

D4: Sviluppo Applicazione

G06

June 3, 2024

Contents

1 Scopo del documento	3
2 User Flows	3
2.1 User Flow Utente	4
2.2 User Flow Admin	5
3 Application implementation and documentation	5
3.1 Struttura	5
3.2 Dependencies del progetto	7
3.3 Modelli nel database	8
3.4 API del progetto	11
3.4.1 Estrazione delle risorse	11
3.4.2 Diagramma delle risorse	13
3.5 Sviluppo delle API	16
3.5.1 API del modello utente	16
3.5.2 API del modello Books	19
3.5.3 API del modello admin	20
3.5.4 Metodi del modello authentication	20
3.5.5 API del modello disponibilità	21
4 API documentation	22
5 Frontend	24
5.1 Homepage	25
5.2 Login	26
5.3 Registrazione	27
5.4 Archivio	28
5.5 Ricerca	29
5.6 Servizi	30
5.7 Contatti	31
5.8 Impostazioni	31
5.9 Appuntamento	32
5.10 Frontend Admin	33

5.10.1	Donazioni Admin	33
6	Testing	36
6.1	Risultati del test	38
7	GitHub repository and Deployment	41
7.1	Deployment	41
7.1.1	Eseguire in locale	41

1 Scopo del documento

In questo documento vengono descritte tutte le informazioni di sviluppo dell'applicazione **EasyLib**.

Nello specifico vengono trattate tutte le parti di login, gestione, e noleggi alla nostra biblioteca digitale, iniziando dalle descrizioni degli User Flows, sia per l'utente normale che per l'amministratore; per poi passare alla struttura del codice scritto, con la descrizione di tutte le varie API sviluppate. Ogni API creata verrà analizzata nelle sue funzioni, avendo la sua documentazione, e poi, nella sezione testing avrà i risultati dei test a cui è stata sottoposta. Per concludere è presente una sezione che descrive il deployment dell'applicazione e tutte le informazioni del Git Repository.

2 User Flows

Lo User flow rappresenta come l'utente interagisce con il design del frontend, e le azioni che può compiere spostandosi tra le varie pagine dell'applicazione. Per prima cosa introduciamo la legenda del diagramma che ci permette di rappresentare lo user flow.



Figure 1: Legenda User Flow

1

Informazioni sulla legenda:

- con **Bivio** si intende un punto in cui la risposta del sistema può variare a seconda della selezione dell'utente.
- per **Attesa** si intende un punto in cui il sistema rimette all'utente la decisione su cosa fare: se ripetere un'operazione appena compiuta o tornare indietro alla pagina precedente e effettuare altre azioni.
- per **Arrivo** si intende il momento in cui il sistema ha soddisfatto le richieste dell'utente. Nel nostro caso per l'utente il punto di 'Arrivo' corrisponde con la ricerca ed eventuale noleggio di un libro, mentre per l'admin corrisponde con la creazione di una multa, l'aggiunta di un nuovo libro al database o la conferma del reso di un libro da parte di un utente.

¹Per visualizzare le foto con una qualità maggiore si visiti il seguente link https://drive.google.com/drive/folders/1mCccEUUQHTccKQIxqfn6cw_Kf0J2SwSW?usp=sharing

Per rendere più chiaro l'intero diagramma e non saturarlo di collegamenti, abbiamo optato per creare 2 User flow diversi: il primo riguardante le funzionalità dell'**utente normale**; mentre il secondo rappresenta le azioni che possono essere svolte da un **utente Admin** in possesso delle dovute credenziali.

2.1 User Flow Utente

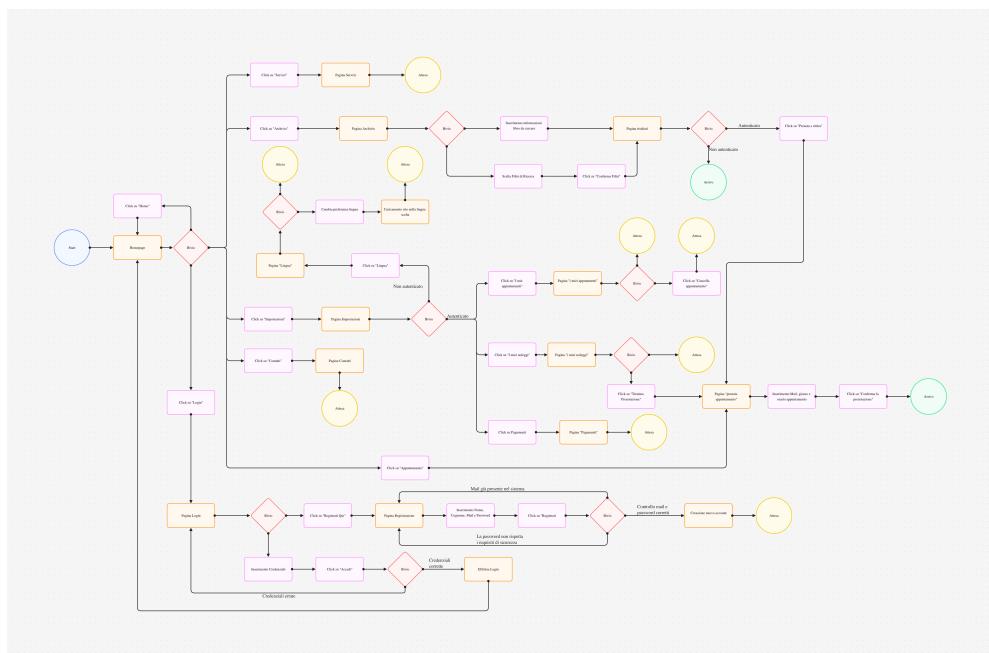


Figure 2: User Flow utente

2.2 User Flow Admin

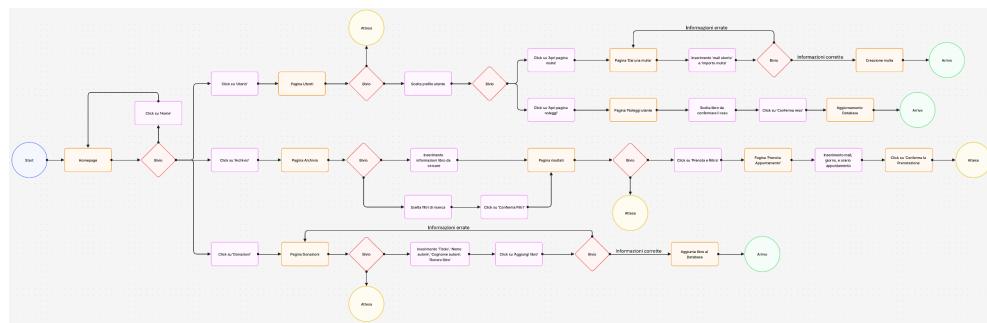


Figure 3: User flow Admin

3 Application implementation and documentation

Nei documenti precedenti vengono descritte tutte le funzionalità e i requisiti che la nostra applicazione EasyLib deve avere. In questa sezione invece verrà descritto come queste funzionalità sono state sviluppate tramite. Lo sviluppo dell'applicazione è stato creato usando HTML, CSS, Javascript e NodeJs; mentre la gestione dei database necessari all'uso dell'applicazione è stato usato MongoDB.

3.1 Struttura

La struttura del progetto è rappresentata nella seguente immagine

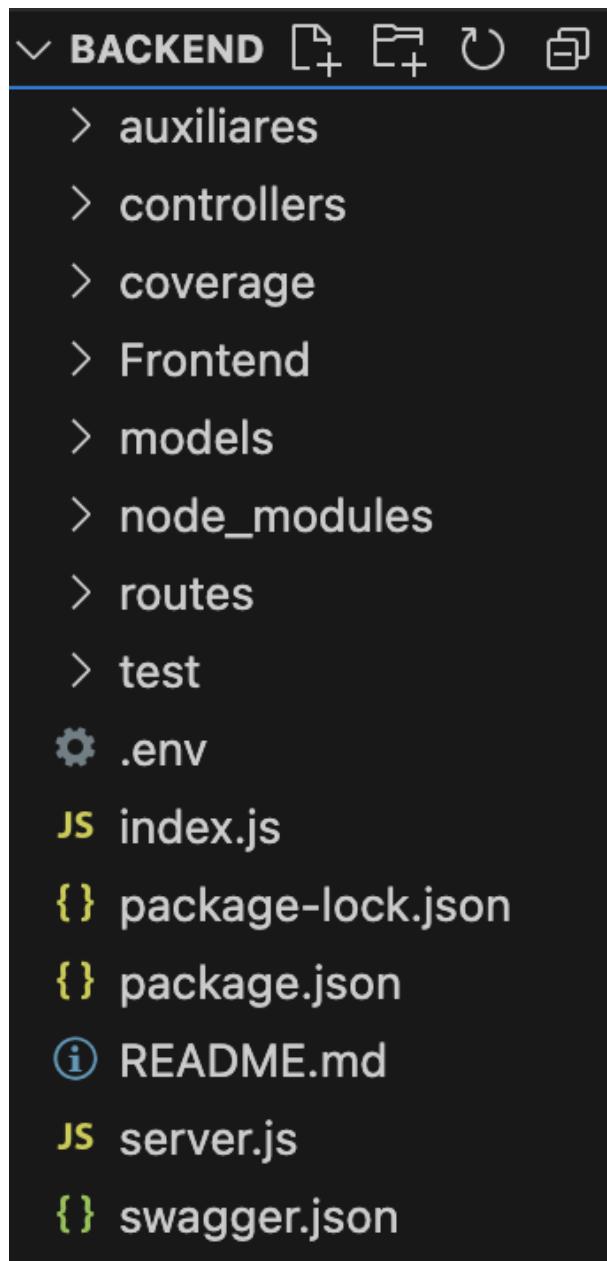


Figure 4: Struttura del progetto

Nella cartella **auxiliares** sono presenti delle funzioni per verificare che la password rispetti i requisiti (lunghezza maggiore o uguale a 8 caratteri, e almeno un carattere speciale) e che la stringa inserita nei campi mail sia effettivamente

una mail valida. La cartella **controllers** contiene le varie api sviluppate divise per i modelli utilizzati. Tutta la parte riguardante il FrontEnd, è contenuta nella cartella **Frontend** dove sono collocati tutti i file CSS e HTML. I modelli con i loro attributi descritti in seguito in questo documento sono contenuti nella cartella **models**; la cartella **Test** contiene i vari test svolti sulle varie API, e sono implementati tramite Jest. La cartella **routes** infine definisce il percorso e la tipologia delle varie API create. Oltre alle cartelle ci sono alcuni file :

