

Progetto EpiOpera

Gruppo T18

Alessandro Marostica

Nicolò Marchini

Nicolò Masellis





Indice

Scopo del documento	3
User flows	3
Leggenda	3
1 Login	4
2 Registrazione	5
3 Esplorazione	6
4 Post Ridotto	7
5 Post Completo	8
6 Header	9
7 Commento	10
8 Pagina Utente	11
Implementazione e Documentazione della Applicazione	12
Struttura del Prototipo	12
Dependencies del Prototipo	12
Database	14
posts	15
utentes	15
commento_posts	16
commento_profilos	16
Progetto API	16
Estrazione Risorse dal Class Diagram	16
Modelli delle Risorse	17
Post	17
Utente	19
Commenti_Post	22
Commenti_Profilo	24
Upload	26
Sviluppo API	27
Documentazione API	45
Implementazione Front End	47
GitHub Repository e Info sul Deployment	53
Testing	54

Scopo del documento

Il presente documento riporta tutte le informazioni necessarie per lo sviluppo di un prototipo dell'applicazione EpiOpera. Per ogni API realizzata, oltre ad una descrizione delle funzionalità fornite, il documento presenta la sua documentazione e i test effettuati. Infine una sezione è dedicata alle informazioni del Git Repository e il deployment dell'applicazione stessa.

User flows

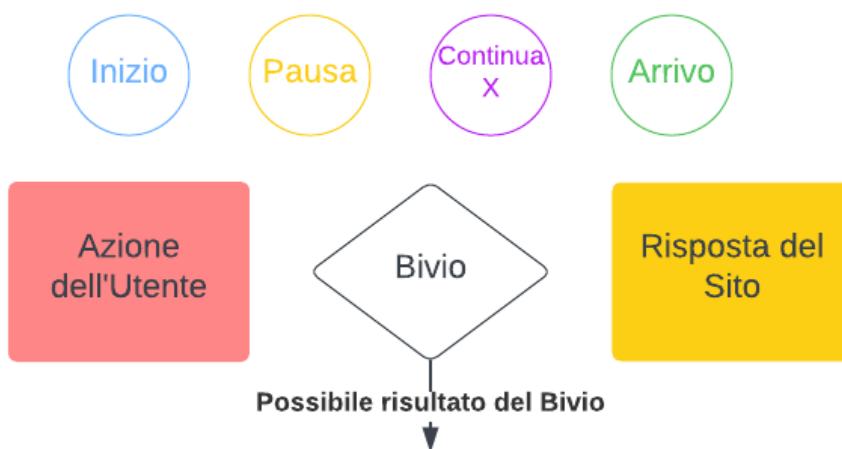
In questa sezione del documento di sviluppo riportiamo gli user flow pianificati per il nostro prototipo per un qualsiasi utente, autenticato o non (il prototipo non implementa gli amministratori) usando dei bivi quando necessario per distinguere i due. Essendo il nostro sito composto da elementi modulari e quindi ripetuti abbiamo deciso, per rendere il diagramma User Flow più leggibile, di dividere l'User Flow in 8 diagrammi che descrivono sezioni specifiche del sito.

Teoricamente usando i link un utente può iniziare a navigare da una qualsiasi parte del sito senza alcun problema, quindi il nostro sito non ha un vero e proprio punto di inizio, con la pagina di esplorazione senza filtri come cosa che si avvicina di più a una pagina iniziale o homepage.

Un utente quando accede al sito potrebbe essere già autenticato se non sono passati più di 15 minuti dall'ultima volta che ha eseguito azioni nel sito come utente autenticato, altrimenti per autenticarsi dovrà eseguire un login.

Leggenda

Sotto si possono vedere i vari blocchi che abbiamo usato per creare il nostro User Flow, oltre a quelli standard abbiamo aggiunto un cerchio viola con dentro scritto “Continua X” per indicare al lettore che da quel blocco in avanti per continuare con l'User Flow si debba andare a visualizzare il diagramma identificato dal numero X (es. “Continua 2” sta ad indicare il diagramma della Registrazione).



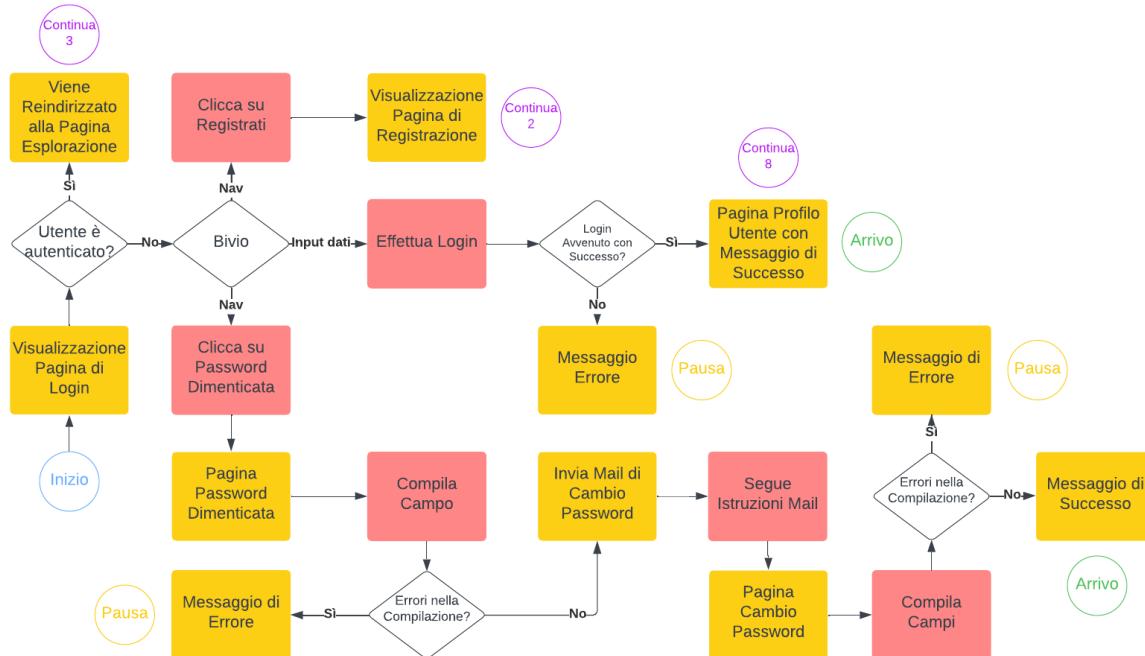
1 Login

Questo diagramma illustra le azioni che un utente può eseguire quando accede alla pagina di login.

Per eseguire l'azione “Effettua Login” l'utente dovrà inserire l'username o la e-mail che ha usato quando si è registrato e la password associata al suo profilo, il login fallisce se non esiste un profilo associato alla e-mail o username o se la password usata non è corretta.

Per eseguire l'azione “Compila Campo” l'utente dovrà inserire la sua e-mail e cliccare “Recupera password”, se la e-mail non è associata a nessun account si avrà un problema nella compilazione.

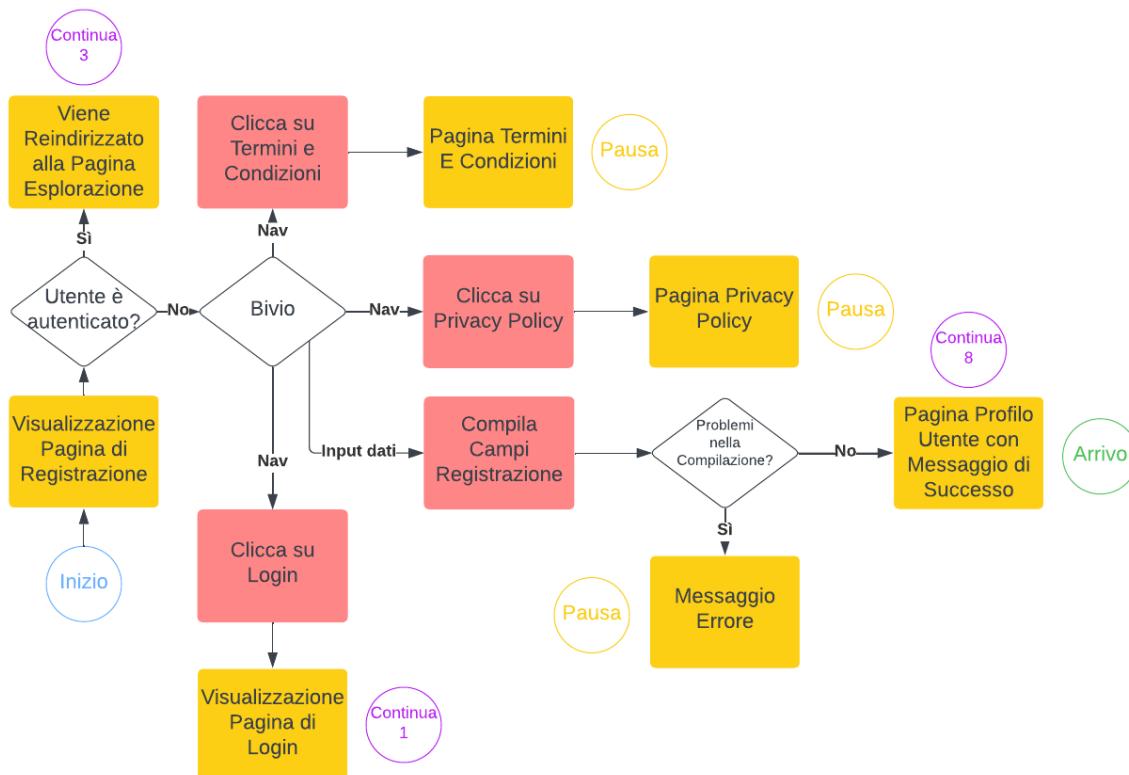
Per eseguire l'azione “Compila Campi” l'utente dovrà compilare due campi con la nuova password che vuole utilizzare e cliccare “Cambia password”, in caso i due campi non coincidano o la password non sia valida si avrà un errore nella compilazione.



2 Registrazione

Questo diagramma illustra le azioni che un utente può eseguire quando accede alla pagina di registrazione.

Per eseguire l'azione “Compila Campi di Registrazione” l'utente dovrà inserire un username, una e-mail universitaria, una password (ripetuta 2 volte) e preme il pulsante “Crea profilo”, se esiste già un profilo associato all'username o alla e-mail, l'username non è valido, la password non è stata ripetuta correttamente o non è valida allora si avrà un problema nella compilazione.



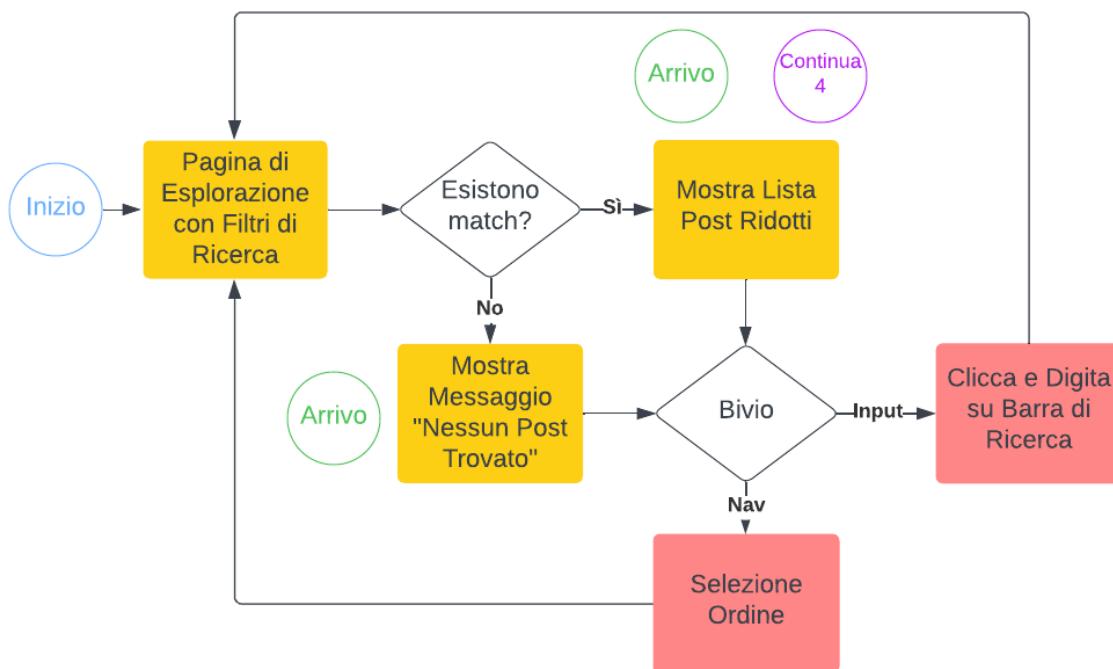
3 Esplorazione

Questo diagramma illustra le azioni che un utente può eseguire quando accede alla pagina di esplorazione.

I filtri di default sono l'ordine “più recenti” e nessuna parola inserita dentro la barra di ricerca.

Se esistono match all'interno di questa pagina compariranno viste di post ridotti che corrispondono al match ordinati secondo l'ordine selezionato.

Selezione ordine consiste nel cliccare in uno dei vari ordini di ricerca disponibili, più votati e più recenti oltre a (se autenticato) favoriti e seguiti.



