('/api/brano', async (req, res) => {

    // Input validation
    if (!req.body.nomeBrano || !req.body.tags) {
        res.status(400).json({ message: 'Parametri insufficienti: sono necessari l\'ID del brano e o il nome del brano, o i tags del brano' });
        return;
    }

    if (!Types.ObjectId.isValid(req.body.idBrano)) {
        res.status(400).json({ message: 'Il valore inserito per il brano non è un ID valido' });
        return;
    }

    // Check if the song exists
    let brano = await Brano.findById(req.body.idBrano).exec();

    if (!brano) {
        res.status(404).json({ message: 'Brano non trovato' });
        return;
    }

    // Check if the update would conflict with an already existing song
    // I can apply the update only if there's no songs
    // by the same artist with the name we want to assign

    let branoEsistente = await Brano.find({
        nome: req.body.nomeBrano,
        artista: brano.artista
    });

    if (branoEsistente.length !== 0 && branoEsistente[0].id !== req.body.idBrano) {
        res.status(409).json({ message: 'Esiste un brano con lo stesso nome' });
        return;
    }

    await Brano.updateOne(
        {
            _id: req.body.idBrano
        },
        {
            $set: {
                nome: req.body.nomeBrano,
                tags: req.body.tags
            }
        }
    );
    res.sendStatus(200);
});
```

Figura 24: API per modificare un brano



```
// Get a song by its id
router.get('/api/brano/:idBrano', async (req, res) => {

    // Input validation
    if (!Types.ObjectId.isValid(req.params.idBrano)) {
        res.status(400).json({ message: 'Il valore inserito per il brano non è un ID valido' });
        return;
    }

    let brano = await Brano.findById(req.params.idBrano);

    // Check if the song exists
    if (!brano) {
        res.status(404).json({ message: 'Brano non trovato' });
        return;
    }

    // Return the song
    res.status(200).json({
        id: brano.id,
        nome: brano.nome,
        artista: brano.artista,
        durata: brano.durata,
        tags: brano.tags
    });
});
```

Figura 25: API per ottenere un brano

## 4.9 Ottieni preferiti

Questa API restituisce la lista preferiti di un utente a partire dal suo ID. Validazione dell'input, si verifica l'esistenza dell'utente e si restituisce la lista. Nel caso in cui non fosse ancora stata creata per quello specifico utente, si crea una lista vuota e si restituisce quella.

## 4.10 Modifica preferiti

Questa API permette l'aggiunta o rimozione di un brano dalla lista dei preferiti. È l'API con il maggior numero di controlli dei dati ricevuti (ID, esistenza utente, esistenza brano, etc.). Nel caso in cui la lista preferiti non sia presente per l'utente considerato, viene creata una nuova lista e si opera su quella. Per un'operazione di "aggiunta" si verifica che il brano non sia già presente; in caso negativo, si inserisce nella lista l'ID del brano fornito. Per un'operazione di "rimozione" si verifica che il brano sia effettivamente nella lista preferiti; in caso positivo, il suo ID viene rimosso dalla lista. In entrambi i casi, la lista aggiornata viene restituita nel corpo della risposta.



```
// Get a user's favorite songs
router.get('/api/prefeRiti/:idUtente', async (req, res) => {
  const idUtente = req.params.idUtente;

  // Input validation
  if (!Types.ObjectId.isValid(idUtente)) {
    res.status(400).json({ message: 'Il valore inserito per l\'utente non è un ID valido' });
    return;
  }

  // Check if the user actually exists
  let utente = await Utente.findById(idUtente).exec();

  if (!utente) {
    res.status(404).json({ message: 'Utente non registrato' });
    return;
  }

  // Check if the document has already been created
  let preferiti = (await Preferiti.find({ utente: idUtente }))[0];