- **Swagger.json** dove sono descritte con tanto di esempi le API sviluppate per il progetto.
- **index.js** è il programma che deve essere eseguito per instaurare la connessione con MongoDB e attivare il server all'indirizzo localhost.
- **Package.json** contiene il file di configurazione generale del progetto.
- **.env** E' il file in cui sono definite le variabili dell'ambiente.
- **server.js** che importa tutti i moduli utilizzati nel Back-end e definisce gli end-point delle API.

3.2 Dependencies del progetto

Qui sono descritti i moduli Node aggiunti a package.json nell'area dependencies, tra cui:

- cors: modulo per permettere alla web app di supportare il Cross-Origin Resource Sharing protocol
- dotenv: permette di utilizzare le variabile d'ambiente definite nel file .env
- express: framework che offre molte funzioni per la gestione di API per le web application.
- jsonwebtoken: modulo per creare e gestire un token d'accesso.
- mongoose: fornisce le varie funzioni per interagire con MongoDB.
- multer: gestisce il body nelle varie API.
- swagger-ui-express: tool usato per documentare e testare la API che abbiamo progettato.
- jest: modulo usato per il testing delle API e delle funzioni nel back-end.
- supertest: modulo usato per chiamare le API in fase di testing.

3.3 Modelli nel database

Per la gestione dei dati abbiamo creato 5 collezioni principali:

1. **Utente:** che contiene tutte le informazioni relative ai vari utenti registrati. I campi ID, Nome, cognome, mail e password che vengono inseriti alla prima registrazione.

Libri_noleggiati Dove sono presenti gli ID univoci dei vari libri noleggiati da un singolo utente.

E *n_libri* che serve per rispettare il RF8 che permette il noleggio di al più 5 libri.

Viene mostrato prima lo schema del modello creato per utente:

```
const mongoose = require("mongoose");

const schema = new mongoose.Schema({
  utente_id : {type : Number, required : true},
  nome : {type : String, required : true},
  cognome : {type : String, required : true},
  mail : {type : String, required : true},
  password : {type : String, required : true},
  libri_noleggiati : {type : Array, default: null},
  n_libri : {type : Number, default: null},
});

const Utente = mongoose.model("Utente", schema);
module.exports = Utente;
```

Figure 5: Modello utente

E successivamente vengono riportati degli esempi delle collezioni utente nel database, di un utente registrato e uno in possesso di credenziali admin per poter provare entrambe le interfacce.

```
_id: ObjectId('664497930f610139804a6b42')
utente_id: 778
nome: "Francesco"
cognome: "Rossi"
mail: "Fra.rossi@gmail.com"
password: "ciao123!"
libri_noleggiati: Array (1)
  n_libri: 1
__v: 0
```

Figure 6: Tipo di dato: Utente

```
_id: ObjectId('6648ac396ef2e2alc9a511f9')
utente_id: 780
nome: "admin"
cognome: "admin"
mail: "admin@easylib.com"
password: "admin123!"
libri_noleggiati: Array (empty)
n_libri: null
autenticato: null
--v: 0
```

Figure 7: Tipo di dato: Utente Admin

2. **Appuntamenti:** in cui sono salvate le informazioni degli appuntamenti effettuati dagli utenti. Gli attributi necessari sono la mail dell'utente che deve essere registrato, la *Data* richiesta per l'appuntamento e il *tipo_app* che descrive la tipologia dell'appuntamento desiderata e può essere un ritiro, un reso o una donazione. Sotto viene mostrato il modello creato.

```
const mongoose = require("mongoose");
const schema = new mongoose.Schema({
  mail: {type: String, required: true},
  data: {type: Date, required: true},
  tipo_app: {type: String, required: true},
});

const Appuntamento = mongoose.model("Appuntamento", schema);
module.exports = Appuntamento;
```

Figure 8: Modello Appuntamento

E successivamente un esempio del tipo di dato Appuntamento presente all'interno del database:

```
_id: ObjectId('66472bb31d4ac16d27faf602')
mail: "Fra.rossi@gmail.com"
data: 2024-05-24T10:30:00.000+00:00
tipo_app: "Restituire un libro"
--v: 0
```

Figure 9: Tipo di dato: Appuntamento

3. **Libri:** dove vengono salvati tutti i vari volumi presenti su **EasyLib**. Gli attributi del modello sono le caratteristiche di ogni volume, tra cui il titolo, il nome e cognome dell'autore e il genere, necessari per la ricerca nell'archivio e filtri di ricerca. Un ID del libro creato automaticamente e sequenziale aggiunto dal sistema all'inserimento di ogni nuovo libro.

Inoltre è presente una *Scadenza* del noleggio, che è settata a NULL se il libro non è in noleggio; infine *Is_available* serve per gestire la disponibilità del libro in caso di ritiro.

Qui sotto viene riportato il modello Book creato:

```
const mongoose = require ("mongoose");

const schema = new mongoose.Schema({
  book_id : {type : Number, required : true},
  titolo : {type : String, required : true},
  Author_name : {type : String, required : true},
  Author_sur : {type : String, required : true},
  Genre : {type : String, required : true},
  Is_available : {type : Boolean, default: true},
  scadenza : {type : Date, default: null}
})

const Libro = mongoose.model("Libro", schema);
module.exports = Libro;
```

Figure 10: Modello Book

E di seguito è rappresentata un immagine di esempio del tipo di dato Book all'interno del database:

```
_id: ObjectId('661d05a6a3e56be383b94b29')
book_id: 22
titolo: "Il mastino dei Baskerville"
Author_name: "Arthur"
Author_sur: "Conan Doyle"
Genre: "Giallo"
Is_available: true
scadenza: null
__v: 0
```

Figure 11: Tipo di dato: Libro

4. **Multa:** in cui vengono riportate le informazioni riguardanti le varie contravvenzioni. Gli attributi del modello sono: la mail a cui è legata la multa, l'*importo* della multa e la data entro cui bisogna pagare la multa, contenuta in *paga entro*.

Prima è raffigurata un immagine del modello Multa creato:

```

const mongoose = require ("mongoose");

const schema = new mongoose.Schema({
  mail: {type: String, required : true},
  importo: {type: Number, required: true},
  paga_entro: {type: Date, required: true},
})

const Multa = mongoose.model("Multa", schema);
module.exports = Multa;

```

Figure 12: Modello Multa

E poi un esempio del tipo di dato multa all'interno del database:

```

_id: ObjectId('6647460903b978ea5726107d')
mail: "Fra.rossi@gmail.com"
importo: 35
paga_entro: 2024-12-11T23:00:00.000+00:00

```

Figure 13: Tipo di dato: Multa

3.4 API del progetto

In questa sottosezione del documento vengono descritte le varie API implementate a partire dal diagramma delle classi mostrato nel documento D3-G06. Useremo un diagramma per rappresentare l'estrazione delle risorse a partire dal class diagram e uno per rappresentare le risorse sviluppate.

3.4.1 Estrazione delle risorse

Questi diagrammi presentano come abbiamo estratto le risorse partendo dal class diagram.

Abbiamo diviso le risorse per la parte utente e quella dell'admin, individuando dal diagramma delle classi dei modelli, i cui attributi sono necessari per la memorizzazione nel database.

Abbiamo convertito i metodi delle classi estratte in API, che pensavamo definissero meglio lo scheletro del progetto. Di queste, nei diagrammi sottostanti viene definito il tipo di API (GET, POST, PATCH, DELETE) e con quali attributi interagiscono.

Inoltre abbiamo etichettato ogni API in base alla loro interazione con Frontend e Backend; nel caso delle API di tipo GET l'effetto più immediato sarà quello sul Frontend, le restanti POST, PATCH e DELETE invece, avranno un interazione più profonda sul Backend in quanto andaranno a modificare, creare e cancellare risorse nel database.