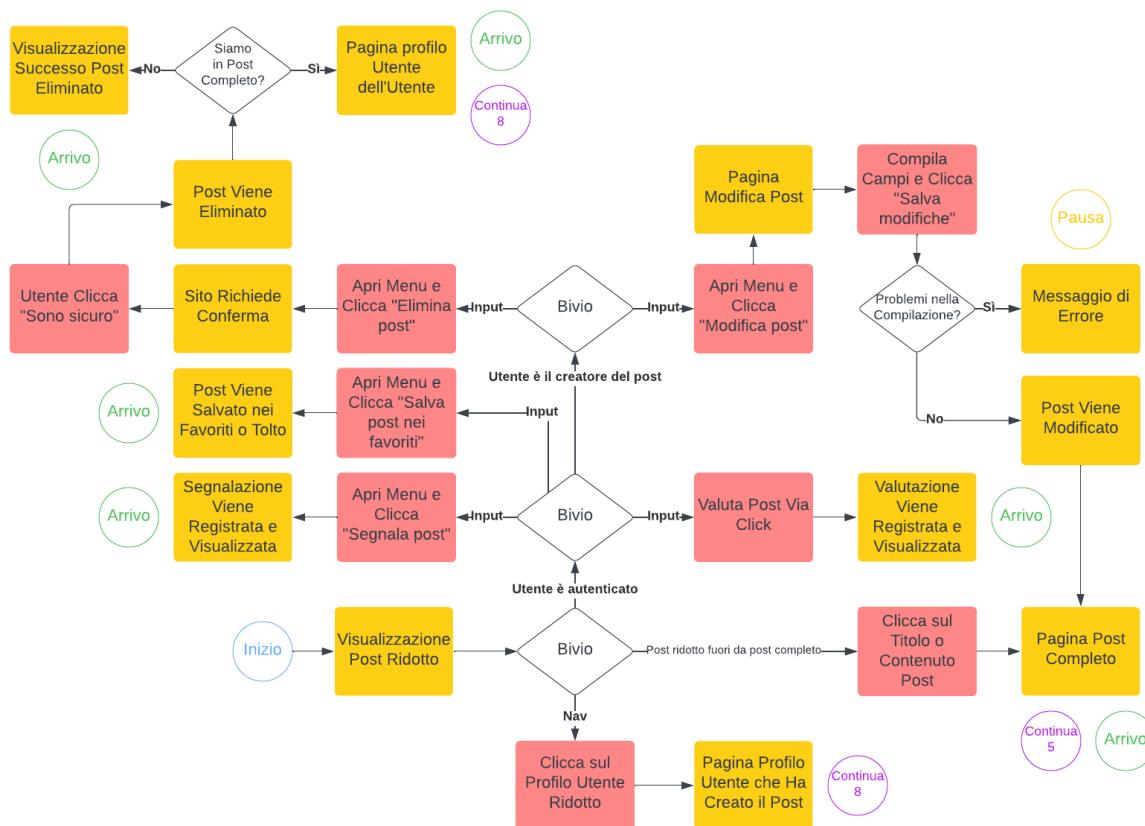
4 Post Ridotto

Questo diagramma illustra le azioni che un utente può eseguire quando visualizza un post nella sua versione ridotta.

Se il post ridotto si trova fuori dalla pagina post completo allora cliccare sul titolo o sul contenuto del post non porterà ad avere un reindirizzamento alla pagina post completo, invece se ci si ritrova nella pagina post completo quando si tenta di eliminare un post allora si verrà reindirizzati alla pagina profilo utente dell'utente.

Quando si parla di menu ci si riferisce a un menu a 3 punti (...).

Per eseguire l'azione "Compila Campi e Clicca "Salva modifiche" bisogna compilare i campi titolo, media e tag (che saranno precompilati con le informazioni presenti nel post originale) e cliccare "Salva modifiche", se il titolo o il media inserito non è valido si avrà un problema nella compilazione.



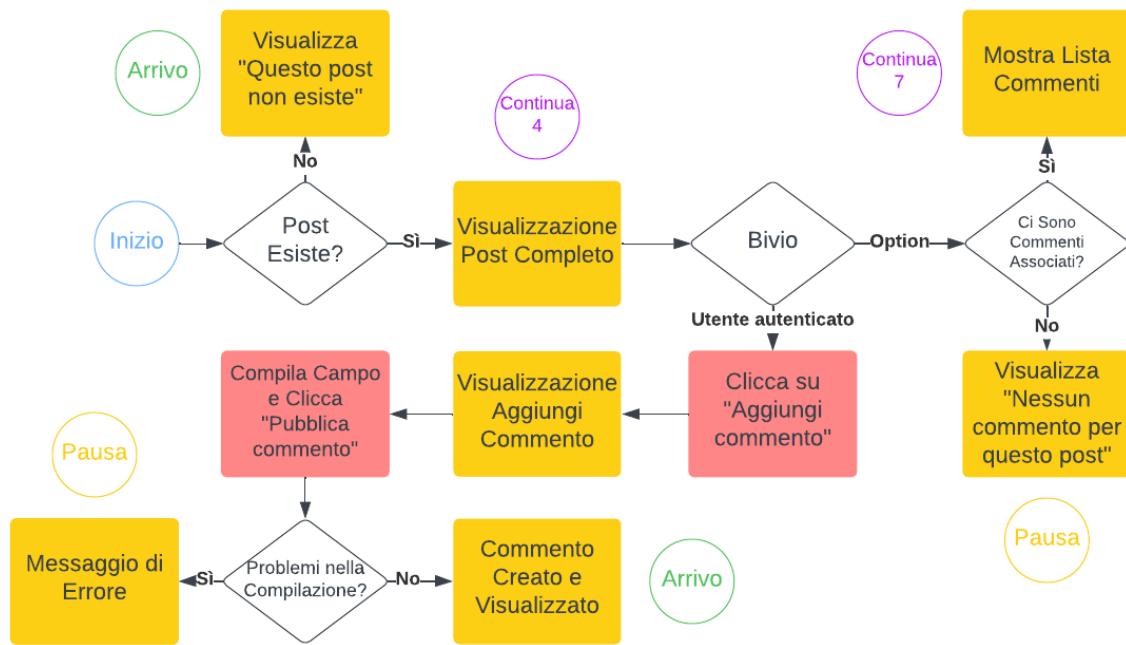
5 Post Completo

Questo diagramma illustra le azioni che un utente può eseguire quando accede alla pagina post completo.

Se il post esiste all'interno di questa pagina vi sarà una visualizzazione del post ridotto.

Se il post ha commenti associati in questa pagina compariranno viste di commenti associati.

Per eseguire l'azione "Compila Campo e Clicca "Pubblica commento"" bisogna scrivere un commento e cliccare "Pubblica commento", se il commento inserito non è valido si avrà un problema nella compilazione.

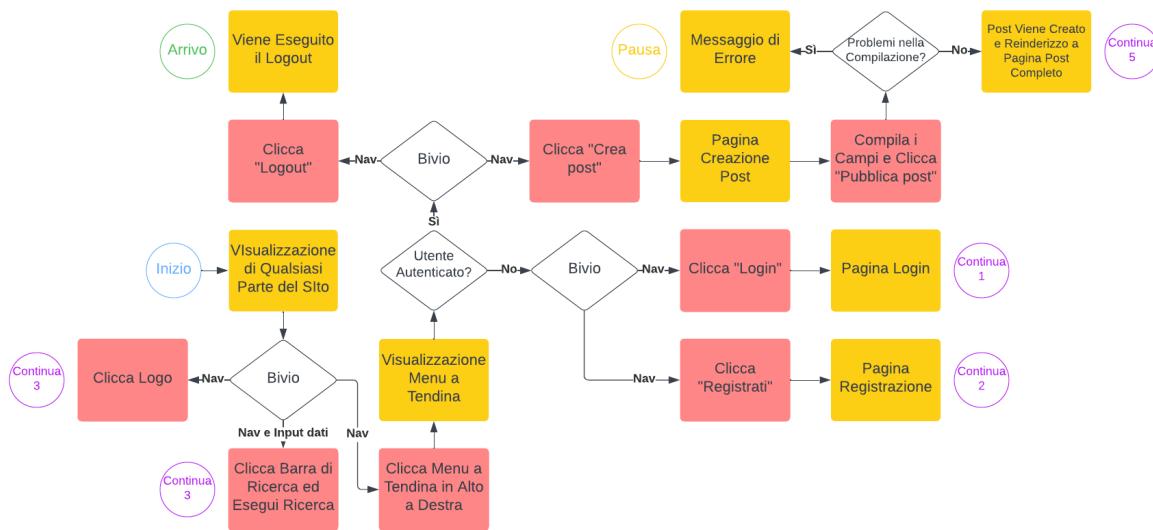


6 Header

Questo diagramma illustra le azioni che un utente può eseguire interagendo con l'header che si trova in ogni parte del sito.

Quando si clicca il Logo si esegue la ricerca di default, invece quando si esegue la ricerca dalla barra di ricerca allora la ricerca si esegue con l'ordine “più recenti” (a meno che non ci si ritrovi nella pagina di esplorazione e l'utente aveva selezionato un altro ordine in precedenza)

Per eseguire l'azione “Compila i Campi e Clicca "Pubblica post"" bisogna compilare i campi titolo, media e tag e cliccare "Pubblica post", se il titolo o il media inserito non è valido si avrà un problema nella compilazione.



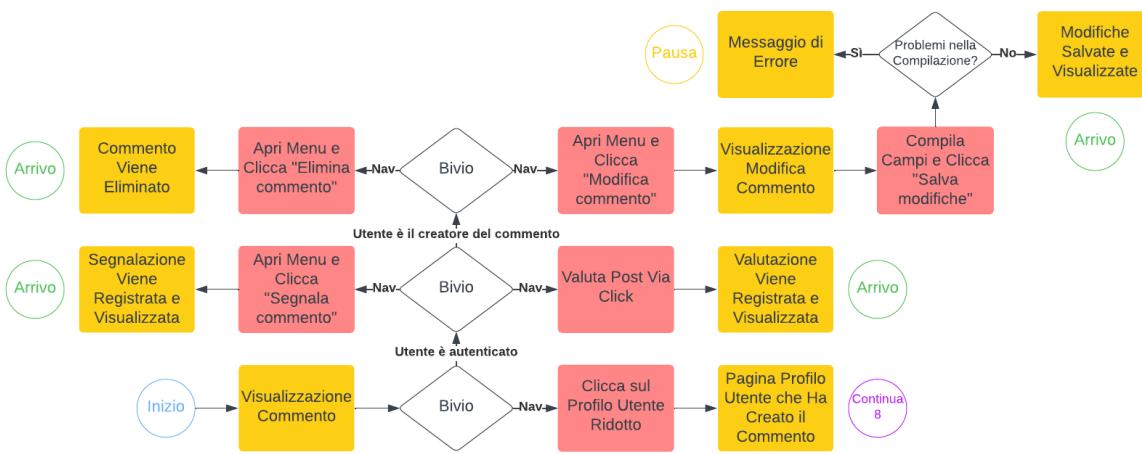
7 Commento

Questo diagramma illustra le azioni che un utente può eseguire quando visualizza un commento.

Un commento di un profilo avrà una visualizzazione diversa da quello di un post, in quanto quello di un profilo avrà un titolo.

Quando si parla di menu ci si riferisce a un menu a 3 punti (...).

Per eseguire l'azione "Compila Campi e Clicca "Salva modifiche" bisogna modificare il commento nel campo apposito che sarà precompilato con il commento originale e cliccare "Salva modifiche", se il commento inserito non è valido si avrà un problema nella compilazione.



8 Pagina Utente

Questo diagramma illustra le azioni che un utente può eseguire quando accede alla pagina profilo di un utente.

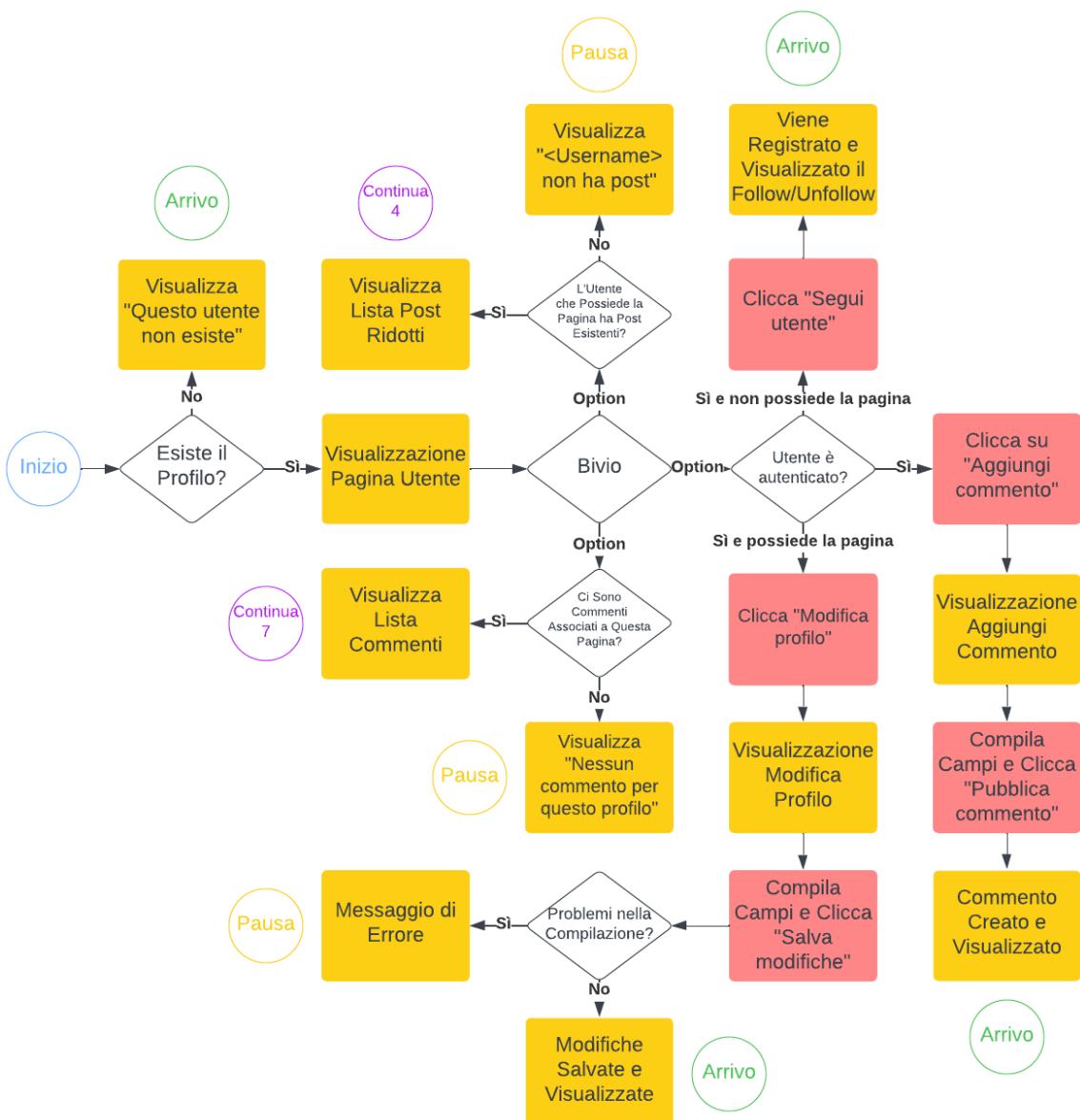
Se l'utente che possiede la pagina ha post allora compariranno viste di tali post ridotti.

