UNIVERSITY OF TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Progetto:

RevHub

Titolo del documento:

Analisi del Programma

Scopo del documento

Il seguente documento si porta il compito di spiegare il procedimento di come è stato realizzato RevHub.
Come prima sezione si vede lo User Flow, ovvero una rappresentazione visiva di tutte le decisioni possibili sotto forma di timelines delle funzionalità sviluppate.
Successivamente è presente la struttura delle cartelle e dei file presenti, con una spiegazione delle librerie esterne usate per la programmazione,
Per poi continuare con un'analisi delle API sviluppate, fatta attraverso il diagramma delle risorse estrapolato dal diagramma delle classi presente nel deliverable 3.
Nel capitolo successivo è presente la documentazione delle API con Swagger.
Procedendo avanti è presente una breve descrizione delle pagine utilizzabili dagli utenti.
Successivamente una sezione con la spiegazione della repository di GitHub e di come provare la piattaforma dal punto di vista di un utente.
Per concludere un report della fase di testing e dei risultati.

1. User Flow

1.1 Legenda

In questo Capitolo viene presentato lo user-flow dell'applicazione, ovvero una descrizione grafica del flusso di azioni che qualsiasi tipologia di utente può seguire all'interno della piattaforma.
Di seguito una didascalia dei componenti utilizzati.

Figura 1.1 Legenda dello User Flow

Ora a seguire varie parti dello User Flow con una loro descrizione per poi mostrare tutto lo User Flow completo.

1.2 Sezioni dello User Flow

Abbiamo suddiviso lo user flow per consentirne una corretta rappresentazione. È stata comunque inclusa alla fine del capitolo una sua versione completa.

1.2.1 Ricerca

Questa sezione rappresenta un frangente riguardante il momento in cui un qualsiasi utente esegue una ricerca, per poi ricavare una lista di risultati e dopo una selezione di uno dei risultati si ottiene la pagina.
Pagina profilo se nella barra di ricerca si è digitato lo username preceduto da "@", pagina della recensione se si è digitato nella barra di ricerca "#" (eseguirà una ricerca

nel database per tag) o sempre di una pagina della recensione con testo senza particolare formattazione nella barra di ricerca (eseguirà una ricerca nel database per titolo).

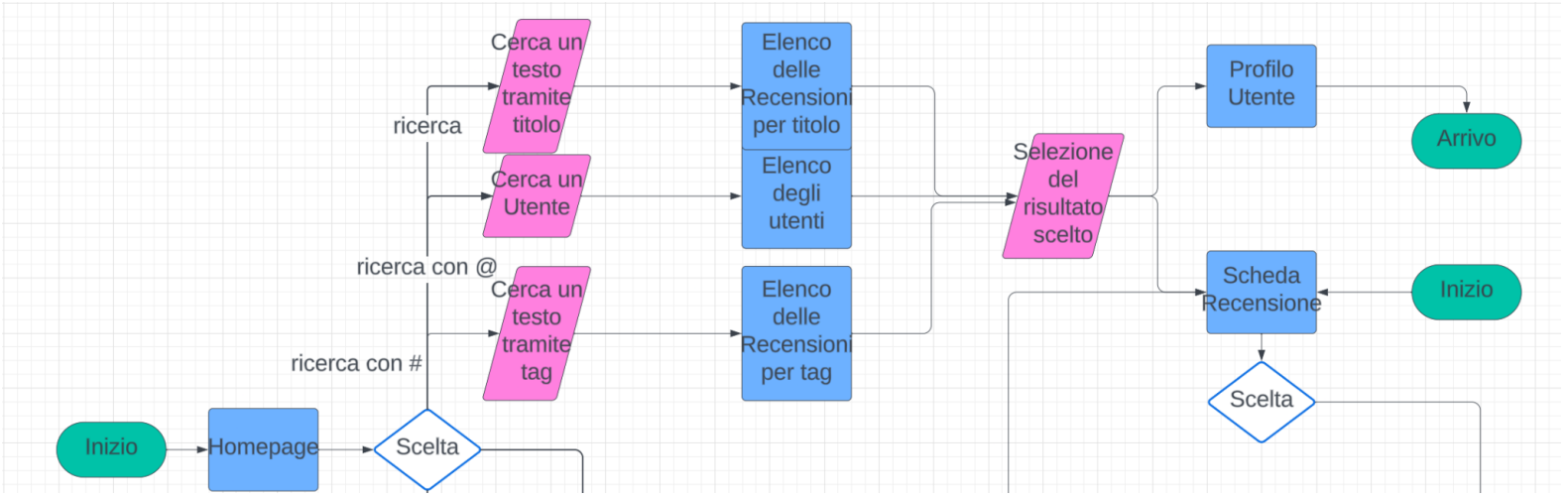


Figura 1.2.1 Ricerca

1.2.2 Menù

Qui è presente la sequenza di azioni possibili per un utente autenticato.
Quindi per lei/lui è possibile selezionare la visualizzazione del profilo personale, fare il logout o di creare una recensione.

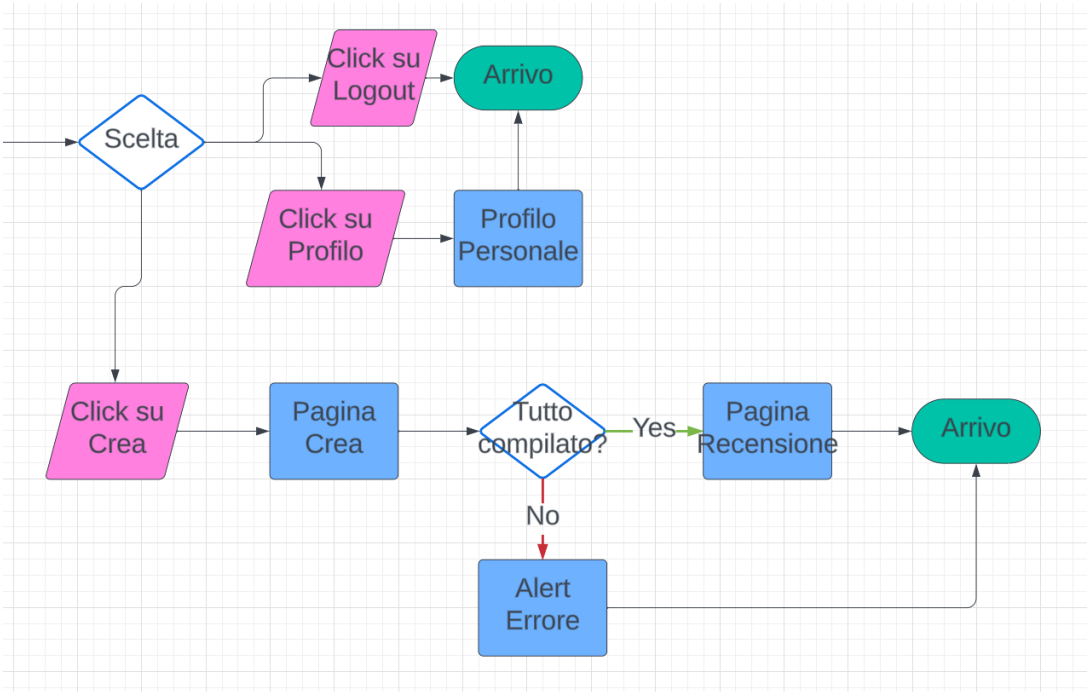


Figura 1.2.2 Menù

1.2.3 Like

Qui un utente non autenticato se cercasse di mettere like riceverebbe un errore, mentre ciò non avviene per quelli autenticati.

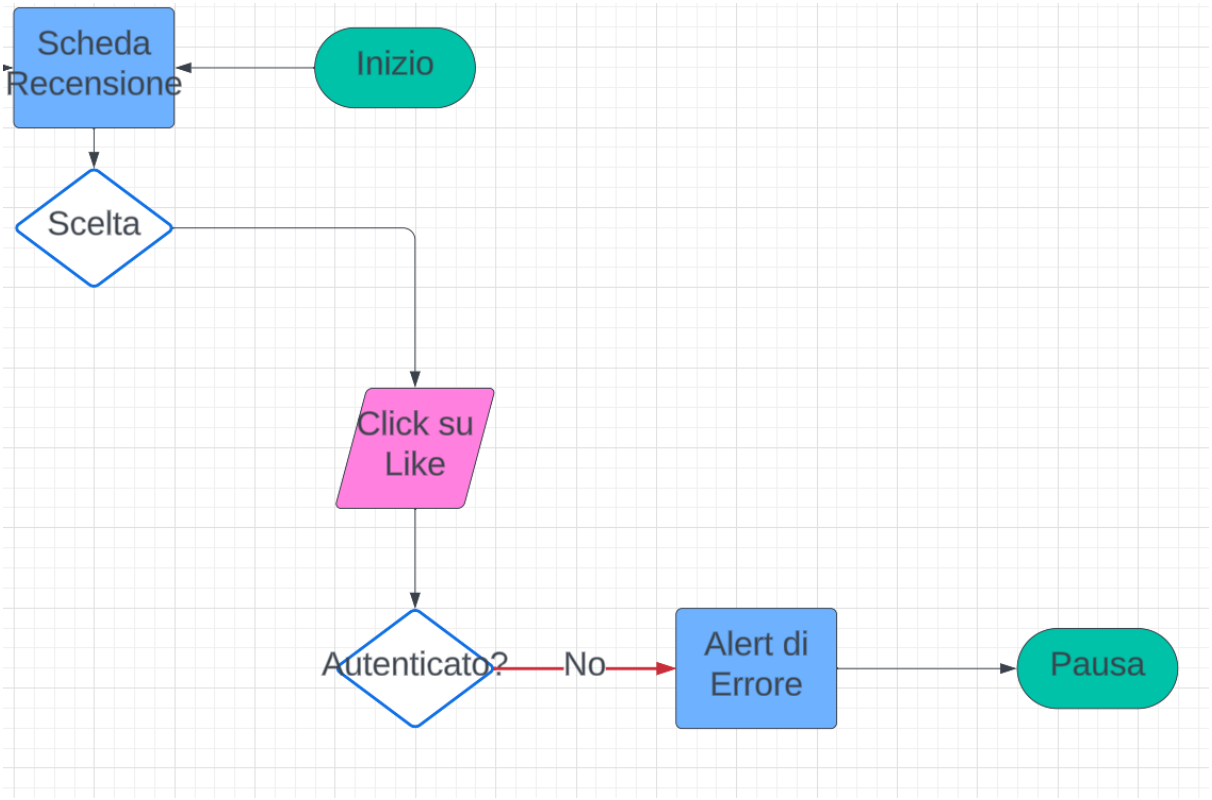


Figura 1.2.3 Like & Commenti

1.2.4 Login

Per concludere è presente il frangente di azioni disponibili durante la creazione di un profilo tramite credenziali già esistenti di Ateneo o Google.
Nello specifico l'utente non autenticato può andare su Login e inserire il profilo che dovrebbe già possedere, se non fosse così può andare nella pagina di registrazione dove deve scegliere una password, username e scegliere se usare le credenziali di Ateneo o di Google con relativa email associata.