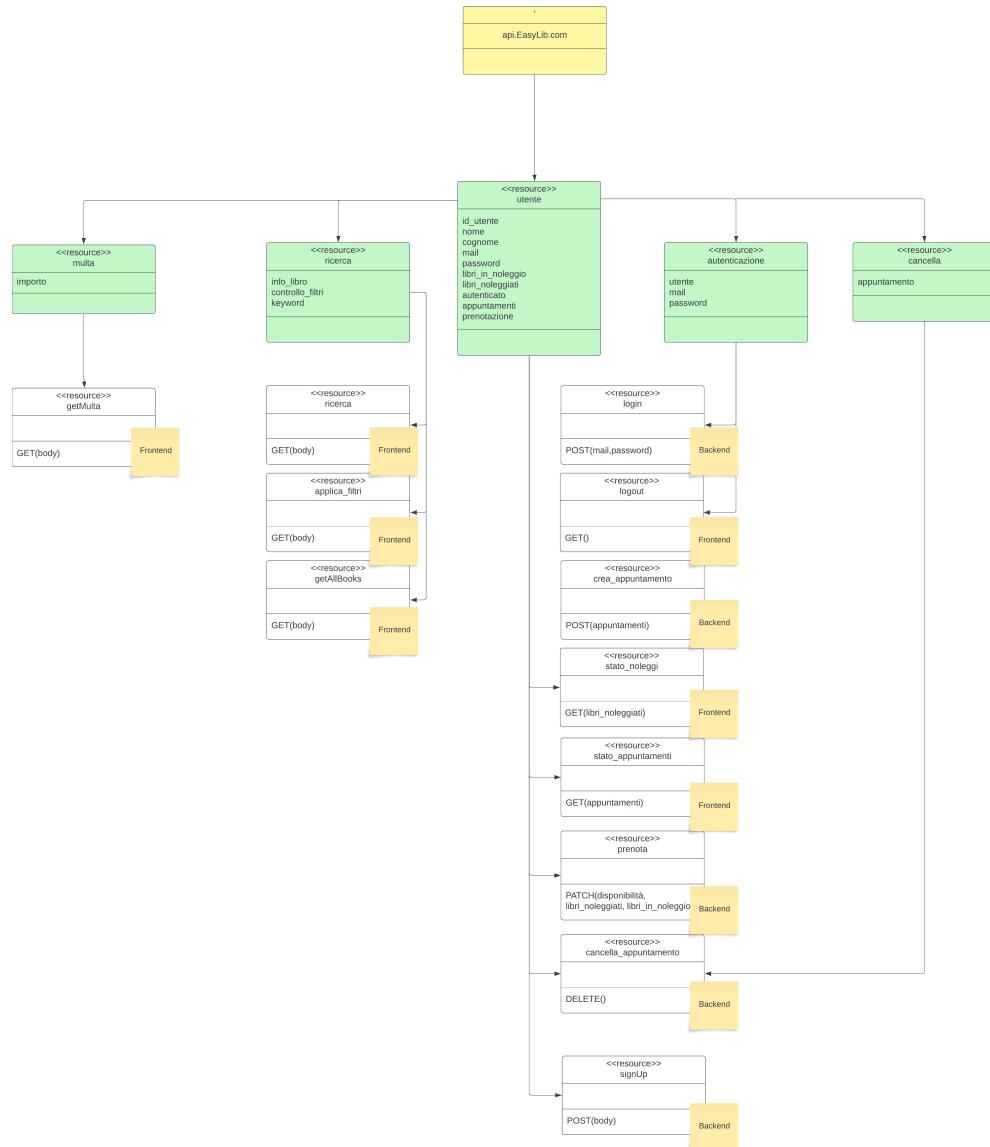


Figure 14: Estrazione risorse relative all'utente

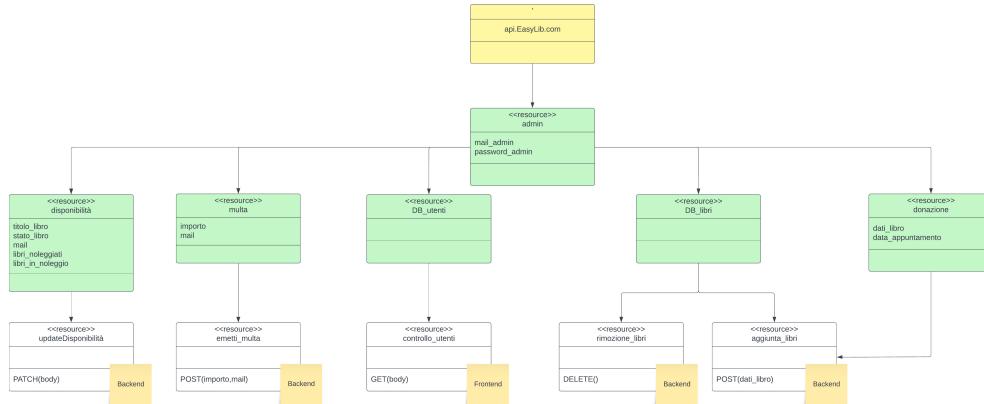


Figure 15: Estrazione risorse relative all'Admin

3.4.2 Diagramma delle risorse

Nel seguenti diagramma abbiamo rappresentato in maniera più specifica le API sviluppate nel progetto. Il diagramma è stato diviso, per permettere più chiarezza e non creare un'immagine troppo compatta, in una parte, come nell'estrazione delle risorse appena descritto, in sezione admin e sezione utente; la seconda parte invece descrive le API a partire dal modello appuntamento e dal modello book.

Si è specificato per ogni interfaccia un input e un output possibile, in base alla correttezza o meno dell'esecuzione dell'API.