Se la pagina utente ha commenti associati allora compariranno viste di tali commenti associati.

Al posto di <Username> deve comparire l'username del possessore della pagina profilo utente.

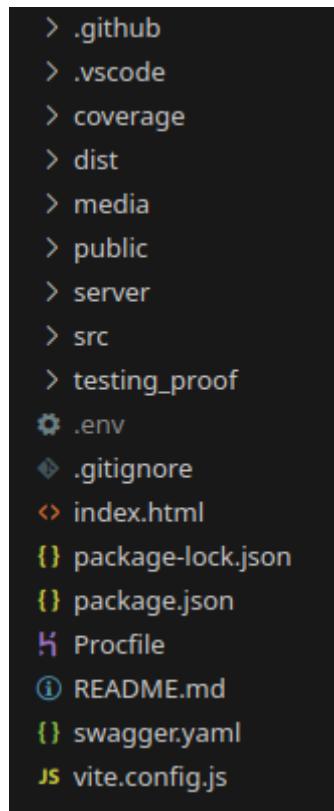
Per eseguire l'azione "Compila Campo e Clicca "Pubblica commento"" bisogna scrivere un commento, se il commento inserito non è valido si avrà un problema nella compilazione.

Per eseguire l'azione "Compila Campi e Clicca "Salva modifiche"" bisogna compilare i campi descrizione, icona profilo e banner (che saranno precompilati con i dati originali) e cliccare "Salva modifiche", se la descrizione inserita non è valida si avrà un problema nella compilazione.



Implementazione e Documentazione della Applicazione

Struttura del Prototipo



```
> .github
> .vscode
> coverage
> dist
> media
> public
> server
> src
> testing_proof
⚙️ .env
↳ .gitignore
↳ index.html
{} package-lock.json
{} package.json
↳ Procfile
ⓘ README.md
{} swagger.yaml
JS vite.config.js
```

Il progetto è composto di una singola repository su GitHub ed è composto di:

- .github: directory della configurazione di GitHub
- .vscode: directory della configurazione di VSCode
- coverage: directory contenente i risultati della fase di testing
- dist: directory contenente il front-end dopo l'operazione di building
- media: directory contenente i file multimediali caricati sul sito
- public: directory contenenti vari file pubblici
- server: directory contenente lo sviluppo del back-end
- src: directory contenente lo sviluppo del front-end
- testing_proof: directory contenente immagini dei risultati del testing
- swagger.yaml: file della documentazione
- altri file di configurazione dei vari tool usati durante lo sviluppo

Dependencies del Prototipo

Questi moduli sono stati aggiunti al file “package.json” e usati nel progetto:

- “@voidsolutions/vue-datepicker”: Utilizzato per strutturare le date in Vue;
- “bcryptjs”: Un port in javascript di bcrypt, libreria usata per fare hash delle password;
- “cookie-parser”: Middleware per il parsing dei cookie in Express;

- “cors”: Middleware per gestire le richieste cross-origin in Express;
- “dotenv”: Carica variabili d’ambiente da un file .env;
- “express”: Principale framework web per Node.js;
- “express-fileupload”: Middleware per l’upload di file in Express;
- “jsonwebtoken”: Implementazione di JSON Web Token (JWT), un formato di token per trasmettere informazioni in modo sicuro;
- “mongoose”: Libreria per l’interazione con MongoDB;
- “swagger-ui-express”: Middleware per la visualizzazione di API Swagger in Express;
- “uuid”: Genera UUID (Universal Unique Identifier), usato per assegnare ID ai post, commenti etc.;
- “vue”: Framework per la creazione di front end;
- “vue-router”: Gestisce il routing in Vue.js;
- “yamljs”: Legge e scrive file YAML in Node.js;
- “path”: Risolve il percorso di directory in base allo schema UNIX
- “jest”: libreria di testing
- “supertest”: libreria per effettuare richieste al server nella fase di testing

```
{  
  "name": "epiopera",  
  "version": "0.0.0",  
  "private": true,  
  "scripts": {  
    "dev": "vite",  
    "build": "vite build",  
    "preview": "vite preview",  
    "server": "node server/server.js",  
    "test": "jest --coverage"  
  },  
  "dependencies": {  
    "@voidsolutions/vue-datepicker": "^0.5.0",  
    "bcryptjs": "^2.4.3",  
    "cookie-parser": "^1.4.6",  
    "cors": "^2.8.5",  
    "dotenv": "^16.0.3",  
    "express": "^4.18.2",  
    "express-fileupload": "^1.4.0",  
    "jsonwebtoken": "^9.0.0",  
    "mongoose": "^6.9.1",  
    "path": "^0.12.7",  
    "swagger-ui-express": "^4.6.3",  
    "uuid": "^9.0.0",  
    "vue": "^3.3.2",  
    "vue-router": "^4.2.1",  
    "yamljs": "^0.3.0"  
  },  
  "devDependencies": {  
    "@vitejs/plugin-vue": "^4.2.3",  
    "jest": "^29.6.3",  
    "supertest": "^6.3.3",  
    "vite": "^4.3.5"  
  }  
}
```

Database

Per la gestione dei dati utili all'applicazione abbiamo definito quattro principali strutture dati che descriviamo qui sotto che sono usate dal nostro database MongoDB, le immagini invece vengono salvate direttamente nel server (infatti la variabile “media” contiene solo il nome dell'immagine che poi il backend si adopera a recuperare quando richiesto).

Abbiamo gli ID delle strutture come alfanumerici per semplicità di implementazione (invece di avere per esempio ID numerici per i post come specificato nel D3 o utente identificato solamente dal suo username che comunque rimane univoco).

Le strutture contengono anche informazioni che per il nostro prototipo non servono ma che servirebbero per il sito completo.

posts

Struttura dati che contiene il post, componente essenziale del nostro sito.

In particolare tag è un array di stringhe, associato_a_contest un array di int (teoricamente sarebbero dovuti essere gli ID di contest ma questi non sono stati implementati nel nostro prototipo) e valutazioni è una lista di username associati alla loro valutazione (1= positiva, 0=neutrale e -1=negativa).

Abbiamo reso tag un array di stringhe invece di una singola stringa per ragioni implementative (invece di parsare ogni volta la stringa così viene parsata una sola volta quando vengono associati i tag, dividendo la stringa di testo con vari tag che l'utente scrive in varie stringhe di testo contenenti un singolo tag l'una).

Abbiamo aggiunto valutazioni alla struttura post rispetto al Class Diagram per ragioni implementative (per sapere chi ha già valutato quale post).

1	_id:	ObjectId('643a95e6dbc2eac41d589977')	ObjectId
2	id:	"9fec6206-b970-4192-852b-29f0e24016c5"	String
3	titolo:	"Abracadabra"	String
4	data:	1681561062446	Double
5	testo:	"W00000000000000000000000000000000"	String
6	media:	"squirrel.jpg"	String
7	► tag:	Array	Array
8	punteggio_post:	14	Int32
9	segnalato:	false	Boolean
10	numero_commenti:	1	Int32
11	► associato_a_contest:	Array	Array
12	creatore_post:	"Ilcalmissimo"	String
13	► valutazioni:	Object	Object
14	__v:	0	Int32

utentes

Struttura dati che contiene gli utenti che sono iscritti al nostro sito.

In particolare utenti_seguiti è un array di string (corrispondenti agli id dei utenti seguiti da questo profilo) e post_favoriti è un array di string (corrispondenti agli id dei profili favoriti da questo profilo).

Aggiunto tolto token e timer alla risorsa utentes rispetto al Class Diagram perché le funzionalità che rappresentavano le abbiamo implementate senza dover richiedere che quei dati venissero salvati nel database.

1	_id:	ObjectId('63f8ede7f6f0ef8c812340a6')	ObjectId
2	username:	"Ilcalmissimo"	String
3	email:	"ilcalmissimo@studenti.unitn.it"	String
4	password:	"\$2a\$12\$4er2YioZjF8oPFueP8DROAKYBClQG0zaQILIBsA67VUpYtT1NXPi"	String
5	descrizione:	"Ciao!"	String
6	icona_profilo:	"DefaultPathPfp.png"	String
7	iconaNSFW:	false	Boolean
8	banner:	"DefaultPathBanner.jpg"	String
9	bannerNSFW:	false	Boolean
10	userscore:	2	Int32
11	lingua:	"inglese"	String
12	isAmministratore:	false	Boolean
13	nsfw:	"yes"	String
14	nome_tema_selezionato:	"default"	String
15	► utenti_seguiti:	Array	Array
16	► post_favoriti:	Array	Array
17	__v:	5	Int32

commento_posts

Struttura dati che contiene i commenti associati a un post.

In particolare valutazioni è una lista di username associati alla loro valutazione (1= positiva, 0=neutrale e -1=negativa).

Abbiamo aggiunto valutazioni alla struttura commento_posts rispetto al Class Diagram per ragioni implementative (per sapere chi ha già valutato quale commento).

1 _id: ObjectId('644418a59e8fc00884dd1d5')	ObjectId
2 id: "f8968b2a-7b23-488b-bddc-ff37b0597c1a"	String
3 id_post: "9fec6206-b970-4192-852b-29f0e24016c5"	String
4 data: 1682184357802	Double
5 testo: "BrUuUhz"	String
6 punteggio_commento: 0	Int32
7 segnalato: false	Boolean
8 creatore_commento: "impostera"	String
9 ▶ valutazioni: Object	Object
10 __v: 0	Int32

commento_profilos

Struttura dati che contiene i commenti associati a una pagina profilo utente.

In particolare valutazioni è una lista di username associati alla loro valutazione (1= positiva, 0=neutrale e -1=negativa).

Abbiamo aggiunto valutazioni alla struttura commento_profilos rispetto al Class Diagram per ragioni implementative (per sapere chi ha già valutato quale commento).

1 _id: ObjectId('644793460a2857c580e0ef64')	ObjectId
2 id: "06fef304-2c14-4f07-a00a-3fd45e11e5dc"	String
3 profilo_commentato: "Ilcalmissimo"	String
4 titolo: "asbestos"	String
5 data: 1682412358452	Double
6 testo: "abracadabra"	String
7 punteggio_commento: -1	Int32
8 segnalato: true	Boolean
9 creatore_commento: "Ilcalmissimo"	String
10 ▶ valutazioni: Object	Object
11 __v: 0	Int32

Progetto API

Estrazione Risorse dal Class Diagram

Questo diagramma fa da ponte tra il Class Diagram, presentato nel documento D3, e le resources che andremo ad implementare con le API nel nostro prototipo.

Quando nei parametri si parla di "Token" ci si sta riferendo a un token che identifica un login. Per semplicità abbiamo deciso che per eseguire il login serve l'username e la mail non serve.



Modelli delle Risorse

In questa parte del documento si prendono le risorse che sono state ricavate dal class diagram e si vanno a descrivere più nel dettaglio il loro scopo e possibile output.

Per maggiori dettagli su come queste funzioni agiscono visionare i use case del documento D2 o il documento D3 per avere una visione completa di come il sito dovrebbe funzionare o la sezione dell'implementazione del [\[Sviluppo API\]](#) per vedere solo ciò che riguarda il prototipo.

Per rendere i modelli delle risorse più leggibili li abbiamo divisi in 5 diagrammi diversi.

Abbiamo deciso di mettere Token come parametro in quanto è contenuto nei cookie del sito, il Token ci indica la sessione di login dell'utente e quindi da quello ricaveremo l'username quando serve.

Per decidere i codici HTML usati ci siamo basati su questa risorsa <https://kb.iu.edu/d/bfrc>, in particolare abbiamo usato:

- 200 OK, azione eseguita con successo.
- 201 Created, struttura creata dentro il database.
- 400 Bad Request, errore nella richiesta.
- 401 Unauthorized, utente non ha il login attivo o non ha l'autorizzazione a eseguire l'azione.
- 404 Not Found, si sta cercando di accedere a qualcosa che non esiste.
- 409 Conflict, non si può completare l'azione per via di un conflitto (es. nella registrazione l'username che si vuole usare è già registrato da un altro utente).
- 500 Server Error, errore generico quando ci sono problemi con il server.

Post

- POST Crea_Post: Ha l'obiettivo di creare un post.



Prende in input la struttura dati di un post con un valore dell'ID casuale e il Token che indica il login di un utente.

Ha le seguenti possibili risposte:

- 201 - Created + l'id generato per il post che è stato creato;
- 400 - Invalid Input + un messaggio che indichi quale sia il problema;
- 401 - Unauthorized + il messaggio "Login non eseguito";
- 500 - Server Error;

- DELETE Elimina_Post: Ha l'obiettivo di eliminare un post.

Prende in input l'ID del post da eliminare e il Token che indica il login di un utente.

Ha le seguenti possibili risposte:

- 200 - OK + l'id del post eliminato;
- 401 - Unauthorized + il messaggio "Utente non autorizzato";
- 404 - Not Found + il messaggio "Post indicato non esiste";
- 500 - Server Error;

- GET Trova_Tutti_Post: Ha l'obiettivo di ritornare tutti i post presenti nel database.

Ha le seguenti possibili risposte:

- 200 - OK + molteplici Post presenti nel database(o nessuno se non ce ne sono);
- 500 - Server Error;

- GET Trova_Tutti_Post_Di_Utente: Ha l'obiettivo di ritornare i post creati da un particolare utente.

Prende in input l'username dell'utente.

Ha le seguenti possibili risposte:

- 200 - OK + molteplici Post creati dall'utente precisato (o nessuno se l'utente precisato non ne ha creati);
- 404 - Not Found + il messaggio "Utente non esiste";
- 500 - Server Error;

- GET Trova_Post_Con_ID: Ha l'obiettivo di ritornare il post associato a un particolare ID.

Prende in input l'ID del post.

Ha le seguenti possibili risposte:

- 200 - OK + il post che si era cercato;
- 404 - Not Found + il messaggio "Post non esiste";
- 500 - Server Error;

- PUT Modifica_Post: Ha l'obiettivo di modificare un post.

Prende in input l'ID del post da modificare, il titolo modificato, il testo modificato, il media modificato, il tag modificato e il Token che indica il login di un utente.

