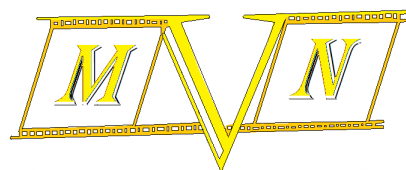




**UNIVERSITÀ
DI TRENTO**

**Dipartimento di Ingegneria e
Scienza dell'Informazione**



Progetto:

MoViewerNet

Titolo del documento:

Sviluppo Sito Web

Gruppo T47, AA 2022/2023

Matteo Javid Battista, Aurora Ottaviani, Alessandro Nitti

“MoViewerNet”

Indice:

User Flows	3.
Implementazione e Documentazione	5.
Documentazione API	25.
Implementazione Frontend	27.
Repository GitHub	30 .
Testing	32.

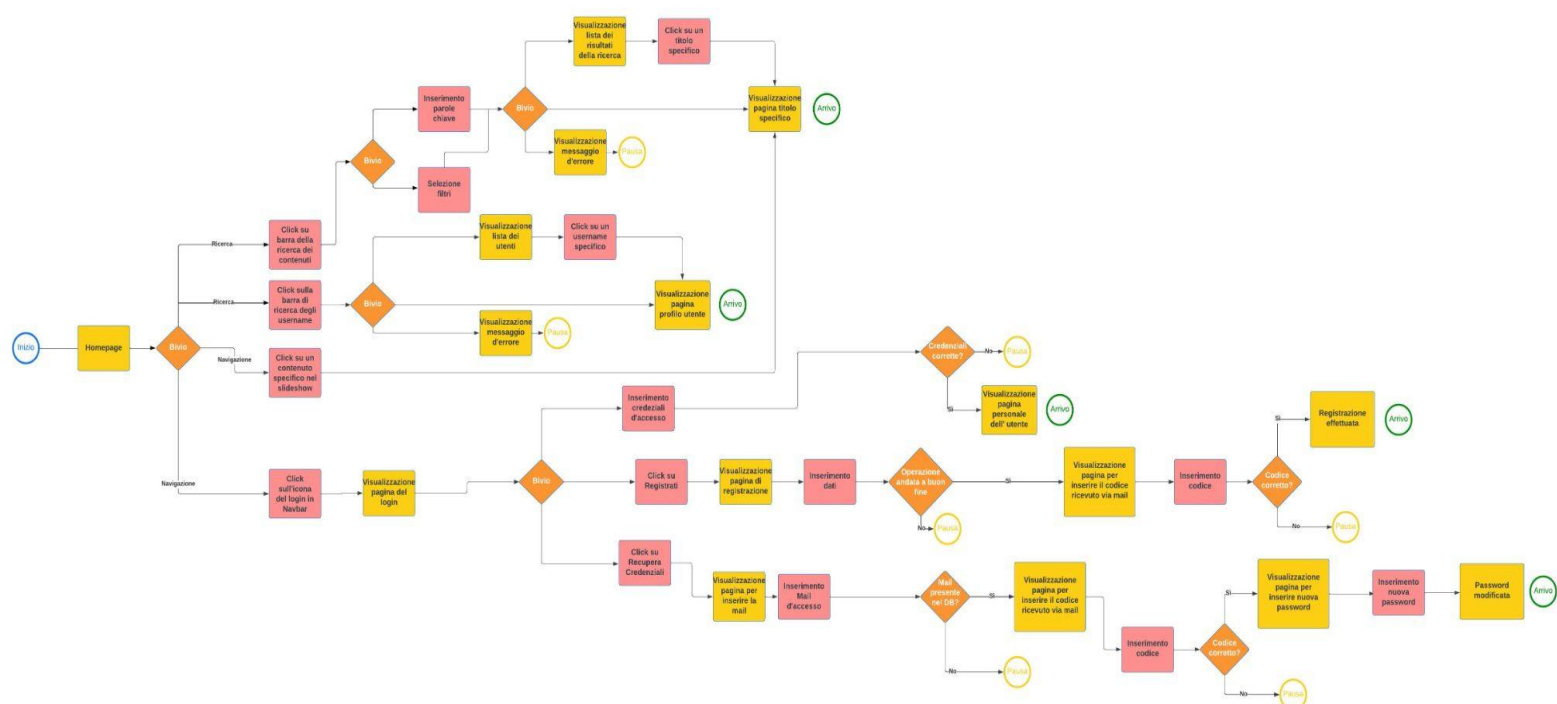
User Flows :

Gli user flow riportati qui di seguito rappresentano le varie azioni che gli utenti di ogni tipologia possono intraprendere sul sito di MVN; per chiarezza e semplicità sono divisi in base alla tipologia di utente, rispettivamente: utente generico e utente loggato. Infine è riportata subito qui sotto anche una legenda per i simboli e le figure presenti negli user flow.

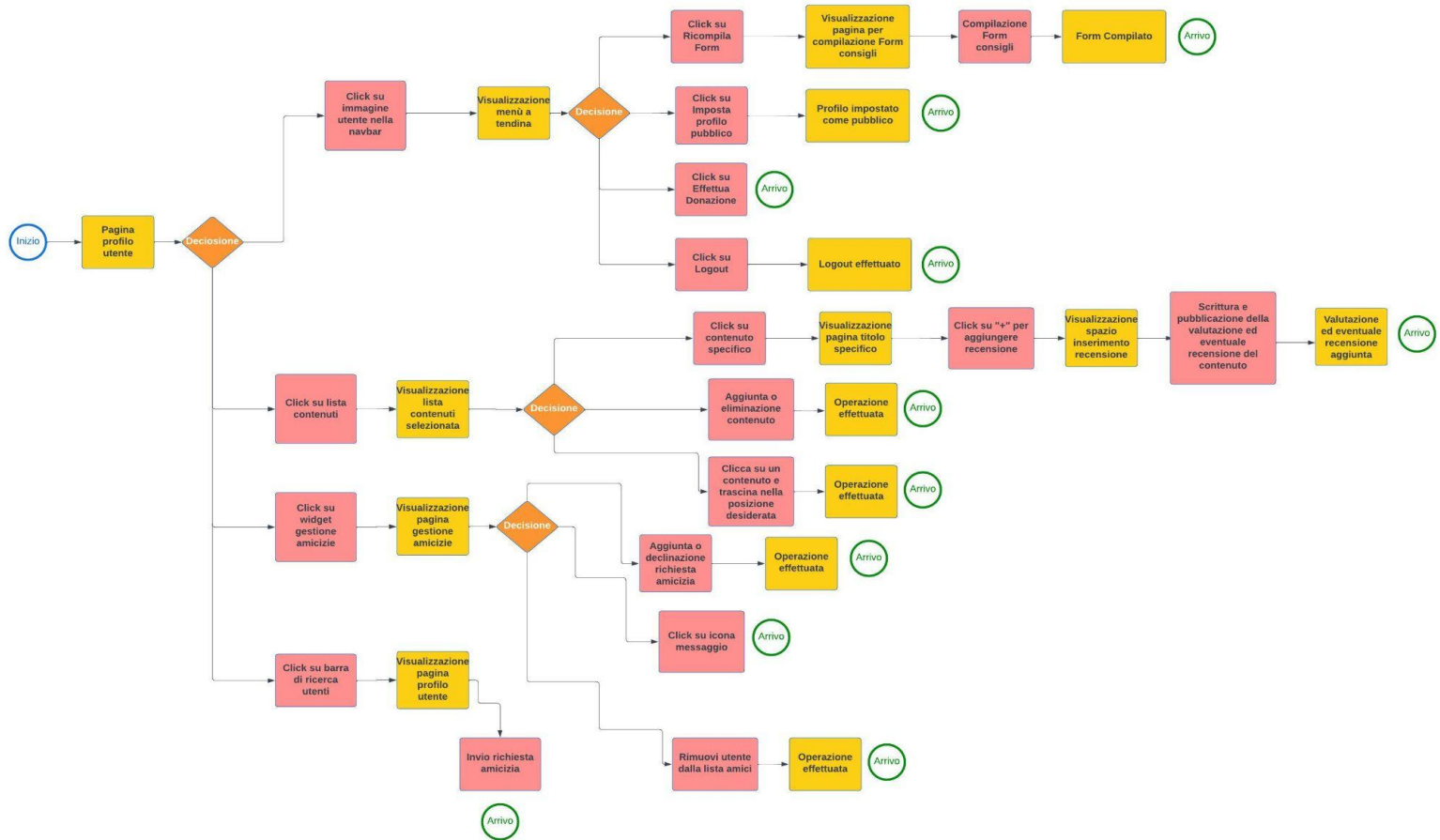


Si allega il [link](#) alla pagina del diagramma originale per migliore lettura.