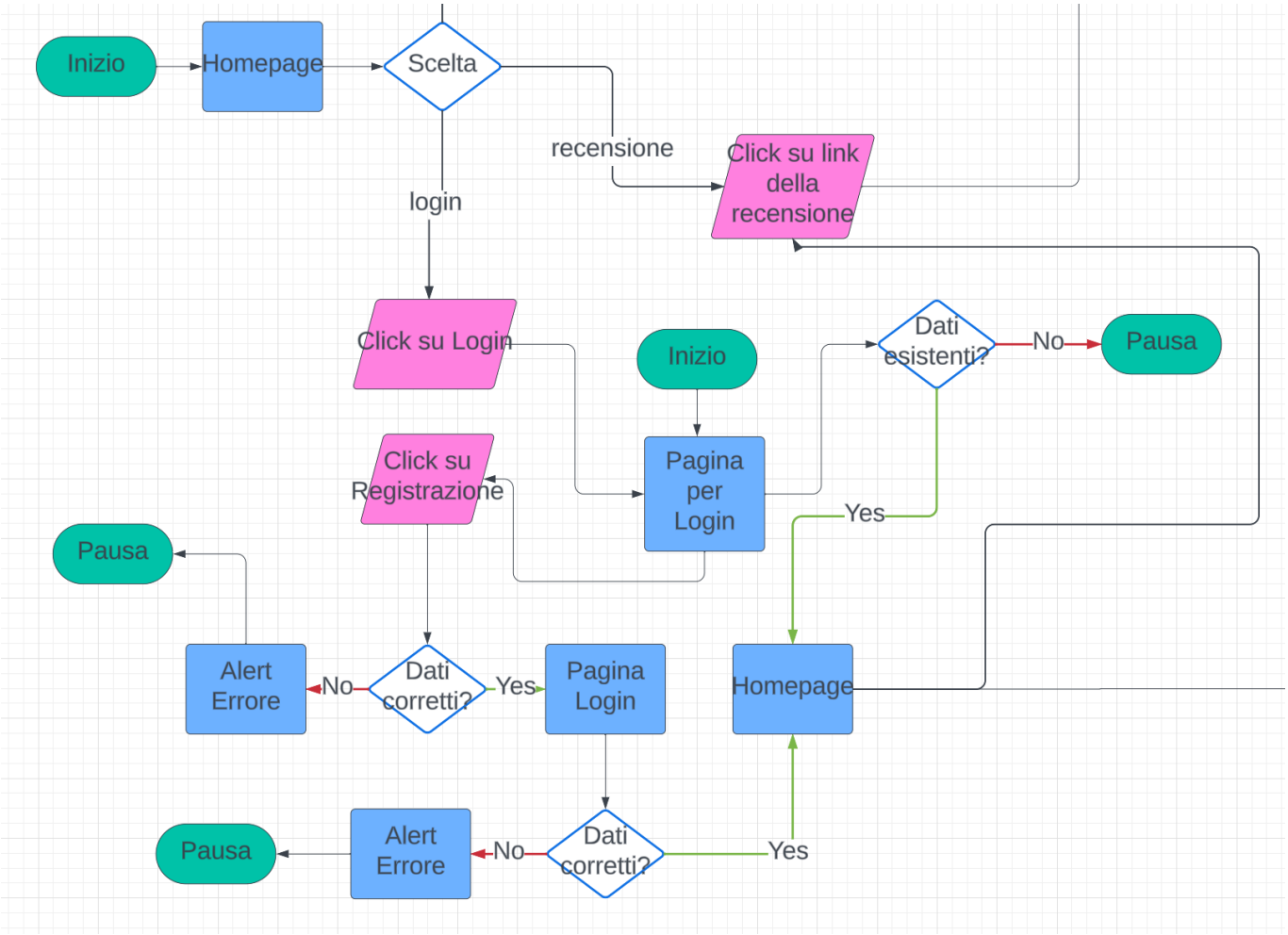


Figura 1.2.4 Login

1.3 User Flow

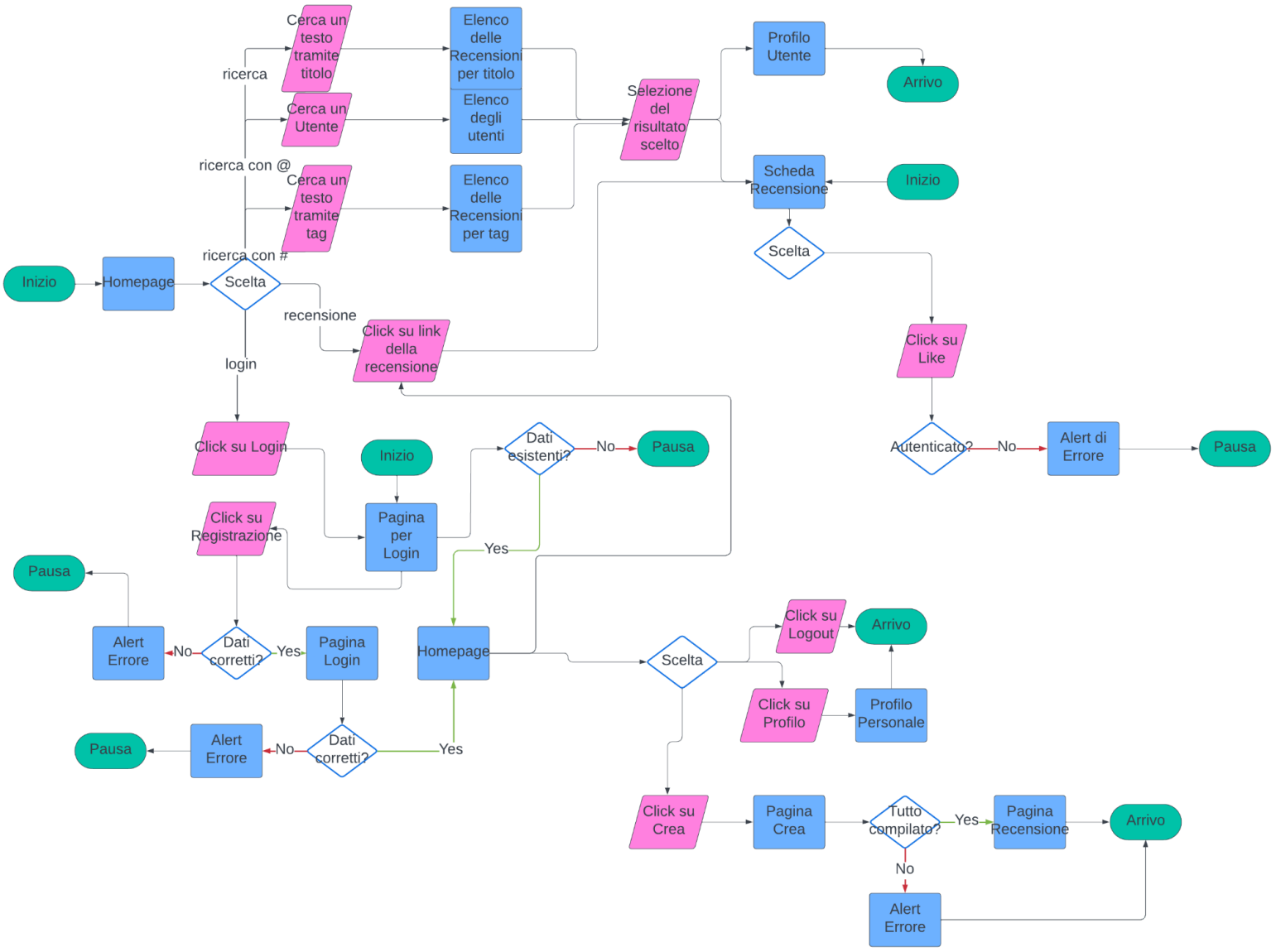
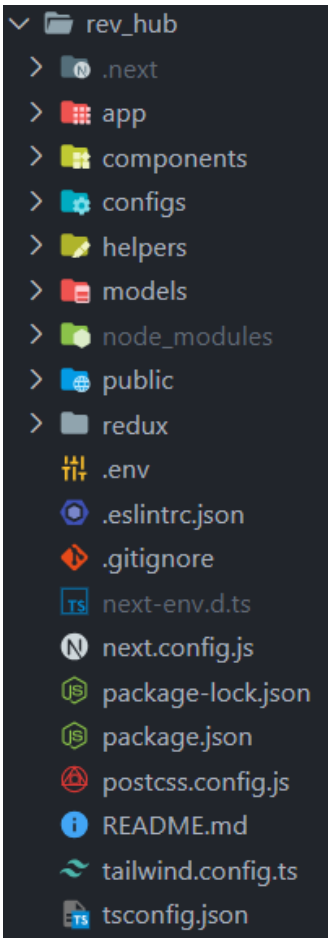


Figura 1.3 User Flow

2. Struttura del Codice

L'applicazione RevHub è stata sviluppata utilizzando per la parte di frontend NodeJS, e a livello di backend è stato usato MongoDB.
Come si può vedere dallo User Flow, sono state implementate tutte le componenti in esso descritte, come la creazione di un profilo con credenziali esterne, il login, la ricerca di un utente o recensione, la pagina profilo, visualizzazione di una recensione e la creazione di una recensione e infine la homepage.

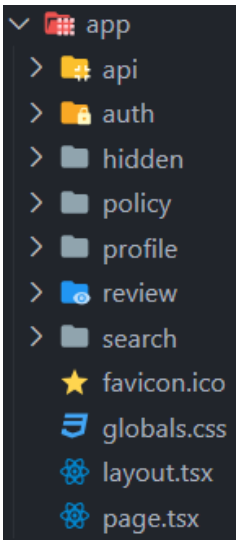


La directory dell'applicazione è **rev_hub** ed è formata dai seguenti file:

- **/.next** - contiene la build dell'applicazione;
- **/app** - contiene le pagine, gli stili e le api;
- **/components** - dove sono definiti la barra di ricerca, del menù, il footer;
- **/configs** - che contiene il modulo per la connessione al db;
- **/helpers** - dove si delega la validazione di un token salvato nei cookies per mantenere il login;
- **/models** - contiene le definizioni dei modelli utilizzati per definire la struttura degli oggetti salvati nelle collezioni del database;
- **/node_modules** - contiene i pacchetti delle dipendenze installate e delle dipendenze del sistema;
- **/public** - contiene i file da rendere disponibili al pubblico;
- **/redux** - contiene i due file usati per fornire un contesto ad un layout consentendo l'uso di variabili globali;
- **.env** - contiene le variabili d'ambiente (come le credenziali di connessione al database);
- **.gitignore** - contiene l'elenco delle risorse da non caricare su git;
- **next.config.js** - contiene la configurazione di Next;
- **package-lock.json** - è una versione più approfondita del fila package.json;
- **package.json** - contiene alcuni dati sull'applicazione, gli scripts richiamabili su di essa e l'elenco di dipendenze e di dipendenze di sviluppo;
- **README.md** - è un file che contenente alcune note lasciate da chi ha scritto il codice dell'applicazione a chi deve consultare tale codice;
- i rimanenti file sono file di configurazione di vario tipo, che nel nostro caso sono stati creati automaticamente e abbiamo lasciato invariati.

