Documentazione progetto Basi di Dati 2: WorldBookShelf

Panoramica

WorldBookShelf è una web application progettata per gli amanti dei libri, offrendo loro una piattaforma interattiva per esplorare, recensire e organizzare la loro collezione di libri. Con oltre 20.000 titoli disponibili, WorldBookShelf consente agli utenti di lasciare recensioni utilizzando un sistema a stelle da 1 a 5 per valutare i libri presenti sulla piattaforma.

Gli utenti possono anche salvare i titoli che hanno già letto nella libreria "Read" e quelli che desiderano leggere nella libreria "Want to Read". Questa funzionalità consente agli utenti di tenere traccia dei loro progressi di lettura e di gestire facilmente la loro lista di desideri letterari.

Oltre alla gestione delle recensioni e delle librerie personali, WorldBookShelf utilizza un sistema di raccomandazione. Basandosi sui generi dei libri che l'utente ha salvato nelle librerie, la webapp suggerisce altri titoli che potrebbero interessare agli utenti. Questo approccio personalizzato aiuta gli utenti a scoprire nuovi libri che potrebbero corrispondere ai loro gusti e interessi letterari.

Fase preliminare - dataset

Nella fase preliminare vengono utilizzati diversi script python per estrarre i dati rilevanti da vari dataset, ognuno con colonne diverse, e mapparli alle variabili della classe Book (file book.py). Questo in modo da creare un unico formato di dati che rappresenta i libri, indipendentemente dalle differenze tra i dataset originali.

Dataset utilizzati:

- https://www.kaggle.com/datasets/bahramjannesarr/goodreads-book-datasets-10m
  Vari dataset contenenti informazioni su rating libri. Questi dataset sono stati utilizzati come base di partenza a cui sono state aggiunte altre informazioni prese dai dataset successivi.
- https://www.kaggle.com/datasets/dk123891/books-dataset-goodreadsmay-2024
- https://www.kaggle.com/datasets/sahilkirpekar/goodreads10k-dataset-cleaned
- https://github.com/zygmuntz/goodbooks-10k
- https://www.kaggle.com/datasets/ishikajohari/best-books-10k-multi-genre-data
- https://www.kaggle.com/datasets/thedevastator/comprehensive-overview-of-52478-goodreads-best-b

I vari script utilizzati in sequenza sono:

1. **save_books.py**
   In questo file attraverso la funzione *get_books_from_csv(...)*, a cui vengono passati come parametri il percorso e i nomi delle colonne del dataset di nostro interesse, vengono letti i valori e costruito un dizionario di libri che ha come chiave l'id del libro e come valore l'oggetto Book costruito con i valori estratti dal dataset.
   Viene costruito un dizionario di libri per ogni dataset utilizzato, e successivamente questi vengono serializzati in modo da avere direttamente i dati estratti per gli script successivi e non dover rileggere e rielaborare i dataset più volte.

2. **books_to_load.py**
   In questo file al dizionario di libri relativo ai dataset scaricati dal primo link, vengono aggiunte informazioni come il genere, la descrizione e l'url della cover del libro, etc.. prese dai dizionari relativi agli altri dataset.
   Da questo dizionario vengono filtrati in un altro dizionario "*books_to_load*" solo i libri che hanno i generi, scartando quelli che non li hanno.
   Successivamente di quest'ultimi vengono eliminati anche i libri che non hanno descrizione e che non hanno editore (publisher).

Successivamente, essendo che le informazioni sulla lingua vengono prese da più dataset, con la conseguenza che ogni dataset utilizza un modo diverso per rappresentare la lingua, ad esempio un dataset utilizza “en”, un altro “English” etc.. , si è usata la funzione *normalize_languages(...)* per uniformare i nomi delle lingue, in modo che siano rappresentate in modo coerente.
A questo punto, il dizionario di libri risultante viene serializzato.