Ha le seguenti possibili risposte:

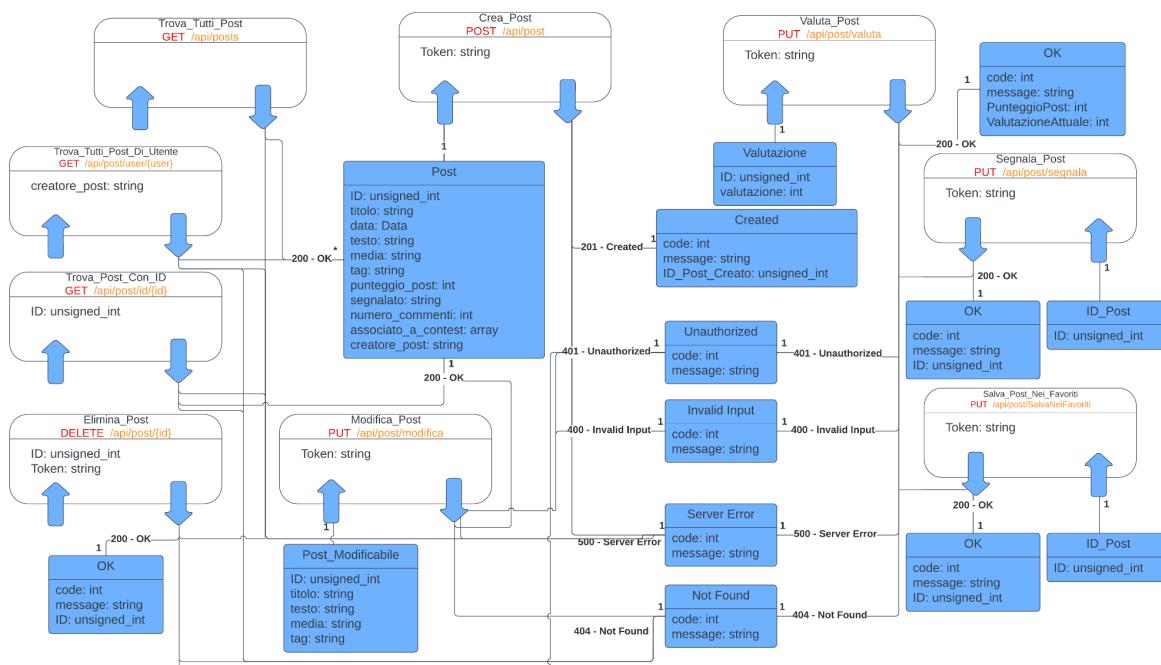
- 200 - OK + Il post modificato;
- 400 - Invalid Input + un messaggio che indichi quale sia il problema;
- 401 - Unauthorized + il messaggio "Utente non autorizzato";
- 404 - Not Found + il messaggio "Post indicato non esiste";
- 500 - Server Error;

- PUT Valuta_Post: Ha l'obiettivo di far valutare a un utente un post.

Prende in input l'ID del post da valutare, la valutazione eseguita e il Token che indica il login di un utente.

Ha le seguenti possibili risposte:

- 200 - OK + il nuovo punteggio del post e la valutazione attuale eseguita dall'utente;
 - 400 - Invalid Input + un messaggio che indichi quale sia il problema;
 - 401 - Unauthorized + il messaggio "Login non effettuato";
 - 404 - Not Found + il messaggio "Post indicato non esiste";
 - 500 - Server Error;- PUT Segnala_Post: Ha l'obiettivo di registrare la segnalazione da un utente di un post.
Prende in input l'ID del post da segnalare e il Token che indica il login di un utente.
Ha le seguenti possibili risposte:
 - 200 - OK + l'id del post segnalato;
 - 400 - Invalid Input + un messaggio che indichi quale sia il problema;
 - 401 - Unauthorized + il messaggio "Login non effettuato";
 - 404 - Not Found + il messaggio "Post indicato non esiste";
 - 500 - Server Error;
- PUT Salva_Post_Nei_Favoriti: Ha l'obiettivo di salvare un post nei favoriti di un utente.
Prende in input l'ID del post da salvare e il Token che indica il login di un utente.
Ha le seguenti possibili risposte:
 - 200 - OK + l'id del post segnalato;
 - 400 - Invalid Input + un messaggio che indichi quale sia il problema;
 - 401 - Unauthorized + il messaggio "Login non effettuato";
 - 404 - Not Found + il messaggio "Post indicato non esiste";
 - 500 - Server Error;



Utente

- POST Crea_Utente: Ha l'obiettivo di registrare un utente al sito. Prende in input la struttura dati di un utente.

Ha le seguenti possibili risposte:

- 201 - Created + username e il Token generato per il login che è stato automaticamente eseguito;
- 400 - Invalid Input + un messaggio che indichi quale sia il problema;
- 409 - Conflict + un messaggio che indichi quale sia il conflitto;
- 500 - Server Error;
- **DELETE Elimina_Utente:** Ha l'obiettivo di eliminare un utente.
Prende in input l'username dell'utente da eliminare e il Token che indica il login di un utente.

Ha le seguenti possibili risposte:

- 200 - OK;
- 401 - Unauthorized + il messaggio "Utente non autorizzato";
- 404 - Not Found + il messaggio "Utente indicato non esiste";
- 500 - Server Error;
- **GET Trova_Tutti_Utenti:** Ha l'obiettivo di ritornare tutti gli utenti presenti nel database.

Ha le seguenti possibili risposte:

- 200 - OK + molteplici utenti presenti nel database(o nessuno se non ce ne sono);
- 500 - Server Error;
- **GET Trova_Utente:** Ha l'obiettivo di ritornare l'utente associato a un particolare username.
Prende in input l'username dell'utente.

Ha le seguenti possibili risposte:

- 200 - OK + l'utente che si era cercato;
- 404 - Not Found + il messaggio "Utente non esiste";
- 500 - Server Error;
- **PUT Modifica_Mail:** Ha l'obiettivo di modificare l'e-mail di un utente.
Prende in input la e-mail nuova e il Token che indica il login di un utente.

Ha le seguenti possibili risposte:

- 200 - OK + la nuova mail;
- 400 - Invalid Input + un messaggio che indichi quale sia il problema;
- 401 - Unauthorized + il messaggio "Utente non autorizzato";
- 409 - Conflict + il messaggio "La e-mail è già utilizzata da un altro account"
- 500 - Server Error;
- **PUT Modifica_Password:** Ha l'obiettivo di modificare la password di un utente.
Prende in input la nuova password e il Token che indica il login di un utente.

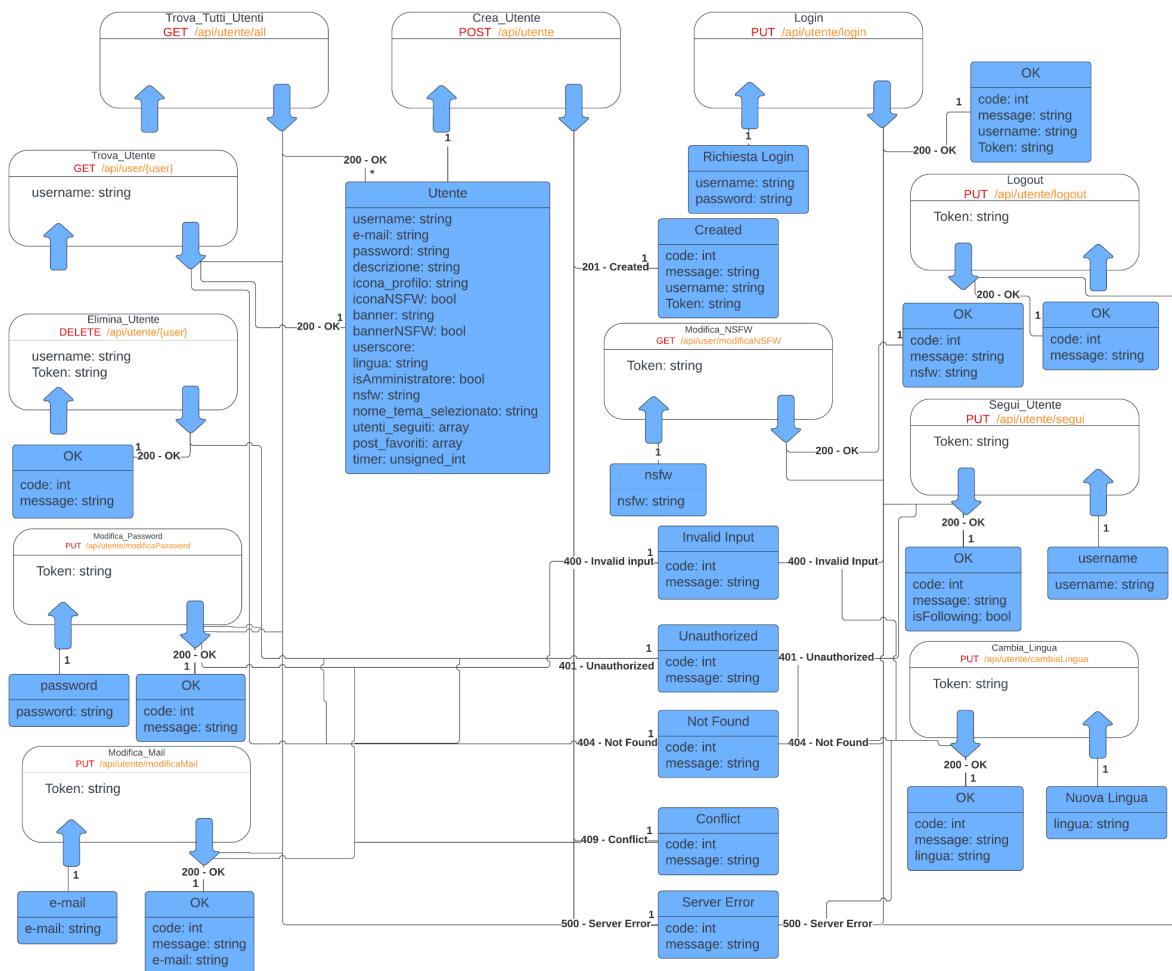
Ha le seguenti possibili risposte:

- 200 - OK;
- 400 - Invalid Input + un messaggio che indichi quale sia il problema;
- 401 - Unauthorized + il messaggio "Utente non autorizzato";
- 500 - Server Error;
- **PUT Modifica_NSFW:** Ha l'obiettivo di modificare lo stato della gestione del contenuto NSFW di un utente.
Prende in input il nuovo stato della impostazione NSFW e il Token che indica il login di un utente.

Ha le seguenti possibili risposte:

- 200 - OK + il nuovo stato della impostazione NSFW;

- 400 - Invalid Input + un messaggio che indichi quale sia il problema;
- 401 - Unauthorized + il messaggio “Utente non autorizzato”;
- 500 - Server Error;
- PUT Cambia_Lingua: Ha l'obiettivo di modificare la lingua usata da un utente.
Prende in input la nuova lingua e il Token che indica il login di un utente.
Ha le seguenti possibili risposte:
 - 200 - OK + la nuova lingua;
 - 400 - Invalid Input + un messaggio che indichi quale sia il problema;
 - 401 - Unauthorized + il messaggio “Utente non autorizzato”;
 - 500 - Server Error;
- PUT Segui_Utente: Ha l'obiettivo di permettere a un utente di followare o unfolloware un altro utente.
Prende in input l'username dell'utente da seguire e il Token che indica il login di un utente.
Ha le seguenti possibili risposte:
 - 200 - OK + un booleano che rappresenta se ora l'utente è seguito o meno;
 - 400 - Invalid Input + un messaggio che indichi quale sia il problema;
 - 401 - Unauthorized + il messaggio “Login non effettuato”;
 - 404 - Not Found + il messaggio “L'username indicato non è associato ad alcun account”
 - 500 - Server Error;
- PUT Login: Ha l'obiettivo di eseguire il login di un utente.
Prende in input l'username e la password.
Ha le seguenti possibili risposte:
 - 200 - OK + il token generato che rappresenta il login e l'username;
 - 400 - Invalid Input + un messaggio che indichi quale sia il problema;
 - 404 - Not Found + il messaggio “L'username indicato non è associato ad alcun account”
 - 500 - Server Error;
- PUT Logout: Ha l'obiettivo di eseguire il logout di un utente.
Ha le seguenti possibili risposte:
 - 200 - OK;
 - 500 - Server Error;



Commenti_Post

- **POST Crea_Commento_Post**: Ha l'obiettivo di creare un commento associato a un post.

Prende in input la struttura dati di un commento post con un valore dell'ID casuale e il Token che indica il login di un utente.

Ha le seguenti possibili risposte:

- 201 - Created + l'id generato per il commento che è stato creato;
- 400 - Invalid Input + un messaggio che indichi quale sia il problema;
- 401 - Unauthorized + il messaggio "Login non eseguito";
- 500 - Server Error;

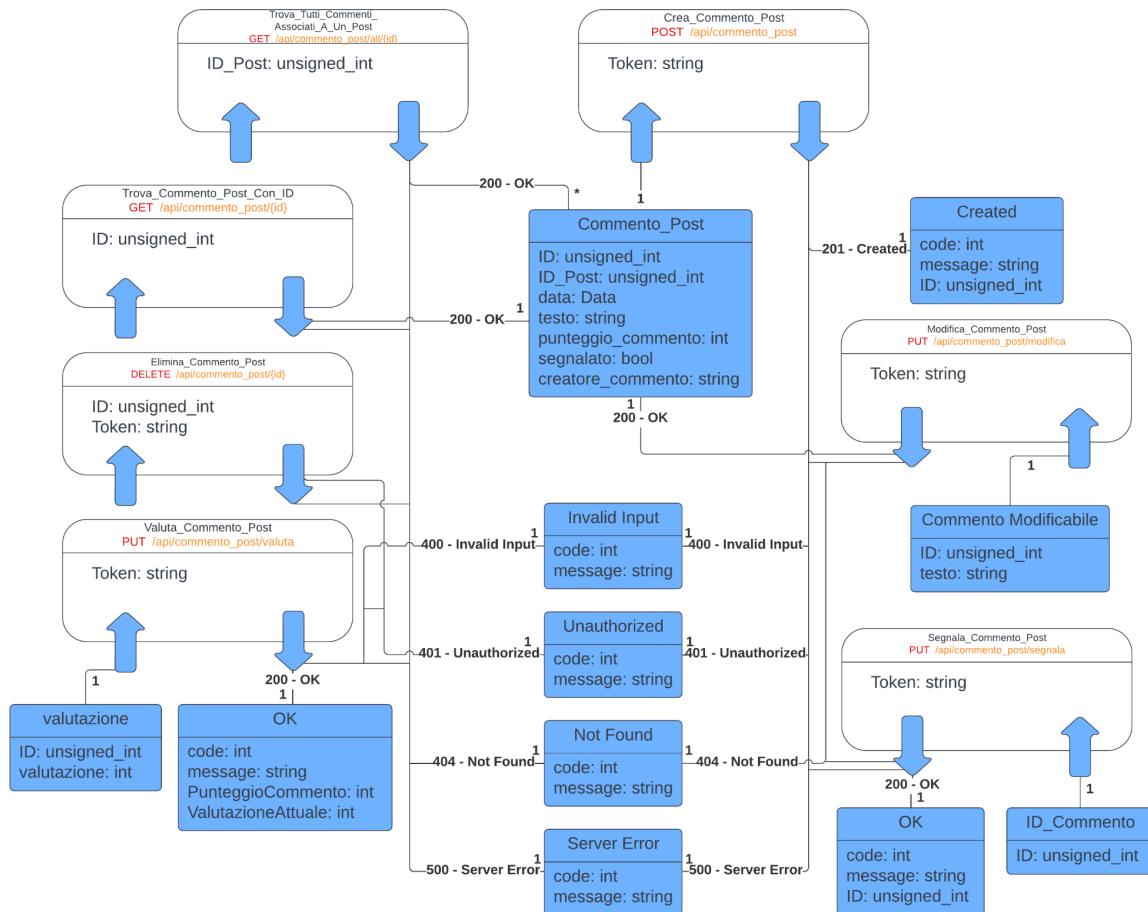
- **DELETE Elimina_Commento_Post**: Ha l'obiettivo di eliminare un commento associato a un post.