La cartella /app è definita come segue:

- **/api** - dove sono definite tutte le API;
- **/auth** - dove sono definite le pagine di login e registrazione;
- **/hidden** - riguarda la creazione di credenziali fruibili successivamente per la creazione di un profilo;
- **/policy** - contenente la pagina coi nostri termini di servizio per adempiere a una parte dei requisiti non funzionali;
- **/profile** - dove è possibile la visualizzazione del profilo di un utente;
- **/review** - contente il codice per creare e visualizzare una recensione;
- **/search** - contenente i risultati della ricerca



2.1 Dipendenze

I moduli Node presenti nel file package.json nel campo dependencies sono:

```
"dependencies": {
  "@headlessui/react": "^1.7.17",
  "@heroicons/react": "^2.0.18",
  "@reduxjs/toolkit": "^2.0.1",
  "@types/bcrypt": "^5.0.2",
  "@types/jsonwebtoken": "^9.0.5",
  "axios": "^1.6.2",
  "bcrypt": "^5.1.1",
  "date-fns": "^2.30.0",
  "headlessui": "^0.0.0",
  "heroicons": "^2.0.18",
  "jsonwebtoken": "^9.0.2",
  "mongoose": "^8.0.2",
  "next": "14.0.3",
  "react": "^18",
  "react-dom": "^18",
  "react-redux": "^9.0.2",
  "sass": "^1.69.5"
},
```

Di questi, quelli installati da noi per soddisfare i requisiti dell'applicazione sono:

- jsonwebtoken: modulo per la gestione dei token di accesso;
- mongoose: per le interazioni con MongoDB;
- date-fns: per la formattazione e la visualizzazione dell'orario di creazione di una recensione;
- axois: per eseguire chiamate http alle api;
- bcrypt: per l'hashing delle password;
- heroicons: raccolta di icone;
- react-redux: per creare un contesto (per avere variabili globali).

2.2 Modelli nel Database

Per la gestione dei dati nel progetto sono stati definiti modelli a partire dal diagramma delle classi presente nel deliverable_3. Di conseguenza le risorse necessarie da gestire hanno fatto sì che si sviluppassero cinque modelli con ognuno una collezione nel database.

Modello Comment

Il seguente modello è stato sviluppato a livello di design ma non è stato usato nello sviluppo del codice, come mostrato da user flow. Esso rappresenta i dati necessari per la memorizzazione di un commento fatto da un utente autenticato.

```
export const Comment = new mongoose.Schema (
    {
        id: { type: Number, required: true },
        review_id: { type: Number, required: true },
        author_id: { type: Number, required: true },
        text: { type: String, required: true },
        date: { type: Date, required:true }
    }
)
```

review_id si riferisce all'identificativo della recensione a cui fa riferimento, mentre author_id il medesimo ma si riferisce all'utente autenticato che ha compilato il commento.

Modello Credential

Necessario per simulare l'uso di credenziali UniTn e Google.

```
export const Credential = new mongoose.Schema(
    {
        organization: { type: String, required: true },
        email: { type: String, required: true },
        password: { type: String, required: true },
        name: { type: String, required: true },
        surname: { type: String, required: true }
    }
)
```

Modello Rate

Questo particolare modello serve per poter memorizzare nel db di MongoDB l'associazione di ogni singola valutazione in termini di like e dislike. Infatti viene memorizzato con un booleano rate se stiamo parlando di un like (con true) o dislike (con false), poi chi ha messo tale valutazione e a quale recensione.

```
export const Rate = new mongoose.Schema (
    {
        author_id: { type: Number, required: true },
        review_id: { type: Number, required: true },
        rate: { type: Boolean, required: true }
    }
)
```

Modello Review

Per memorizzare le recensioni sono stati usati i seguenti valori, di cui tags, title e text possono essere compilati dall'utente e solo gli ultimi due sono obbligatori. Gli altri dati vengono ottenuti o generati automaticamente dal sistema.

```
export const Review = new mongoose.Schema(
    {
        id: { type: Number, required: true, unique: true },
        title: { type: String, required: true },
        author_id: { type: Number, required: true },
        date: { type: Date, default: Date.now, required: true, },
        tags: { type: [String], default: [], required: false },
```

```
        text: { type: String, required: true },
        views: { type: Number, default: 0, required: false }
    }
)
```

Modello User

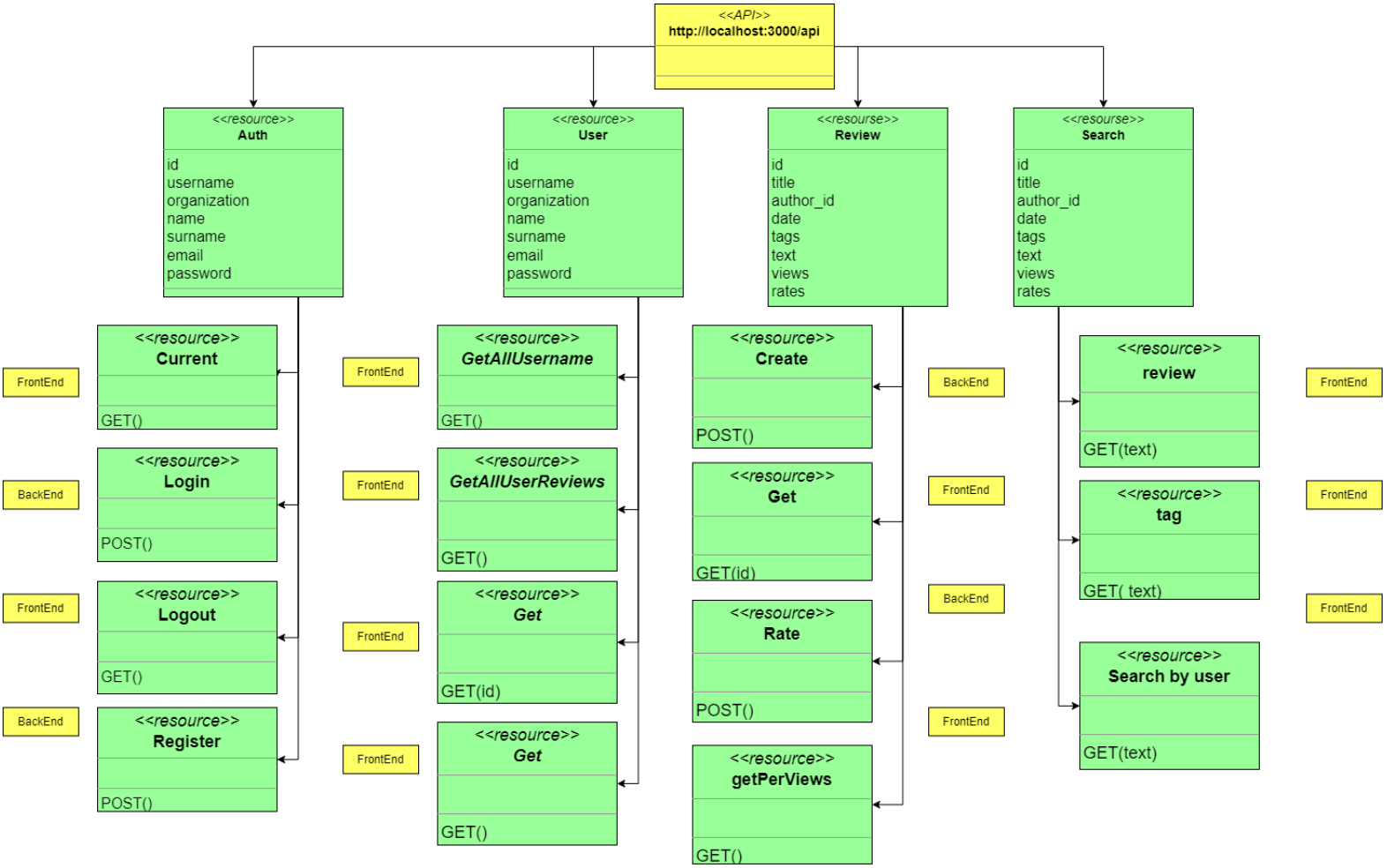
Per memorizzare i dati sensibili dell'utente ed effettuare la sua registrazione e la verifica delle sue credenziali, come anche la visualizzazione dei suoi dati nella pagina profilo è disponibile il seguente schema usato.

```
export const User = new mongoose.Schema(
    {
        id: { type: Number, required: true, unique: true },
        name: { type: String, required: true },
        surname: { type: String, required: true },
        username: { type: String, required: true, unique: true },
        email: { type: String, required: true, unique: true },
        password: { type: String, required: true },
        followers: { type: [Number], default: [], required: false },
        isAdmin: { type: Boolean, default: false, required: false, }
    }
)
```

Da notare come sia l'*id* che lo *username* sono etichettati come unique, ciò è stato scelto per garantire l'unicità dello username, senza tuttavia usarlo come chiave primaria (scopo riservato all'id).
Successivamente sono stati raccolti i dati sensibili come name, surname, email e password.
followers è un vettore contenente gli id degli utenti che seguono la persona. Inoltre *isAdmin* è un booleano che identifica se un utente autenticato è anche admin. Questi due ultimi dati con le relative funzionalità non sono state sviluppate (come da user flow) ma sono state incluse nel modello (come da diagramma delle classi).

3. Estratto delle risorse

In questa sezione del documento verranno descritte le API sviluppate dal diagramma delle classi del documento deliverable_3.
Di seguito è presente una estrazione delle risorse dal diagramma delle classi. Questo diagramma mostra come sono state estratte le “risorse” a partire dal diagramme delle classi.
Si parte dalla directory /api per poi avere quattro sotto cartelle contenenti le proprie API: Auth, User, Reviews e Search (con ognuno i dati che utilizza presi dai modelli nel db precedentemente descritti).
Sono state omesse le API della cartella /hidden, in quanto servono per la creazione dei profili per l’accesso con credenziali esterne, allo scopo di simulare l’uso di credenziali UniTn e Google.