3. **export_books_db,py**
   In questo file viene deserializzato il dizionario serializzato nello script precedente e per ogni libro in esso presente (con numero di pagine > 0) , i generi che erano rappresentati da un’unica stringa vengono trasformati in un array di stringhe, e il numero di stelle viene convertito nell’intero corrispondente. Dopo viene caricato sul database attraverso una richiesta HTTP POST al server locale all’endpoint specificato:
   response = requests.post('http://localhost:4000/api/book/addBook', json=new_document)

WebApplication

1. **backend**
   è stato utilizzato **Node.js** con **Express** e **MongoDB** come database. In particolare, è stato utilizzato **Mongoose**, una libreria Node.js per l’interazione con MongoDB.

   I vari *schemi* utilizzati sono:
   - ***User*** che rappresenta le informazioni di un utente

```
const userSchema = new mongoose.Schema({
    username : {type: String, required: true, unique:true},
    name:  {type: String, required: true},
    surname: {type: String, required: true},
    date: {type: String, required: true},
    email : {type: String, required: true, unique:true},
    password : {type: String, required: true},
})
```

```
//middleware pre save
userSchema.pre('save', async function (next) {
    const user = this; //riferimento all'istanza dell'utente su cui viene chiamato il metodo save()

    /**
     * Verifica se la password dell'utente è stata modificata.
     * Se la password non è stata modificata (ad esempio, quando si aggiorna un'email senza cambiare la password),
     * il middleware passa al prossimo middleware nella catena senza fare nulla.
     */
    if (!user.isModified('password'))
        return next();

    /**
     * Genera un "salt" casuale utilizzato per hashing della password.
     * Il numero 10 rappresenta il costo, cioè il numero di iterazioni dell'algoritmo di hash.
     */
    const salt = await bcrypt.genSalt(10);
    const hash = await bcrypt.hash(user.password, salt);
    user.password = hash;

    next(); //per procedere con il processo di salvataggio dell'utente nel database.
});
```

   In questo secondo screen possiamo vedere che viene utilizzato l'approccio del middleware pre di Mongoose per gestire l'hashing della password prima che venga salvata nel database. Questo approccio assicura che la password inserita dall'utente sia criptata utilizzando l'algoritmo bcrypt prima di essere memorizzata nel database.

- ***Book*** che rappresenta le informazioni di un libro

```
const bookSchema = new mongoose.Schema({
    id_book:{ type: String, required: true, unique: true},
    title: { type: String, required: true },
    authors: { type: String, required: true },
    cover : {type: String},
    genres: [{ type: String }],  // Array di stringhe per i generi
    description: { type:String , required: true },
    publisher: { type: String},
    publish_year: { type: String, required: true },
    pages_number: { type: Number, required: true, default: 0 },
    language:  { type: String },
    number_stars_1: { type: Number, default: 0 },
    number_stars_2: { type: Number, default: 0 },
    number_stars_3: { type: Number, default: 0 },
    number_stars_4: { type: Number, default: 0 },
    number_stars_5: { type: Number, default: 0 },
})
```

- ***Review*** che rappresenta la recensione lasciata da un utente ad un libro

```
const reviewSchema = new mongoose.Schema({
    user: { type: mongoose.Schema.Types.ObjectId, ref: 'user', required: true },
    book: { type: mongoose.Schema.Types.ObjectId, ref: 'book', required: true },
    rating: { type: Number, min: 1, max: 5, required: true },
})
```

- ***Shelf*** che rappresenta l'insieme di libri salvati da un utente

```
const shelfSchema = new mongoose.Schema({
    user: { type: mongoose.Schema.Types.ObjectId, ref: 'user', required: true },
    books_read: [{ type: mongoose.Schema.Types.ObjectId, ref: 'book' }], // Array di riferimenti ai libri già letti
    books_to_read: [{ type: mongoose.Schema.Types.ObjectId, ref: 'book' }], // Array di riferimenti ai libri da leggere
});
```

Operazioni:

- Login al sito

  Funzione **login** in back-end/controllers/authController.js

```
export const login = async (req, res) => {
    //username e password inseriti nel form
    const { username, password } = req.body;

    try {
        //cerco utente, con username inserito, nel db
        const user = await UserModel.findOne({ username });

        //se non esiste utente con quell'username
        if (!user)
            return res.json({
                success:false,
                message: 'user [' + username + '] does not exist'
            });

        //confronto se la password inserita e la password dell'utente nel db coincidono
        const isMatch = await bcrypt.compare(password, user.password);
        if (!isMatch)
            return res.json({
                success: false,
                message: 'invalid password'
            });

        res.json({
            success: true,
            message: ''
        });

    } catch (error) {
        console.error(error);
        res.status(500).json({
            success: false,
            message: 'Errore del server'
        });
    }
};
```

- Registrazione al sito
  Funzione **signup** in back-end/controllers/authController.js

```
export const signup = async (req, res) => {
    //valori inseriti nel form da utente
    const { username, password, name, surname, email, date } = req.body;

    try {
        // Verifico se l'utente esiste già nel database
        const existingUser = await UserModel.findOne({ username });
        if (existingUser) {
            return res.json({
                success: false,
                message: 'username already exists'
            });
        }

        const existingEmail = await UserModel.findOne({ email });
        if (existingEmail) {
            return res.json({
                success: false,
                message: 'email already exists'
            });
        }

        // Creazione di un nuovo utente nel database
        const newUser = new UserModel({
            username,
            name,
            surname,
            date,
            email,
            password    //la password viene automaticamente hashata dal middleware [vedi userModel.js]
        });

        await newUser.save();

        res.json({
            success: true,
            message:'User Added'
        });

    } catch (error) {
        console.error(error);
        res.status(500).json({
            success: false,
            message: 'Errore del server'
        });
```

- Aggiunta libri al database (usata in script python per salvare libri su db)
  Funzione **addBook** in back-end/controllers/bookController.js

```
export const addBook = async (req, res) => {
    //Dati dei libri da salvare
    const { id_book,
            title,
            authors,
            genres,
            description,
            publisher,
            publish_year,
            pages_number,
            language,
            number_stars_1,
            number_stars_2,
            number_stars_3,
            number_stars_4,
            number_stars_5,
            coverUrl } = req.body;
    try {
        // Verifico se id libro esiste già nel database
        const existingBookId = await bookModel.findOne({ id_book });
        if (existingBookId) {
            return res.json({
                message: 'Id Book ['+id_book+'] already exist'
            });
        }

        // Scarica l'immagine da URL e convertila in Base64
        let cover = '';
        if (coverUrl) {
            try {
                const response = await axios.get(coverUrl, { responseType: 'arraybuffer' });
                const buffer = Buffer.from(response.data, 'binary').toString('base64');
                cover = `data:${response.headers['content-type']};base64,${buffer}`;
            } catch (error) {
                console.error('Errore durante il download dell\'immagine:', error.message);
                // Se si verifica un errore nel download dell'immagine, loggo l'errore ma continuo senza copertina
            }
        }
```

```
        // Creazione di un nuovo utente nel database
        const newBook = new bookModel({
            id_book,
            title,
            authors,
            cover,
            genres,
            description,
            publisher,
            publish_year,
            pages_number,
            language,
            number_stars_1,
            number_stars_2,
            number_stars_3,
            number_stars_4,
            number_stars_5
        });

        await newBook.save();

        res.json({
            success: true,
            message:'Book ['+id_book+'] Added',
        });

    } catch (error) {
        console.error(error);
        res.status(500).json({
            success: false,
            message: 'Errore del server'
        });
    }
}
```