  // If not, create one and reassign the 'preferiti' variable
  if (!preferiti) {
    preferiti = await creaPreferiti(idUtente);
  }

  res.status(200).json({ idBrani: preferiti.listaBrani });
});
```

Figura 26: API per ottenere i preferiti

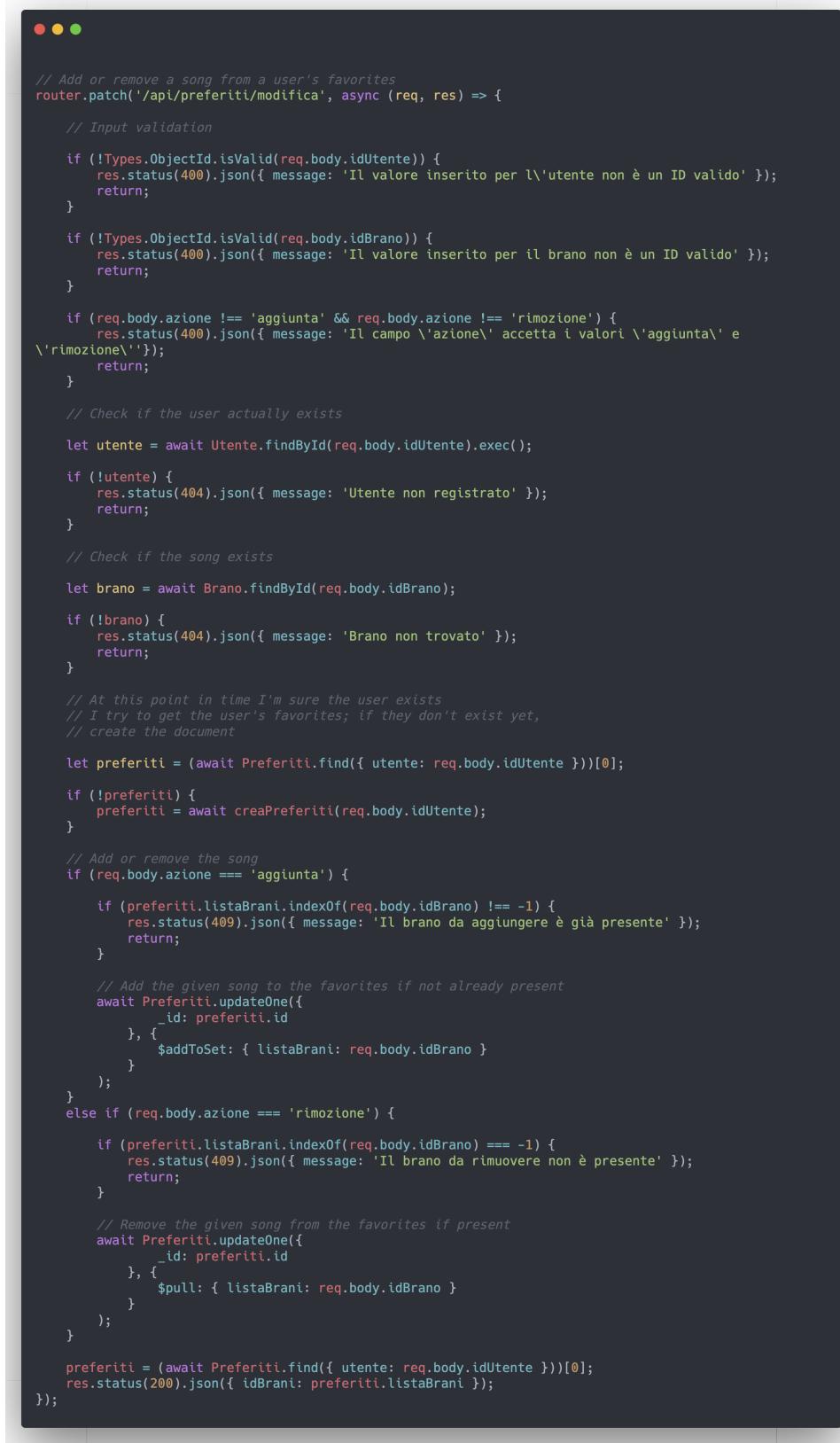
## 5 Documentazione delle API

Le API elencate nella sezione precedente sono state documentate utilizzando il modulo `swagger-ui-express`, seguendo lo standard OpenAPI. Sono state descritti i vari endpoint e i modelli dati che abbiamo considerato fin'ora: Utente, Brano, Preferiti.

Per permettere un facile accesso alla documentazione è possibile visitare una pagina dove è presente l'interfaccia **Swagger UI**, un modo semplice e chiaro per visualizzare ciò che riguarda l'API: dall'URI, dal metodo HTTP, alla struttura della richiesta e alle possibili risposte. La modalità interattiva di Swagger UI permette il testing dell'API direttamente sul sito.

La documentazione è consultabile tramite l'endpoint `/api-dpCs`.

Di seguito riportiamo l'esempio di quattro API che sfruttano metodi HTTP differenti: una GET, una POST, una PATCH e una DELETE.