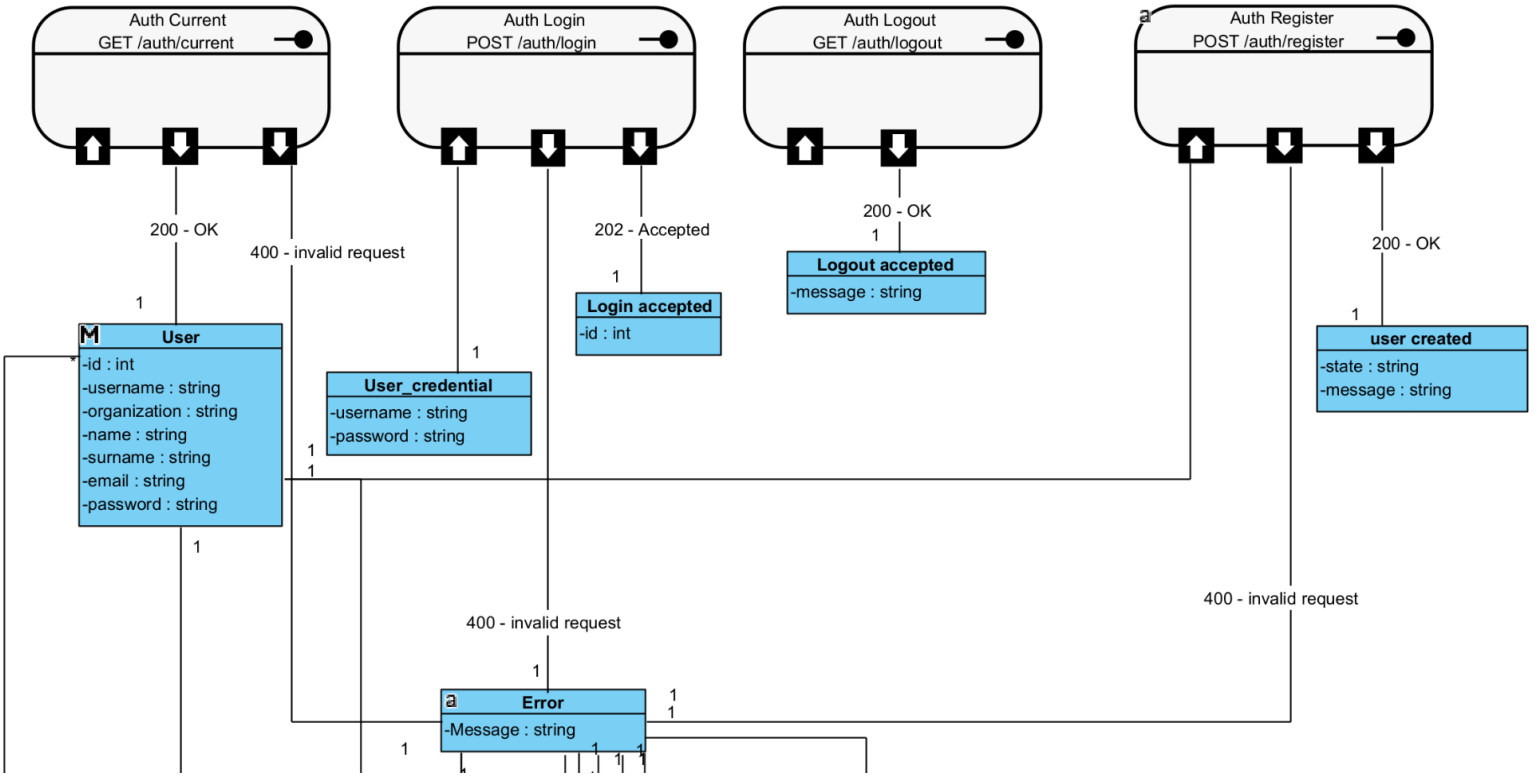


4. Diagramma delle risorse

Di seguito una rappresentazione in sezioni del diagramma delle risorse per poi avere tutto il diagramma. In tutte le parti del diagramma sono state specificate le componenti in input e in output. Ovviamente le varie API possono ritornare un file json con il risultato desiderato, oppure un errore etichettato; perciò è stata creata una unica classe Error a cui abbiamo collegato tutto per evitare di duplicare classi e complessità.

4.1 /auth

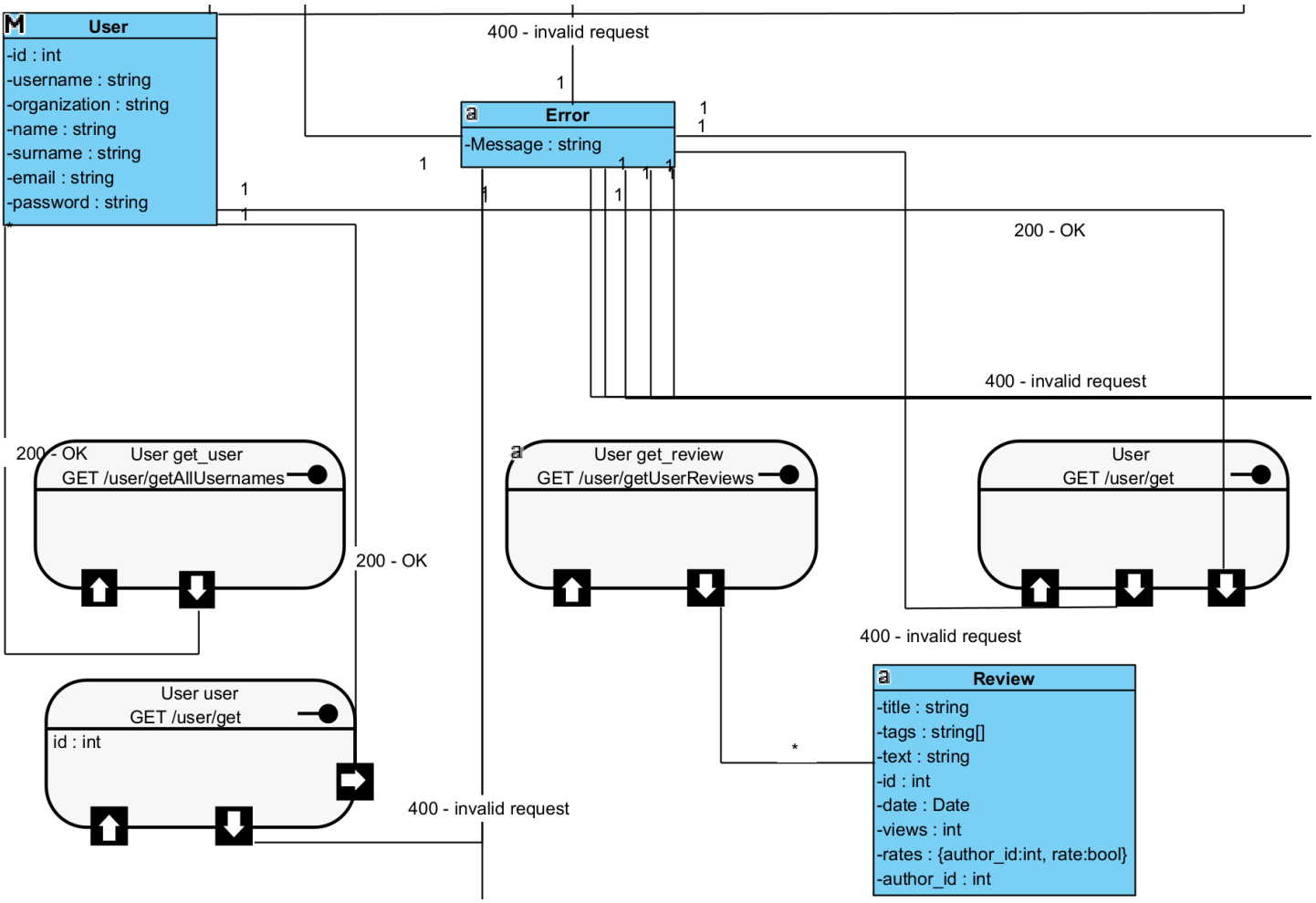
In questa sezione di diagramma le possibilità di output sono un 200 - ok per rappresentare che l'azione è stata effettuata con successo, oppure un 400 - invalid request tutti collegati alla classe Error per identificare una risposta invalida e quindi un errore. E' possibile ricevere come output la classe User che rappresenta la risorsa User analizzata nell'estratto delle risorse precedenti. Da notare la differenza tra User e User_credential, in quanto la prima si riferisce a tutti i dati che si possono estrarre dal db di un utente autenticato, mentre il secondo si riferiscono ai soli dati inseriti dall'utente non autenticato per ottenere l'autorizzazione ad accedere con tali credenziali.



4.1 Sezione /auth

4.2 /user

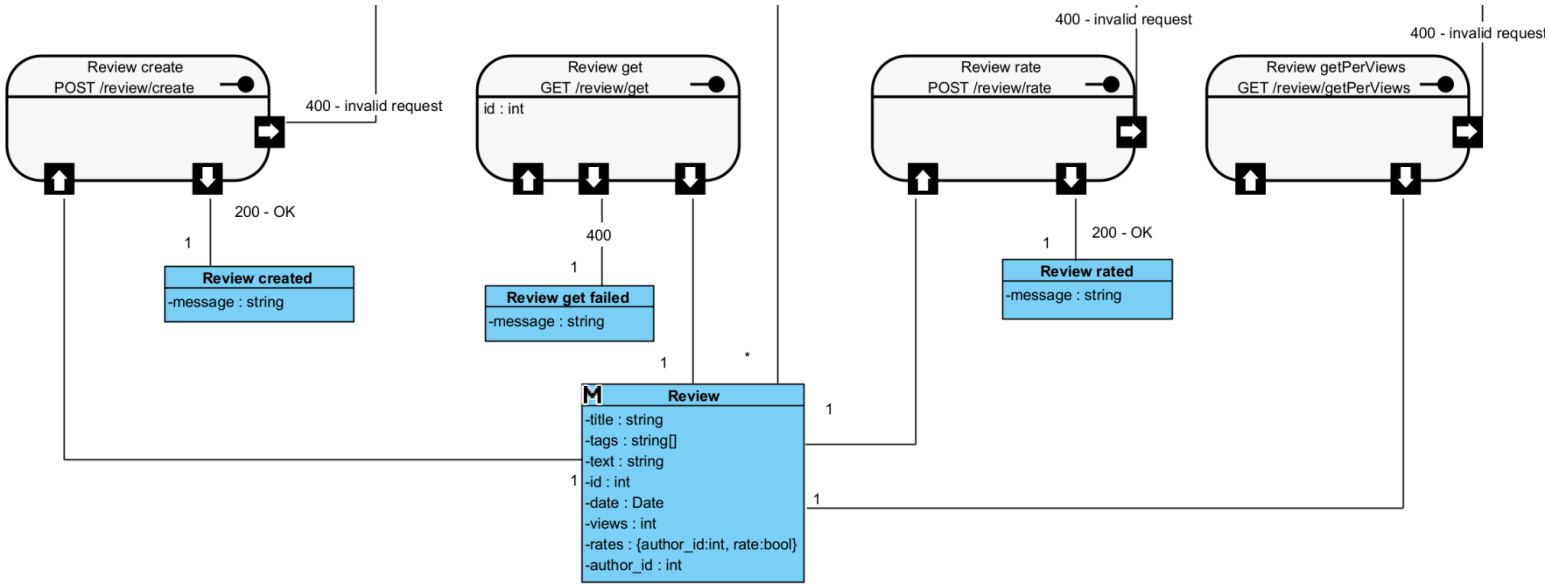
Qui è possibile ricevere in output la classe User, le recensioni associate ad un utente oppure la classe standard Error.