Scaricamento e conversione dell'immagine: Nei dati estratti dai file csv abbiamo il link delle varie immagini di copertina dei libri. Nel database vogliamo salvare direttamente queste immagini quindi il codice tenta di scaricare l'immagine dalla rete. Utilizza Axios per fare una richiesta GET al 'coverUrl' con 'responseType' impostato su 'arraybuffer'. L'immagine scaricata viene quindi convertita da 'arraybuffer' a una stringa in formato 'base64' (il formato Base64 è un modo per rappresentare dati binari in formato testuale). Questa stringa rappresenta i dati dell'immagine convertiti che possono essere inclusi direttamente come parte dei dati del libro salvato nel database. Se il download dell'immagine fallisce per qualche motivo, viene visualizzato un messaggio di errore ma il processo di salvataggio del libro continua senza l'immagine di copertina.

- Lista dei 10 libri con valutazione media più alta
  Funzione **topRatingBooks** in back-end/controllers/bookController.js

```
export const topRatingBooks = async (req, res) => {
    try {
        const pipeline = getTopRatingBooksAggregationPipeline();
        const books = await bookModel.aggregate(pipeline);

        if (books.length > 0) {
            return res.json({
                success: true,
                message: 'Top 10 books sorted by highest rating',
                books: books
            });
        } else {
            return res.json({
                success: false,
                message: 'No books found'
            });
        }
    } catch (error) {
        console.error('Error while fetching top rated books:', error);
        return res.status(500).json({
            success: false,
            message: 'Error while fetching top rated books'
        });
    }
}
```

Questa funzione recupera i primi 10 libri con la valutazione media più alta dalla collezione di libri nel database.

La funzione richiama la funzione `getTopRatingBooksAggregationPipeline()` per ottenere la pipeline di aggregazione. Questa pipeline calcola e ordina i libri in base alla loro valutazione media [vedi giù].

Funzione di aggregazione:

```
const getTopRatingBooksAggregationPipeline = () => [
    {
        $addFields: {
            totalReviews: {
                $add: ["$number_stars_1", "$number_stars_2", "$number_stars_3", "$number_stars_4", "$number_stars_5"]
            }
        }
    },
    {
        $addFields: {
            averageRating: {
                $cond: {
                    if: { $eq: ["$totalReviews", 0] },
                    then: 0,
                    else: {
                        $divide: [
                            {$add: [
                                    { $multiply: ["$number_stars_1", 1] },
                                    { $multiply: ["$number_stars_2", 2] },
                                    { $multiply: ["$number_stars_3", 3] },
                                    { $multiply: ["$number_stars_4", 4] },
                                    { $multiply: ["$number_stars_5", 5] }
                                ]},
                            "$totalReviews"
                        ]
                    }
                }
            }
        }
    },
    {
        $project: {
            id_book: 1,
            title: 1,
            authors: 1,
            cover: 1,
            totalReviews: 1,
            averageRating: 1
        }
    },
    {
        $sort: { averageRating: -1 }  // Ordina per la media della valutazione in ordine decrescente
    },
    {
        $limit: 10  // Limita il risultato ai primi 10 libri
```

Questa funzione getTopRatingBooksAggregationPipeline definisce una serie di stadi di aggregazione per calcolare e recuperare i libri con la valutazione media più alta, ordinati in ordine decrescente di media di valutazione e limitati ai primi 10 risultati.

1. $addFields:
    a. totalReviews: aggiunge un nuovo campo totalReviews al documento. Questo campo è calcolato sommando i valori dei campi number_stars_1, number_stars_2, number_stars_3, number_stars_4, e number_stars_5. Questi campi rappresentano il numero di recensioni per ogni stella da 1 a 5 di ogni libro.

2. $addFields:
    a. averageRating: aggiunge un campo averageRating al documento.
    b. $cond: operatore condizionale che valuta se totalReviews è uguale a 0.
    Se è uguale a 0, allora il libro non ha recensioni e l'averageRating viene impostato a 0.