```

// Add or remove a song from a user's favorites
router.patch('/api/prefertiti/modifica', async (req, res) => {
    // Input validation
    if (!Types.ObjectId.isValid(req.body.idUtente)) {
        res.status(400).json({ message: 'Il valore inserito per l\'utente non è un ID valido' });
        return;
    }
    if (!Types.ObjectId.isValid(req.body.idBrano)) {
        res.status(400).json({ message: 'Il valore inserito per il brano non è un ID valido' });
        return;
    }
    if (req.body.azione !== 'aggiunta' && req.body.azione !== 'rimozione') {
        res.status(400).json({ message: 'Il campo \'azione\' accetta i valori \'aggiunta\' e \'rimozione\'' });
        return;
    }

    // Check if the user actually exists
    let utente = await Utente.findById(req.body.idUtente).exec();
    if (!utente) {
        res.status(404).json({ message: 'Utente non registrato' });
        return;
    }

    // Check if the song exists
    let brano = await Brano.findById(req.body.idBrano);
    if (!brano) {
        res.status(404).json({ message: 'Brano non trovato' });
        return;
    }

    // At this point in time I'm sure the user exists
    // I try to get the user's favorites; if they don't exist yet,
    // create the document
    let preferiti = (await Preferiti.find({ utente: req.body.idUtente }))[0];
    if (!preferiti) {
        preferiti = await creaPreferiti(req.body.idUtente);
    }

    // Add or remove the song
    if (req.body.azione === 'aggiunta') {
        if (preferiti.listaBrani.indexOf(req.body.idBrano) === -1) {
            res.status(409).json({ message: 'Il brano da aggiungere è già presente' });
            return;
        }

        // Add the given song to the favorites if not already present
        await Preferiti.updateOne({
            _id: preferiti.id
        }, {
            $addToSet: { listaBrani: req.body.idBrano }
        });
    } else if (req.body.azione === 'rimozione') {
        if (preferiti.listaBrani.indexOf(req.body.idBrano) === -1) {
            res.status(409).json({ message: 'Il brano da rimuovere non è presente' });
            return;
        }

        // Remove the given song from the favorites if present
        await Preferiti.updateOne({
            _id: preferiti.id
        }, {
            $pull: { listaBrani: req.body.idBrano }
        });
    }

    preferiti = (await Preferiti.find({ utente: req.body.idUtente }))[0];
    res.status(200).json({ idBrani: preferiti.listaBrani });
});

```

Figura 27: API per modificare i preferiti

The screenshot shows the Swagger UI interface for the Web Music Player API, version 1.0.0, running at localhost:8080. The interface is organized into sections for different endpoints:

- Schemes:** HTTP
- Utenti:** Gestione degli account utente
  - POST /api/auth/registrazione** Aggiungi un utente al sistema
  - POST /api/auth/accesso** Effettua l'accesso al sistema
  - DELETE /api/eliminaAccount** Rimuovi un account dal sistema
- Brani:** Operazioni relative ai brani
  - GET /api/brano/{idBrano}** Ottieni un brano dal suo ID
  - POST /api/brano** Aggiungi un brano al sistema
  - PATCH /api/brano** Modifica un brano presente nel sistema
  - DELETE /api/brano** Rimuovi un brano dal sistema
  - GET /api/ricerca/{testo}** Ricerca un brano all'interno del sistema
- Preferiti:** Visualizzazione e modifica dei preferiti
  - GET /api/preferiti/{idUtente}** Ottieni l'elenco dei preferiti di un utente dal suo ID
  - PATCH /api/preferiti/modifica** Aggiungi o rimuovi un brano dai preferiti di un utente
- Models:**
  - Utente >
  - Brano >
  - Preferiti >

Figura 28: SwaggerUI

**GET /api/brano/{idBrano}** Ottieni un brano dal suo ID

**Parameters**

Name	Description
<b>idBrano</b> * required (path)	63a18edd25800766fed3d137

**Responses**

Code	Description
200	Brano trovato
400	Richiesta non valida
404	Brano non trovato

Example Value | Model