4.2 Sezione /user

4.3 /review

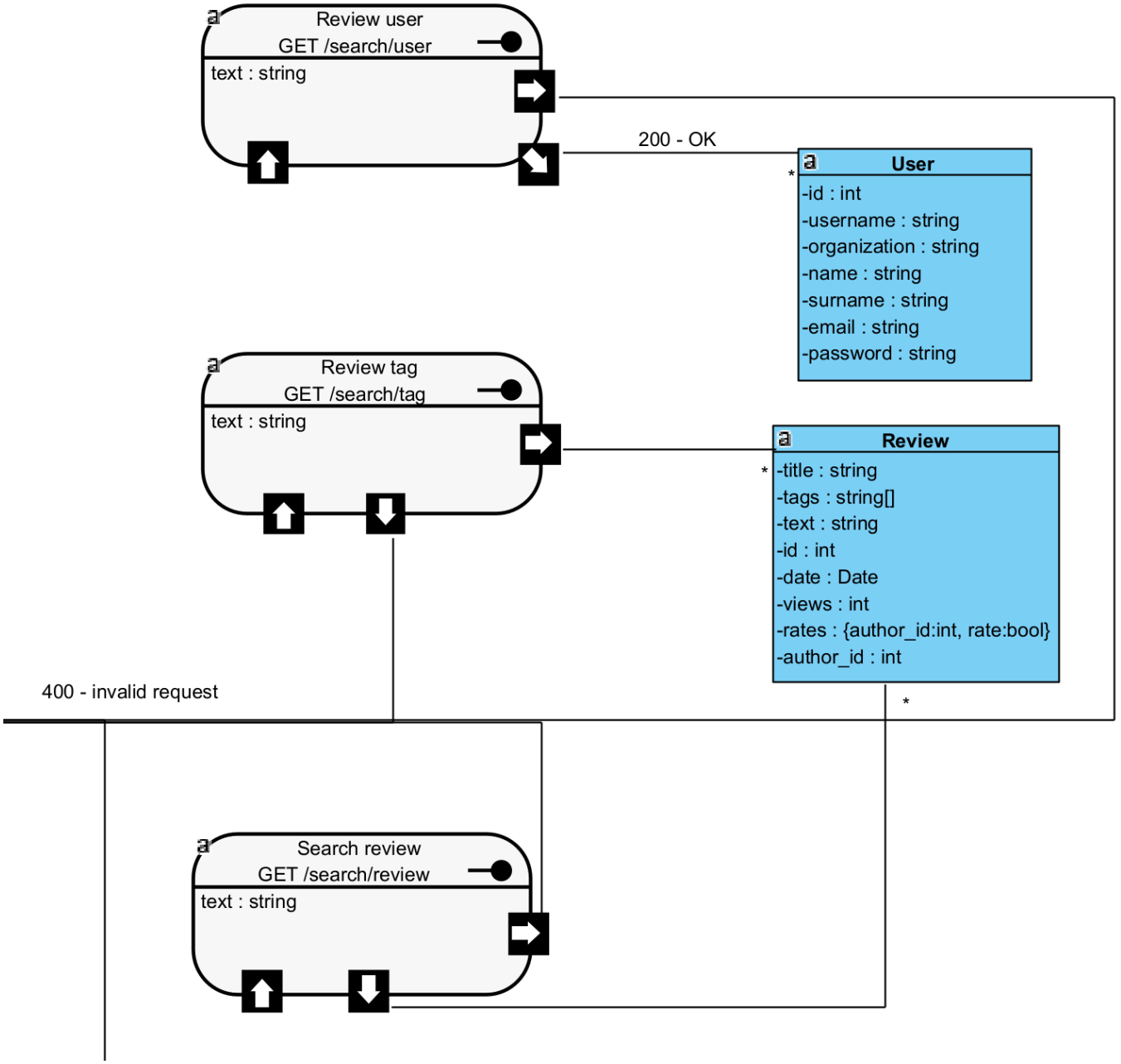
Qui si può ottenere come input e anche output la classe Review oppure ricevere un 200 - ok o un errore 400.
Anche questi, come i precedenti, possono dare un errore 400 ed essere quindi collegati alla classe precedente.



4.3 Sezione /review

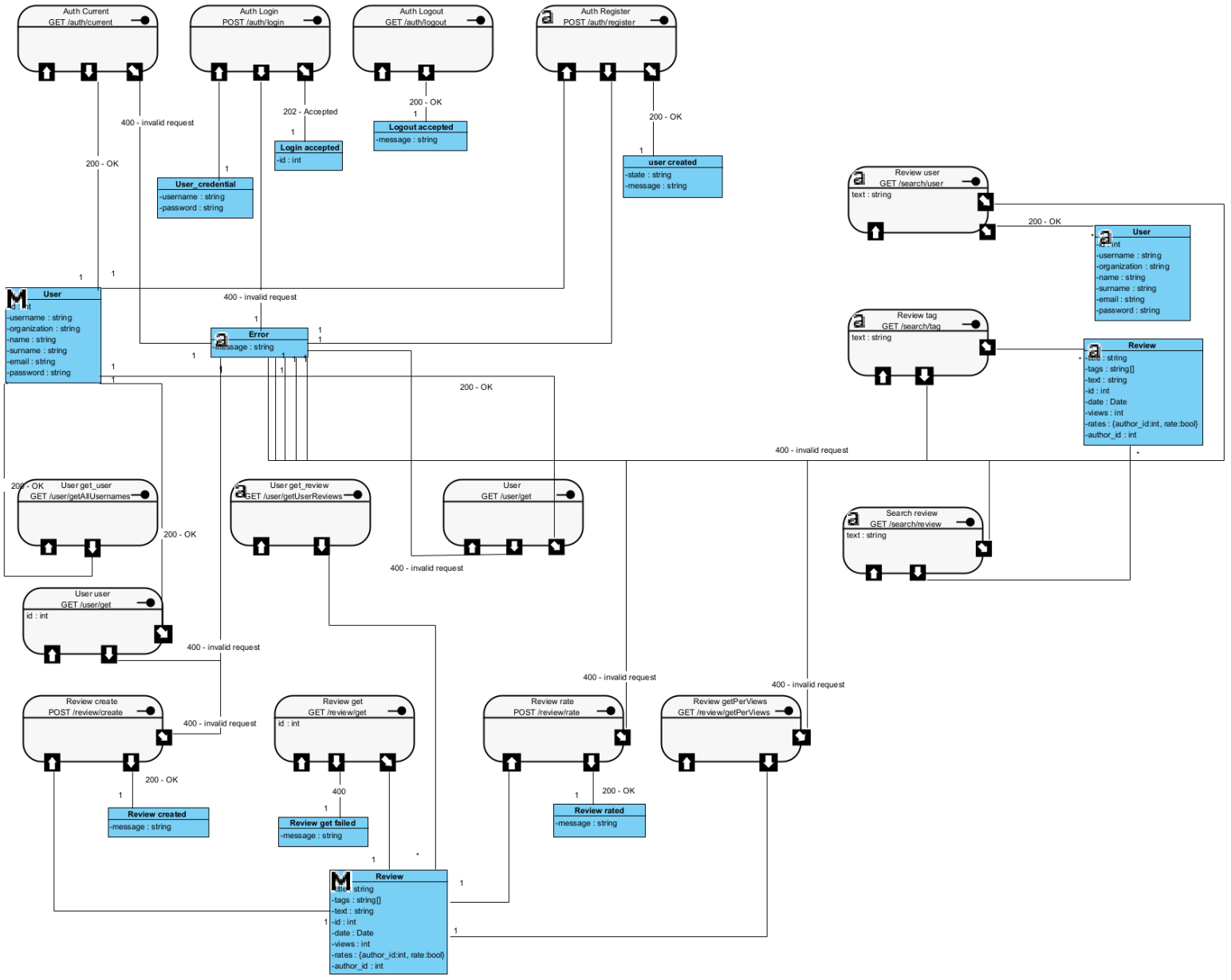
4.4 /search

Infine questa sezione per la ricerca di una recensione tramite titolo o tag, oppure in un utente si può ricevere in input solo il parametro text compilato dall'utente e in output una lista coi risultati coerenti.
Dove tutte e tre le api sono collegate alla classe Error citata precedentemente, ovvero possono dare come output anche un errore.



4.4 Sezione /search

4.5 Diagramma delle risorse



4.5 Diagramma delle risorse completo

5. API Documentation

Qui di seguito verrà descritto il funzionamento delle API sviluppate.
Tali API consentono di interagire con il database, consentendo di aggiungere, modificare e visionare il suo contenuto.

Le API locali fornite da RevHub e i parametri da esse coinvolti sono stati documentati in modo approfondito tramite il file *revhub_api.yaml* contenuto nel progetto all'interno della cartella *RevHub > api_documentation*.
È possibile osservare l'output di tale file recandosi su Swagger Editor e importando il file yaml.
Abbiamo riportato qui sotto una breve descrizione della funzione delle varie API. Eseguono tutte compiti molto semplici, riassumibili quasi tutte con uno schema del tipo:

1. connessione al database;
2. esecuzione di una o più query;
3. restituzione del risultato dell'operazione eseguita;

con alcune eccezioni per alcune delle api che richiedono una banale elaborazione dei dati (ad esempio la composizione della risposta a partire dal risultato di più query).

5.1 API di /auth

Usate per permettere all'utente di registrarsi, fare login e logout e per verificare se l'utente ha fatto o meno il login.

5.1.1 /current

Controlla se nei cookies della request è contenuto un token che corrisponde ad un id chiave di un utente registrato.

5.1.2 /login

Tramite una query al database, viene controllato se le credenziali inserite nel login sono valide. Se lo sono viene inserito nella response un token associato all'utente a cui appartengono tali credenziali.

5.1.3 /logout

Per effettuare il logout si elimina, se esiste, il token dell'utente contenuto nella request.

5.1.4 /register

Permette di creare un nuovo utente collegato alle credenziali UniTn o Google inserite.

5.2 API di /review

Sono le api usate per creare, visualizzare ed interagire con le recensioni.

5.2.1 /create

Crea una nuova recensione usando i dati passati nel corpo della request.

5.2.2 /get/{id}

Restituisce la recensione con ID pari all'ID passato come parametro.

5.2.3 /getPerViews

Restituisce le recensioni con il maggior numero di visualizzazioni.

5.2.4 /rate

Gestisce la votazione degli utenti, aggiornando la raccolta di Rate nel database in base ai dati ricevuti in modo da rispecchiare gli input degli utenti.

5.3 API di /search

Sono le api che consentono di cercare utenti e recensioni.

5.3.1 /review/{text}

Restituisce quelle recensioni che hanno un match tra il testo usato per la ricerca ed il loro titolo.

5.3.2 /tag/{text}

Restituisce quelle recensioni che possiedono tra i tag quello inserito nella ricerca.

5.3.3 /user/{text}

Restituisce quegli utenti che hanno un match tra il testo usato per la ricerca ed il loro username.

5.4 API di /user

Sono le api usate per ottenere dei dati riguardanti gli utenti.

5.4.1 /get/{id}

Restituisce l'utente con ID pari all l'ID passato come parametro.

5.4.2 /get

Restituisce l'utente corrente, utilizzando il token contenuto nella request.

5.4.3 /getAllUsernames

Restituisce una lista con tutti gli username di tutti gli utenti esistenti.