USER FLOW - UTENTE GENERICO:



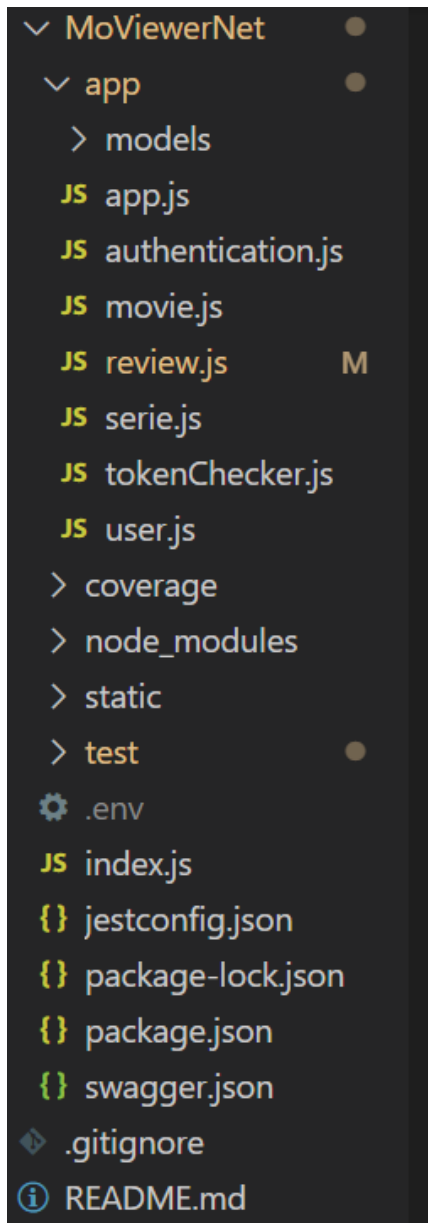
USER FLOW - UTENTE LOGGATO:



Implementazione e documentazione:

1. Struttura del progetto.

Si riporta un'immagine raffigurante la struttura del progetto, dove si può osservare che la cartella MoViewerNet è suddivisa in 5 sottocartelle principali:



- **app:**
in questa cartella sono presenti:
 - la cartella models, contenente i modelli (file .js) utilizzati per rappresentare utenti, film, serie tv e recensioni all'interno della piattaforma;
 - il file app.js, dove sono contenute tutte le definizioni utili, e soprattutto le route delle API
 - il file authentication.js, dove è definita l'API per il login di un utente alla piattaforma. Questa API è separata dalle altre in quanto differente, siccome effettua l'assegnazione del token;
 - i file movie.js, review.js, serie.js e user.js, dove sono definite tutte le API sviluppate
 - il file tokenChecker.js, dove è definita la funzione che controlla l'esistenza e la correttezza dei token. Viene chiamata prima dell'esecuzione di alcune API riservate agli utenti loggati
- **coverage:** cartella contenente file prodotti dalla libreria jest. Quest'ultima contiene dati utili riguardanti lo svolgimento dei casi di test definiti
- **node_modules:** cartella contenente tutte le dipendenze scaricate
- **static:** cartella contenente un file index.html e un file script.js (gli elementi che definiscono la parte di frontend). Nel primo è definita l'interfaccia utente, mentre nel secondo sono definiti gli script che permettono di far eseguire le API all'utente e fargli visualizzare i risultati a schermo
- **test:** cartella contenente dei file .js dove sono definiti i vari casi di test implementati (per eseguire i test, si può scrivere a terminale il comando `npm run test`)

Segnaliamo, inoltre, la presenza dei seguenti file:

- .env: file segreto contenente le variabili d'ambiente del sistema
- index.js: file del progetto all'interno del quale è implementata la connessione al database
- jestconfig.json: file di configurazione per la libreria jest
- package-lock.json: file contenente dati riguardanti tutte le dipendenze delle librerie installate e la loro versione
- package.json: file contenente i dati chiave del progetto. All'interno sono definiti anche gli script di esecuzione
- swagger.json: file con all'interno definita la documentazione delle API
- .gitignore: file con all'interno i nomi dei file e delle cartelle che non devono essere sottoposti a controllo di revisione
- README.md: file contenente una descrizione del progetto, utile ai nuovi utenti per orientarsi e sapere come eseguire il progetto

2. Project Dependencies.

Sono stati utilizzati i seguenti moduli node per lo sviluppo del sito web:

```
cors: version 2.8.5,  
dotenv: version 16.0.3,  
express: version 4.18.2,  
jsonwebtoken: version 9.0.0,  
mongoose: version 6.8.1,  
mongoose-express-api: version 0.0.3,  
open: version 8.4.0,  
swagger-ui-express: version 4.6.0  
jest: version 29.4.2,  
supertest: version 6.3.3
```

3. Project Data or DB

Per gestire i dati del sito web, si è utilizzato MongoDB.

Sono state definite tre collezioni principali: “User”, “Movie”, “Serie” e “Review”, visibili nell’immagine sottostante:

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
movies	4	990B	248B	36KB	1	36KB	36KB
reviews	7	1.02KB	150B	36KB	1	36KB	36KB
series	3	669B	223B	36KB	1	20KB	20KB
users	5	575B	115B	36KB	1	36KB	36KB

- “Movie” rappresenta i film presenti nel database.
Si riporta un esempio di film:

```

_id: ObjectId('63ee0ada6d1663df87bddeb0')
titolo: "quo vado"
regista: "gennaro nunziante"
etaCons: "6"
copertina: "https://www.vicini.to.it/wp-content/uploads/2016/09/Quo-Vado-esordio-
durata: "86"
  ✓ generi: Array
    0: "comico"
  ✓ piattaforme: Array
    0: "netflix"
  __v: 0

```

- “Serie” rappresenta le serie che sono presenti sul sito MVN.
Si riporta un esempio dei dati per un film:

```

_id: ObjectId('63ee10f5988344258cbc310f')
titolo: "brooklyn 99"
regista: "mike schur"
etaCons: "6"
copertina: "https://i0.wp.com/www.meganerd.it/wp-content/uploads/2021/02/b99.jpg?f..."
  ✓ generi: Array
    0: "poliziesco"
    1: "comico"
  ✓ piattaforme: Array
    0: "netflix"
  > stagioni: Array
  valutazione: 3.5

```

- “User” rappresenta gli utenti registrati nel database
Si riporta un esempio dei dati per un film:

```

_id: ObjectId('63ea1fc445859c9a9c196e4d')
mail: "pluto@gmail.com"
username: "pluto"
password: "password"
isPrivate: true

```


- “Review” rappresenta le recensioni ai contenuti del database
Si riporta un esempio dei dati di una recensione:

```
_id: ObjectId('63ee13f56d1663df87bdded8')
titolo: "brooklyn 99"
regista: "mike schur"
mailAutore: "pluto@gmail.com"
voto: 5
testo: "wow, i enjoyed this sitcom so much!"
__v: 0
```

4. Project APIs

4.1) Estrazione delle risorse dal Class Diagram: analizzando il Class Diagram sono state rilevate 4 risorse principali: Utente, Film, Serie TV, Recensioni:

L'**Utente** presenta i seguenti metodi:

- registrazione (corrispondente a "Sign Up" nello sviluppo): è un metodo POST che prende in input i dati dell'utente (ossia mail, username, password e passwordSupp) e crea la risorsa Utente;
- login (corrispondente ad "Authentication" nello sviluppo): è un metodo POST che prende in input username/mail e password, e crea la risorsa Utente;
- impostaPrivacyProfilo (corrispondente a "SetMyPrivacy" nello sviluppo): è un metodo PATCH che prende in input una mail e un attributo booleano, e modifica l'attributo isPrivate dell'utente in questione;
- dona (corrispondente a "Donation" nello sviluppo): è un metodo GET che apre un link alla pagina del servizio esterno PayPal, che prende in carico la donazione agli sviluppatori
- searchUser (corrispondente a "Find" nello sviluppo): è un metodo GET che prende in input una stringa rappresentante l'username e ritorna almeno una risorsa Utente se trova corrispondenza (se la stringa in questione è "Leonardo", verranno restituiti tutti gli User che contengono "Leonardo" nell'username

I **Film** usano i seguenti metodi:

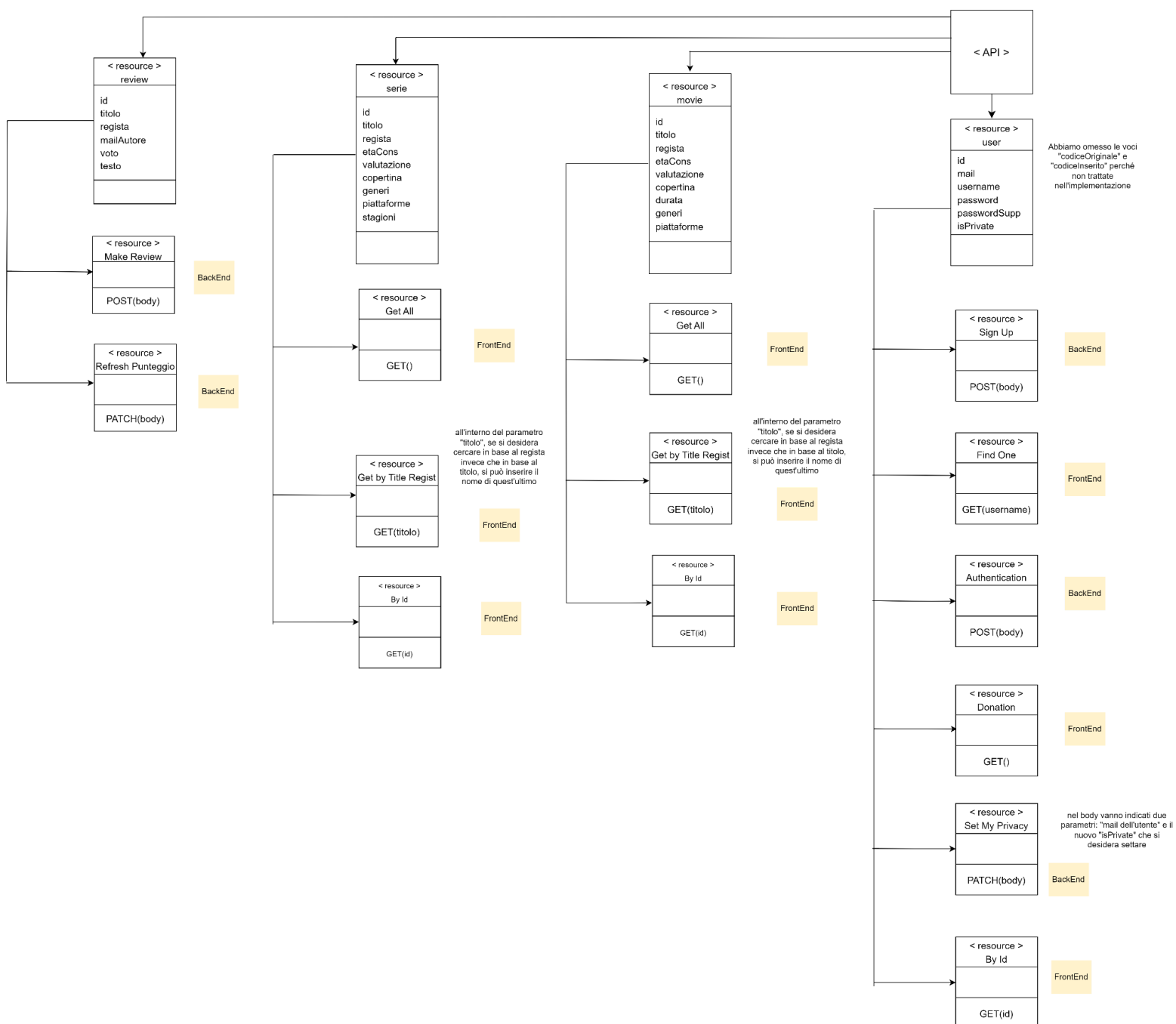
- searchTitleOrReg (corrispondente a "Get by Title Regist" nello sviluppo): è un metodo GET che prende in input la stringa inserita nella barra di ricerca e ritorna una o più risorse Film in caso di corrispondenza: se la stringa passata è "steven", verranno restituiti tutti i film che nel titolo contengono la parola steve, o hanno un regista di nome steven

Le **Serie** usano i seguenti metodi:

- searchTitleOrReg (corrispondente a "Get by Title Regist" nello sviluppo): è un metodo GET che prende in input la stringa inserita nella barra di ricerca e ritorna una o più risorse Serie. La logica è la stessa del metodo descritto subito sopra per i film

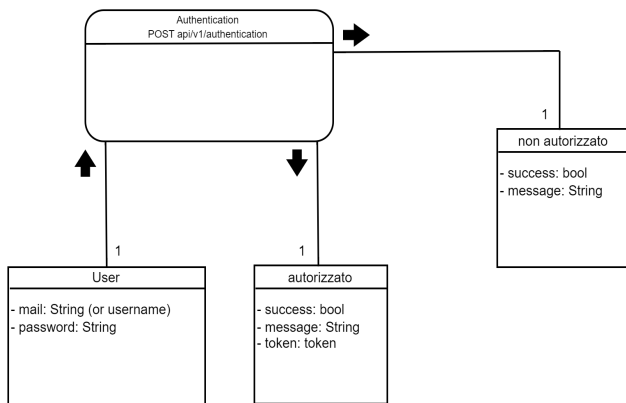
Le **Recensioni** usano i seguenti metodi:

- calcolaMediaPunteggio (corrispondente a “Refresh Punteggio” nello sviluppo): metodo PATCH che prende in entrata una risorsa Film o Serie (passando titolo e regista) e ne modifica l’attributo valutazione;
- faRecensione (corrispondente a “Make Review” nello sviluppo): è un metodo POST, che in entrata prende tutti i dati della recensione (titolo, regista, mailAutore, voto e testo facoltativo) e con questi crea la risorsa recensione corrispondente.



4.2) Modelli di Risorse: nelle didascalie a seguire si vogliono descrivere le funzionalità dei modelli di risorse utilizzati:

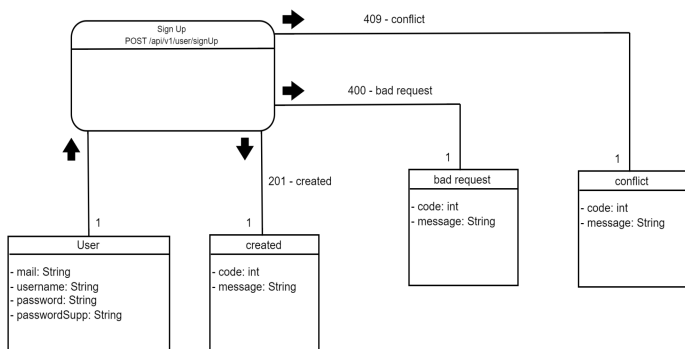
Authentication:



Il metodo Authentication prende in input un'istanza di User con mail (o username) e Password. Se il controllo sulle credenziali va a buon fine, viene creata e restituita la risorsa token, che sfrutta la mail dell'utente, assieme a una variabile success contenente true e un messaggio.

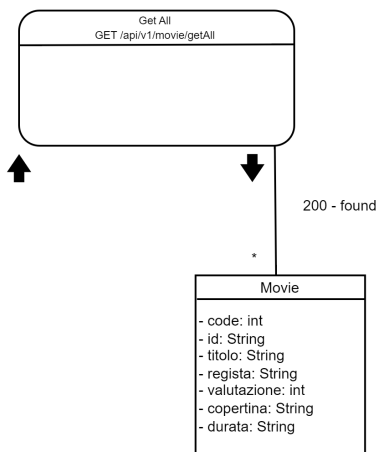
Nel caso in cui, invece, le credenziali inserite non trovino corrispondenza, non si ottiene l'autorizzazione necessaria e viene restituita la variabile success contenente false e il messaggio di errore

Sign Up:



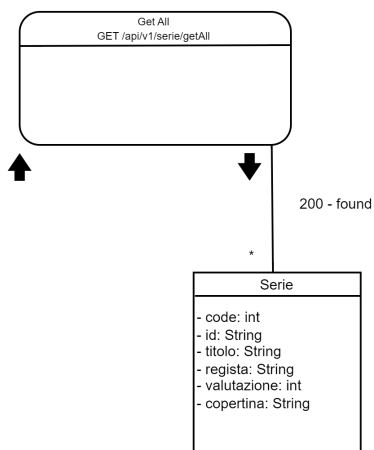
Quando un utente si registra, viene eseguita una POST dei dati dell'utente, che crea la risorsa utente e salva i dati (status code: 201), se l'operazione va a buon fine.

Diversamente, questo processo può non avere successo nel caso di una richiesta malformata (status code: 400) o di un conflitto nel caso di credenziali già presenti nel database (status code: 409).