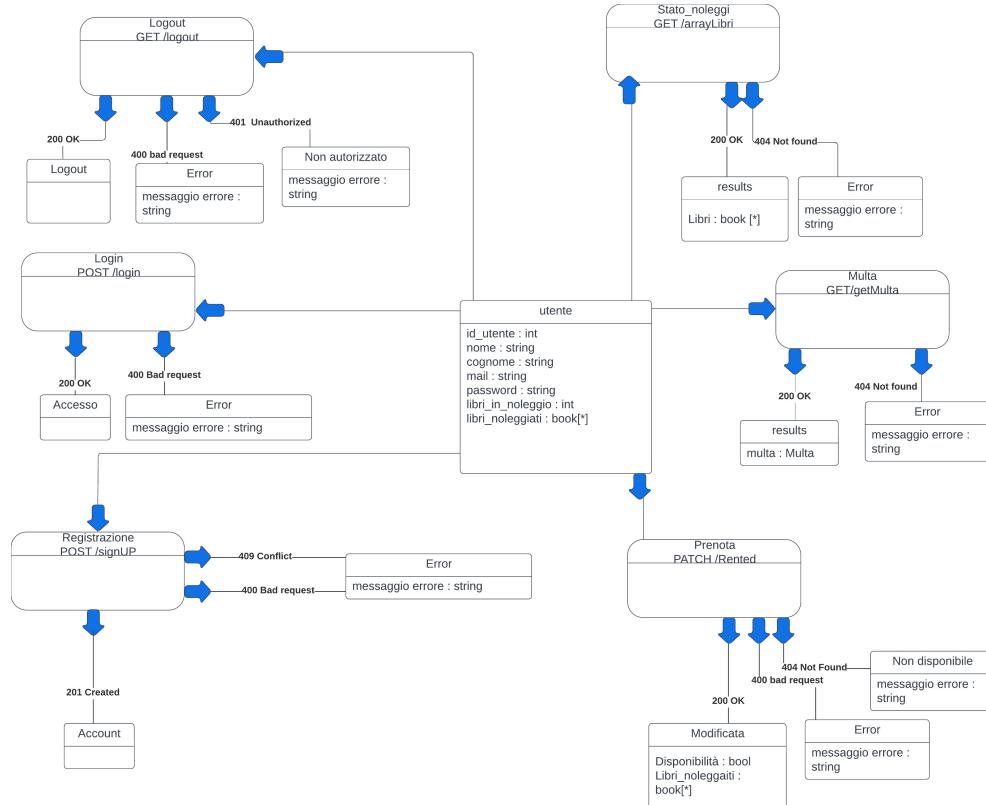


Figure 16: Diagramma delle risorse: Sezione utente

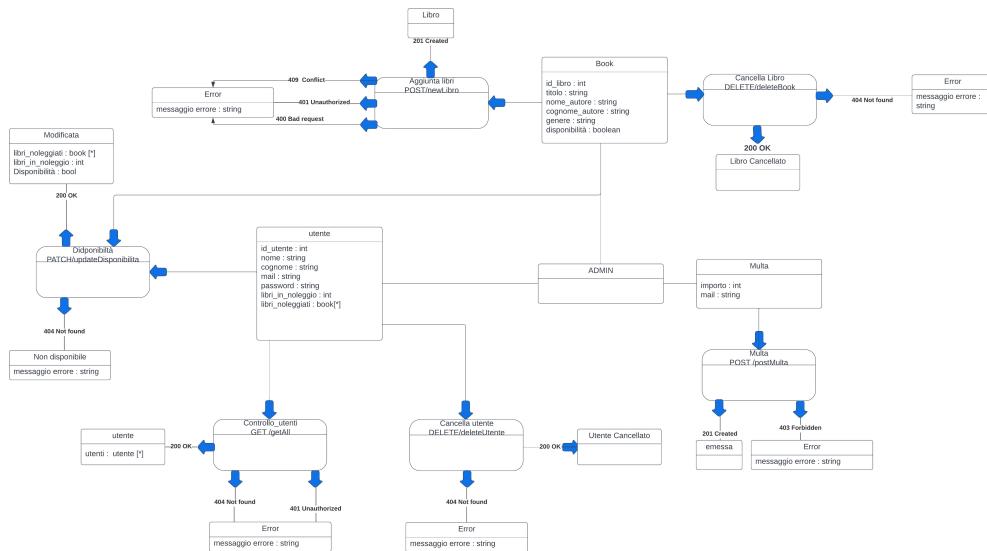


Figure 17: Diagramma delle risorse: Sezione Admin

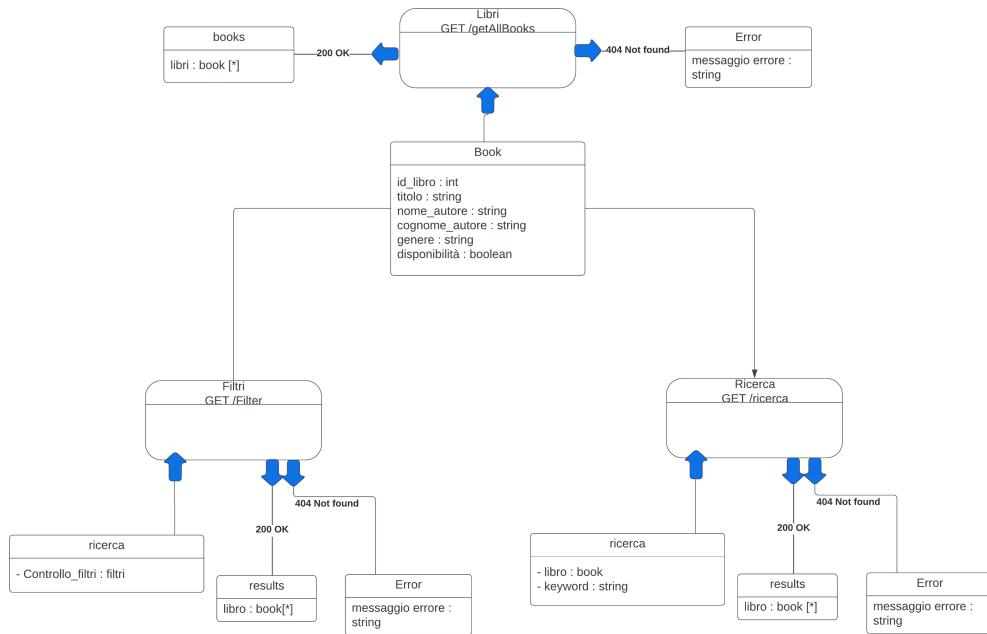


Figure 18: Diagramma delle risorse: parte Book

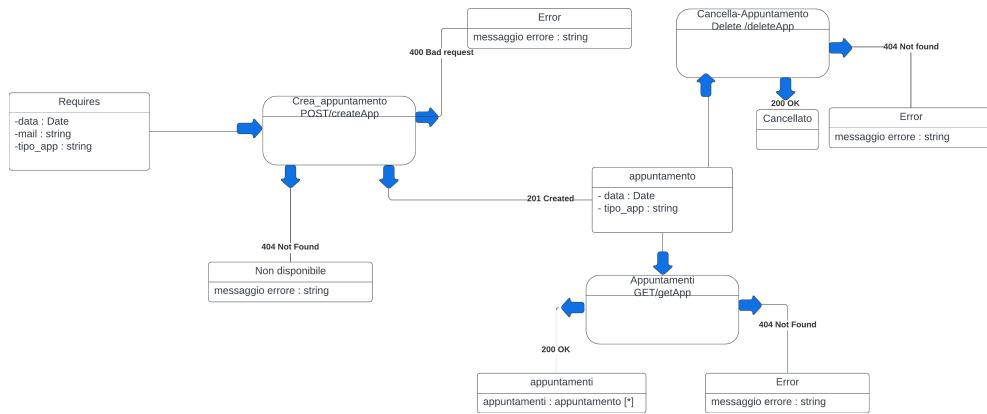


Figure 19: Diagramma delle risorse: parte appuntamento

3.5 Sviluppo delle API

In questa parte verranno descritti i funzionamenti di tutte le API implementate del progetto. Viene riportato con delle immagini solo un esempio di API per ogni tipologia (GET, POST, PATCH e DELETE) perché il codice è molto lungo e saturerebbe il documento di immagini. In ogni caso il codice corrispondente a tutte le altre interfacce può essere trovato nella repository del Backend, nella cartella **controllers**, divise in più file.

3.5.1 API del modello utente

Sotto descritte le API relative alle varie azioni possibili sulla risorsa Utente.

SignUp

L'API SignUp viene utilizzata per creare un nuovo utente, riceve in input un body con i dati del nuovo utente da registrare: nome cognome, mail, password, tutti necessari. Restituisce in output l'utente creato se non ci sono stati errori, oppure vari messaggi di errore, ad esempio "mail o password non valida" "mail già in uso". Si utilizza il metodo `findOne()` per verificare l'unicità della mail, il modello ausiliare Counter per assegnare un ID univoco per ogni utente e il metodo `save()` per salvare la risorsa nel database.

Creazione di un appuntamento

L'API `createApp` viene utilizzata per creare un nuovo appuntamento, riceve in input un body con i dati necessari per creare l'appuntamento: mail dell'utente, data dell'appuntamento, e il tipo di appuntamento(donazione, ritiro e restituzione) tutti necessari. Restituisce in output l'appuntamento creato se non ci sono stati errori, oppure vari messaggi di errore. Si utilizza il metodo `findOne()` per trovare l'utente che desidera creare un nuovo appuntamento e il metodo

save() per salvare la risorsa nel database. Può essere usata da utenti che hanno effettuato il login.

```
const createApp = async(req, res) => {
    let data_u = await utente.findOne ({mail: req.body.mail}).exec()
    if (!data_u){
        return res.status(404).json({success : false, message:"Utente non trovato"});
    }
    const newAppuntamento = new appuntamento({
        mail: data_u.mail,
        data: req.body.data,
        tipo_app: req.body.tipo_app
    })

    if(!compareDates(req.body.data)){
        return res.status(400).json({ success: false, message: "Data non valida" });
    }

    newAppuntamento.save((err,data)=>{
        if (err){
            return res.status(500).json({success: false, message:"Errore del server"})
        } else {
            return res.status(200).json({success: true, message: "Appuntamento creato", data})
        }
    })
}
```

Figure 20: createApp: API POST

Cancellazione di un appuntamento

L'API deleteApp viene utilizzata per cancellare un appuntamento di cui un utente non vuole usufruire. Riceve in input la mail dell'utente e ritorna in output un messaggio di cancellazione avvenuta con successo o un errore vario. Si utilizza il metodo findOne() per trovare l'utente che desidera cancellare un appuntamento. Può essere usata da utenti che hanno effettuato il login.

Trova i Libri noleggiati da un utente

L'API getBooks viene utilizzata per vedere i libri noleggiati da un utente, viene preso in input la mail dell'utente e ritorna l'elenco dei libri, sotto forma di array, in possesso di un utente se non ha problemi; mentre ritorna errori vari se l'input non è corretto. Si utilizza il metodo findOne() per trovare la mail dell'utente di cui ritornare la lista dei libri. I libri vengono cercati dall'API tramite l'id del libro, che poi viene utilizzato per stampare le informazioni dei volumi (titolo, nome e cognome dell'autore, genere e scadenza del noleggio). Può essere usata da utenti che hanno effettuato il login.

```

const getBooks = async (req, res) => {
    try {
        let data = await utente.findOne({ mail: req.query.mail }).exec();

        if (!data) {
            return res.status(404).json({ success: false, message: "Utente non trovato" });
        }

        var libri = [];
        var l = {
            libri_noleggiati: data.libri_noleggiati
        };
        for (let i = 0; i < l.libri_noleggiati.length; i++) {
            let element = l.libri_noleggiati[i];
            let faa = await libro.findOne({ 'book_id': element }).exec();

            var p = {
                book_id: faa.book_id,
                titolo: faa.titolo,
                Author_name: faa.Author_name,
                Author_sur: faa.Author_sur,
                Genre: faa.Genre,
                scadenza: faa.scadenza
            };

            libri.push(p);
        }

        return res.status(200).json({libri });
    } catch (error) {
        return res.status(500).json({ success: false, message: "Errore del server" });
    }
}

```

Figure 21: getBooks: API GET

Noleggiare un libro

L'API RentedBooks serve per poter noleggiare un libro, e di conseguenza inserire il libro all'interno dei libri noleggiati dall'utente, settare la scadenza del noleggio e modificare la disponibilità del libro a FALSE. Viene usato il metodo updateOne() per modificare entrambe le collezioni (utenti e libri); e findOne() per trovare la mail dell'utente che sta noleggiando e il titolo del libro noleggiato. Per settare la data di scadenza del noleggio viene usato il metodo date(). L'output ritorna un messaggio di conferma se tutto è andato a buon fine, oppure messaggi di errore vari. Può essere usata da utenti che hanno effettuato il login.

Trovare eventuali multe da pagare

L'API getMultas serve all'utente per visualizzare le eventuali multe emesse nei suoi confronti dal sistema. Utilizza il metodo find() per ritornare le ipotetiche multe (anche più di una). L'output è un array di multe se ce ne sono, altrimenti un array vuoto. Può essere usata da utenti che hanno effettuato il login.

Trova gli appuntamenti di un utente

L'API getAppuntamenti serve all'utente per visualizzare gli eventuali appuntamenti. Utilizza il metodo find() per ritornare i vari appuntamenti (anche più di uno). L'output è un array di appuntamenti se ce ne sono, altrimenti un array vuoto. Può essere usata da utenti che hanno effettuato il login.

3.5.2 API del modello Books

Trova tutti i libri

l'API getAllBooks viene utilizzata per poter ottenere tutti i libri presenti nell'archivio. utilizza il metodo find() per ritornare le informazioni riguardo i libri (nome e cognome dell'autore, titolo, genere e disponibilità). Nel caso di nessun libro trovato ritorna un messaggio di errore.