5.4.4 /getUserReviews/{id}

Restituisce le recensioni dell'utente il cui ID è stato passato come parametro

6. Descrizione pagine

In questa sezione vengono descritte le schermate disponibili del frontend e di come possono essere utilizzate.

Homepage (autenticato)

Partendo dalla pagina principale, visibile a tutti gli utenti, è possibile in prima azione e senza fare nulla visionare la classifica delle recensioni più visualizzate. Se si volesse interagire con la barra di navigazione si possono aprire delle scelte per gli utenti autenticati, per quelli non autenticati verrà trattato successivamente.
Se si è autenticati si possono fare cinque azioni col menù.
Cliccare in alto a sinistra il logo per ritornare alla homepage.
Cliccare la barra di ricerca e dopo aver scritto qualcosa premere “Cerca” per ottenere i risultati desiderati. Si possono vedere i risultati delle ricerche solo se si compila l’input della barra di ricerca.
Cliccare “Profilo” per visionare i propri dati di profilo e vedere le recensioni scritte associate a quel profilo. Per visionare il profilo di altri utenti autenticati, bisogna digitare “@”+<nome_utente> nella barra di ricerca.
Cliccare “Crea” per accedere alla pagina per creare una recensione.
Oppure cliccare “logout” per effettuare il logout.

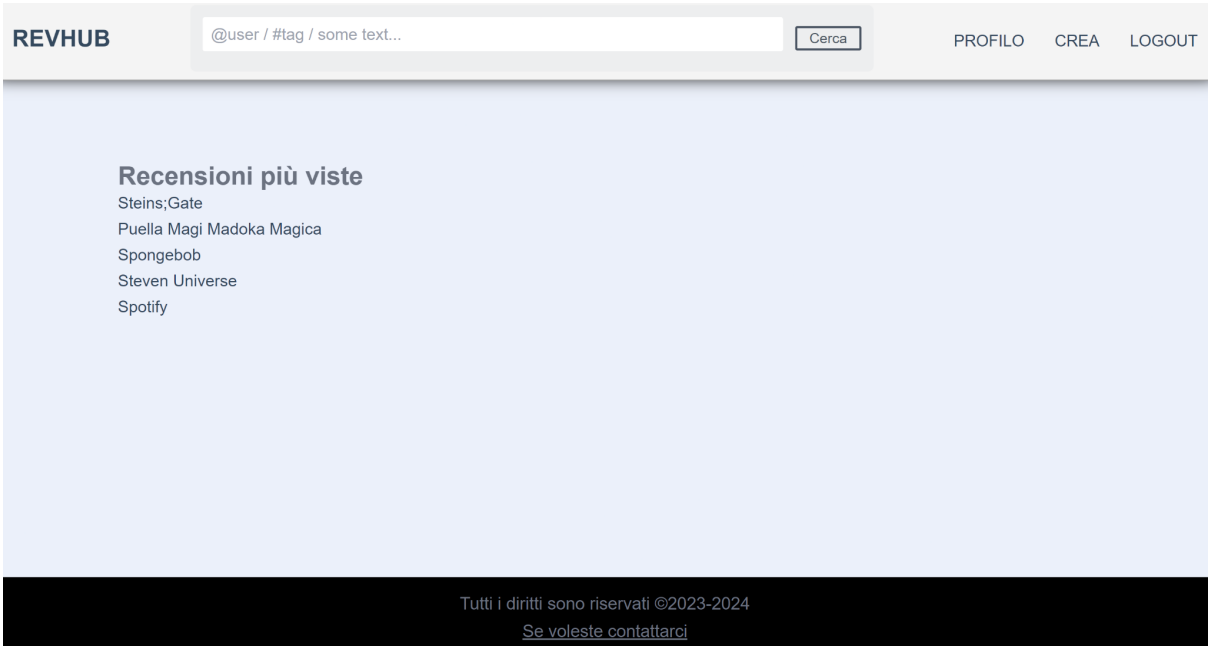


Figura 6.1 - schermata home di un utente loggato

Visualizzazione di una recensione

Dopo aver scritto un campo di testo nella barra di ricerca e aver selezionato una delle scelte ottenute per le recensioni, oppure dopo aver cliccato una delle recensioni esistenti nella pagina profilo di qualcuno si arriva alla sua visualizzazione.
In questa schermata, fruibili per tutti gli utenti, è possibile visualizzare la recensione e mettere like o dislike (questa opzione è disponibile dopo essersi autenticati).

Figura 6.2 - schermata visualizzazione di una recensione

Pagina profilo

Dopo aver cliccato “Profilo” nella barra di navigazione, o aver selezionato uno dei risultati nella pagina di ricerca, si possono vedere i dati associati a quel profilo e tutte le recensioni che ha scritto. Qui si possono cliccare i titoli delle recensioni per andare alla pagina descritta precedentemente e visualizzare il contenuto.

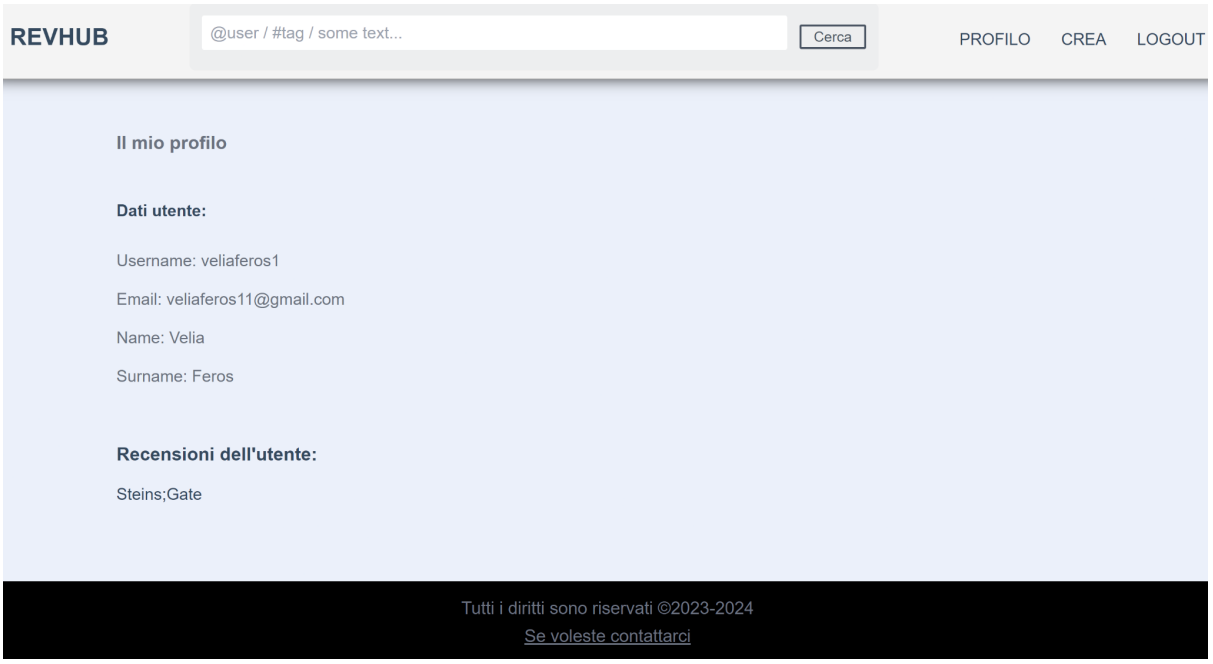


Figura 6.3 - schermata di profilo

Pagina Creazione

Dopo aver cliccato “Crea” nel menù si accede a questa pagina con un form da compilare per creare la propria recensione, si può accedere qui solo se autenticati.
Il campo Titolo e Testo sono obbligatori, viene specificato con un placeholder quali sono i caratteri accettati e non. Per aggiungere tags bisogna scrivere una singola parola o più unite da “_” e poi cliccare su “+”.

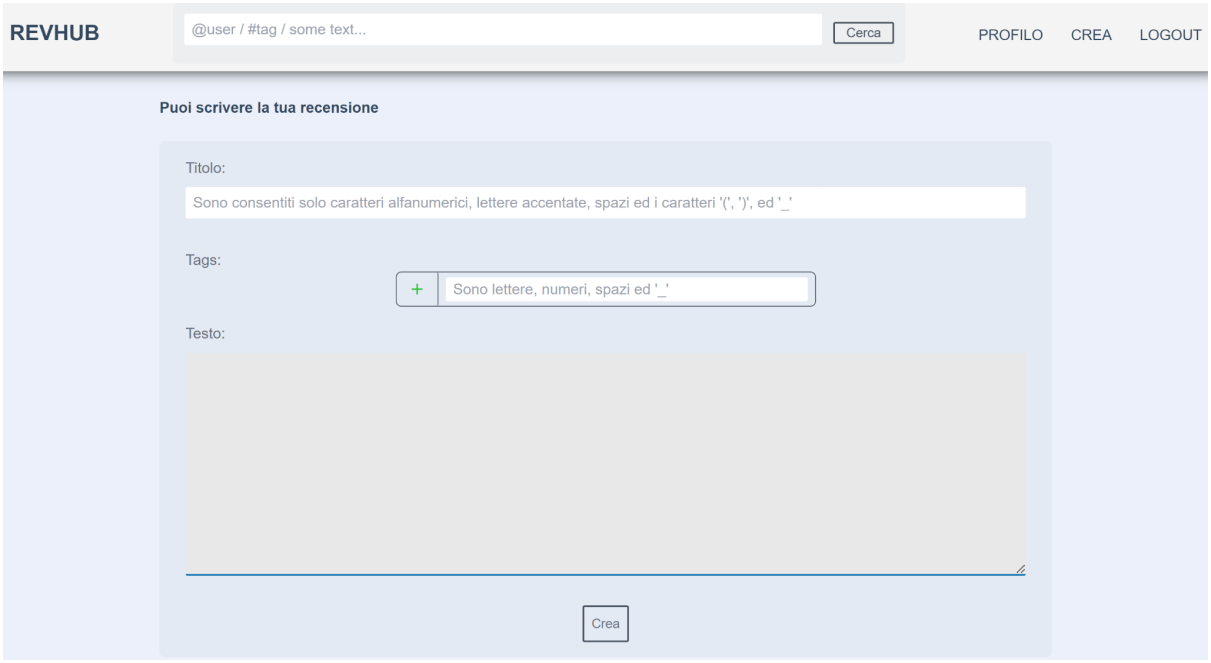


Figura 6.4 - schermata creazione di una recensione

Homepage (non autenticato)

Questa è la stessa schermata tranne per la barra di navigazione dove compare solo il logo, la barra di ricerca e il pulsante “Login” che porta alla pagina di ricerca, successivamente descritta.