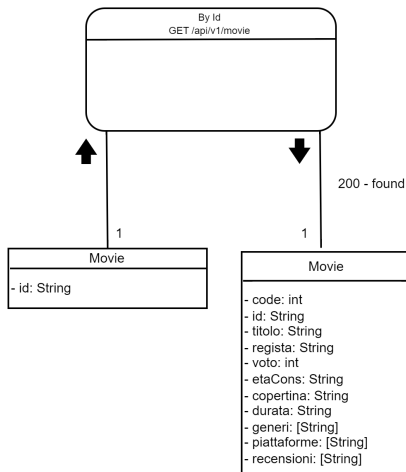
Get All (movie):

Questa funzione non ha un input, e come unica uscita ha tutti i film presenti nel database di MVN, assieme al codice di stato 200



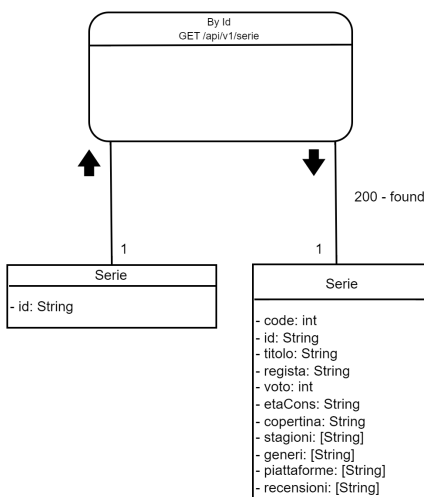
Get All (serie):

Questa funzione lavora in maniera del tutto simile a quella descritta subito sopra, ma al posto di restituire tutti i film, restituisce tutte le serie TV presenti nel database (assieme al solito status code 200)



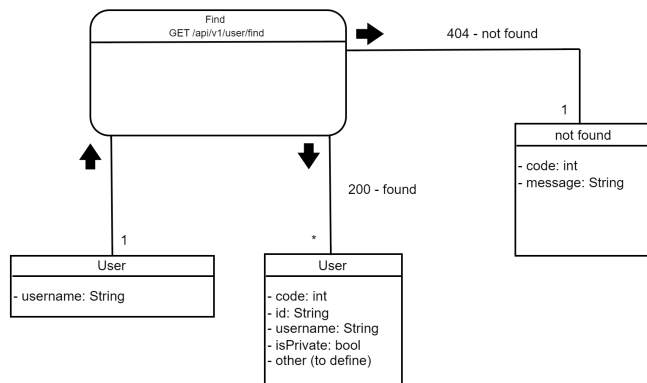
By Id (movie):

Metodo get che cerca nel database il film con l'id passato in input. Restituisce lo status code 200



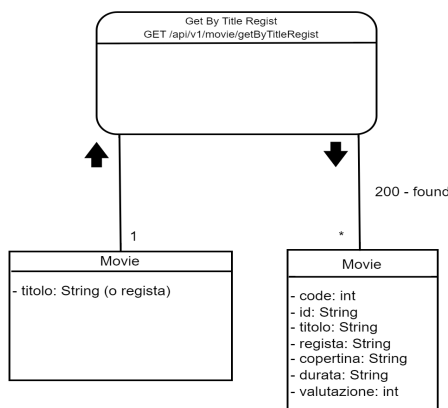
By Id (serie):

Metodo get equivalente a quello descritto sopra, che cerca nel database il film con l'id passato in input. Restituisce lo status code 200 anch'esso



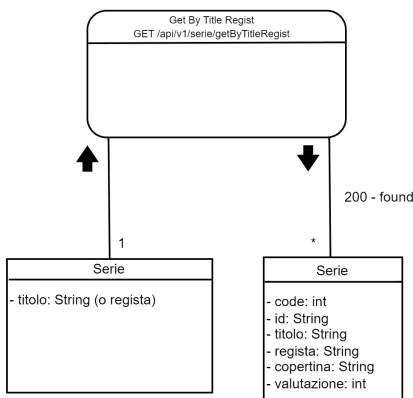
Find:

Tramite un metodo GET questa funzione ritorna, in base all'input di una stringa contenente l'username desiderato, tutti gli user che contengono il testo contenuto nella stringa in questione. Viene restituito lo status code: 200 se qualcosa è stato trovato, altrimenti lo (status code: 404)



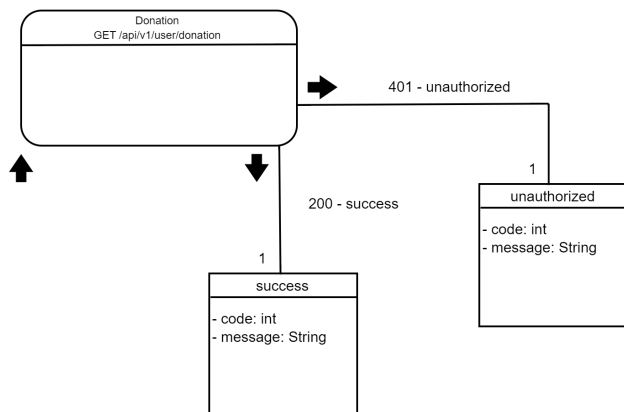
Get By Title Regist (movie):

Un metodo GET che, preso in input un parametro rappresentante titolo o regista, restituisce tutti i film che contengono quella stringa nel nome oppure nel regista, oppure un array vuoto. Viene settato a prescindere lo status code 200



Get By Title Regist (serie):

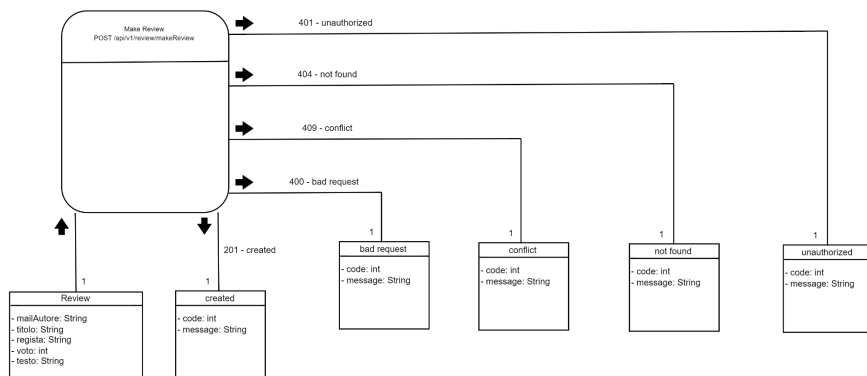
Un metodo GET che, preso in input un parametro rappresentante titolo o regista, restituisce tutte le serie che contengono quella stringa nel nome oppure nel regista, oppure un array vuoto. Viene settato a prescindere lo status code 200



Donation:

Questa funzione GET non ha un input, e serve a reindirizzare l'utente verso il servizio esterno di PayPal che poi prenderà in carico la donazione effettiva. Restituisce lo status code: 200 se tutto va a buon fine, oppure lo status code: 401 se l'utente non possiede un token valido

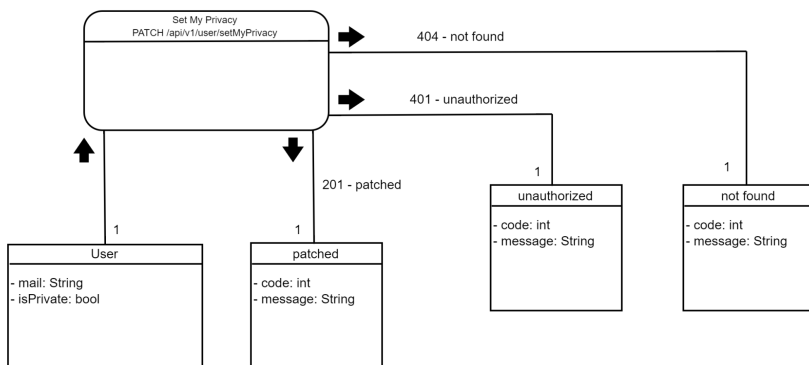
Make Review:



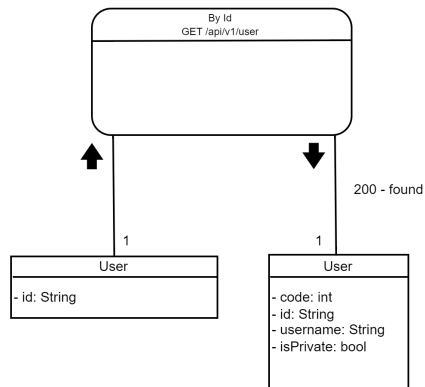
Questo metodo tramite una POST crea una nuova risorsa Recensione, in base ai dati inseriti (mail autore, titolo, regista, voto e testo). Se tutto va a buon fine, viene restituito lo status code 201. Se invece non esiste l'utente autore, o il contenuto in questione, viene restituito lo status code 404. Se il voto non rientra nel range imposto viene restituito il codice

400, mentre se l'utente prova a fare per la seconda volta una recensione sullo stesso titolo viene restituito il codice 409. Se questa API viene chiamata da un utente senza un token valido, viene restituito il codice di stato 401

Set My Privacy:

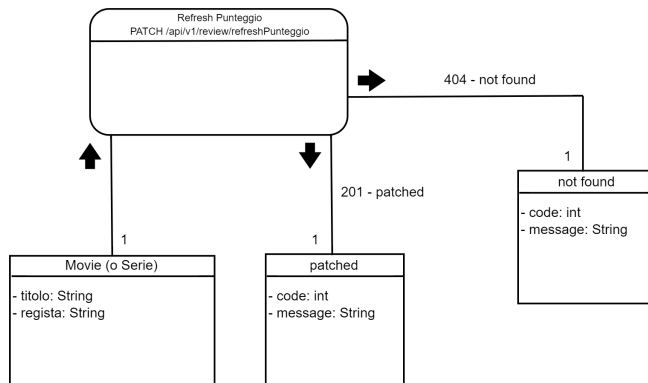


Questo è un metodo PATCH che va a modificare l'attributo isPrivate. Se tutto va bene, viene restituito lo status code 201. Se invece il token è malformato o non c'è viene mostrato un messaggio d'errore (status code: 401), altrimenti se non si trova l'utente passato in input, si cade in not found (status code: 404)



By Id (user):

Tale metodo GET cerca nel database l'utente con id uguale a quello passato in input, e restituisce i dati dell'utente in questione



Refresh Punteggio:

Questo metodo PATCH modifica (in seguito ad un input di Movie o Serie) il voto relativo alla risorsa stessa ricalcolando la media dei punteggi delle recensioni (status code: 201). Se non trova la coppia un contenuto con dato titolo e regista restituisce un not found, con codice di stato 404. Questa API andrebbe chiamata ogni volta che viene inserita una recensione

5. Sviluppo API

API RELATIVE AI FILM:

```
router.get('/getAll', async (req, res) => {
  let myMovies = await Movie.find({});
  myMovies = myMovies.map( (myMovies) => {
    return {
      self: '/api/v1/movie/' + myMovies.id, //dati miniatura
      titolo: myMovies.titolo,
      regista: myMovies.regista,
      valutazione: myMovies.valutazione,
      copertina: myMovies.copertina,
      durata: myMovies.durata,
    };
  });
  res.status(200).json(myMovies); //200 found
});
```

Questa API ha lo scopo di restituire tutti i film contenuti nel database a prescindere dal loro titolo o dal loro regista, restituendo un array contenente per ciascun elemento i dati sufficienti alla rappresentazione della miniatura

```
router.get('/getTitleRegist/:parametro', async (req, res) => {
  let myMovies = await Movie.find({$or: [{ titolo: {"$regex":
req.params.parametro.toLowerCase(), "$options": "i"} }, { regista:
{"$regex": req.params.parametro.toLowerCase(), "$options": "i"} }]});
  myMovies = myMovies.map( (myMovies) => {
    return {
      self: '/api/v1/movie/' + myMovies.id, //dati miniaturz
      titolo: myMovies.titolo,
      regista: myMovies.regista,
      valutazione: myMovies.valutazione,
      copertina: myMovies.copertina,
      durata: myMovies.durata,
    };
  });
  res.status(200).json(myMovies); //200 found
});
```

API implementata in modo che possa restituire tutti i film contenuti nel database che contengono nel titolo o nel regista il parametro (stringa) passato nell'header. Anche questa restituisce solo i dati necessari a mostrare la copertina

```
router.get('/:id', async (req, res) => {
  let myMovie = await Movie.findById(req.params.id);
  res.status(200).json( { //200 success
    self: '/api/v1/movie/' + myMovie.id,
    titolo: myMovie.titolo,
    regista: myMovie.regista,
    etaCons: myMovie.etaCons,
    valutazione: myMovie.valutazione,
    copertina: myMovie.copertina,
    generi: myMovie.generi,
    piattaforme: myMovie.piattaforme,
    durata: myMovie.durata,
  }); });
```

Questa API ha invece lo scopo di restituire un film in base al suo id, passato nell'header. A differenza di quelle precedenti, questa API restituisce tutti i dati relativi al film in questione, non solo quelli sufficienti per mostrare la miniatura

API DELLE SERIE:

```
router.get('/getAll', async (req, res) => {
  let mySeries = await Serie.find({});
  mySeries = mySeries.map( (mySeries) => {
    return {
      self: '/api/v1/serie/' + mySeries.id, //dati miniatura
      titolo: mySeries.titolo,
      regista: mySeries.regista,
      valutazione: mySeries.valutazione,
      copertina: mySeries.copertina
    };
  });
  res.status(200).json(myMovies); //200 found
});
```

Questa API ha lo scopo di restituire tutti e le serie contenute nel database a prescindere dal loro titolo o dal loro regista, restituendo un array contenente per ciascun elemento i dati sufficienti alla rappresentazione della miniatura

```

router.get('/getByTitleRegist/:parametro', async (req, res) => {
  let mySeries = await Serie.find({$or: [{ titolo: {"$regex":
req.params.parametro.toLowerCase(), "$options": "i"} }, { regista:
{"$regex": req.params.parametro.toLowerCase(), "$options": "i"} }]});
  mySeries = mySeries.map( (mySeries) => {
    return {
      self: '/api/v1/serie/' + mySeries.id, //dati miniaturz
      titolo: mySeries.titolo,
      regista: mySeries.regista,
      valutazione: mySeries.valutazione,
      copertina: mySeries.copertina
    };
  });
  res.status(200).json(myMovies); //200 found
});

```

API implementata in modo che possa restituire tutte le serie contenute nel database che contengono nel titolo o nel regista il parametro passato nell'header. Anche questa restituisce solo i dati necessari a mostrare la copertina

```

router.get('/:id', async (req, res) => {
  let mySerie = await Serie.findById(req.params.id);
  res.status(200).json( { //200 success
    self: '/api/v1/serie/' + mySerie.id,
    titolo: mySerie.titolo,
    regista: mySerie.regista,
    etaCons: mySerie.etaCons,
    valutazione: mySerie.valutazione,
    copertina: mySerie.copertina,
    generi: mySerie.generi,
    piattaforme: mySerie.piattaforme,
    stagioni: mySerie.stagioni
  }); });

```

Questa API ha invece lo scopo di restituire una serie in base al suo id, passato nell'header. A differenza di quelle precedenti, questa API restituisce tutti i dati relativi al film, non solo quelli sufficienti per mostrare la miniatura

API DEGLI UTENTI:

Nello sviluppo di login e registrazione non è stata implementata la crittografia delle password per questioni di tempo

```
router.post('/signUp', async (req, res) => {

  let myUser = await User.findOne({ $or: [{ mail: req.body.mail }, { username: req.body.username }] });

  if (!myUser && req.body.password==req.body.passwordSupp && req.body.password.length>=8&&req.body.username)
  {

    let newUser = new User ({
      mail: req.body.mail,
      username: req.body.username,
      password: req.body.password
      isPrivate: true, //default
    });

    if(!newUser.mail || typeof newUser.mail != 'string' || !checkIfEmailInString(newUser.mail)){
      res.status(400).json({ error: 'Quella inserita non risulta essere una mail valida' });
      console.log("Quella inserita non risulta essere una mail valida");
      return;
    }

    newUser = await newUser.save();
    let userId = newUser.id;
    res.location("/api/v1/user/" + userId).status(201).send(); //201 posted
    console.log('Utente salvato con successo nel database');
  }
  else {
    if(req.body.password != req.body.passwordSupp) {
      res.status(400).json({ error: 'Hai inserito due password diverse' }); //400 bad request
      console.log("Hai inserito due password diverse");
    } else if(req.body.password.length < 8) {
      res.status(400).json({ error: 'Hai inserito una password non conforme' }); //400
      console.log("Hai inserito una password non conforme");
    } else if(!req.body.username) {
      res.status(400).json({ error: 'Non hai inserito un username' }); //400 bad request
      console.log("Non hai inserito un username");
    }
    else {
      res.status(409).json({error:'Esiste già un utente che utilizza una di queste credenziali' });
      console.log("Nel database esiste già un utente che utilizza una di queste credenziali");
    }
  }
});
```

Questa API serve per effettuare la registrazione di un utente. Vengono effettuati anche dei controlli per:

- evitare che un utente utilizzi credenziali (mail o username) uguali a quelle di un altro utente
- fare in modo che la mail inserita sia effettivamente formattata come una mail vera e propria
- l'utente inserisca una password di conferma diversa da quella ufficiale
- l'utente inserisca una password non conforme ai criteri di sicurezza
- l'utente non lasci nessun campo vuoto