Altrimenti, calcola la media delle valutazioni:

- Usiamo l'operatore $multiply per moltiplicare il numero di recensioni per stella per il valore della stella (1 * $number_stars_1, 2 *$number_stars_2, etc....).
- Sommiamo i risultati di tutte le moltiplicazioni usando l'operatore $add.
- Divide la somma totale delle valutazioni per stella per totalReview per ottenere la media effettiva delle valutazioni per quel libro.

3. $project:
   Definisce quali campi del documento devono essere inclusi nel risultato, in questo caso: id_book, title, authors, cover, totalReviews, e averageRating.

4. $sort:
   Ordina i documenti in base ad averageRating in ordine decrescente (-1). Quindi i libri con la valutazione media più alta verranno posizionati all'inizio.

5. $limit:
   Limita il numero di documenti restituiti a 10. Quindi solo i primi 10 libri con la valutazione media più alta vengano restituiti come risultato.

- Lista dei 10 libri con valutazione media più alta che abbiano come genere, i generi facenti parte delle librerie dell'utente [se l'utente ha almeno un libro nella libreria]
  Funzione **topRatingBooksBasedOnUserShelves** in back-end/controllers/bookController.js

```
export const topRatingBooksBasedOnUserShelves = async (req, res) => {
    const { username } = req.body;

    try {
        // Trova l'utente basato sullo username
        const user = await UserModel.findOne({ username });
        if (!user) {
            return res.status(404).json({ success: false, error: 'User not found' });
        }

        // Trova le shelf dell'utente
        const shelves = await shelfModel.findOne({ user: user._id });
        //se non ha trovato nessuna shelves (utente non ha mai salvato libro)
        if(!shelves){
            return res.json({
                success: false,
                message: 'No books found in user shelves'
            });
        }

        // Ottieni tutti i generi dei libri presenti nelle shelf dell'utente
        let genres = [];

        // Trova i libri nelle due mensole dell'utente
        const readBooks = await bookModel.find({ _id: { $in: shelves.books_read } });
        const toReadBooks = await bookModel.find({ _id: { $in: shelves.books_to_read } });

        if(readBooks.length + toReadBooks.length === 0){
            return res.json({
                success: false,
                message: 'No books found in user shelves'
            });
        }

        //Concateno i generi dei libri trovati
        readBooks.forEach(book => {
            genres = genres.concat(book.genres);
            console.log("genres books_read: "  + genres);
        });
```

const readBooks = await bookModel.find({ _id: { $in: shelves.books_read } });
cerca tutti i libri che sono stati contrassegnati come letti dall'utente.

shelves.books_read contiene un array di ID dei libri che l'utente ha contrassegnato come letti. Quindi la query utilizza l'operatore $in per trovare tutti i documenti nel modello bookModel il cui _id è presente nell'array shelves.books_read.

Lo stesso viene fatto per toReadBooks.

```
    toReadBooks.forEach(book => {
        genres = genres.concat(book.genres);
        console.log("genres books_to_read: "  + genres);
    });

    // Rimuovi i duplicati dai generi
    genres = [...new Set(genres)];

    // Trova i libri che hanno almeno uno dei generi trovati nelle shelf dell'utente
    const topRatingPipeline = getTopRatingBooksAggregationPipeline();
    const pipeline = [
        {
            $match: { genres: { $in: genres } }
        },
        ...topRatingPipeline
    ];
    const books = await bookModel.aggregate(pipeline);

    if (books.length > 0) {
        return res.json({
            success: true,
            message: 'Top 10 books sorted by highest rating based on user shelves genres',
            books: books
        });
    } else {
        return res.json({
            success: false,
            message: 'No books found based on user shelves genres'
        });
    }
} catch (error) {
    console.error('Error while fetching top rated books based on user shelves genres:', error);
    return res.status(500).json({
        success: false,
        message: 'Error while fetching top rated books based on user shelves genres'
    });
}
};
```

genres è un array che contiene i generi dei libri presenti nelle librerie dell'utente. Questi generi sono stati ottenuti iterando sui libri letti e da leggere dell'utente e recuperando i generi di ciascun libro.