Eliminare un libro

L'API cancellaLibro serve per poter eliminare un libro dal database è accessibile solo ad un utente in possesso di credenziali Admin. Prende in input il titolo di un libro e se non ci sono errori il dato libro sarà eliminato dal database, sennò ci saranno dei messaggi di errore. Utilizza metodi findOne() per trovare il libro e deleteOne() per eliminarlo.

```
const CancellLibro = async (req, res) => {
    let data = await libro.findOne({titolo : req.query.titolo}) .exec()

    if (!data) {
        return res.status(404).json({success : false, message : "Libro non trovato"})
    } else {
        await libro.deleteOne({book_id : data.book_id});
        return res.status(200).send()
    }
}
```

Figure 22: CancellLibro: API DELETE

Ricerca in archivio

L'API ricercaLibro viene utilizzata per cercare uno specifico libro nel database, può prendere in input il titolo del libro, il nome o cognome dell'autore e ritorna i libri che rispettano l'input inserito, se valido o messaggi di errore. Utilizza il metodo find() per la ricerca.

Filtro

L'API Filter viene utilizzata per filtrare i risultati di una ricerca, può prendere in input il cognome dell'autore oppure il genere del libro e ritorna i libri che rispettano l'input inserito, se valido o messaggi di errore. Utilizza il metodo find() per la ricerca.

3.5.3 API del modello admin

Trova tutti gli utent

l'API getAllUsers viene utilizzata per poter ottenere tutti gli utenti registrati. Utilizza il metodo find() per ritornare le informazioni riguardo gli utenti(nome, cognome e mail). Nel caso di nessun utente trovato ritorna un messaggio di errore. Può essere utilizzata solo dagli user in possesso delle credenziali Admin.

Inserire un nuovo libro nel database

L'API newLibro serve all'admin per aggiungere un libro nel database. Prende come input un body di informazioni riguardo al libro (titolo, nome e cognome dell'autore e genere del libro). Ritorna in output un errore se il libro è già presente nel database. Utilizza il metodo findOne() per verificare se il titolo sia già presente in archivio e il metodo save() per salvare la risorsa nella collezione. Può essere utilizzata solo dagli user in possesso delle credenziali Admin.

Emetti multa

L'API multa viene utilizzata per emettere una multa nei confronti di un utente. Prende in input la mail dell'utente a cui emettere la multa e un body degli attributi della multa (mail dell'utente, importo e la scadenza della multa). Utilizza il metodo findOne() per trovare la mail dell'utente, date() per settare la scadenza della multa e save() per il salvataggio della multa appena creata. Ritorna un messaggio d'errore se l'utente non esiste. Può essere utilizzata solo dagli user in possesso delle credenziali Admin.

Cancella un utente

L'API deleteUtente serve per poter eliminare un utente dal database è accessibile solo ad un utente in possesso di credenziali Admin. Prende in input la mail di un utente e se non ci sono errori il dato utente sarà eliminato dal database, sennò ci saranno dei messaggi di errore. Utilizza metodi findOne() per trovare l'utente e deleOne() per eliminarlo.

3.5.4 Metodi del modello authentication

Login

L'API login serve per poter effettuare il login nella piattaforma. Prende in input un body contenente la mail e la password corrispondente, ritorna dei messaggi di errore se una o l'altra sono errati. Utilizza il metodo findOne() per verificare se la mail esiste nel database.

Logout

L'API logout serve all'utente per poter effettuare il logout. Prende in input il token di accesso e ritorna un messaggio di errore se l'operazione non è andata a buon fine.

3.5.5 API del modello disponibilità

Aggiornamento della disponibilità di un libro

L'API updateDisponibilità è utilizzata per modificare la disponibilità da FALSE a TRUE di un libro che è stato restituito e setta anche la scadenza a NULL. Inoltre tramite la funzione locale 'rimuoviElemento()' elimina l'ID del libro dall'array di libri in noleggio di un utente e sottrae di '1' il numero di noleggi. Prende in input la mail dell'utente che ha riportato il libro e il titolo del libro. Ritorna dei messaggi di errore se l'utente o il libro non esistono. Utilizza i metodi findOne() per cercare la mail dell'utente e il titolo del libro e il metodo updateOne() per modificare la scadenza e l'array di libri. Può essere utilizzata solo dagli user in possesso delle credenziali Admin.

```
const updateDisponibilità = async (req, res) => {
    // Attendere la promessa restituita da findOne()
    let book = await libro.findOne({ titolo: req.query.titolo }).exec();
    let user = await utente.findOne({ mail: req.query.mail }).exec();

    if(!book) {
        return res.status(404).json({ success: false, message: "Libro non trovato" });
    }
    if(!user) {
        return res.status(404).json({ success: false, message: "Utente non trovato" });
    }

    // Eseguire updateOne() con i dati da aggiornare
    await libro.updateOne({ titolo: book.titolo }, {
        $set: {
            Is_available: true,
            scadenza: null
        }
    });

    function rimuoviElemento(array, elementoDaRimuovere) {
        // Trova l'indice dell'elemento da rimuovere
        const indice = array.indexOf(elementoDaRimuovere);

        // Se l'elemento è presente nell'array, rimuovilo
        if (indice !== -1) {
            array.splice(indice, 1);
        }

        // Ritorna l'array modificato
        return array;
    }

    await utente.updateOne({ mail: user.mail }, {
        $set: {
            n_libri: user.n_libri - 1,
            libri_noleggiati: rimuoviElemento(user.libri_noleggiati, book.book_id)
        }
    });
    return res.status(200).json({ success: true, message: "Disponibilità aggiornata" });
};
```

Figure 23: updateDsiponibilità: API PATCH

4 API documentation

Tutte le API sviluppate per l'applicazione e descritte nella sezione precedente sono state documentate utilizzando **Swagger UI**, che ci permette di avere una pagina web disponibile per visionare la documentazione delle API, e rende anche possibile testarne il funzionamento. Le API all'inizio vengono divise per modello

Figure 24: Pagina Swagger delle API

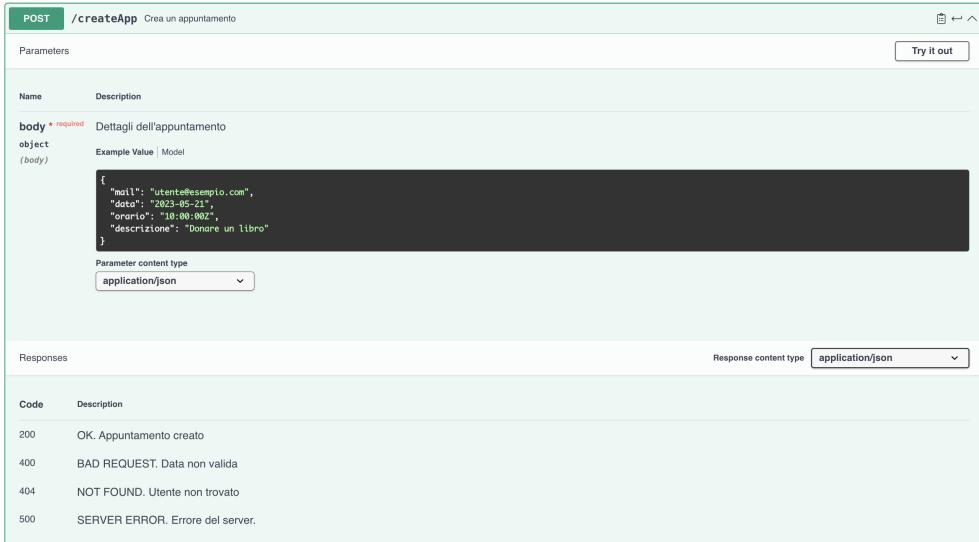
Aprendo poi un modello compaiono tutte le API sviluppate per quel modello divise per le tipologie, che possono essere:

- GET: Usata per ottenere dei dati dal sistema.
- POST: Usata per inviare dei nuovi dati dal sistema
- PATCH: usata per modificare una determinata risorsa all'interno del database.
- DELETE: Usata per eliminare una determinata risorsa dal database.

Di seguito viene lasciata un'immagine di esempio delle API realizzate per il modello utente:

Figure 25: API del modello Utente

Tutte le API sono sviluppate allo stesso modo, presentano una descrizione iniziale di come operano, sono presenti dei campi dove inserire i parametri in input, viene anche scritto il tipo di dato che l'input deve avere, e alla fine sono presenti i vari codici di ritorno possibili per quell'API. Di seguito un esempio di createApp del modello utente



Name	Description
body * required	Dettagli dell'appuntamento
object (body)	Example Value Model
<pre>{ "mail": "utente@esempio.com", "data": "2023-05-21", "orario": "10:00:00Z", "descrizione": "Donare un libro" }</pre>	
Parameter content type	application/json

Code	Description
200	OK. Appuntamento creato
400	BAD REQUEST. Data non valida
404	NOT FOUND. Utente non trovato
500	SERVER ERROR. Errore del server.

Figure 26: Esempio struttura di un API

5 Frontend

in questa sezione viene descritta l'implementazione del FrontEnd di EasyLib, ovvero l'interfaccia con cui l'utente andrà a interagire. Le pagine con cui l'utente potrà interagire sono 9, ovvero:

- Homepage
- Login
- Registrazione
- Archivio
- Ricerca
- Servizi
- Contatti
- Impostazioni

- Appuntamento

5.1 Homepage

La Homepage viene presentata come dalla figura sottostante



Figure 27: Pagina Homepage

Vediamo subito in alto l'Header contente i link per le varie pagine, ovvero Home, servizi, impostazioni e contatti, oltre al bottone per effettuare il Login. L'header resterà presente in ogni schermata per facilitare al meglio la navigazione. In fondo alla Homepage è presente anche un link diretto all'archivio, come riportato di seguito.