```
{
  "id": "63a18edd25800766fed3d137",
  "nome": "Titolo",
  "artista": "639b88a1e5e03f310f6f8294",
  "durata": 1,
  "tags": [
    "Pop",
    "Rock"
  ]
}
```

Response content type application/json

Figura 29: Documentazione API di tipo GET

**POST /api/brano** Aggiungi un brano al sistema

**Parameters**

Name	Description
<b>Nuovo brano</b> * required object (body)	Example Value   Model

Parameter content type application/json

Example Value | Model

```
{
  "nuovoBrano": "Titolo",
  "idUtente": "639b88a1e5e03f310f6f8294",
  "durata": 1,
  "tags": [
    "Pop",
    "Rock"
  ]
}
```

**Responses**

Code	Description
201	Brano aggiunto al sistema
400	Richiesta non valida
404	Artista non trovato
409	Un brano con lo stesso nome è già presente

Response content type application/json

Figura 30: Documentazione API di tipo POST

**PATCH /api/preferriti/modifica** Aggiungi o rimuovi un brano dai preferiti di un utente

**Parameters**

Name	Description
<b>Preferiti</b> * required object (body)	Example Value   Model <pre>{   "idUtente": "639b88a1e5e03f310f6f8294",   "idBrano": "63a18edd25800766fed3d137",   "azione": "aggiunte" }</pre>
Parameter content type	application/json

**Responses**

Code	Description	Response content type
200	Modifica effettuata	application/json
400	Richiesta non valida	
404	Utente/brano non trovato	
409	Azione non valida	

Figura 31: Documentazione API di tipo PATCH

**DELETE /api/eliminaAccount** Rimuovi un account dal sistema

**Parameters**

Name	Description
<b>ID utente</b> * required object (body)	Example Value   Model <pre>{   "idUtente": "639b88a1e5e03f310f6f8294" }</pre>
Parameter content type	application/json

**Responses**

Code	Description	Response content type