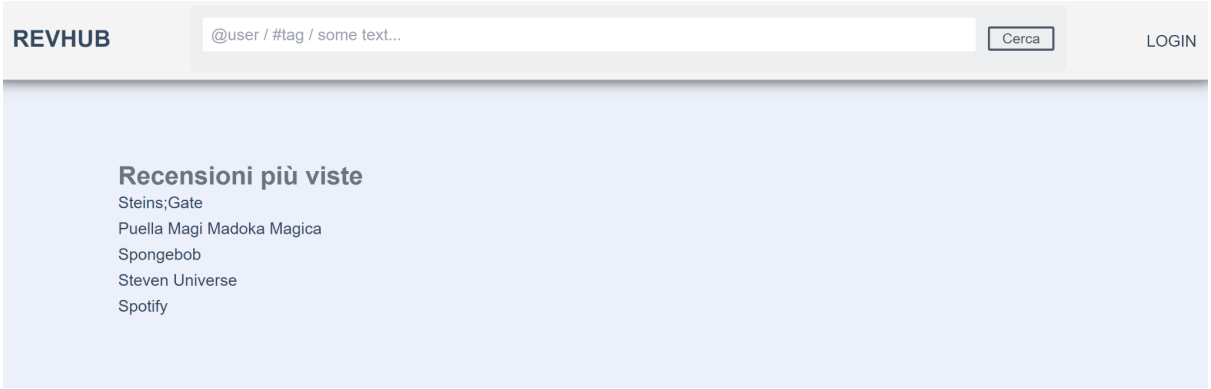


Figura 6.5 - schermata home di un utente non loggato

Pagina Login

In questa schermata è possibile effettuare il login con le credenziali già create nel form di registrazione che verrà spiegato in seguito. Se non si dovessero possedere le credenziali, si possono usare le credenziali di ateneo o Google da inserire durante la registrazione.

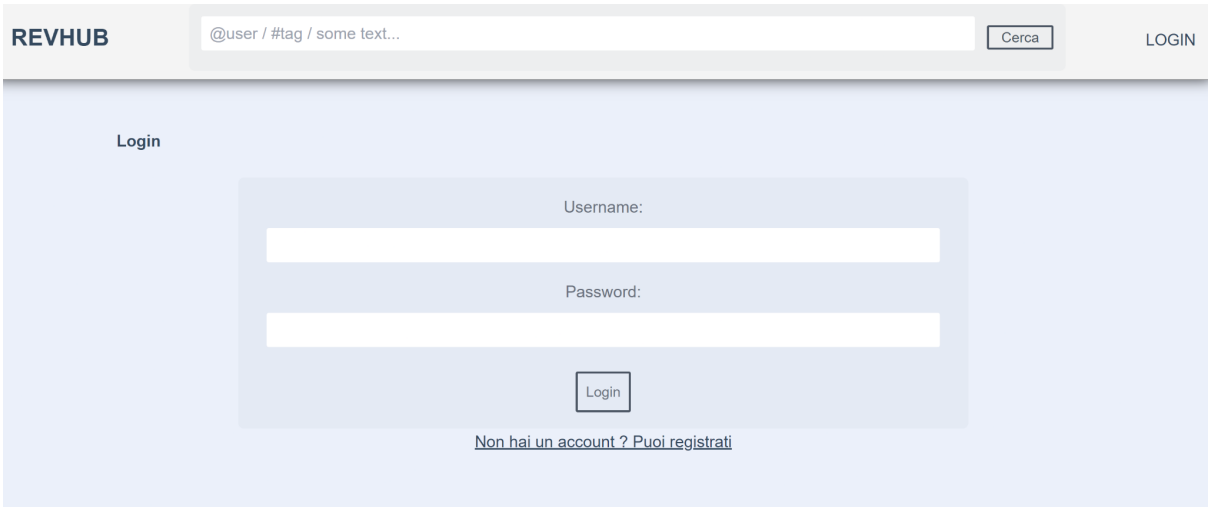


Figura 6.6 - schermata di login

Pagina di Registrazione

Questa è la schermata per gli utenti non autenticati che non possiedono credenziali RevHub o per chi volesse fare un altro profilo. Per compilarla servono credenziali di Ateneo o di Google proprie, quindi si seleziona su “Organizzazione” quale tipologia di credenziali si userà, poi si inserisce la email associata alla tipologia di credenziali terze che si è deciso di utilizzare e la password (che sarà la stessa associata all’utente, dato che si simula l’uso delle credenziali esterne) e infine un username unico.

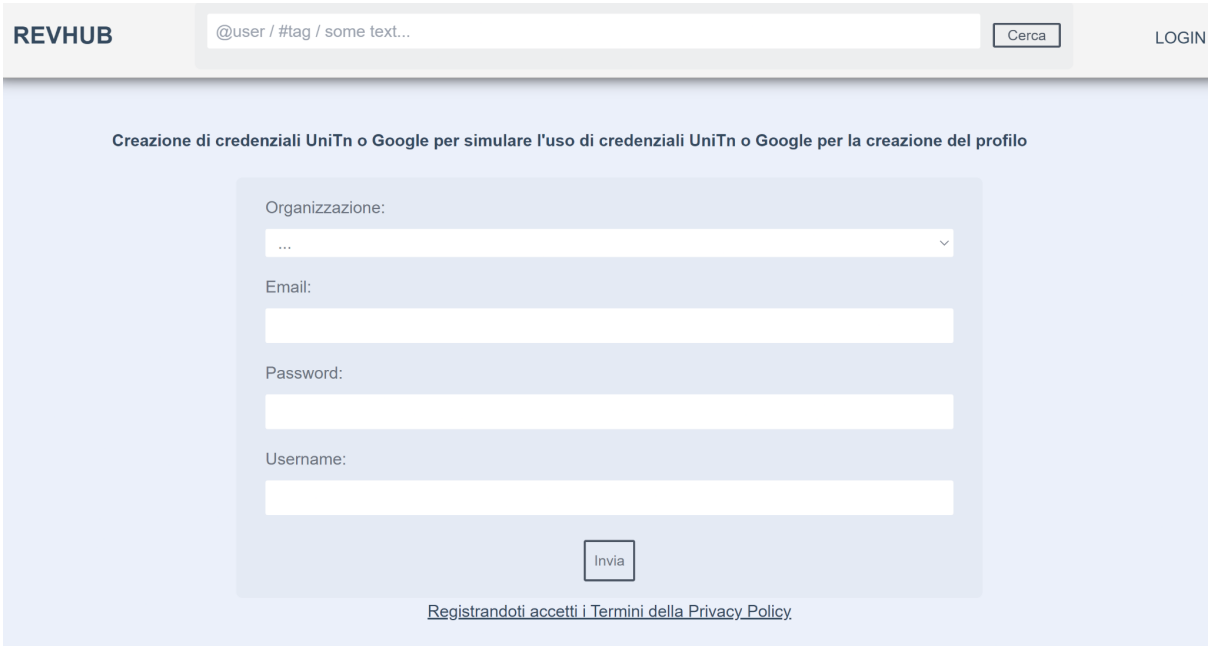


Figura 6.7 - schermata di creazione account

Pagina Ricerca

Per concludere è presente la pagina di ricerca a cui si può accedere esclusivamente dopo aver scritto qualcosa nel campo di ricerca della barra di ricerca presente nel menù e aver premuto “Cerca”. Si possono cercare utenti, tramite il loro username, e recensioni, tramite i loro tags associati o tramite titolo:

- per cercare un utente si può digitare tutto il suo username o anche una sua parte tramite il formato “@” + <username>;
- per cercare una recensione tramite tags bisogna scrivere nel campo testo della barra di ricerca “#” + <tag>;
- infine per cercare tramite titolo non serve alcuna formattazione, ma solo il <titolo> da cercare (come nell'immagine d’esempio qui di seguito).

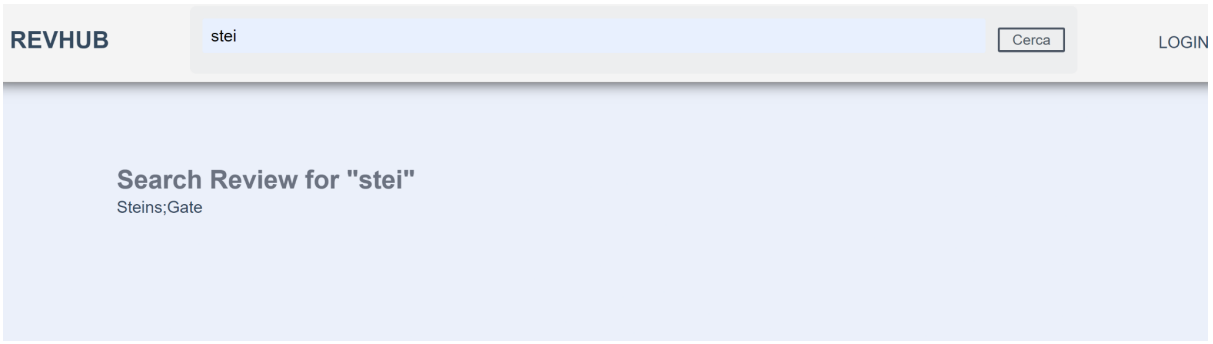


Figura 6.8 - schermata di visualizzazione dei risultati di una ricerca

7. Repository & Deployment

Nel seguente link è presente tutto il progetto sotto forma di codice open source: https://github.com/G13-RevHub/RevHub.

Per eseguire il codice in locale bisogna seguire i passaggi seguenti, descritti anche nel file README.md:

- clonare il progetto tramite il link https://github.com/G13-RevHub/RevHub.git
- entrare nella cartella *RevHub* > *rev_hub*, da cui può essere eseguita l'applicazione.
- installare tutte le dipendenze dell'applicazione tramite il comando *npm install*
- a questo punto è possibile eseguire l'applicazione, scegliendo tra due modalità:
  - in modalità developement tramite il comando ***npm run dev***, usata normalmente durante lo sviluppo in quanto consente di utilizzare la funzionalità Fast Reload per aggiornare in tempo reale il risultato ogni volta che il codice viene modificato e salvato. È anche possibile utilizzare il comando ***npm run dev —turbo*** per un'esecuzione più rapida, tenendo a mente che tale funzione richiede un uso massiccio di RAM (fino a 2 GB circa).
  - oppure creando prima una build col comando ***npm run build*** e poi lanciando quest'ultima col comando ***npm run start***. Scegliere quest'opzione richiederà un tempo maggiore, in quanto tutto il codice dovrà essere compilato (mentre invece in modalità development vengono compilati solo i file di cui è richiesto l'uso durante l'utilizzo dell'applicazione), ma una volta lanciata la build l'applicazione risulterà più veloce e richiederà meno risorse di sistema per essere eseguita.
- Quindi apri il seguente indirizzo http://localhost:3000 nel suo browser (pagina accessibile anche cliccando sull'url che appare nel terminale).

La preghiamo di non utilizzare la connessione di rete di Ateneo durante l'esecuzione del codice, perché è presente un firmware che non ci consente di eseguire la connessione al database di MongoDB, di conseguenza falliranno tutte le chiamate ad api che interagiscono col database.
I parametri per la connessione al database e per la creazione del token di sessione sono contenuti nel file **.env**, anch'esso interno alla repository.