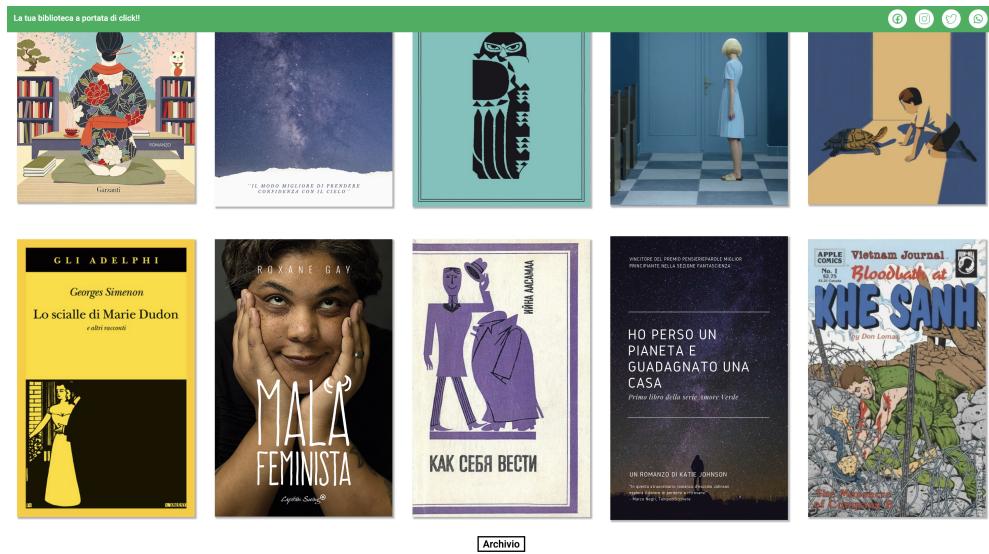


Figure 28: Pagina Homepage 2

5.2 Login

La pagina di **Login** presenta un campo dove inserire la mail e la corrispondente password per gli utenti che si sono già registrati.

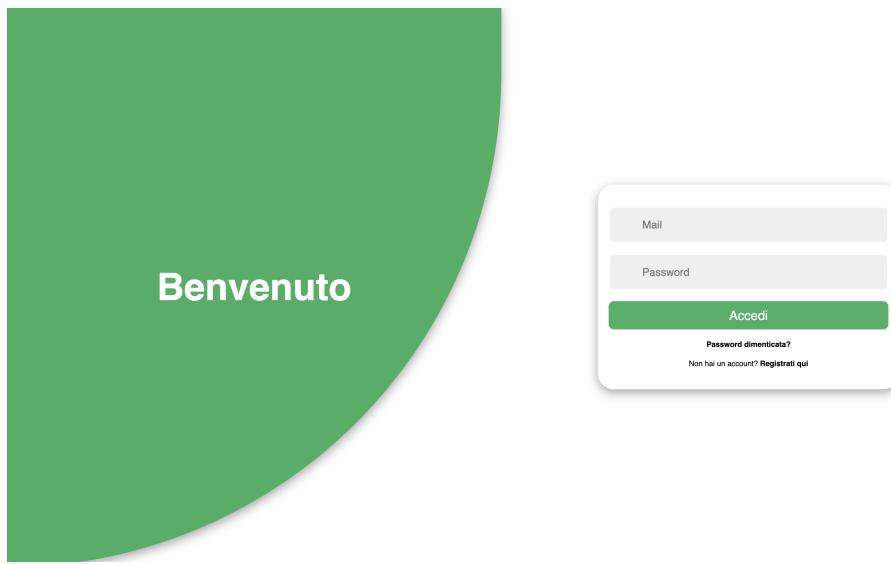


Figure 29: Pagina di Login

5.3 Registrazione

La pagina di **Registrazione** comprende un form da compilare con le varie informazioni per ogni nuovo utente, ovvero:

- Nome
- Cognome
- Mail
- Password
- Conferma password

La password inoltre deve rispettare i criteri di una password sicura, cioè lunghezza minima 8 caratteri con almeno 1 carattere speciale.

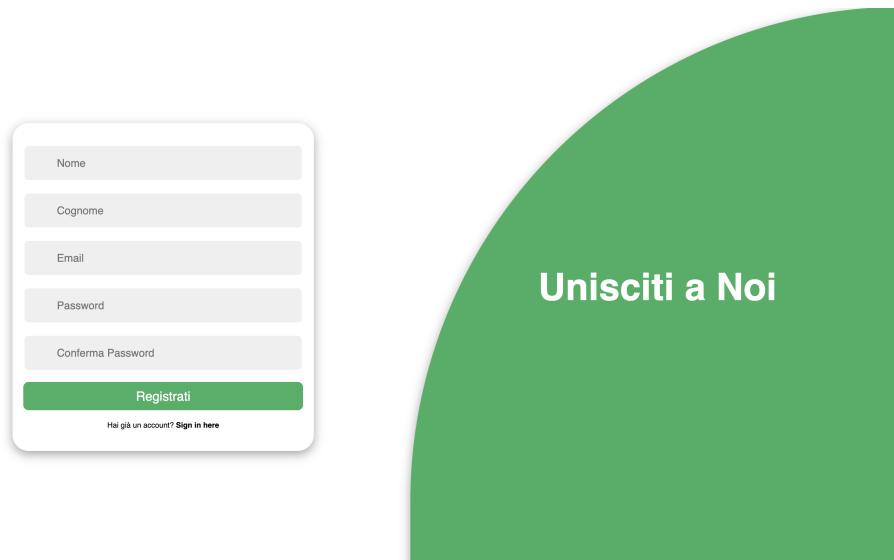


Figure 30: Pagina di Registrazione

5.4 Archivio

La pagina di **Archivio** presenta tutti i libri presenti all'interno del Database MongoDB con anche la barra di ricerca per cercare uno specifico volume con anche la possibilità di utilizzare dei filtri di ricerca

D4-G06

The screenshot shows the EasyLib library archive page. At the top, there's a green header bar with the text "La tua biblioteca a portata di click!" and the EasyLib logo. Below the header is a search bar with the placeholder "Search". The main content area displays three book entries:

- Il mastino dei Baskerville** by Arthur Conan Doyle. It's a yellow book cover featuring a black dog. The genre is listed as "Giallo" and it's marked as "Disponibile" (Available). A blue "Prenota e ritira" button is present.
- Dieci piccoli indiani** by Agatha Christie. It's a dark book cover with a woman's face. The genre is "Giallo" and it's marked as "Disponibile". A blue "Prenota e ritira" button is present.
- Il nome della rosa** by Umberto Eco. It's a white book cover with a circular design. The genre is "Giallo" and it's marked as "Disponibile". A blue "Prenota e ritira" button is present.

On the left side, there are sidebar filters for "Autore" and "Genere". The "Autore" sidebar includes names like Kafka, Calvino, Pirandello, Doyle, Christie, Eco, Baricco, Machiavelli, Orwell, De Cervantes, Shakespeare, and Herbert. The "Genere" sidebar includes categories like Romanzo, Fantascienza, Tragedia, Giallo, Storia, and Satira. A green "Conferma Filtri" button is located at the bottom of the sidebar.

At the top right of the page, there are social media icons for Facebook, Twitter, and LinkedIn, along with a link to "Luca Medici's profile". The footer contains standard navigation links: Home, Servizi, Appuntamento, Impostazioni, and Contatti.

Figure 31: Pagina Archivio

5.5 Ricerca

La pagina di ricerca rappresenta i risultati consecutivi a una ricerca fatta utilizzando l'apposita barra oppure selezionando uno dei filtri disponibili nella sidebar, presentando i volumi che corrispondono ai parametri inseriti.

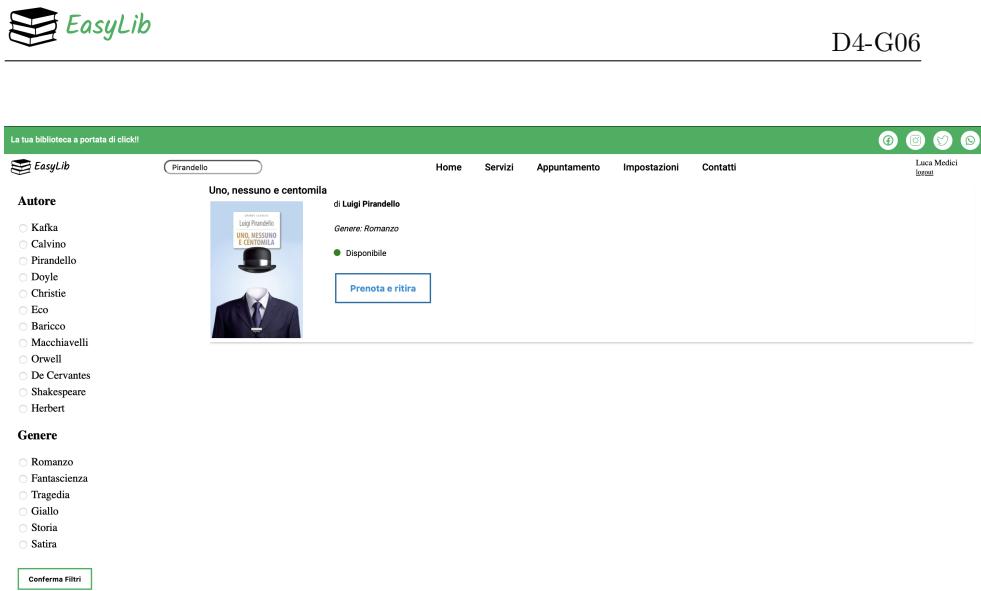


Figure 32: Pagina di ricerca

5.6 Servizi

Nella pagine dei servizi vengono descritti i servizi dell'applicazione nel formato di FAQ.