204	Utente rimosso dal sistema	application/json
400	Richiesta non valida	
404	Utente non trovato	

Figura 32: Documentazione API di tipo DELETE

## 6 Implementazione del frontend

Il frontend consiste in un'interfaccia grafica interattiva tramite la quale è possibile eseguire diverse azioni grazie alle API descritte nelle sezioni precedenti. È stato realizzato con il framework **VueJS**.

La schermata principale presenta due schede tramite le quali è possibile effettuare il login o registrarsi sulla piattaforma. Queste richiamano le corrispettive API per accesso registrazione viste precedentemente. Una volta effettuato l'accesso, il token restituito dall'API viene mantenuto in memoria dal frontend. Sarà necessario per le successive richieste alle API.

Effettuare l'accesso sblocca le varie funzionalità della piattaforma, presenti nella metà destra dell'interfaccia. È possibile fare il logout tramite pulsante apposito.

La prima funzionalità comparsa è la ricerca. È possibile cercare un brano tramite il titolo, e una volta ottenuti i risultati è possibile aggiungerli ai preferiti. Successivamente troviamo l'elenco dei preferiti, dai quali è possibile rimuovere un brano se desiderato. Segue infine una sezione dedicata alla rimozione del proprio account, decisione che prevede una conferma da parte dell'utente.

Nel caso in cui l'utente però sia creator, l'interfaccia guadagna alcune funzionalità. È presente una nuova sezione dedicata al caricamento di un brano, e per ciascun risultato nelle ricerche, se pubblicato da quel creator, è possibile tramite pulsanti appositi modificare o eliminare quel brano.

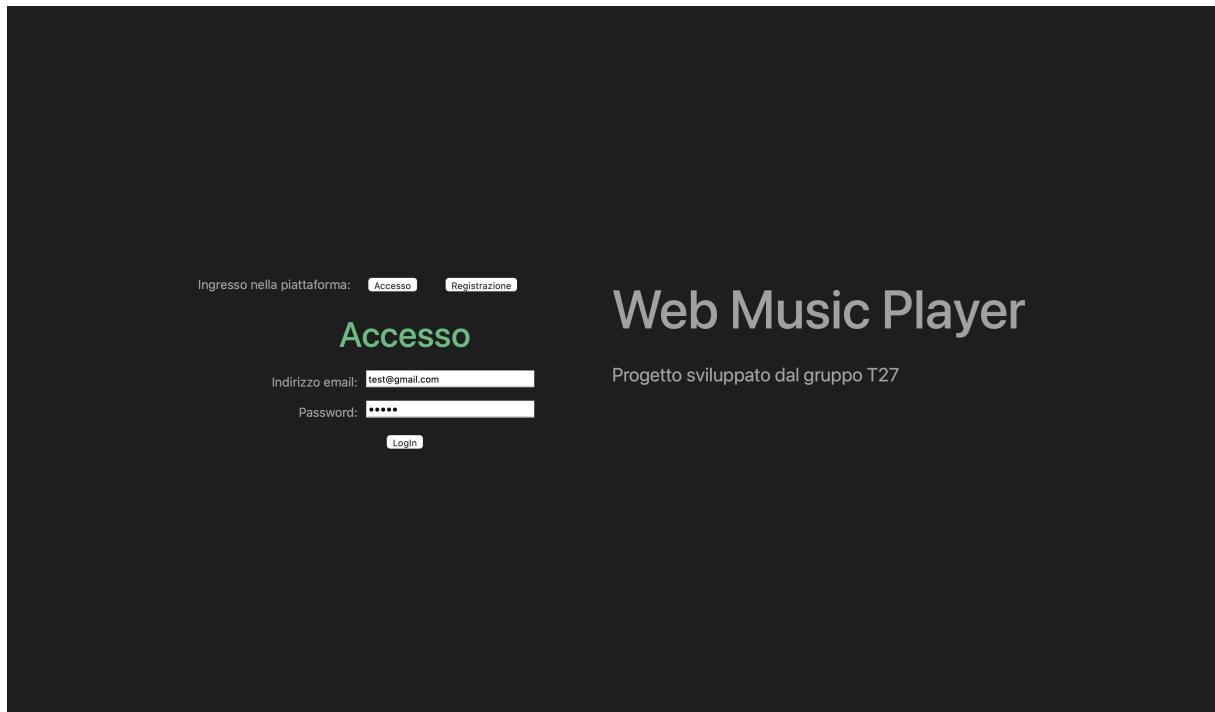


Figura 33: Schermata di accesso

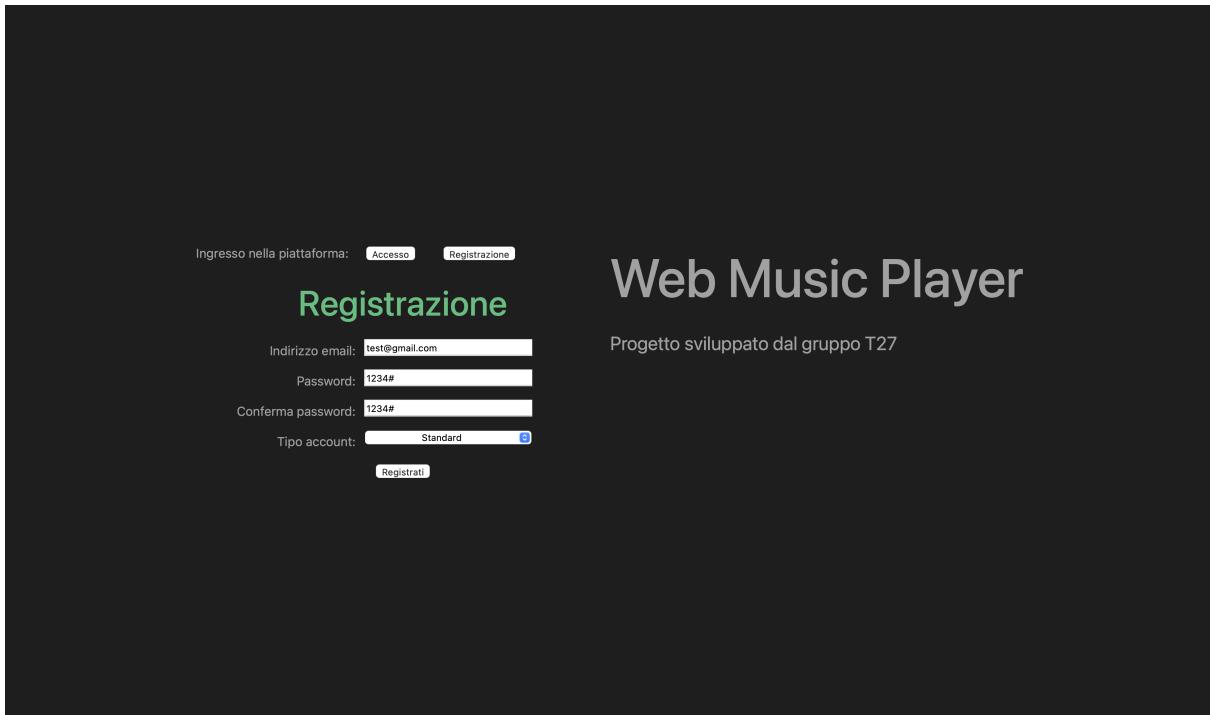