Prende in input l'ID del commento da eliminare e il Token che indica il login di un utente.

Ha le seguenti possibili risposte:

- 200 - OK + l'id del commento eliminato;
- 401 - Unauthorized + il messaggio "Utente non autorizzato";
- 404 - Not Found + il messaggio "Commento indicato non esiste";
- 500 - Server Error;

- GET Trova_Tutti_Commenti_Associati_A_Un_Post: Ha l'obiettivo di ritornare tutti i commenti associati a un post.
 Ha le seguenti possibili risposte:
 - 200 - OK + molteplici commenti associati a un post (o nessuno se non ce ne sono);
 - 500 - Server Error;
- GET Trova_Commento_Post_Con_ID: Ha l'obiettivo di ritornare un commento associato a un post dato il suo ID.
 Prende in input l'ID del commento da cercare.
 Ha le seguenti possibili risposte:
 - 200 - OK + il commento cercato;
 - 404 - Not Found + il messaggio "Commento non esiste";
 - 500 - Server Error;
- PUT Modifica_Commento_Post: Ha l'obiettivo di modificare un commento associato a un post.
 Prende in input l'ID del commento da modificare, il testo modificato e il Token che indica il login di un utente.
 Ha le seguenti possibili risposte:
 - 200 - OK + Il commento modificato;
 - 400 - Invalid Input + un messaggio che indichi quale sia il problema;
 - 401 - Unauthorized + il messaggio "Utente non autorizzato";
 - 404 - Not Found + il messaggio "Commento indicato non esiste";
 - 500 - Server Error;
- PUT Valuta_Commento_Post: Ha l'obiettivo di far valutare a un utente un commento associato a un post.
 Prende in input l'ID del commento da valutare, la valutazione eseguita e il Token che indica il login di un utente.
 Ha le seguenti possibili risposte:
 - 200 - OK + il nuovo punteggio del commento e la valutazione attuale eseguita dall'utente;
 - 400 - Invalid Input + un messaggio che indichi quale sia il problema;
 - 401 - Unauthorized + il messaggio "Login non effettuato";
 - 404 - Not Found + il messaggio "Commento indicato non esiste";
 - 500 - Server Error;
- PUT Segnala_Commento_Post: Ha l'obiettivo di registrare la segnalazione da un utente di un commento associato a un post.
 Prende in input l'ID del commento da segnalare e il Token che indica il login di un utente.
 Ha le seguenti possibili risposte:
 - 200 - OK + l'id del commento segnalato;
 - 400 - Invalid Input + un messaggio che indichi quale sia il problema;
 - 401 - Unauthorized + il messaggio "Login non effettuato";
 - 404 - Not Found + il messaggio "Post indicato non esiste";
 - 500 - Server Error;



Commenti_Profilo

- POST Crea_Commento_Profilo: Ha l'obiettivo di creare un commento associato a un profilo.
Prende in input la struttura dati di un commento profilo con un valore dell'ID casuale e il Token che indica il login di un utente.
Ha le seguenti possibili risposte:
 - 201 - Created + l'id generato per il commento che è stato creato;
 - 400 - Invalid Input + un messaggio che indichi quale sia il problema;
 - 401 - Unauthorized + il messaggio "Login non eseguito";
 - 500 - Server Error;
 - DELETE Elimina_Commento_Profilo: Ha l'obiettivo di eliminare un commento associato a un profilo.
Prende in input l'ID del commento da eliminare e il Token che indica il login di un utente.
Ha le seguenti possibili risposte:
 - 200 - OK + l'id del commento eliminato;
 - 401 - Unauthorized + il messaggio "Utente non autorizzato";
 - 404 - Not Found + il messaggio "Commento indicato non esiste";
 - 500 - Server Error;
 - GET Trova_Tutti_Commenti_Associati_A_Un_Profilo: Ha l'obiettivo di ritornare tutti i commenti associati a un profilo.

Ha le seguenti possibili risposte:

- 200 - OK + molteplici commenti associati a un profilo (o nessuno se non ce ne sono);
- 500 - Server Error;
- GET Trova_Commento_Profilo_Con_ID: Ha l'obiettivo di ritornare un commento associato a un profilo dato il suo ID.

Prende in input l'ID del commento da cercare.

Ha le seguenti possibili risposte:

- 200 - OK + il commento cercato;
- 404 - Not Found + il messaggio "Commento non esiste";
- 500 - Server Error;
- PUT Modifica_Commento_Profilo: Ha l'obiettivo di modificare un commento associato a un profilo.

Prende in input l'ID del commento da modificare, il titolo modificato, il testo modificato e il Token che indica il login di un utente.

Ha le seguenti possibili risposte:

- 200 - OK + Il commento modificato;
- 400 - Invalid Input + un messaggio che indichi quale sia il problema;
- 401 - Unauthorized + il messaggio "Utente non autorizzato";
- 404 - Not Found + il messaggio "Commento indicato non esiste";
- 500 - Server Error;
- PUT Valuta_Commento_Profilo: Ha l'obiettivo di far valutare a un utente un commento associato a un profilo.

Prende in input l'ID del commento da valutare, la valutazione eseguita e il Token che indica il login di un utente.

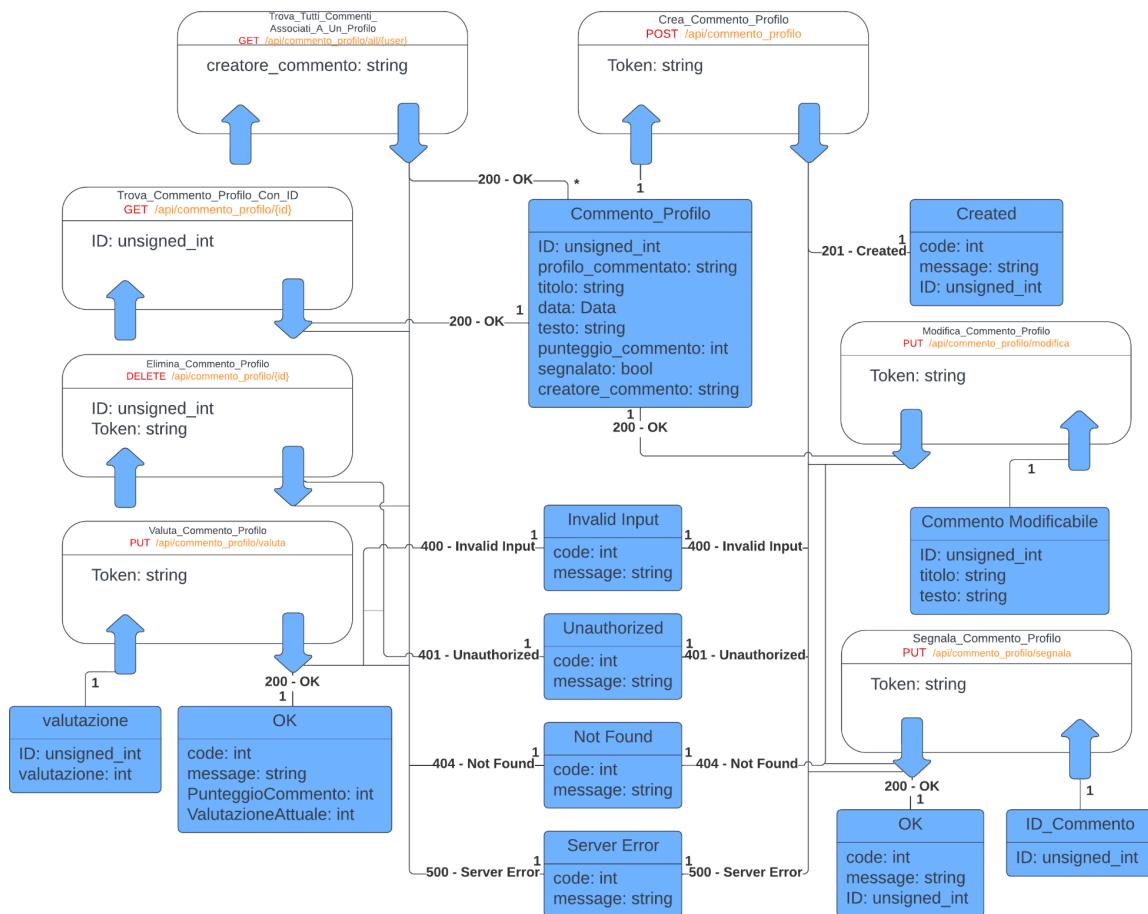
Ha le seguenti possibili risposte:

- 200 - OK + il nuovo punteggio del commento e la valutazione attuale eseguita dall'utente;
- 400 - Invalid Input + un messaggio che indichi quale sia il problema;
- 401 - Unauthorized + il messaggio "Login non effettuato";
- 404 - Not Found + il messaggio "Commento indicato non esiste";
- 500 - Server Error;
- PUT Segnala_Commento_Profilo: Ha l'obiettivo di registrare la segnalazione da un utente di un commento associato a un profilo.

Prende in input l'ID del commento da segnalare e il Token che indica il login di un utente.

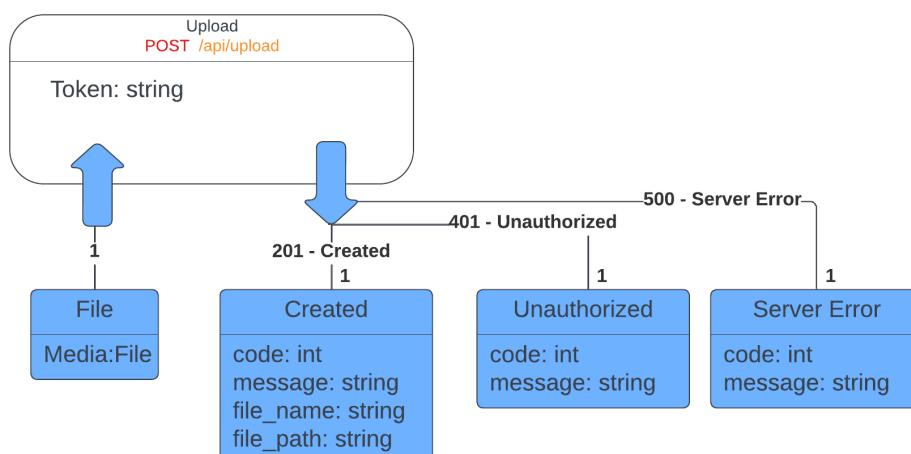
Ha le seguenti possibili risposte:

- 200 - OK + l'id del commento segnalato;
- 400 - Invalid Input + un messaggio che indichi quale sia il problema;
- 401 - Unauthorized + il messaggio "Login non effettuato";
- 404 - Not Found + il messaggio "Post indicato non esiste";
- 500 - Server Error;



Upload

- POST Upload: Ha l'obiettivo di permettere a un utente di caricare file nel sito. Prende in input un file e il Token che indica il login di un utente. Ha le seguenti possibili risposte:
 - 201 - Created + il nome assegnato al file e il percorso per arrivare ad esso;
 - 400 - Invalid Input + un messaggio che indichi quale sia il problema;
 - 401 - Unauthorized + il messaggio “Login non eseguito”;
 - 500 - Server Error;



Sviluppo API

- **Lista utenti**

```
// Get all users
const getUtenti = (req, res) => {

    console.log("Listing all users...");

    // Retireves all users
    Utente.find({}, (err, data) => {

        if (err) {
            return res.status(500).send();
        } else {
            var usernames = [];
            data.forEach(user => {
                usernames.push(user.username);
            });

            // Returns an array with all users
            return res.status(200).json({ Utenti: usernames });
        }
    });
};
```

- **Elimina utente**

```
// Delete user from the database
const deleteUtente = async (req, res) => {

    const username_utente = req.params.user;
    const username = req.body.username;

    // Tries to find the user
    var query = { username: username };
    const utente = await Utente.findOne(query).exec();

    // Check if the deleting user it either admin or the user itself else returns an error
    if(!utente.isAmministratore && utente.username != username_utente){
        console.log(utente.username + " " + username_utente);
        res.status(401).json({ Error: "Utente non autorizzato." });
    }

    // Deletes the user itself
    query = { username: username_utente };
    Utente.deleteMany(query).exec();

    // Cascade deletes all posts by the user
    query = { creatore_post: username_utente };
    Post.deleteMany(query).exec();

    // Cascade deletes all comment by the user
    query = { creatore_commento: username_utente };
    Commento_Post.deleteMany(query).exec();
    Commento_Profilo.deleteMany(query).exec();

    res.status(200).send();
};
```