Figure 33: Pagina di Servizi

5.7 Contatti

La pagina dei **contatti** presenta un collegamento a Google maps per ottenere le indicazioni per la sede di EasyLIB. In più è presente la mail di riferimento e il numero di telefono della segreteria per avere modalità di comunicazione con la sede.

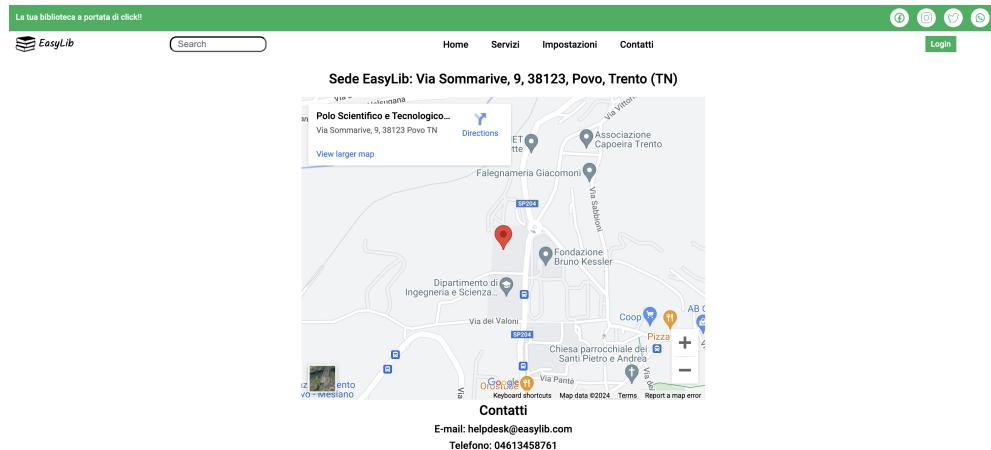


Figure 34: Pagina dei Contatti

5.8 Impostazioni

La pagina delle impostazioni è suddivisa nelle parti di:

- Lingua: dove si può passare dalla lingua italiana a quella inglese.
- I miei noleggi: dove si può vedere i libri noleggiati e terminare la prenotazione.
- I miei appuntamenti: dove si possono vedere lo stato degli appuntamenti già riservati.
- Pagamenti: dove vengono inviate le multe dal sistema.

Di seguito alcune immagini della sezione impostazioni.



Figure 35: Pagina di impostazioni Lingua



Figure 36: Pagina di impostazioni Pagamenti



Figure 37: Pagina di impostazioni appuntamenti

5.9 Appuntamento

Nella pagina Appuntamento è presente un form da compilare per la prenotazione di un appuntamento in fisico all'interno degli uffici di EasyLib. Le informazioni necessarie per effettuare la prenotazione, e presenti nel form, sono:

- Indirizzo email (di un utente già registrato)

- Data richiesta dell'appuntamento
- L'orario dell'appuntamento

in più va specificato fra 2 campi, il motivo della richiesta dell'appuntamento fra quelli presenti, ovvero per restituire un libro o donarlo.

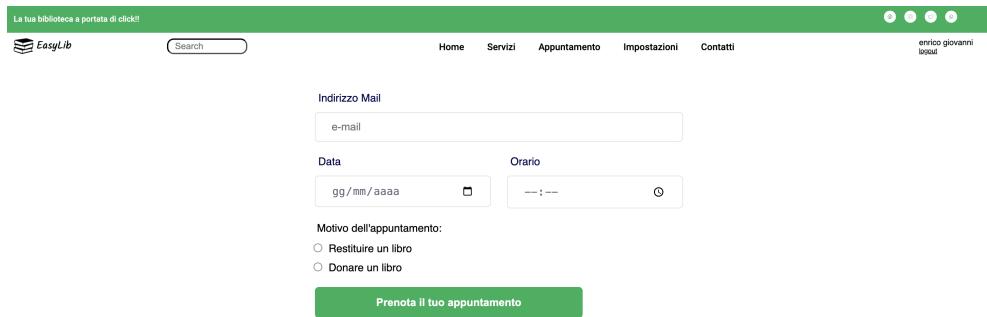


Figure 38: Pagina di Prenotazione appuntamenti

5.10 Frontend Admin

Alla sezione Admin si accede inserendo le apposite credenziali nella fase di login. Le sue pagine di Frontend presentano alcune differenze con quelle dell'utente normale, le sue pagine di navigazione sono:

- Homepage (vedi sezione Homepage 5.1)
- Archivio (vedi sezione Archivio 5.4)

Che sono praticamente uguali a quelle disponibili per l'utente normale, e poi:

- Donazioni
- Utenti

Che sono specifiche dell'admin e vengono descritte qui sotto.

5.10.1 Donazioni Admin

La pagina di donazioni per l'admin comprende un form necessario all'amministratore per inserire all'interno del sistema i dati del libro donato da un utente. I campi necessari sono i dati dell'autore, il titolo del volume e il suo genere (si veda figura sottostante).

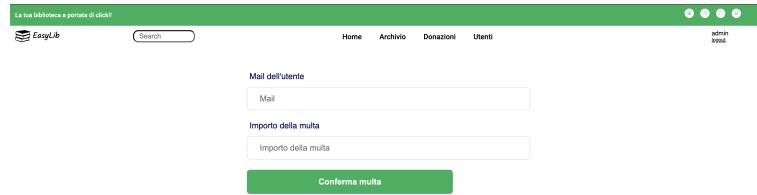


Figure 39: Pagina di donazioni

La pagina utenti dell'admin, mostra tutti gli utenti registrati e attraverso due bottoni "Apri pagina multe" e "Apri pagina noleggi" si aprono le rispettive pagine che permetto all'admin le sue operazioni di gestione.

EasyLib		Search	Home	Archivio	Donazioni	Utenti	admin luca
Nome: Stefano	Cognome: Bianchi	Mail: Ste.bianchi@gmail.com					
Nome: Giuseppe	Cognome: Verdi	Mail: Giuseppe.verdi@gmail.com					
Nome: Mario	Cognome: Rossi	Mail: ciaosonmario@gmail.com					
Nome: Enrico	Cognome: Giovanni	Mail: enrico.giovanni@gmail.com					

Figure 40: Pagina utenti per l'admin



The screenshot shows a software interface with a green header bar. On the left, there's a small icon of three books and the text "La tua biblioteca a portata di click!". In the center, the "EasyLib" logo is displayed next to a search bar with the placeholder "Search". To the right of the search bar are navigation links: "Home", "Archivio", "Donazioni", and "Utenti". On the far right of the header, there are four small circular icons and the text "admin" and "S006". Below the header, the main content area has two input fields: "Mail dell'utente" (with "Mail" in it) and "Importo della multa" (with "Importo della multa" in it). At the bottom of the form is a green button labeled "Conferma multa".

Figure 41: Form per creazione Multa

6 Testing

In questa sezione vengono descritti i casi di test per le varie API che vengono usate su **EasyLib**. I test sono realizzati utilizzando Jest. Di seguito viene descritta la loro realizzazione e il funzionamento. I casi di test sono divisi in 5 file di tipo `test.js` a seconda del tipo delle API che vengono testate. Alla fine della sezione vengono anche mostrati i risultati del test con la copertura del codice diviso per ogni cartella.

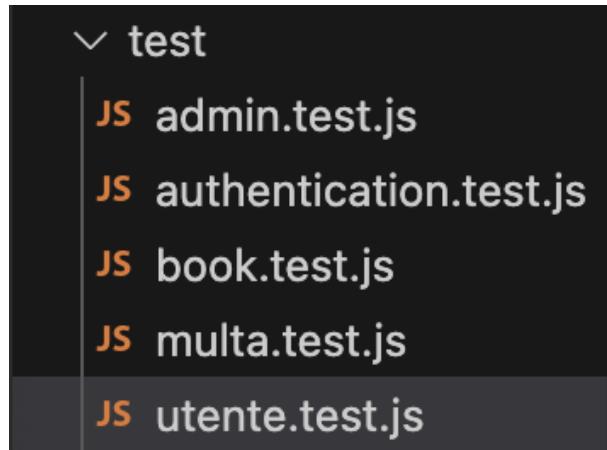


Figure 42: Test Folder

Verrà adesso descritta la struttura di un tipico file `.test.js` per il testing delle API. Tutti i nostri file sono strutturati allo stesso modo, quindi prenderemo come esempio e riporteremo le foto qua sotto, del file **Utente.test.js**.



```

test > js utente.test.js ...
1  const {default : mongoose} = require('mongoose');
2  const request = require('supertest');
3  require('dotenv').config();
4  const app = require('../server');
5  const jwt = require('jsonwebtoken');
6  const { unchangedTextChangeRange } = require('typescript');
7  let server = app.listen(process.env.PORT || 8080);
8
9  module.exports = {
10    setupFilesAfterEnv : ['./jest.setup.js']
11  }
12  beforeAll(async () => {
13    jest.setTimeout(30000)
14    app.locals.db = mongoose.connect(process.env.MONGODB_URI);
15  });
16  afterAll(async () => {
17    mongoose.connection.close();
18    server.close();
19  });
20
21  var dataValida = new Date();
22  dataValida.setDate(dataValida.getDate());
23  dataValida.setHours(dataValida.getHours() + 1);
24  dataValida.setMonth(dataValida.getMonth());
25  dataValida.setFullYear(dataValida.getFullYear());
26
27  var dataNonValida = new Date();
28  dataNonValida.setDate(dataValida.getDate()-1);
29  dataNonValida.setHours(dataValida.getHours() + 1);
30  dataNonValida.setMonth(dataValida.getMonth() -1);
31  dataNonValida.setFullYear(dataValida.getFullYear());
32
33

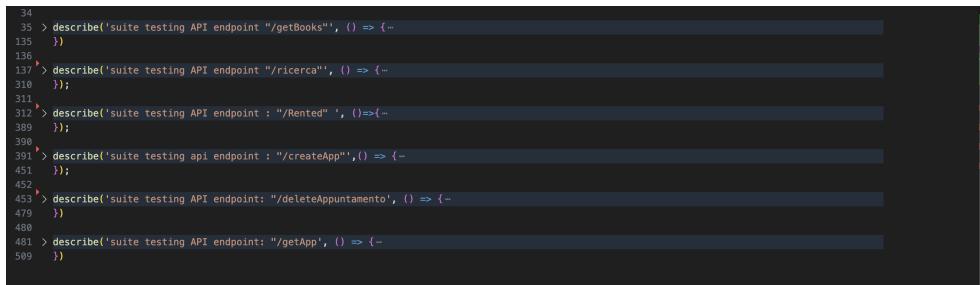
```

Figure 43: Struttura file test

All'inizio si ha un importazione di **Mongoose** e del modulo **Superjest** che effettuano la connessione al database. Viene importato il **jsonwebtoken** che serve per la creazione di un token per alcune API che ne hanno bisogno.