Viene riutilizzata la funzione getTopRatingBooksAggregationPipeline() definita prima, che restituisce una serie di operazioni di aggregazione per calcolare la media delle valutazioni e ordinare i libri per valutazione media in ordine decrescente, limitandoli ai primi 10.

1. $match:
   Filtra i documenti nel modello bookModel in base al campo genres. L'operatore $in è usato per verificare se il campo genres di ciascun libro contiene almeno uno dei generi presenti nell'array genres ottenuto dalle librerie dell'utente.

2. ...topRatingPipeline:
   Aggiunge a pipeline, oltre che il match anche le operazioni di aggregazione definite in getTopRatingBooksAggregationPipeline().

- Ricerca libro attraverso titolo [possibilità di ordinare i libri per titolo, autore, numero di pagine]
  Funzione **searchBooks** in back-end/controllers/bookController.js

```
export const searchBooks = async (req, res) => {
    const limit = 15;
    const { query, page = 1, orderBy } = req.query;

    try {
        const skip = (page - 1) * limit;

        let sortCriteria = {};
        if (orderBy === 'title') {
            sortCriteria = { title: 1 }; // Ordina per titolo in ordine crescente
        } else if (orderBy === 'author') {
            sortCriteria = { authors: 1 }; // Ordina per autore in ordine crescente
        } else if (orderBy === 'pages') {
            sortCriteria = { pages_number: 1 }; // Ordina per numero di pagine in ordine crescente
        } else {
            sortCriteria = { title: 1 }; // Ordine per titolo di default
        }

        const books = await bookModel.find({
            title: { $regex: query, $options: 'i' }  // 'i' per rendere la ricerca case-insensitive
        }).skip(skip)
        .limit(limit)
        .sort(sortCriteria); //criterio di ordinamento

        // Conta il totale dei libri che corrispondono alla query
        const totalBooks = await bookModel.countDocuments({
            title: { $regex: query, $options: 'i' }
        });

        console.log("libri trovati per [" + query + "] : "+ books.length );
        return res.json({
            success: true,
            books,
            totalBooks,
            totalPages: Math.ceil(totalBooks / limit),
            currentPage: parseInt(page)
        });
    } catch (error) {
        console.error('Error while searching for books:', error);
        return res.status(500).json({
            success: false,
            message: 'Error while searching for books'
        });
    }
```

Questa funzione è progettata per gestire la ricerca dei libri, con la possibilità di paginare i risultati per mostrare solo una porzione dei libri per volta sulla pagina web.

1. Parametri di input:
   a. query: È il termine di ricerca inserito dall'utente.
   Quando viene eseguita la query bookModel.find({ title: { $regex: query, $options: 'i' } }), MongoDB cercherà tutti i documenti in bookModel in cui il campo title contiene la sequenza di caratteri inserita dall'utente (query), ignorando le differenze tra maiuscole e minuscole (i).
   b. page: Indica la pagina dei risultati da visualizzare. Di default la prima pagina è 1.
   c. orderBy: Specifica il criterio di ordinamento dei libri. Può essere per titolo, autore o numero di pagine. Se non è specificato, l'ordinamento predefinito è per titolo.

2. skip calcola il numero di documenti da saltare prima di iniziare a restituire i risultati. Questo è calcolato in base al numero della pagina corrente (page) e al limite di libri per pagina (limit).

3. sortCriteria determina come i risultati della ricerca saranno ordinati in base al valore di orderBy. Se orderBy è "title" ad esempio, allora i libri saranno ordinati per titolo.

4. totalBooks conta il numero totale di libri che corrispondono alla query senza limitazione di paginazione. Questo ci serve per calcolare dopo il numero totale di pagine totalPages che l'utente potrà sfogliare, basato sul numero totale di libri trovati e sul limite per pagina.