- **Nuovo utente**

```
// Post a new user
const newUtente = async (req, res) => {
    console.log("Trying to register new user...");

    // Tries to find a user with the same username
    let utente = await Utente.findOne({ username: req.body.username }).exec();

    // Encrypts password
    const encPsw = await bcrypt.hash(req.body.password, 10);

    // Checks password validity
    if (!textRequirements.checkMail(req.body.email)) {
        return res.status(400).json({ Error: 'La mail usata per la registrazione deve essere del dominio unitn.it' });
    }
    if (!textRequirements.checkUsername(req.body.username)) {
        return res.status(400).json({ Error: 'L\'username deve essere almeno lungo 3 caratteri e non deve contenere i caratteri "@" e "#" ' });
    }
    if (!textRequirements.checkPassword(req.body.password)) {
        return res.status(400).json({ Error: 'La password deve avere almeno una maiuscola, minuscola, numero, carattere speciale ed essere almeno 12 caratteri' });
    }

    // If no user with the given username is found it creates one
    if (!utente) {
        const newUtente = new Utente({
            username: req.body.username,
            email: req.body.email,
            password: encPsw,
            icona_profilo: req.body.icona_profilo,
            iconaNSFW: req.body.iconaNSFW,
            banner: req.body.banner,
            bannerNSFW: req.body.bannerNSFW,
            lingua: req.body.lingua,
            isAmministratore: req.body.isAmministratore,
            nsfw: req.body.nsfw,
            nome_tema_selezionato: req.body.nome_tema_selezionato
        });

        // Attaches cookie to response
        const token = tokenManager.setCookie(res, { username: req.body.username });

        // New user is saved to the database
        newUtente.save((err, data) => {
            if (err) return res.status(500).send();
            return res.status(201).json({ username: req.body.username, token: token });
        });
    } else {
        // Returns an error if the user already exists
        return res.status(409).json({ Error: "Utente già esistente." });
    }
};
```

- **Mostra utente specifico**

```
// Get a user by its username
const getUtente = (req, res) => {

    const username = req.params.user;

    const query = { username: username };

    // Tries to find the user
    Utente.find(query, (err, utente) => {

        // If no user is found returns an error
        if (utente.length <= 0) {
            return res.status(404).json({ Error: "L'utente non esiste." })
        }

        if (err) {
            return res.status(500).json({ Error: "Errore interno al server" });
        } else {
            // Returns the body of the user
            return res.status(200).json(utente);
        }
    });
};
```

- Login

```
// Generates a login token for a user
const login = async (req, res) => {

  var username = req.body.username;
  var psw = req.body.password;

  var query = { username: username };

  // Tries to find a user by the username
  Utente.findOne(query, (err, utente) => {

    if(err) {
      return res.status(500).send();
    }

    console.log(utente);

    // If no user is found returns an error
    if(!utente) {
      return res.status(404).json({ Error: "Utente non trovato." });
    }

    // Compares passwords with the bcryptjs dependency
    bcrypt.compare(psw, utente.password, (err, result) => {

      // If password correct generates a token and attaches it to the response
      if(result){
        token = tokenManager.setCookie(res, { username: utente.username });
        // Returns the token and the username
        return res.status(200).json({ token: token, username: username });
      }

      // If wrong password returns an error
      return res.status(401).json({ Error: "Password sbagliata." });
    })
  });
};
```

- Logout

```
// Logs out
const logout = async (req, res) => {

  // Clears the cookie
  res.clearCookie('tokenEpiOpera');

  return res.status(200).send();
}
```



● Segui utente

```
// Adds a user id to this user's following list
const seguiUtente = async (req, res) => {
  const username = req.body.username;
  const utenteDaSeguire = req.body.utenteDaSeguire;

  // Tries to find the user to be followed
  const query = { username: utenteDaSeguire };
  var utente = await Utente.findOne(query).exec();

  // If no user is found returns an error
  if (!utente) return res.status(404).json({ Error: "Utente da seguire non trovato." });

  utente = await Utente.findOne({ username: username }).exec();

  // If the user isn't already followed it adds it to the user's following list else it removes from the following list
  if (!utente.utenti_seguiti.includes(utenteDaSeguire)) utente.utenti_seguiti.push(utenteDaSeguire);
  else utente.utenti_seguiti = utente.utenti_seguiti.filter(entry => entry != utenteDaSeguire);

  utente.save();

  // Returns a boolean telling if the user is now following the user to be followed
  return res.status(200).json({ IsFollowing: utente.utenti_seguiti.includes(utenteDaSeguire) });
}
```

● Modifica Mail

```
// Edit a user's email address
const modificaMail = async (req, res) => {

  const username = req.body.username;
  const email = req.body.email;

  // Checks the validity of the mail else returns an error
  if (!textRequirements.checkMail(email)) {
    return res.status(400).json({ Error: 'La mail usata deve essere del dominio unitn.it' });
  }

  // Tries to find the user and updates accordingly
  const query = { username: username };
  const utente = await Utente.findOneAndUpdate(query, { email: email }).exec();

  // Returns the new email address
  res.status(200).json({ email: email });
}
```

● Modifica Password

```
// Edit a user's password
const modificaPassword = async (req, res) => {
  // Checks password validity else returns an error
  if (!textRequirements.checkPassword(req.body.newPassword)) {
    return res.status(400).json({ Error: 'La password deve avere almeno una maiuscola, minuscola, numero, carattere speciale ed essere almeno 12 caratteri' });
  }

  // Encrypts password
  const username = req.body.username;
  //const encPsw = await bcrypt.hash(req.body.password, 10);
  const newPassword = await bcrypt.hash(req.body.newPassword, 10);

  // Tries to find a user by its username
  const query = { username: username };
  const utente = await Utente.findOne(query).exec();

  //if (utente.password != encPsw){
  //  res.status(401).send("Password sbagliata.")
  //}

  // Saves new password to the database
  utente.password = newPassword;
  utente.save();

  return res.status(200).send();
}
```

- **Modifica NSFW**

```
// Edit a user's NSFW property
const modificaNSFW = async (req, res) => {
  const username = req.body.username;
  const nsfw = req.body.nsfw;

  // Checks inputted setting validity
  if (nsfw != "no" && nsfw != "blur" && nsfw != "yes"){
    return res.status(400).json({ Error: "NSFW non valido." });
  }

  // Tries to find the user by its username and updates accordingly
  const query = { username: username };
  const utente = await Utente.findOneAndUpdate(query, { nsfw: nsfw }).exec();

  // Returns the new setting
  return res.status(200).json({ nsfw: nsfw });
}
```

- **Cambia lingua**

```
// Edit a user's language property
const cambiaLingua = async (req, res) => {

  const username = req.body.username;
  const lingua = req.body.lingua;

  // Checks inputted setting validity
  if (lingua != "italiano" && lingua != "inglese"){
    return res.status(400).json({ Error: "Lingua non valida." });
  }

  // Tries to find the user by its username and updates accordingly
  const query = { username: username };
  const utente = await Utente.findOneAndUpdate(query, { lingua: lingua }).exec();

  // Returns the new setting
  return res.status(200).json({ lingua: lingua });
}
```

- **Lista post**

```
// Get all posts fromt he database
const getPosts = (req, res) => {

  console.log("Listing all Posts...");

  // Retrieves all posts from the database
  Post.find({}, (err, data) => {

    if (err) {
      return res.status(500).send();
    }

    const retval = [];

    data.forEach(element => {
      retval.push(element);
    });

    // Returns an array with all the retrieved posts
    return res.status(200).json(retval);
  });
};
```

- **Mostra post con un certo id**

```
// Get a specific post by its id
const getPostById = (req, res) => {

  let postId = req.params.id;
  var query = { id: postId };

  console.log("Getting post by id...");

  // Tries to find the post by its id
  Post.findOne(query, (err,data) => {

    if (err) {
      return res.status(500).send();
    }

    // If no post is found returns an error
    if (!data) {
      return res.status(404).json({ Error: "Il post non esiste."});
    }

    // Returns the post object
    return res.status(200).json(data);
  });
};
```

- **Mostra i post di un utente**

```
// Get all posts by a specific user
const getPostByUser = async (req, res) => {

  let postUser = req.params.user;
  var query = { creatore_post: postUser };
  console.log("Getting post by username " + postUser + "...");

  // Tries to find the user whose posts are being retrieved
  const userExists = await Utente.findOne({ username: postUser }).exec();

  // If the user is not found returns an error
  if (!userExists) return res.status(404).json({ Error: "L'utente non esiste." });

  // Retrieves all post by the specific user's id
  Post.find(query, (err,data) => {

    if (err) {
      return res.status(500).send();
    }
    // Returns an array with all the retrieved posts
    return res.status(200).json(data);
  });
}
```

- **Segnala post**

```
// Flags a post by its id
const segnalaPost = (req, res) => {

  var postId = req.body.id;
  var query = { id: postId };

  console.log(postId);

  console.log(`Flagging post with id ${postId}...`);

  // Tries to find the specific post by its id
  Post.findOne(query, (err,data) => {

    if (err) {
      return res.status(500).send();
    }

    // If the post is not found returns an error
    if(!data){
      return res.status(404).json({ Error: "Post non trovato." });
    }

    // The flag property is saved to the database
    data.segnalato = true;
    data.save();

    // Returns the id of the flagged post
    return res.status(200).json({ id: postId });
  });
}
```

- **Valuta Post**

```
// Adds an evaluation to a specific post
const valutaPost = (req, res) => {

  const postId = req.body.id;
  const username = req.body.username;
  const valutazione = req.body.valutazione;
  var query = { id: postId };

  // Tries to find the post to be evaluated by its id
  Post.findOne(query, (err,data) => [
    if (err) {
      return res.status(500).send();
    }

    // Checks the validity of the inputted evaluation
    if (valutazione != -1 && valutazione != 0 && valutazione != 1) {
      return res.status(500).send();
    }

    // If the post is not found returns an error
    if (!data) {
      return res.status(404).json({ Error: "Post non trovato." });
    }

    // Evaluations are stored as a map with the user who created the evaluation and the evaluation itself
    // If an entry in the map with the evaluating user is not present it is created here and its value is set to 0
    var valutazionePrecedente = data.valutazioni.get(username);
    if (isNaN(valutazionePrecedente)) valutazionePrecedente = 0;

    // The new evaluation is set and the comment's score is updated
    data.valutazioni.set(username, valutazione);
    const cambioPunteggio = valutazione - valutazionePrecedente;
    data.punteggio_post += cambioPunteggio;

    query = { username: data.creatore_post };

    //Finds the evaluated post's creator and updates its userscore
    Utente.findOne(query, (err, utente) => {
      if (err) {
        return res.status(500).send();
      }
      if (!utente) {
        return res.status(404).json({ Error: "Creatore del post non trovato." });
      }

      utente.userscore += cambioPunteggio;
      utente.save();
      data.save();

      // Returns both the new and old post's scores
      return res.status(200).json({ PunteggioPost: data.punteggio_post, ValutazioneAttuale: valutazione });
    })
  ])
}
```

- Modifica post

```
// Edits a post
const modificaPost = (req, res) => {

  const postId = req.body.id;
  const username = req.body.username;
  const titolo = req.body.titolo;
  const testo = req.body.testo;
  const media = req.body.media;
  const tag = req.body.tag;

  const query = { id: postId };

  // Tries to find the post by its id
  Post.findOne(query, (err, post) => {

    if (err) {
      return res.status(500).send();
    }

    // If the post is not found returns an error
    if (!post) {
      return res.status(404).json({ Error: "Il post non esiste." });
    }

    // If the user trying to edit the post is not the post's creator returns an error
    if (post.creatore_post != username){
      return res.status(401).json({ Error: "Utente non autorizzato." });
    }

    post.titolo = titolo;
    post.testo = testo;
    post.media = media;
    post.tag = tag;

    post.save();

    // Returns the updated post
    return res.status(200).json(post);
  })
}
```

- Salva post nei favoriti

```
// Save a post in a user's favourite list
const salvaNeiFavoriti = async (req, res) => {

  const postId = req.body.id;
  const username = req.body.username;

  const query = { username: username };

  // Checks if the post exists in the database
  const postExists = await Post.findOne({ id: postId }).exec();

  // If the post is not found returns an error
  if (!postExists) {
    return res.status(404).json({ Error: "Il post non esiste." });
  }

  // Retrieves the user trying to save the post
  Utente.findOne(query, (err, utente) => {

    if (err) {
      return res.status(500).send();
    }

    // If a post is already in a user's favourite list nothing happens, else the post is added to the user's favourite list
    if (!utente.post_favoriti.includes(postId)) utente.post_favoriti.push(postId);
    utente.save();

    // Returns the id of the post and the username of the user saving the post
    return res.status(200).json({ id: postId, username: username });
  })
}
```

- **Crea nuovo post**

```
//Post a new post
const newPost = async (req, res) => {

    console.log(req.body);
    console.log('Trying to add new post, checking if a post with the same title exists...');

    const id = uuidv4();

    // A post cannot have both media and text, if so an error is returned
    if (req.body.testo != null && req.body.media != null) {
        return res.status(400).json({ Error: "Un post non può avere sia media che testo." });
    }

    const newPost = new Post({
        id,
        titolo: req.body.titolo,
        data: Date.now(),
        testo: req.body.testo === '' ? null : req.body.testo,
        media: req.body.media === '' ? null : req.body.media,
        tag: req.body.tag,
        associato_a_contest: req.body.associato_a_contest,
        creatore_post: req.body.username
    });

    // Saves the new post to the database and returns its id
    newPost.save((err, data) => {
        if (err) return res.status(500).send();
        return res.status(201).json({ Id: id });
    });
};


```

- **Elimina post**

```
// Delete a post from the database
const deletePost = async (req, res) => {

    const postId = req.params.id;
    const username = req.body.username;

    // Tries to retrieve the post from the database
    var query = { id: postId };
    const post = await Post.findOne(query).exec();

    // If the post is not found returns an error
    if (!post) {
        return res.status(404).json({ Error: "Il post non esiste." });
    }

    // Retrieves the user trying to delete the post
    query = { username: username };
    const utente = await Utente.findOne(query).exec();

    // Deletes post only if the user trying to delete is either admin or the post's creator itself, else returns an error
    if (!utente.isAdmin && utente.username != post.creatore_post){
        return res.status(401).json({ Error: "Utente non autorizzato." });
    }

    // Retrieves the creator of the post
    query = { username: post.creatore_post };
    const creatore_post = await Utente.findOne(query).exec();

    // Deletes the post
    query = { id: postId };
    Post.deleteOne(query).exec();

    // Subtracts the deleted post's score from the total of the user's userscore
    creatore_post.userscore -= post.punteggio_post;

    creatore_post.save();

    // Returns the id of the deleted post
    return res.status(200).json({ id: postId });
};