Successivamente ci sono i metodo **BeforeAll()** che definisce tutto ciò che va fatto nel momento iniziale di esecuzione del file; e **AfterAll()** che è l'ultimo metodo chiamato che si occupa di chiudere la connessione al database e disattivare il server. Questi metodi sono presenti in ogni API

Le API poi vengono testate nelle reciproche parti chiamate dai metodi **Describe()**



```

34 > describe('suite testing API endpoint "/getBooks"', () => {-
35   })
36
37 > describe('suite testing API endpoint "/ricerca"', () => {-
38   })
39
40 > describe('suite testing API endpoint : "/Rented"', ()=>{-
41   })
42
43 > describe('suite testing api endpoint : "/createApp"',() => {-
44   })
45
46 > describe('suite testing API endpoint: "/deleteAppuntamento', () => {-
47   })
48
49 > describe('suite testing API endpoint: "/getApp', () => {-
50   })
51

```

Figure 44: Casi di test Utente

Di seguito viene anche riportato un esempio di un metodo **describe()**, in questo caso per semplicità viene riportato il test di **/deleteAppuntamento**

```

453 describe('suite testing API endpoint: "/deleteAppuntamento', () => {
454   test('Chiamata API corretta', async () => {
455     const res = await request(app)
456       .delete('/deleteAppuntamento')
457       .query(
458         {
459           mail : 'utenteprova@gmail.com',
460         }
461       )
462       .expect(200);
463       expect(res.body.success).toBe(true);
464       expect(res.body.message).toBe('Appuntamento cancellato con successo');
465   })
466
467   test('Chiamata API con appuntamento non prenotato', async () => {
468     const res = await request(app)
469       .delete('/deleteAppuntamento')
470       .expect(404)
471       .query(
472         {
473           mail : 'Test3@gmail.com',
474         }
475       );
476       expect(res.body.success).toBe(false);
477       expect(res.body.message).toBe('Appuntamento non trovato');
478   })
479 })
480

```

Figure 45: Testing /deleteAppuntamento

In questa API i casi di test sono 2: uno di successo in cui viene passata una mail con un appuntamento fissato che andrà poi cancellato, con response code **200**. E uno in caso di insuccesso dove alla mail passata come input non corrisponde nessun appuntamento, il response code **404**.

6.1 Risultati del test

Per fare il testing abbiamo aggiunto il seguente script al file package.json :
 'test': 'jest --coverage --detectOpenHandles'. Tramite il comando npm test, fatto partire dalla root del progetto otterremo tutti i risultati dei test definiti nei file '.test.js'.

La flag --coverage di jest serve a creare dei report finali del nostro testing, mentre la flag --detectOpenHandles serve a identificare elementi che non permettono al testing di terminare. I risultati:

```

Test Suites: 5 passed, 5 total
Tests:      55 passed, 55 total
S_snapshots: 0 total
Time:        14.329 s, estimated 27 s
Ran all test suites.

```

Figure 46: Test Suites

Le 5 suites definite e i 53 test case, definiti nelle funzioni test(), sono stati eseguiti e risultano passati.

Il report dei risultati del testing generato da coverage si può trovare nel file **index.html**, locato, a partire della root del progetto, nella cartella **coverage/lvcon-report/**. La pagina ipertestuale risultante è la seguente:

Backend		96.66%	29/30	100%	0/0	0%	0/1	96.66%	29/30
Backend/auxiliares		100%	10/10	100%	11/11	100%	2/2	100%	10/10
Backend/controllers		95%	228/240	92.5%	74/80	100%	28/28	94.89%	223/235
Backend/models		100%	20/20	100%	0/0	100%	0/0	100%	20/20
Backend/routes		100%	127/127	100%	0/0	100%	0/0	100%	126/126

Figure 47: Coverage Progetto

Questa prima pagina presenta un riassunto della copertura di tutte le funzioni, i branches, gli statements e le linee di codice del progetto prese in carico dal testing.

La cartella che ci interessa è Backend/controllers, che è quella dove abbiamo definito tutte le api. Si nota che abbiamo circa un 93% di copertura per i branches, le linee di codice e gli statements, mentre le funzioni sono coperte al 100%.

Cliccando sopra Backend/controllers ci viene mostrata la pagina seguente, che rappresenta il coverage delle api:

admin.js		89.79%	44/49	75%	9/12	100%	6/6	89.58%	43/48
authentication.js		100%	21/21	100%	6/6	100%	2/2	100%	21/21
book.js		96.22%	51/53	90%	18/20	100%	6/6	96.07%	49/51
disponibilita.js		100%	17/17	100%	6/6	100%	2/2	100%	17/17
utente.js		95%	95/100	97.22%	35/36	100%	12/12	94.89%	93/98

Figure 48: Coverage delle API

Si nota che c'è un'oscillazione tra le percentuali della copertura, questa differenza tra questi numeri è data da errori del seguente tipo, che non sono dipendenti da noi, ma dal server:

```
173 } catch (error) {
174   return res.status(500).json({ success: false, message: "Errore del server" });
175 }
```

Figure 49: Errori del server

Inoltre nelle API di creazione di un nuovo libro e nuovo utente abbiamo utilizzato una funzione specializzata per l'id dei rispettivi, i.e. la mancata copertura delle seguenti linee di codice:

```

20 5x
21
22
23
24      I if(cd==null){
          const newVal = new Counter({id: "autoval", seq: 1})
          newVal.save();
          seqId = 1
}

```

Figure 50: Copertura di ID utente e libro non presente

Altro problema riscontrato è nelle API di getAll dei libri e degli utenti:

```

10 1x
11 1x
12      E if(data){
          I if(data.length === 0){
              return res.status(404).json({success: false, message : "Nessun utente presente"})
          }
13

```

Figure 51: Esempio dell'API su utente

per far si che queste linee vengano coperte bisognerebbe avere il database vuoto o inesistente, e nel nostro caso non lo è mai stato. Per l'API getAllBooks è la stessa cosa.

Abbiamo anche usato delle funzioni ausiliare per la validazione della mail e della password, definite in 'Backend/auxiliaries/', qui sotto il coverage:



Figure 52: Coverage Progetto

7 GitHub repository and Deployment

Al seguente URL è presente la repository Github del nostro progetto:

[https://github.com/orgs/G06-Progetto-IS/repositories.](https://github.com/orgs/G06-Progetto-IS/repositories)

Il nome della cartella è **G06-Progetto-IS**. Nella sezione **Deliverables** sono presenti i vari deliverables da consegnare. Nella repository **Documentazione** sono presenti le bozze dei documenti iniziali, in seguito abbiamo iniziato ad utilizzare il sito **Overleaf** per produrre documenti .pdf usando il linguaggio **LAT_EX**. Inoltre sono presenti i vari aggiornamenti delle ore di lavoro. Nella repository **Backend** è presente tutto il codice relativo alla nostra applicazione, FrontEnd compreso. Mentre nella repository **Frontend** è presente solo il codice HTML e CSS.

7.1 Deployment

Il deployment è stato effettuato utilizzando <https://render.com> e la nostra applicazione è raggiungibile utilizzando l' URL:

<https://easylib-wf03.onrender.com>

Nota: Il primo caricamento del nostro sito a causa dell'utilizzo del servizio 'free' di 'Render' può richiedere fino a 50 secondi. Una volta eseguito il primo caricamento poi girerà tranquillamente con dei tempi di attesa minori di 1 secondo.

La documentazione delle API svolta con Swagger è raggiungibile tramite l'URL:

<https://easylib-wf03.onrender.com/api-docs>

Selezionare lo schema HTTPS per chiamare le varie API.

7.1.1 Eseguire in locale

Nel caso in cui si voglia eseguire il server sulla propria macchina, bisogna prima di tutto usare l'URL: <https://github.com/G06-Progetto-IS/Backend> e clonare la repository sulla propria macchina. Successivamente aprire la cartella Backend utilizzando **Visual studio Code**, eseguire su terminale il comando **npm start**. Non appena vengono stampate in console le seguenti righe: Server in ascolto sulla porta: 3000 MongoDB Connection – Ready state is: 1 è possibile collegarsi tramite il browser alla webApp, utilizzando l'URL <http://localhost:3000/> si riuscirà a navigare sul sito.

Note per la documentazione La documentazione delle API svolta con Swagger è raggiungibile tramite l'URL:

<http://localhost:3000/api-docs>

Selezionare lo schema HTTP per chiamare le varie API..