I dati: username, mail, password e passwordSup vanno passati all'interno del body

```

router.get('/find/:username', async (req, res) => {
  let myUser = await User.find({ username: {"$regex":
req.params.username, "$options": "i"} });

  if(!myUser) {
    res.status(404).json({ error: 'Non ho trovato un utente con
questo username' }); //404 not found
    console.log("Non ho trovato un utente con questo username");
    return;
  }
  myUser = myUser.map((myUser) => {
    return {
      self: '/api/v1/user/' + myUser.id,
      username: myUser.username,
      isPrivate: myUser.isPrivate
    };
  });
  res.status(200).json(myUser); //200 found
  console.log('Ho trovato un utente con questo username');
});

```

API per la ricerca di un utente in base al suo username. Quest'ultimo va passato nell'header. Se la ricerca va a buon fine, restituisce i dati pubblici di un utente

```

router.patch('/setMyPrivacy', async (req, res) => {
  let myUser = await User.findOne({ mail: req.body.mail });
  if (myUser) {
    myUser.isPrivate = req.body.isPrivate;
    myUser = await myUser.save();
    let userId = myUser.id;
    res.location("/api/v1/user/" + userId).status(201).send();
    console.log('Impostazione di privacy aggiornate con successo');
  }
  else {
    res.status(404).json({ error: 'Non esiste nel database un utente
con questo username' }); //404 not found
    console.log('Non esiste nel database un utente con questo
username');
  }
});

```

Questa API serve per modificare l'attributo di privacy di un utente. I parametri di passare all'interno del body sono mail e isPrivate (quest'ultimo di tipo boolean, corrispondente alla nuova impostazione di privacy che si vuole settare)

```
router.get('/donation', async (req, res) => {  
  
  open('https://www.paypal.com/donate/?hosted_button_id=DQ387XP5GBANN');  
  res.status(200).json({ message: 'Ho aperto la pagina per effettuare  
la donazione' }); //200 success  
  console.log("Ho aperto la pagina per effettuare la donazione");  
});
```

Questa api ha lo scopo di aprire la pagina esterna di PayPal per effettuare la donazione

```
router.get('/:id', async (req, res) => {  
  let myUser = await User.findById(req.params.id);  
  res.status(200).json( {  
    self: '/api/v1/user/' + myUser.id,  
    username: myUser.username,  
    isPrivate: myUser.isPrivate  
  });  
});
```

Questa API ha invece lo scopo di restituire i dati pubblici di un utente in base all'id, passato nell'header. Per ora restituisce solamente username e l'attributo isPrivate

API DELLE RECENSIONI:

```
router.post('/makeReview', async (req, res) => {
  let utenteAutore = await User.findOne( { mail: req.body.mailAutore } );
  let contenuto = await Movie.findOne( { titolo: req.body.titolo.toLowerCase(),
  regista: req.body.regista.toLowerCase() } );
  if(!contenuto)
    contenuto = await Serie.findOne( { titolo: req.body.titolo.toLowerCase(),
  regista: req.body.regista.toLowerCase() } );
  if(!utenteAutore || !contenuto) {
    res.status(404).json({ error: 'Questo utente o questo titolo non sono esistenti'
}); //404 not found
    console.log("Questo utente o questo titolo non sono esistenti");
    return;
  }
  if(typeof req.body.voto != "number" || req.body.voto<1 || req.body.voto>5) {
    res.status(400).json({ error: 'Il voto inserito non è accettabile' }); //400
    console.log("Il voto inserito non è accettabile");
    return;
  }

  let searchForOtherReviews = await Review.findOne( { mailAutore: req.body.mailAutore,
  titolo: req.body.titolo.toLowerCase(), regista: req.body.regista.toLowerCase() } );
  if(!searchForOtherReviews) { //User non può avere più recensioni dello stesso titolo
    let newReview = new Review ( {
      titolo: req.body.titolo.toLowerCase(),
      regista: req.body.regista.toLowerCase(),
      mailAutore: req.body.mailAutore,
      voto: req.body.voto,
      testo: req.body.testo
    });
    newReview = await newReview.save();
    let reviewId = newReview.id;
    res.location("/api/v1/review/" + reviewId).status(201).send(); //201 created
    console.log('Recensione salvata con successo');
  } else {
    res.status(409).json({ error: 'Questo utente ha già scritto una recensione per
questo titolo' }); //409 conflict
    console.log("Questo utente ha già scritto una recensione per questo titolo");
  }
});
```

Questa API ha lo scopo di pubblicare nel database una nuova recensione. Vengono prese, prima della pubblicazione:

- va verificato che l'utente autore della recensione esista nel database
- va verificato che il contenuto con titolo e regista passato esista
- il voto dato rientri nel range che va tra 1 e 5
- l'utente non abbia già recensito in passato quel contenuto

```

router.patch('/refreshPunteggio', async (req, res) => {
    let contenuto = await Movie.findOne( { titolo:
req.body.titolo.toLowerCase(), regista: req.body.regista.toLowerCase()
} );
    if(!contenuto)
        contenuto = await Serie.findOne( { titolo:
req.body.titolo.toLowerCase(), regista: req.body.regista.toLowerCase()
} );
    if(!contenuto) {
        res.status(404).json({ error: 'Non abbiamo trovato un titolo
con questi dati' }); //404 not found
        console.log("Non abbiamo trovato un titolo con questi dati");
        return;
    }

    let reviewList = await Review.find( { titolo:
req.body.titolo.toLowerCase(), regista: req.body.regista.toLowerCase()
} );
    let punteggio = 0;
    let numeroRecensioni = 0;
    reviewList = reviewList.map( (reviewList) => {
        numeroRecensioni++;
        punteggio += reviewList.voto;
    } );
    if(numeroRecensioni == 0) return ;
    punteggio = punteggio/numeroRecensioni;


    contenuto.valutazione = punteggio;
    contenuto = await contenuto.save();
    res.status(201).json({ message: 'Valutazione del titolo aggiornata
con successo' });
    console.log("Valutazione del titolo aggiornata con successo");
});

```

API che permette di aggiornare la valutazione complessiva di un titolo. Il funzionamento prevede l'estrazione di tutte le recensioni di un determinato contenuto (titolo e regista sono passati all'interno del body) e il calcolo della media dei punteggi delle recensioni. Va verificato che il titolo e il regista passati corrispondano effettivamente a un contenuto (tra film o serie)

Documentazione API :

La documentazione relativa alle API che abbiamo implementato è stata inserita all'interno del file "swagger.json". Per consultare la pagina contenente la documentazione è sufficiente scrivere: <http://localhost:8080/api-docs> dopo aver eseguito l'applicazione (di questo si parlerà più nel dettaglio dopo), Segue uno screenshot della pagina di Swagger che è stata scritta:

user APIs for the users ^	
POST	/user/signUp Insert a user in the database v
GET	/user/find Get a specific user passing the username v
PATCH	/user/setMyPrivacy Change the privacy attribute of an user v
GET	/user/donation Open the donation PayPal page v
GET	/user find a user with a specific id v
movie APIs for the movies ^	
GET	/movie/getAll Get all movies from the database v
GET	/movie/getByTitleRegist Get a list of movies with a specific title or a specific regist v
GET	/movie find a movie with a specific id v
serie APIs for the series ^	
GET	/serie/getAll Get all series from the database v
GET	/serie/getByTitleRegist Get a list of series with a specific title or a specific regist v
GET	/serie find a serie with a specific id v
review APIs for the reviews ^	
POST	/review/makeReview Insert a review in the database v
PATCH	/review/refreshPunteggio Refresh a content's score v
GET	/review/getAll Get all reviews from the database v 

Si riportano anche i modelli, con relativi attributi e tipi, degli elementi principali del sito. Si trovano anche i modelli per `miniaturaMovie` e `miniaturaSerie` (corrispondono a ciò che è visibile dei singoli contenuti quando una ricerca riporta come risultato una lista di contenuti. Guardando i mockup si può capire meglio ciò che è stato detto nell'ultima frase)

```
miniaturaMovie ▾ {  
  self  
  titolo*      string  
  regista*     string  
  copertina*   string  
  durata*      string  
  valutazione  number  
}
```

```
movie ▾ {  
  self  
  titolo*      string  
  regista*     string  
  etaCons      string  
  valutazione  number  
  copertina*   string  
  generi*      ▾ [string]  
  piattaforme* ▾ [string]  
  durata*      string  
}
```

```
miniaturaSerie ▾ {  
  self  
  titolo*      string  
  regista*     string  
  copertina*   string  
  valutazione  number  
}
```

```
serie ▾ {  
  self  
  titolo*      string  
  regista*     string  
  etaCons      string  
  valutazione  number  
  copertina*   string  
  generi*      ▾ [string]  
  piattaforme* ▾ [string]  
  stagioni*    ▾ [[number]]  
}
```

```
user ▾ {  
  self  
  username*    string  
  isPrivate    bool  
}
```

```
review ▾ {  
  self  
  titolo*      string  
  regista*     string  
  mailAutore*  string  
  voto*        integer  
  testo        string  
}
```

Implementazione Frontend:

Nel frontend troviamo un'unica pagina dove si possono testare le varie funzionalità implementate, rispettivamente:

Login:

Utente loggato: none

1. **Login:** Da la possibilità di fare il login tramite la funzione Authentication, inserendo (mail o username) e password;

Registrati:

Feedback:

2. **Registrati:** Offre la possibilità di registrarsi tramite la funzione Sign Up inserendo email, username, password e password di conferma;

Cerca un utente in base al suo username:

3. **Cerca un utente in base al suo username:**
Grazie a questo form possiamo cercare un insieme di utenti (o un utente unico) sfruttando il relativo username, tramite la funzione Find di User;

Modifica la tua clausola di visibilità:

Ora sei: utente non loggato

☐ privato ☒ pubblico

4. **Modifica la tua clausola di visibilità:** Offre la possibilità di modificare la visibilità del proprio profilo dopo essersi ovviamente loggati, tramite la funzione Set My Privacy. Nota: Il metodo va a modificare il valore relativo all'impostazione di privacy dell'utente, tuttavia non sono state implementate le differenze tra profilo privato e profilo pubblico non avendo implementato liste e altri elementi visibili solo se l'utente ha un profilo pubblico

Fai una donazione:

Feedback:

5. **Fai una donazione:** Da la possibilità di effettuare una donazione, tramite la funzione Donation, una volta loggati

Oltre a ciò che è stato spiegato nella pagina sopra, sono stati sviluppati gli elementi di frontend legati ai film alle serie. A sinistra troviamo la ricerca in base a titolo e regista dei film, e la visualizzazione di tutti gli elementi appena nominati. A destra abbiamo invece l'equivalente per quanto riguarda le serie. Queste API dovevano essere utilizzate per creare la ricerca del requisito funzionale RF2, che permette di cercare un contenuto generico (sia esso serie tv o film) inserendo un titolo o un regista, ma, per motivi di tempo, non si è riusciti poi ad implementarla come un'unica ricerca. Cliccando sui link in blu, si possono consultare gli attributi relativi ai film e alle serie

API relative ai film:

Visualizza tutti i film:

- [quo vado., gennaro nunziane](#)

Cerca un film in base a titolo o regista:

API relative alle serie:

Visualizza tutte le serie:

- [brooklyn 99, mike schur](#)

Cerca una serie in base a titolo o regista:

In fine, troviamo la possibilità di valutare un titolo ed, eventualmente, recensirlo. Nota: poichè sono state implementate solo alcune funzionalità del sito MVN, per inserire una recensione è necessario inserire il titolo e il regista del contenuto specificato, diversamente da quanto succederebbe sul sito completo (dove la recensione viene scritta già all'interno della pagina dedicata allo specifico titolo di cui si parla). Dopo aver inserito una recensione, la valutazione del titolo recensito risulterà aggiornata

API relative alle recensioni:

Inserisci una nuova recensione:

Feedback:

titolo
regista
testo

Visualizza tutte le recensioni:

- brooklyn 99, mike schur, pluto@gmail.com, 5, wow, i enjoyed this sitcom so much!
- brooklyn 99, mike schur, pippo@gmail.com, 2, a little boring

Repository GitHub:

Il progetto MVN è presente su GitHub. Alleghiamo il link:
<https://github.com/Organizzazione-IngSoftware>.

Questo è suddiviso in varie repository:

- Deliverables: contiene tutti i file .pdf relativi ad i vari deliverables dall' 1 al 5;
- D1Mockup: raggruppa tutti i mock-up delle pagine di front end, che sono presenti nel documento di analisi (D1);
- D1Backend: Qui si trova il file usato per la progettazione del backend nel documento di analisi (D1);
- BancoLavoroD2: Qui sono presenti i file che sono serviti per scrivere il secondo documento di progetto;
- bancoLavoroD4: In questo repository si trovano tutti i file scritti per il documento di progetto D4. Quest'ultimi si trovano all'interno del branch master invece che nel main branch

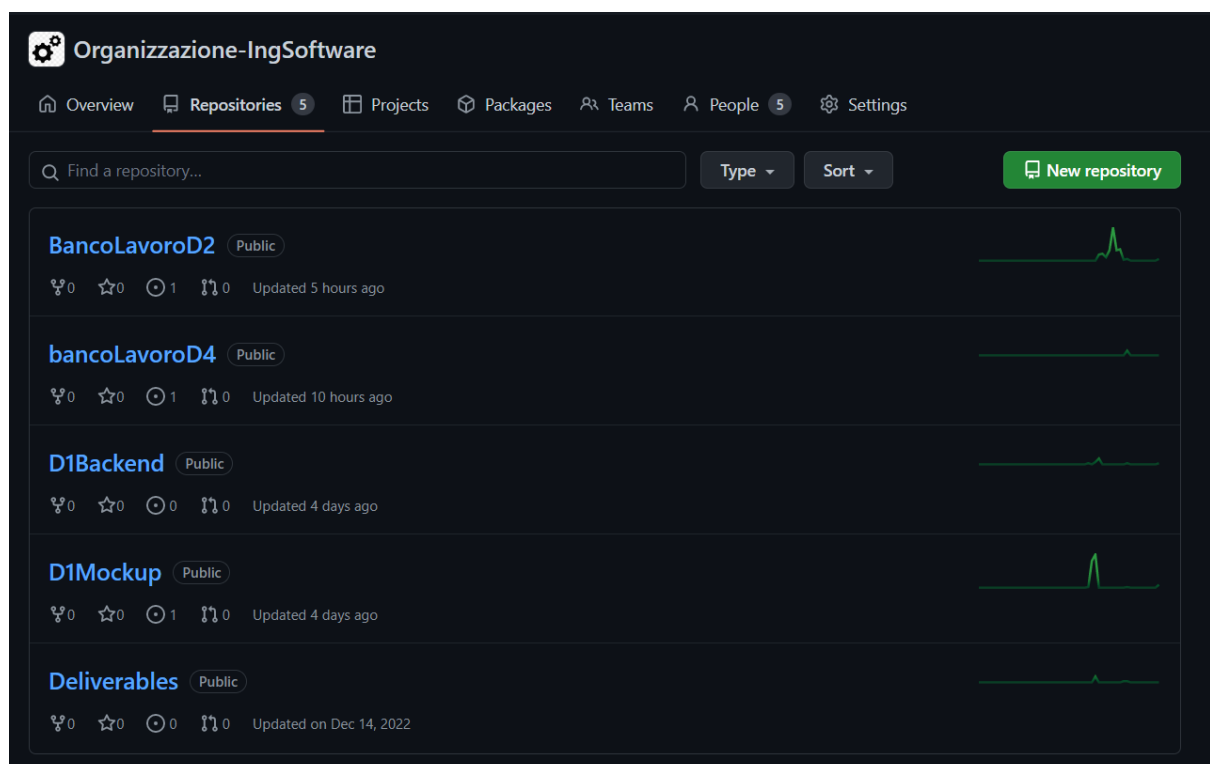
Gli account dei membri del gruppo che hanno lavorato al progetto MVN sono:

Matteo Javid Battista: <https://github.com/MatteoJavidBattista>

Aurora Ottaviani: <https://github.com/AuroraOttaviani>

Alessandro Nitti: <https://github.com/AlessandroNTI>

Si allega uno screenshot della nostra organizzazione su GitHub:



Come avviare il sito web:

Prerequisiti: Avere NodeJS con npm, che servirà per installare tutte le dependencies. Dopo averle installate con il comando npm install digitare sulla console le seguenti istruzioni:

Istruzioni:

- clonare il repository GitHub (o scaricarla da GitHub):
\$ git clone
<https://github.com/Organizzazione-IngSoftware/bancoLavoroD4>
- spostarsi sul master branch
- creare un file ".env" nella cartella MoViewerNet del progetto (vedere structure in caso di dubbi) contenente esattamente il codice a fondo pagina
- digitare cd moViewerNet nel terminale per accedere alla cartella;
- scrivere npm run start nel terminale (o npm run dev in caso di problemi)
- Collegarsi al sito indicando come url su browser localhost:8080
- Per visualizzare la documentazione, scrivere come url su browser la stringa <http://localhost:8080/api-docs>
- se si vogliono eseguire i test, scrivere invece npm run test nel terminale

```
DB_HOST = localhost

DB_USER = root

PORT = 8080

MONGODB_URI =
'mongodb+srv://moViewerNet:FileMignon546@clusterprinc.qjxmkkq.mongodb.
net/?retryWrites=true&w=majority'

SUPER_SECRET = 'sarabanda44353sarabanda'
```

Testing:

CASI DI TEST DEGLI USER:

```
test('POST /api/v1/user/signUp returns 409 if someone is already logged  
with this credentials', async () => {  
  expect.assertions(1);  
  const response = await  
request(app).post('/api/v1/user/signUp').send({ mail:  
'pippo@gmail.com', username: 'pippo', password: 'password',  
passwordSupp: 'password' });  
  expect(response.statusCode).toBe(409);  
});
```

Test per verificare che, se provo a registrarmi utilizzando un username o una password già presenti nel database sfruttando l'API signUp, ricevo l'errore 409 (conflict)

```
test('POST /api/v1/user/signUp returns 400 if the password is not  
formatted in the right way', async () => {  
  expect.assertions(1);  
  const response = await  
request(app).post('/api/v1/user/signUp').send({ mail:  
'randomemail@gmail.com', username: 'randomm', password: 'pas',  
passwordSupp: 'pas' });  
  expect(response.statusCode).toBe(400);  
});
```

Test per verificare che se la password non è correttamente formattata si riceve l'errore 400 (bad request)


```
test('POST /api/v1/user/signUp returns 400 if the mail is not formatted
in the right way', async () => {

  expect.assertions(1);

  const response = await
request(app).post('/api/v1/user/signUp').send({ mail: 'vrivbernib',
username: 'verberbernb', password: 'mypassw', passwordSupp: 'mypassw'
});

  expect(response.statusCode).toBe(400);

});
```

Test per verificare che se la mail non è formattata nel modo giusto si riceve l'errore 400 (bad request)

```
test('POST /api/v1/user/signUp returns 400 if you inserted two
different passwords', async () => {
  expect.assertions(1);
  const response = await
request(app).post('/api/v1/user/signUp').send({ mail:
'acaso@gmail.com', username: 'acaso', password: 'mypassword1',
passwordSupp: 'mypassword2' });
  expect(response.statusCode).toBe(400);
});
```

Test per verificare che, se inserisco la password di conferma diversa dalla password originale, ricevo l'errore 400 (bad request)

```
test('GET /api/v1/user/donation with no token should return 401', async
() => {
  const response1 = await request(app).get('/api/v1/user/donation');
  expect(response1.statusCode).toBe(401);
});
```

Test per verificare che, se provo a chiamare l'API per effettuare le donazioni (donation) senza avere un token valido, si riceve l'errore 401 (unauthorized)

```
test('GET /api/v1/user/setMyPrivacy with no token should return 401',
  async () => {
    const response2 = await
request(app).get('/api/v1/user/setMyPrivacy');
    expect(response2.statusCode).toBe(401);
  });
```

Test per verificare che, se provo a chiamare l'API per cambiare l'attributo di privacy senza essere loggato sfruttando l'API setMyPrivacy (senza un token valido), si riceve l'errore 401 (unauthorized)

```
test('GET /api/v1/user/donation with valid token should return 200',
  async () => {
    expect.assertions(1);
    const response3 = await
request(app).get('/api/v1/user/donation?token='+token);
    expect(response3.statusCode).toBe(200);
  });
```

Test per verificare che, se provo a chiamare l'API per effettuare la donazione con un token valido, si riceve il codice di stato 201 (success)

CASI DI TEST DEI FILM:

```
test('GET /api/v1/movie/getAll should respond with an array of books
and code 200', async () => {
  return request(app)
    .get('/api/v1/movie/getAll')
    .expect('Content-Type', /json/)
    .expect(200)
    .then((res) => {
      if(res.body && res.body[0]) {
        expect(res.body[0]).toEqual({
          self: '/api/v1/movie/1010',
          titolo: 'Software Engineering 2',
          regista: 'unitn',
          copertina: 'simple jpeg',
          durata: '6'
        });
      }
    });
});
```

Test per verificare che l'API per ottenere tutti i film dal database (getAll) restituisca ciò che ci aspettiamo

```
test('GET /api/v1/movie/:id should respond with json and code 200',
async () => {
  return request(app)
    .get('/api/v1/movie/1010')
    .expect('Content-Type', /json/)
    .expect(200, {
      self: '/api/v1/movie/1010',
      titolo: 'Software Engineering 2',
      regista: 'unitn',
      copertina: 'simple jpeg',
      durata: '6'
    });
});
```

Test per verificare che l'API per ottenere i dati del film passato per ID (definito nel mock, come nel caso di test definito sopra) restituisca il risultato atteso

```
test('GET /api/v1/movie/getByTitleRegist:param should respond with an
array of movies (json) and code 200', async () => {
  return request(app)
    .get('/api/v1/movie/getByTitleRegist/Software Engineering 2')
    .expect('Content-Type', /json/)
    .expect(200, [
      {
        self: '/api/v1/movie/1010',
        titolo: 'Software Engineering 2',
        regista: 'unitn',
        copertina: 'simple jpeg',
        durata: '6'
      }
    ]);
});
```

Test per verificare che l'API per la ricerca di un film in base al titolo o al regista (getByTitleRegist) dia i risultati attesi

CASI DI TEST DELLE SERIE:

```
test('GET /api/v1/serie/getAll should respond with an array of serie
and code 200', async () => {
  return request(app)
    .get('/api/v1/serie/getAll')
    .expect('Content-Type', /json/)
    .expect(200)
    .then((res) => {
      if(res.body && res.body[0]) {
        expect(res.body[0]).toEqual({
          self: '/api/v1/serie/1010',
          titolo: 'Software Engineering serie 2',
          regista: 'unitn',
          copertina: 'simple jpeg'
        });
      }
    });
});
```

Test per verificare che l'API per ottenere tutte le serie dal database (getAll) restituisca ciò che ci aspettiamo

```
test('GET /api/v1/serie/:id should respond with json and code 200',
  async () => {
    return request(app)
      .get('/api/v1/serie/1010')
      .expect('Content-Type', /json/)
      .expect(200, {
        self: '/api/v1/serie/1010',
        titolo: 'Software Engineering serie 2',
        regista: 'unitn',
        copertina: 'simple jpeg'
      });
  });
```

Test per verificare che l'API per ottenere i dati della serie passata per ID (definita nel mock, come nel caso di test definito sopra) restituisca il risultato atteso

```
test('GET /api/v1/serie/getByTitleRegist:param should respond with an
array of series (json) and code 200', async () => {
  return request(app)
    .get('/api/v1/serie/getByTitleRegist/Software Engineering serie
2')
    .expect('Content-Type', /json/)
    .expect(200, [
      {
        self: '/api/v1/serie/1010',
        titolo: 'Software Engineering serie 2',
        regista: 'unitn',
        copertina: 'simple jpeg'
      }
    ]);
});
```

Test per verificare che l'API per la ricerca di un film in base al titolo o al regista (getByTitleRegist) dia i risultati attesi

CASI DI TEST DEL FILE APP:

```
test('app module should be defined', () => {  
  expect(app).toBeDefined();  
});
```

Test per verificare che il modulo app sia definito

MOCK UTILIZZATI:

```
let movieSpy;
let movieSpyFindById;
beforeAll( () => {
  const Movie = require('../app/models/movie.js');
  movieSpy = jest.spyOn(Movie, 'find').mockImplementation((criterias)
=> {
    return [{
      id: 1010,
      titolo: 'Software Engineering 2',
      regista: 'unitn',
      copertina: 'simple jpeg',
      durata: '6'
    }];
  });
  movieSpyFindById = jest.spyOn(Movie,
'findById').mockImplementation((id) => {
    if (id==1010)
      return {
        id: 1010,
        titolo: 'Software Engineering 2',
        regista: 'unitn',
        copertina: 'simple jpeg',
        durata: '6'
      };
    else
      return {};
  });
});
afterAll(async () => {
  movieSpy.mockRestore();
  movieSpyFindById.mockRestore();
});
```

```
let serieSpy;
  let serieSpyFindById;
  beforeAll( () => {
    const Serie = require('../app/models/serie.js');
    serieSpy = jest.spyOn(Serie, 'find').mockImplementation((criterias)
=> {
      return [{
        id: 1010,
        titolo: 'Software Engineering serie 2',
        regista: 'unitn',
        copertina: 'simple jpeg'
      }];
    });
    serieSpyFindById = jest.spyOn(Serie,
'findById').mockImplementation((id) => {
      if (id==1010)
        return {
          id: 1010,
          titolo: 'Software Engineering serie 2',
          regista: 'unitn',
          copertina: 'simple jpeg'
        };
      else
        return {};
    });
  });
  afterAll(async () => {
    serieSpy.mockRestore();
    serieSpyFindById.mockRestore();
  });
```



```
let userSpy;
beforeAll( () => {
  const User = require('../app/models/user');
  userSpy = jest.spyOn(User,
'findOne').mockImplementation((criterias) => {
    return {
      id: 1212,
      mail: 'john@gmail.com',
      username: 'john'
    };
  });
});
afterAll(async () => {
  userSpy.mockRestore();
});
```