```

- Crea un commento su un post

```
// Post a new post comment
const newCommento_Post = async (req, res) => {

    console.log(req.body);
    console.log('Trying to add new comment...');

    // New post comment object is created
    const newCommento_Post = new Commento_Post({
        id: uuidv4(),
        id_post: req.body.id_post,
        data: Date.now(),
        testo: req.body.testo,
        creatore_commento: req.body.username
    });

    // Finds the post to which the comment should be associated
    Post.findOne({ id: req.body.id_post }, (err, post) => {

        if (err) {
            return res.status(500).send();
        }

        console.log(post.numero_commenti);
        post.numero_commenti += 1;
        console.log(post.numero_commenti);

        post.save();

        // New comment is saved to the database
        newCommento_Post.save((err, data) => {
            if (err) return res.status(500).send();
            // Returns the id of the created post
            return res.status(201).json({ Id: newCommento_Post.id });
        });
    });
};
```

- Lista commenti di uno specifico post

```
const postId = req.params.id_post;

// Retrieves the post comments by the the post's id
Commento_Post.find({ id_post: postId }, (err, data) => {

    if (err) {
        return res.status(500).send();
    }

    const retval = [];

    data.forEach(element => {
        retval.push(element.id);
    });

    // Returns an array containing the retrieved comments
    return res.status(200).json(retval);
});
```

- **Elimina un commento di un post**

```
// Delete a specific post comment]
const deleteCommento_Post = (req, res) => {

  const commentId = req.params.id;

  console.log(`Deleting comment with id ${commentId}...`);

  const username = req.body.username;
  const query = { username: username };

  // Retrieves the user trying to delete the post comment
  Utente.findOne(query, (err, utente) => {

    if (err){
      throw err
    } else {
      const query = { id: commentId };

      // Retrieves the comment to be deleted by its id
      Commento_Post.findOne(query, (err, commento) => {

        if (err){
          throw err
        } else {
          // Deletes comment only if the user trying to delete is either admin or the comment's creator itself, else returns an error
          if(!utente.isAmministratore && utente.username != commento.creatore_commento){
            return res.status(401).json({ Error: "Utente non autorizzato." });
          }

          // Deletes the post comment itself
          Commento_Post.deleteOne(query, async (err, data) => {

            if (err){
              throw err
            } else {
              console.log(`Comment with id ${commentId} deleted successfully.`);

              utente = await Utente.findOne({ username: commento.creatore_commento }).exec();

              // Subtracts the post's score from the total user's userscore
              utente.userscore -= commento.punteggio_commento;
              utente.save();

              // Edits the number of comments from the post
              Post.findOne({ id: commento.id_post }, (err, post) => {

                if (err) {
                  console.log(err);
                }
                post.numero_commenti -= 1;
                post.save();
              })
            }

            // Returns the deleted comment's id
            return res.status(200).json({ id: commentId });
          })
        }
      })
    }
  })
}
```

- **Mostra un commento di un post tramite il suo id**

```
// Get a specific post comment by its id
const getCommento_Post = (req, res) => {

  const commentAssoc = req.params.id;
  const query = { id: commentAssoc };

  console.log(`Getting comment with association id ${commentAssoc}...`);

  // Tries to find the post comment by its id
  Commento_Post.findOne(query, (err, commento) => {

    if (err) {
      return res.status(500).send();
    }

    // If no comment is found returns an error
    if (!commento) {
      return res.status(404).json({ Error: "Commento non trovato." })
    }

    // Returns the comment object
    console.log(`Comment with association id ${commentAssoc} retrieved successfully.`);
    return res.status(200).json(commento);
  });
}
```

- **Segnala un commento di un post**

```
// Flag a post comment by its id
const segnalaCommento_Post = (req, res) => {

    const commentId = req.body.id;
    var query = { id: commentId };

    console.log(`Flagging comment with id ${commentId}...`);

    // Tries to find the comment by its id
    Commento_Post.findOne(query, (err,data) => {

        if (err) {
            return res.status(500).send();
        }

        // If the comment is not found returns an error
        if (!data) {
            return res.status(404).json({ Error: "Il commento non esiste." });
        }

        // Saves the new property to the database
        data.segnalato = true;
        data.save();

        // Returns the flagged post comment's id
        return res.status(200).json({ id: commentId });
    });
}
```

- **Valuta un commento di un post**

```
// Edit a post comment by its id
const modificaCommento_Post = (req, res) => [
    const commentId = req.body.id;
    const testo = req.body.testo;
    const username = req.body.username;

    var query = { id: commentId };

    // Tries to find the post comment to be edited
    Commento_Post.findOne(query, (err, commento) => {

        if (err) {
            return res.status(500).send();
        }

        query = { username: username };

        // Retrieves the user trying to edit the comment
        Utente.findOne(query, (err, utente) => {

            if (err) {
                return res.status(500).send();
            }

            // Edits comment only if the editing user is admin or the creator of the comment itself, else returns an error
            if(!utente.isAdmin && utente.username != commento.creatore_commento){
                return res.status(401).json({ Error: "Utente non autorizzato." });
            }

            // Sabe the edited post comment to the database
            commento.testo = testo;
            commento.save();

            // Returns the edited comment
            return res.status(200).json({ id: commentId, testo: testo });
        });
    });
}
```

- **Modifica un commento di un post**

```
// Edit a post comment by its id
const modificaCommento_Post = (req, res) => [
  |
  const commentId = req.body.id;
  const testo = req.body.testo;
  const username = req.body.username;

  var query = { id: commentId };

  // Tries to find the post comment to be edited
  Commento_Post.findOne(query, (err, commento) => {

    if (err) {
      return res.status(500).send();
    }

    query = { username: username };

    // Retrieves the user trying to edit the comment
    Utente.findOne(query, (err, utente) => {

      if (err) {
        return res.status(500).send();
      }

      // Edits comment only if the editing user is admin or the creator of the comment itself, else returns an error
      if(!utente.isAmministratore && utente.username != commento.creatore_commento){
        return res.status(401).json({ Error: "Utente non autorizzato." });
      }

      // Sobe the edited post comment to the database
      commento.testo = testo;
      commento.save();

      // Returns the edited comment
      return res.status(200).json({ id: commentId, testo: testo });
    })
  })
]
```

- **Crea un commento su un profilo**

```
// Post a new profile comment
const newCommento_Profilo = async (req, res) => {

  //console.log(req.body);
  //console.log('Trying to add new comment...');

  // Creates a new profile comment object
  const newCommento_Profilo = new Commento_Profilo({
    id: uuidv4(),
    profilo_commentato: req.body.profilo_commentato,
    titolo: req.body.titolo,
    data: Date.now(),
    testo: req.body.testo,
    creatore_commento: req.body.username
  });

  // Saves to the database and returns the new profile comment
  newCommento_Profilo.save((err, data) => {
    if (err) { console.log(err); return res.status(500).send(); }
    return res.status(201).json({ Id: newCommento_Profilo.id });
  });
};
```

- **Lista i commenti su un profilo specifico**

```
// Get all profile comments associated to a specific user
const getCommenti_Profilo = (req, res) => [
  const profiloUsername = req.params.user;

  // Retrieves all profile comments of a specific user by its username
  Commento_Profilo.find({ profilo_commentato: profiloUsername }, (err, data) => {

    if (err) {
      return res.status(500).send();
    }

    const retval = [];

    data.forEach(element => {
      retval.push(element.id);
    });

    // Returns an array with every profile comment found
    return res.status(200).json(retval);
  });
];
```

- **Elimina un commento su un profilo**

```
// Delete a profile comment by its id
const deleteCommento_Profilo = (req, res) => [
  const commentId = req.params.id;
  //console.log(`Deleting comment with id ${commentId}...`);

  const username = req.body.username;
  const query = { username: username };

  // Tries to retrieve the user trying to delete the profile comment
  Utente.findOne(query, (err, utente) => {
    if (err){
      return res.status(500).send();
    } else {
      const query = { id: commentId };

      // Tries to find the comment to be deleted
      Commento_Profilo.findOne(query, (err, commento) => {
        if (err){
          return res.status(500).send();
        } else if (!commento) {
          // If the comment is not found returns an error
          return res.status(404).json({ Error: "Commento non trovato." });
        } else {
          // Deletes the comment only if the deleting user is either admin or the user which created the comment itself
          if(!utente.isAdministatore && utente.username != commento.creatore_commento){
            return res.status(401).json({ Error: "Utente non autorizzato." });
          }

          // Deletes the comment
          Commento_Profilo.deleteOne(query, async (err, data) => {
            if (err){
              return res.status(500).send();
            } else {
              //console.log(`Comment with id ${commentId} deleted succesfully.`);

              utente = await Utente.findOne({ username: commento.creatore_commento }).exec();

              // Updates the userscore by subtracting the score of the deleted comment
              utente.userscore -= commento.punteggio_commento;
              utente.save();

              // Returns the deleted comment's id
              return res.status(200).json({ id: commentId });
            }
          });
        }
      });
    }
  });
];
```

- Mostra un commento di profilo specifico

```
// Get a profile comment by its id
const getCommento_Profilo = (req, res) => {

  const id = req.params.id;
  const query = { id: id };

  //console.log(`Getting comment with association id ${username}...`);

  // Tries to find a profile comment by its id
  Commento_Profilo.findOne(query, (err, commento) => {

    if (err) {
      return res.status(500).send();
    }

    // If no profile comment is found returns an error
    if (!commento) {
      return res.status(404).json({ Error: "Commento non trovato. " });
    }

    // If comment is found returns it
    return res.status(200).json(commento);
  });
};
```

- Segnala un commento di profilo

```
// Flags a profile comment
const segnalaCommento_Profilo = (req, res) => {

  commentId = req.body.id;
  var query = { id: commentId };

  //console.log(`Flagging comment with id ${commentId}...`);

  // Tries to find the profile comment by its id
  Commento_Profilo.findOne(query, (err,data) => {

    if (err) {
      return res.status(500).send();
    }

    // If no comment is found return an error
    if (!data) {
      return res.status(404).json({ Error: "Commento non trovato." });
    }

    // Sets the segnalato property to true
    data.segnalato = true;
    data.save();

    // Returns the id of the flagged comment
    return res.status(200).json({ id: commentId });
  });
};
```

- **Modifica un commento di profilo**

```
// Edit a profile comment
const modificaCommento_Profilo = (req, res) => {

  const commentId = req.body.id;
  const titolo = req.body.titolo
  const testo = req.body.testo;
  const username = req.body.username;

  var query = { id: commentId };

  // Tries to find the comment by its id
  Commento_Profilo.findOne(query, (err, commento) => {
    if (err) {
      return res.status(500).send();
    }

    // If no comment is found returns an error
    if (!commento) {
      return res.status(404).json({ Error: "Commento non trovato." });
    }

    query = { username: username };

    // Tries to retrieve the editing user
    Utente.findOne(query, (err, utente) => {
      if (err) {
        return res.status(500).send();
      }

      // Edits comment only if the editing user is either admin or the user which created the comment itself
      if(!utente.isAdmin && utente.username != commento.creatore_commento){
        return res.status(401).json({ Error: "Utente non autorizzato." });
      }

      // Saves the updated comment to the database
      commento.titolo = titolo;
      commento.testo = testo;
      commento.save();

      // Returns the updated comment
      return res.status(200).json(commento);
    })
  })
}
```

- **Valuta un commento di profilo**

```
// Adds an evaluation to a profile comment
const valutaCommento_Profilo = (req, res) => {

  const commentId = req.body.id;
  const username = req.body.username;
  const valutazione = req.body.valutazione;
  var query = { id: commentId };

  // Tries to find the target profile comment
  Commento_Profilo.findOne(query, (err,data) => {

    if (err) {
      return res.status(500).send();
    }

    // If the comment is not found returns an error
    if (!data) {
      return res.status(404).json({ Error: "Commento non trovato." });
    }

    // Checks the validity of the evaluation
    if (valutazione != -1 && valutazione != 0 && valutazione != 1) {
      return res.status(400).json({ Error: "Valutazione non valida." });
    }

    // Evaluations are stored as a map with the user who created the evaluation and the evaluation itself
    // If an entry in the map with the evaluating user is not present it is created here and its value is set to 0
    var valutazionePrecedente = data.valutazioni.get(username);
    if (isNaN(valutazionePrecedente)) valutazionePrecedente = 0;

    // The new evaluation is set and the comment's score is updated
    data.valutazioni.set(username, valutazione);
    const cambioPunteggio = valutazione - valutazionePrecedente;
    data.punteggio_commento += cambioPunteggio

    query = { username: data.creatore_commento };

    // The comment's owner is retrieved from the database
    Utente.findOne(query, (err, utente) => {

      if (err) {
        return res.status(500).send();
      }

      // If the user is not found returns an error
      if (utente == null) {
        return res.status(404).json({ Error: "Creatore del commento non trovato." });
      }

      // This user's userscore is updated and saved
      utente.userscore += cambioPunteggio;
      utente.save();
      data.save();

      // Both new and old comment's scores are returned
      return res.status(200).json({ PunteggioCommento: data.punteggio_commento, ValutazioneAttuale: valutazione });
    });
  });
};
```

- **Upload**

```
//Upload API

const path = require('path');

const upload = (req, res) => {
    console.log("Upload invoked");

    if(!req.files){
        return res.status(500).send({ Error: "file not found" });
    }

    const file = req.files.file;
    const parentDir = path.resolve(__dirname, '../../../../../');

    file.mv(`${parentDir}/media/${file.name}`, (err) =>{
        if(err){
            console.error(err);
            return res.status(500).send({ Error: "Error occurred" });
        }
        return res.status(201).json({ name: file.name, path: `./media/${file.name}` });
    });
};
```

Documentazione API

Le API fornite dal nostro prototipo e già descritte nelle sezioni precedenti sono state poi documentate utilizzando swagger. In questo modo la documentazione relativa alle API è direttamente disponibile a chiunque veda il codice sorgente. Per poter generare l'endpoint dedicato alla presentazione delle API abbiamo utilizzato "swagger-ui-express" in quanto crea una pagina web secondo la documentazione scritta.

La documentazione è divisa in quattro sezioni, Utenti, Post, Commenti Post e Commenti Profilo, più una extra che contiene la API di Upload dei file.