Figura 34: Schermata di registrazione

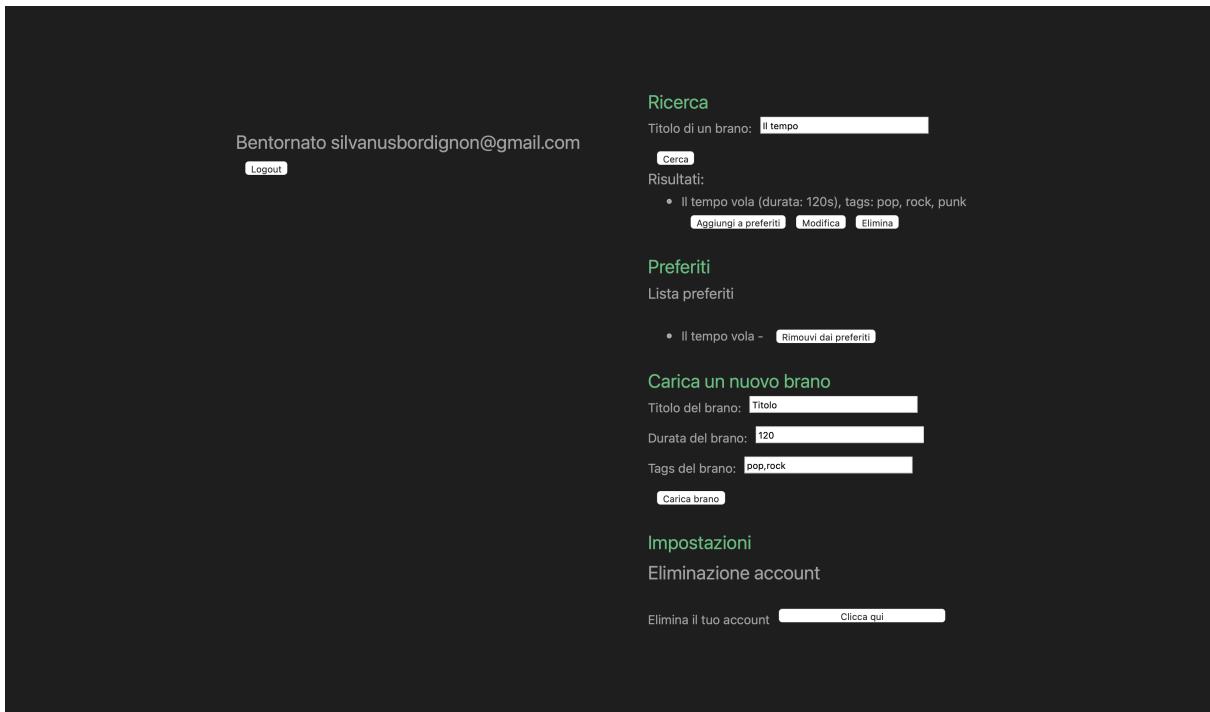


Figura 35: Schermata principale

## 7 GitHub repository e deployment

Il progetto è presente su GitHub. Le repository sono gestite dall'organizzazione Web Music Player. Il backend e il frontend sono salvati su due repository distinte.

Il deployment è gestito da due servizi diversi: **Railway** gestisce il deployment del backend, mentre **Netlify** si occupa del deployment del frontend. Le corrispettive applicazioni su GitHub sono collegate alle repository, in modo da assemblare ed effettuare il deploy ad ogni commit nel branch `main`.

- Backend: <https://wmp-backend.up.railway.app>
- Frontend: <https://wmp-frontend.netlify.app>

Nel caso in cui backend o frontend non dovessero essere online, è possibile seguire le istruzioni del `/README.md` all'interno della repository per avviare il progetto in locale.

## 8 Testing delle API

Il testing delle API è stato realizzato utilizzando due librerie di NodeJS: **Jest** e **supertest**. Jest è il framework principale, che gestisce la divisione in suite di test, l'esecuzione dei vari unit test e la produzione del report sulla test coverage. supertest è stato utilizzato per occuparsi delle richieste da inviare ai vari endpoint.

All'interno della cartella `/src/test` sono presenti i file di test. Sono presenti quattro suite di test, una per gruppo di API simili; corrispondono ai file presenti in `/src` che si occupano delle API.

All'interno della suite di test sono presenti tanti test case quante le possibili risposte che un'API può restituire. Di seguito il report sulla coverage dei test e un esempio di suite di test.

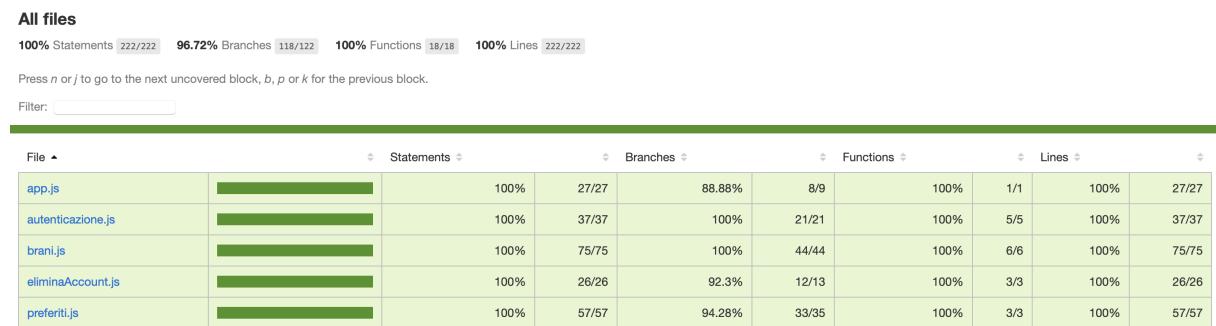


Figura 36: Coverage dei casi di test