- Aggiunta/Rimozione di un libro alla propria libreria
  Funzione **addBookToShelf** in back-end/controllers/shelfController.js

```
export const addBookToShelf = async (req, res) => {
    const { username, id_book, type } = req.body; // type può essere 'read' o 'to-read'

    try {
        // Trova l'utente basato sullo username
        const user = await UserModel.findOne({ username });
        if (!user) {
            return res.status(404).json({ success: false, error: 'User not found' });
        }

        // Verifica se il libro esiste
        const book = await bookModel.findOne({ id_book });
        if (!book) {
            return res.json({ success: false, error: 'Book not found' });
        }

        // Trova o crea la mensola per l'utente
        let shelf = await shelfModel.findOne({ user: user._id });
        if (!shelf) {
            shelf = new shelfModel({ user: user._id, books_read: [], books_to_read: [] });
        }

        // Determina quale array aggiornare
        let targetArray, otherArray;
        if (type === 'read') {
            targetArray = 'books_read';
            otherArray = 'books_to_read';
        } else if (type === 'to-read') {
            targetArray = 'books_to_read';
            otherArray = 'books_read';
        } else {
            return res.status(400).json({ success: false, error: 'Invalid type' });
        }
```

```
        // Verifica se il libro è già nella mensola richiesta
        const bookIndexInShelf = shelf[targetArray].indexOf(book._id);
        if (bookIndexInShelf !== -1) {
            // Se il libro è già presente nella mensola, rimuovilo
            shelf[targetArray].splice(bookIndexInShelf, 1);
            await shelf.save();
            return res.json({
                success: true,
                message: 'Book removed from shelf successfully',
                removed: true,
                shelf
            });
        }

        // Rimuovi il libro dall'altra mensola se presente
        shelf[otherArray] = shelf[otherArray].filter(bookId => !bookId.equals(book._id));

        // Aggiungi il libro alla mensola richiesta
        shelf[targetArray].push(book._id);
        await shelf.save();

        return res.json({
            success: true,
            message: 'Book added to shelf successfully',
            removed: false,
            shelf
        });

    } catch (error) {
        console.error('Error adding book to shelf:', error);
        return res.status(500).json({ success: false, error: 'Internal server error' });
    }
};
```

- Lista libri presenti nelle librerie dell'utente
  Funzione **getBooksByUserShelves** in back-end/controllers/shelfController.js

```
//lista dei libri nelle shelves dell'utente
export const getBooksByUserShelves = async (req, res) => {
    const { username } = req.body;

    try {
        // Trova l'utente basato sullo username
        const user = await UserModel.findOne({ username });
        if (!user) {
            return res.status(404).json({ error: 'User not found' });
        }

        // Trova la mensola dell'utente
        const shelf = await shelfModel.findOne({ user: user._id });

        if (!shelf) {
            return res.json({
                success: false,
                message: 'No shelves found for user'
            });
        }

        // Trova i libri nelle due mensole dell'utente
        const readBooks = await bookModel.find({ _id: { $in: shelf.books_read } });
        const toReadBooks = await bookModel.find({ _id: { $in: shelf.books_to_read } });

        return res.json({
            success: true,
            books_read: readBooks,
            books_to_read: toReadBooks
        });

    } catch (error) {
        console.error('Error fetching books by user shelves:', error);
        return res.status(500).json({ success: false, error: 'Internal server error' });
    }
};
```