Per ovvi motivi l'applicazione non può davvero interfacciarsi coi sistemi di credenziali UniTn e Google, per cui abbiamo creato delle credenziali fittizie che possono essere usate in fase di registrazione per creare un nuovo profilo utente.
Se vuole testare la registrazione di un utente può utilizzare le seguenti credenziali:

- organizzazione: *UniTn* email: *marco@unitn.it* password: *pass123*
- organizzazione: *Google* email: *mario@gmail.com* password: *mario99*
- organizzazione: *Google* email: *luigi@gmail.com* password: *luigi66*

oppure può creare delle nuove credenziali fittizie alla pagina nascosta /hidden/createCredential.
In alternativa può anche fare direttamente il login con uno degli utenti già esistenti:

- username: *SuperGuido* password: *gui2024*
- username: *Mari4* password: *m4ri4*

8. Testing

Per effettuare il testing è stata usata la libreria Jest, come citato nella struttura del codice.
Naturalmente è stata anche svolta una fase di testing preliminare durante lo sviluppo del codice, basata semplicemente sul verificare che l'applicazione rispondesse ai nostri input nel modo previsto. In particolare, la maggior parte del codice e delle funzionalità dell'applicazione non richiedevano di eseguire particolari elaborazioni sui dati, ma piuttosto di interagire in un certo modo con il database (ossia operazioni a carico di moduli e funzioni forniti di pacchetti importati nell'applicazione) o di permettere all'utente di interfacciarsi in un certo modo con l'applicazione.

Il testing con Jest prevede la creazione di file di tipo *.test.js* che contengono i test da eseguire.
Ci sono due tipi di test:

- il test di funzioni e moduli da importare da altri file, che producono un risultato dettagliato sulla correttezza dei risultati e sulla copertura del codice testato.
- il test dei risultati delle chiamate alle API, che può solo verificare la correttezza delle risposte delle API a fronte degli input che gli abbiamo passato (se erano richiesti).

Esempi di Testing

Un esempio del primo tipo di test potrebbe essere il seguente.
Potremmo voler testare la funzione *sum* del seguente file ***sum.js***.

```
function sum(a, b) {
    return a + b;
}

export default sum;
```

Per fare ciò dovremo creare un file che gestisca il test relativo a tale funzione: lo possiamo chiamare ***sum.test.js*** e conterrà il codice seguente.

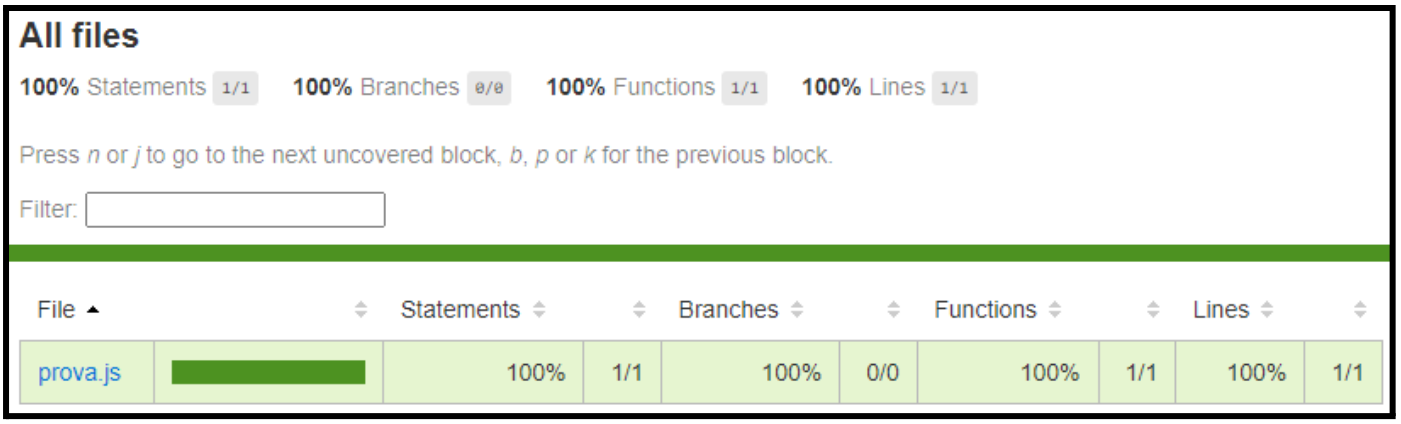
```
import sum from './sum.js';

describe("Questo blocco testa la funzione somma", () => {
    test('1 + 4 = 5', () => {
        expect(sum(1, 4)).toBe(5);
    })
    test('8 + 3 = 11', () => {
```

```
        expect(sum(8, 3)).toBe(11);
    })
    test('7 + (-2) = 5', () => {
        expect(sum(7, -2)).toBe(5);
    })
})
```

La struttura del file è la seguente:

1. per prima cosa vengono importate le funzioni da testare, in questo caso la funzione ***sum***;
2. poi si dichiara col metodo ***describe*** un blocco che conterrà i test. L'uso di tale metodo per incapsulare i test è opzionale, ma è comodo per organizzare meglio i nostri test. Il primo parametro è una stringa da usare per descrivere i test al suo interno, mentre il secondo parametro è una funzione che conterrà i test.
3. tramite la funzione ***test*** si eseguono i test veri e propri. Come ***describe***, anche ***test*** ha un primo parametro di tipo stringa da usare per descrivere il singolo test che verrà eseguito nella funzione passata come secondo parametro. Dentro questa funzione, tra il resto del codice, va inserita una chiamata alla funzione ***expect(****input****).toBe(****output****)***, dove *input* sarà un valore di cui testare la correttezza (nell'esempio sopra è l'esito della funzione) mentre *output* è l'esito atteso.

Andando ad eseguire il test con questi due file l'output che si otterrà sarà il seguente:



Per quanto riguarda invece il testing delle api invece, non vi è alcun file da cui importare delle funzioni da eseguire. L'unico file è quello di test, il quale dovrà eseguire una chiamata all'API da testare e, una volta ottenuta la risposta, verificarne la correttezza in modo simile a come viene fatto per il primo caso.
Qui di seguito si può vedere una versione ridotta del file di testing usato per testare le api riguardanti le review:

```
const api_review_url = "http://localhost:3000/api/review"

describe("This block tests all the review APIs", () => {
    it("create review fail", async () => {
        expect.assertions(1)
        var response = await fetch(api_review_url + "/create", {
            method: 'POST', body: JSON.stringify({
                title: "recensione bella",
                tags: [],
                text: "testo della recensione bella"
            }),
            headers: { 'Content-Type': 'application/json' }
        })
        expect(response.status).toEqual(400)
    })

    it("get review", async () => {
        expect.assertions(1)
        var review_id = "1"
        var response = await fetch(api_review_url + "/get/" + review_id,
{ method: 'GET' })
        expect(response.status).toEqual(200)
    })
    it("rate review fail", async () => {
        expect.assertions(1)
        var response = await fetch(api_review_url + "/rate", {
            method: 'PUT', body: JSON.stringify({
                author_id: 1,
                review_id: 1,
                rate: true
            }),
            headers: { 'Content-Type': 'application/json' }
        })
        expect(response.status).toEqual(400)
    })
})
```

È possibile notare qualche differenza rispetto al testing di moduli importati, ma il principio è lo stesso:

1. all'interno della funzione ***describe*** (sempre opzionale) si inseriscono i test tramite la funzione ***it*** (analoga alla precedente funzione ***test***) alla quale viene passata una funzione asincrona, in quanto deve essere effettuata una chiamata ad un'API;

2. va poi definito il numero di asserzioni che si intendono eseguire all'interno di quel test tramite la funzione ***expect.assertions(n)***;
3. si esegue poi la chiamata all'API;
4. infine si valuta la correttezza del risultato ottenuto, usando il metodo ***.toEqual()*** (invece che il metodo ***.toBe()*** usato prima).

Sfortunatamente, data la natura di questi test, non è possibile ottenere un'analisi sui test dettagliata quanto la precedente, ma solo l'esito dei singoli testi e il tempo richiesto per eseguirli, come è possibile vedere nella sezione successiva.

Risultati e Considerazioni sui Test svolti

La nostra applicazione non possedeva funzioni complicate o logiche di qualche tipo, ma si basava su una massiccia esecuzione di interazione col database ed una corretta rappresentazione dei risultati da mostrare all'utente. Per tale motivo in questa fase di testing con Jest non abbiamo mai testato singole funzioni da file (poiché non ne avevamo) o la correttezza delle risposte delle API (verifica che è stata invece svolta durante il debugging avvenuto con durante la scrittura del codice), ma piuttosto dei test minori sul fatto che restituissero risposte non vuote e con esito positivo (codice di stato 200) o negativo (codice di stato 400).
Il risultato di tali test è il seguente:

```
PASS ./api-search.test.js
  This block tests all the search APIs
    √ search per tag (243 ms)
    √ search per title (313 ms)
    √ search per user (141 ms)

PASS ./api-auth.test.js
  This block tests all the auth APIs
    √ current user fail (162 ms)
    √ login (346 ms)
    √ login fail (121 ms)
    √ logout (37 ms)
    √ register fail (103 ms)
```

```
PASS ./api-review.test.js
  This block tests all the review APIs
    √ create review fail (197 ms)
    √ get review (406 ms)
    √ get review fail (33 ms)
    √ get reviews per views (166 ms)
    √ rate review fail (35 ms)

PASS ./api-user.test.js
  This block tests all the user APIs
    √ get self user fail (168 ms)
    √ get user (294 ms)
    √ get user fail (49 ms)
    √ get all usernames (156 ms)
    √ get user reviews (177 ms)
```

```
Test Suites: 4 passed, 4 total
Tests:       18 passed, 18 total
Snapshots:   0 total
Time:        2.137 s, estimated 6 s
Ran all test suites.
```

È possibile analizzare ed eseguire i test utilizzati per ottenere tale output recandosi alla cartella *RevHub* > *testing* ed eseguendo il comando ***npm test***.

Va aggiunto che per alcune API non è corretto testarne la correttezza tramite questo tipo di approccio, infatti tutte le api che hanno come scopo la creazione di nuovi record all'interno del database richiederebbero un input diverso e possibilmente sensato ad ogni asserzione del test per poter preservare la correttezza dei dati nel database, condizione che andrebbe a vanificare completamente un test di questo tipo, oltre al fatto che non hanno un output tale da verificare l'effettiva creazione dell'oggetto nel database. Per tale motivo per le api destinate alla creazione di nuovi record è stato creato un test che verifica solo per la restituzione di errori (sotto le condizioni previste).

Considerazioni finali sul Testing

Questa fase del lavoro ci ha permesso di scovare alcune falle nelle nostre API. In particolare (contro ogni aspettativa) i test che si sono rivelati più utili sono stati quelli che avevano come obiettivo il testare le risposte ottenute in caso venissero inviati all'API dati non corretti o incompatibili col tipo di input richiesto. Spesso infatti ci siamo accorti come non venisse restituito l'errore che ci aspettavamo, o addirittura veniva restituita una risposta positiva, senza che si fosse verificato alcun tipo di errore all'interno della logica dell'API.
Probabilmente tali mancanze sono dovute al fatto che durante lo sviluppo si ci concentra maggiormente su come il nostro codice risponde con un tipo di input corretto e prevedibile e meno di come si comporti sotto condizioni palesemente errate e con input inverosimili.