```
● ● ●

import request from 'supertest';
import mongoose from 'mongoose';
import app from '../src/app';
import { generaUtenteTest, generaBranoTest } from '../scripts';

describe('Testing dell\'API di eliminazione account', () => {

  let connection;
  let id: string, token: string, idBrano: string;

  beforeAll(async () => {
    mongoose.set('strictQuery', true);
    connection = await mongoose.connect(process.env.MONGODB_URI || "");
    ({ id, token } = await generaUtenteTest());
    idBrano = await generaBranoTest(id);
  });

  afterAll(async () => {
    mongoose.connection.close(true);
  });

  test('ID non valido', async () => {
    const response = await request(app)
      .delete('/api/eliminaAccount')
      .set('Accept', 'application/json')
      .set('x-access-token', token)
      .send({
        idUtente: 'ID non valido'
      })

    expect(response.statusCode).toBe(400)
    expect(response.body).toEqual({ message: 'Il valore inserito per l\'utente non è un ID valido' });
  });

  test('Utente non trovato', async () => {
    const response = await request(app)
      .delete('/api/eliminaAccount')
      .set('Accept', 'application/json')
      .set('x-access-token', token)
      .send({
        idUtente: '00000000000000000000000000000000'
      })

    expect(response.statusCode).toBe(404)
    expect(response.body).toEqual({ message: 'Utente non trovato' });
  });

  test('Eliminazione utente', async() => {

    const response = await request(app)
      .delete('/api/eliminaAccount')
      .set('Accept', 'application/json')
      .set('x-access-token', token)
      .send({
        idUtente: id
      })

    expect(response.statusCode).toBe(204);
  });
});
```

Figura 37: Esempio suite di test - Eliminazione account