- Aggiunta/Rimozione/Aggiornamento di una valutazione ad un libro
  Funzione **saveReview** in back-end/controllers/reviewController.js

```
export const saveReview = async (req, res) => {
    const { username, id_book, rating } = req.body;

    try {
        // Trova l'utente basato sullo username
        const user = await UserModel.findOne({ username });
        if (!user) {
            return res.status(404).json({ success: false, error: 'User not found' });
        }

        // Verifica se il libro esiste
        const book = await bookModel.findOne({ id_book });
        if (!book) {
            return res.json({ success: false, error: 'Book not found' });
        }

        // Verifica se l'utente ha già recensito questo libro
        let existingReview = await reviewModel.findOne({ user: user._id, book: book._id });

        // Se esiste già una recensione dell'utente per questo libro
        if (existingReview) {
            // Controlla se la nuova valutazione è diversa da quella esistente
            if (rating !== null && existingReview.rating !== rating) {
                // Aggiorna la recensione con la nuova valutazione
                const oldRating = existingReview.rating;
                existingReview.rating = rating;
                const updatedReview = await existingReview.save();

                // Aggiorna il conteggio delle stelle nel libro
                book[`number_stars_${oldRating}`] -= 1; // Decrementa il vecchio punteggio
                book[`number_stars_${rating}`] += 1;   // Incrementa il nuovo punteggio
                await book.save();

                return res.json({
                    success: true,
                    removed: false,
                    message: 'Review updated successfully',
                    review: updatedReview
                });
            } else if (rating === null) {
                // Rimuovi la recensione
```

```
        } else if (rating === null) {
            // Rimuovi la recensione
            await reviewModel.deleteOne({ _id: existingReview._id });

            // Decrementa il conteggio delle stelle nel libro
            book[`number_stars_${existingReview.rating}`] -= 1;
            await book.save();

            return res.json({
                success: true,
                removed: true,
                message: 'Review removed successfully'
            });
        } else {
            //non fa nulla in altri casi
            return res.json({
                success: true,
                removed: false,
                message: 'Review rating unchanged'
            });
        }
    } else {
        // Se l'utente non ha ancora recensito questo libro, crea una nuova recensione
        const newReview = new reviewModel({
            user: user._id,
            book: book._id,
            rating
        });

        const savedReview = await newReview.save();

        // Aggiorna il conteggio delle stelle nel libro
        book[`number_stars_${rating}`] += 1;
        await book.save();

        return res.json({
            success: true,
            removed: false,
            message: 'Review saved successfully',
            review: savedReview
        });
    }
} catch (error) {
    console.error('Error saving/retrieving review:', error);
```

**(rating == null) si riferisce al caso in cui ad esempio un utente che aveva inserito 3 stelle ad un libro, riclicca sulle 3 stelle, quindi in questo caso, la review viene rimossa.

Connessione al database:

```
import mongoose from "mongoose"

const project_name = "worldbookshelf_db";

const string = "mongodb://127.0.0.1:27017/" + project_name;


export const connectDB = async () => {
    try {
        await mongoose.connect(string);
        console.log("DB connected");
    } catch (error) {
        console.error("Error connecting to MongoDB:", error.message);
    }
};
```

2. **frontend**
   è stato utilizzato **React.js** e altre librerie come **Bootstrap**, **MUI** e **FontAwesome** per realizzare la parte grafica della WebApplication. Le richieste dirette al backend vengono effettuate utilizzando **axios**.

   Contenuto pagine:
   - Nella *pagina di login* un utente può loggarsi alla piattaforma
   - Nella *pagina di registrazione* un utente può registrarsi alla piattaforma

- Nella *home page di un utente non loggato* vengono visualizzati i 10 libri che hanno la valutazione media più alta.
- Nella *home page di un utente loggato* vengono visualizzati i 10 libri consigliati
- Nella *pagina Account* vengono visualizzate le informazioni dell'utente come: username, nome, cognome. Vengono poi visualizzate le due librerie dell'utente: "want to read" e "read", e i libri a cui l'utente ha aggiunto una review.
- Nella *pagina relativa ad un libro* vengono visualizzate le informazioni del libro, come: titolo, autore, immagine di copertina, descrizione, generi, anno pubblicazione, publisher, lingua, numero di pagine, la media delle valutazioni, numero di stelle. Un utente in questa pagina può lasciare una valutazione al libro e/o salvarlo in una delle sue due librerie.
- Nella *pagina di ricerca* di un libro vengono visualizzati i libri che contengono nel titolo l'input inserito dall'utente.