Per accedere a questa documentazione bisogna avviare il server e poi usare un browser per navigare alla pagina <https://epiopera-4f1c76fdd577.herokuapp.com/api-docs/#/>, qui sotto possono anche essere visionate le varie sezioni.

**Utenti** APIs for the Utente class

GET	/api/utente/{user}	Retrieves a user registered to the database	
DELETE	/api/utente/{user}	Deletes a user from the database by username, requires authentication via login API	
POST	/api/utente	Register a new user in the database	
GET	/api/utente/all	Retrieves all users registered on the database	
PUT	/api/utente/login	Generates a new login token for the user	
PUT	/api/utente/logout	Logs out a user by clearing the associated cookie	
PUT	/api/utente/seguì	Associates a user to the requesting user's followed list, requires authentication via login API	
PUT	/api/utente/modificaMail	Change the mail field of a logged in user, requires authentication via login API	
PUT	/api/utente/modificaPassword	Change the password field of a logged in user, requires authentication via login API	
PUT	/api/utente/modificaNSFW	Change the NSFW setting of a user, requires authentication via login API	
PUT	/api/utente/cambiaLingua	Change the language setting of a user, requires authentication via login API	

Post APIs for the Post class

GET	/api/posts	Retrieves all posts in the database	
GET	/api/post/id/{id}	Retrieves a post by its Id	
GET	/api/post/user/{user}	Retrieves all posts posted by a user	
PUT	/api/post/segnalà	Flags a post, requires authentication via login API	
PUT	/api/post/valuta	Increments or decrements a post's score, requires authentication via login API	
PUT	/api/post/modifica	Edits a post, requires authentication via login API	
PUT	/api/post/salvaNeiFavoriti	Saves a post in a user's favourites, requires authentication via login API	
POST	/api/post	Uploads a new post to the database, requires authentication via login API	
DELETE	/api/post/{id}	Deletes a post by its id	

Commenti Post APIs for the Commento_Post class

POST	/api/commento_post	Post a new post comment, requires authentication via login API	
GET	/api/commento_post/all/{id}	Get post comments by association Id	
GET	/api/commento_post/{id}	Get a post comment by its ID	
DELETE	/api/commento_post/{id}	Deletes a post comment by id, requires authentication via login API	
PUT	/api/commento_post/segnalà/{id}	Flags a post comment, requires authentication via login API	
PUT	/api/commento_post/valuta	Increments or decrements a post comment's score, requires authentication via login API	
PUT	/api/commento_post/modifica	Edit the comment, requires authentication via login API	

Commenti Profilo APIs for the Commento_Profilo class

POST	/api/commento_profilo	Uploads a new post comment on the database, requires authentication via login API	
GET	/api/commento_profilo/all/{user}	Retrieves all comments on the database relating to a specific profile	
DELETE	/api/commento_profilo/{id}	Deletes a profile comment by its id, requires authentication via login API	
GET	/api/commento_profilo/{id}	Retrieves a specific profile comment by its ID	
PUT	/api/commento_profilo/segnalà/{id}	Flags a profile comment, requires authentication via login API	
PUT	/api/commento_profilo/modifica	Edit the comment, requires authentication via login API	
PUT	/api/commento_profilo/valuta	Increments or decrements a profile comment's score, requires authentication via login API	

Upload API for the file upload function

POST /api/upload Upload a file

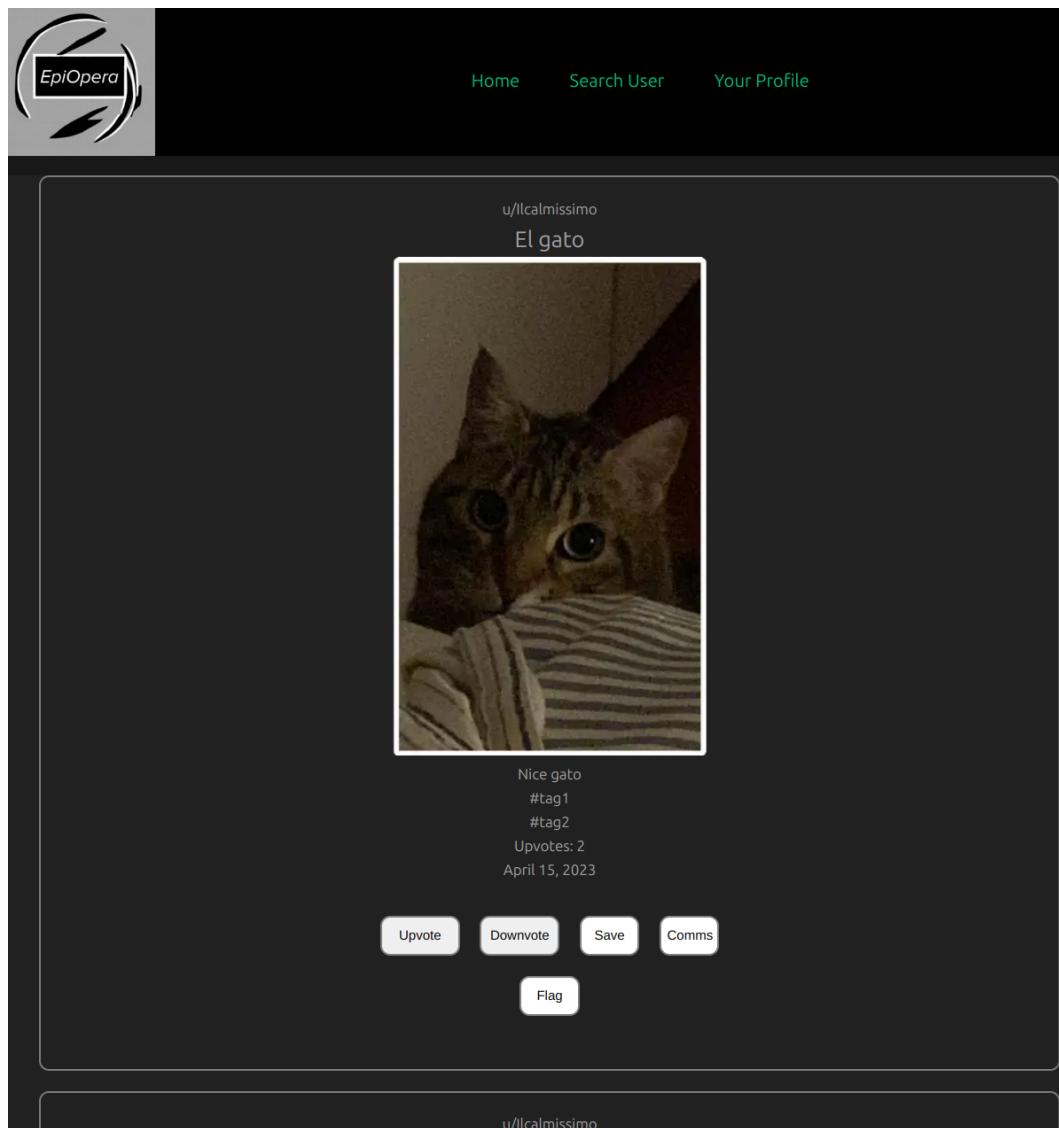
Implementazione Front End

Le sezioni visualizzabili nel sito sono 3:

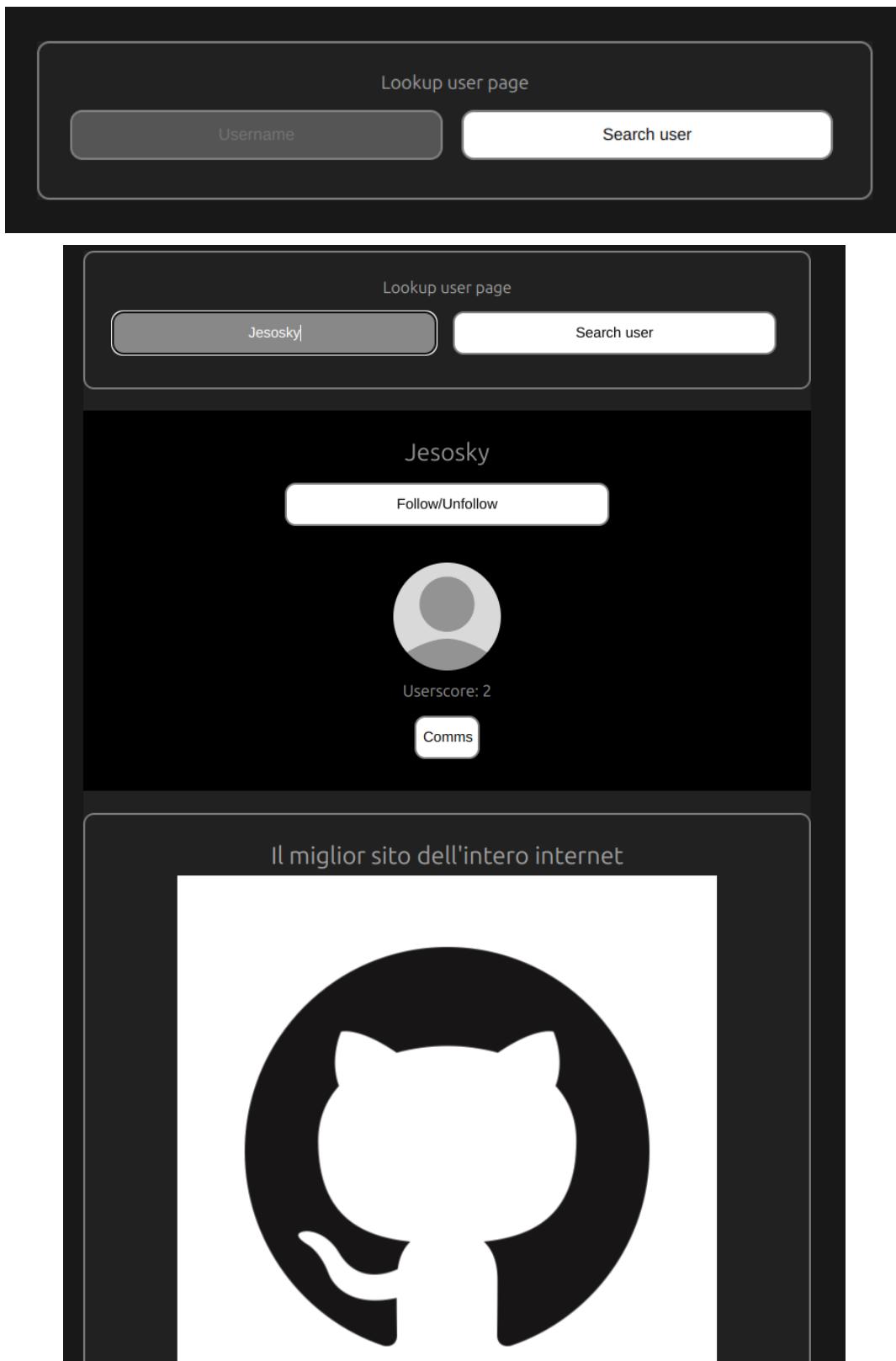
- Home
- Search User
- Your profile

In queste pagine si può rispettivamente:

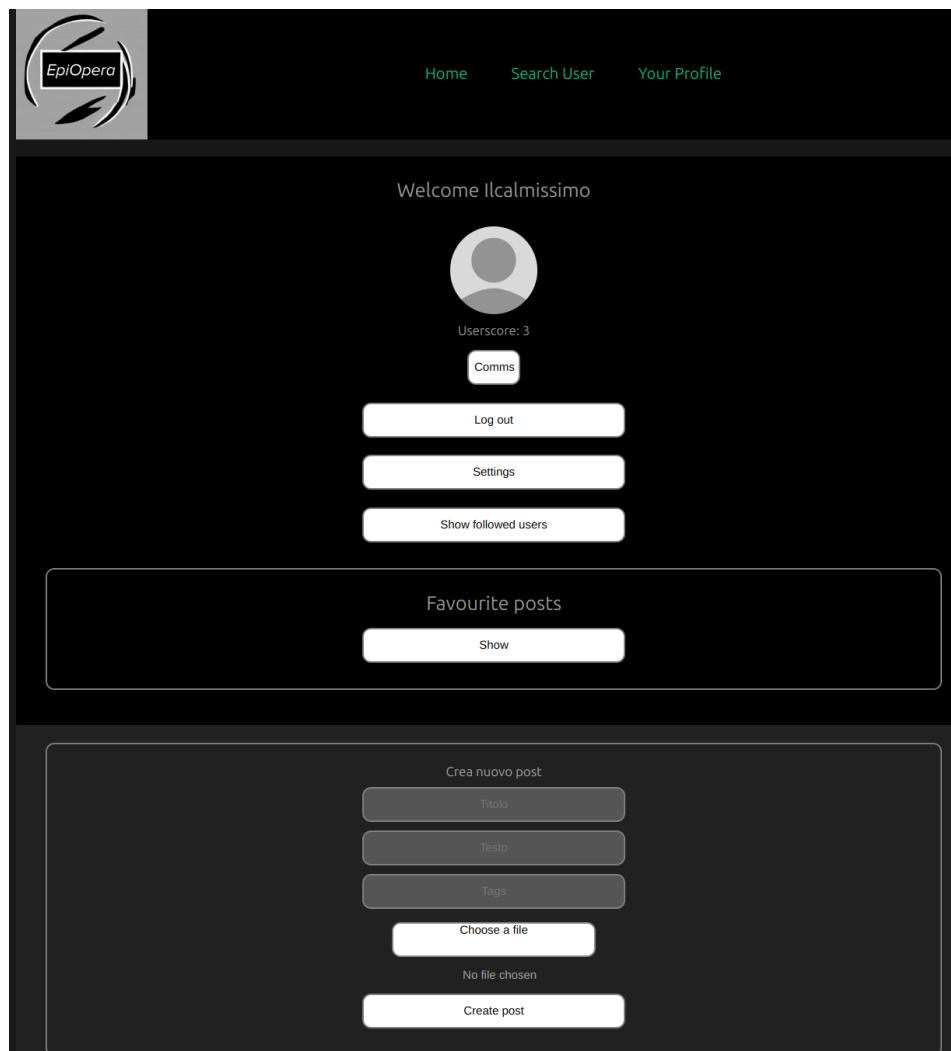
- Scorrere la lista di post presenti sul sito ed interagire tramite i 5 pulsanti mostrati in immagine. In questa pagina è possibile notare tutte le informazioni riguardanti il post ottenute dal database.

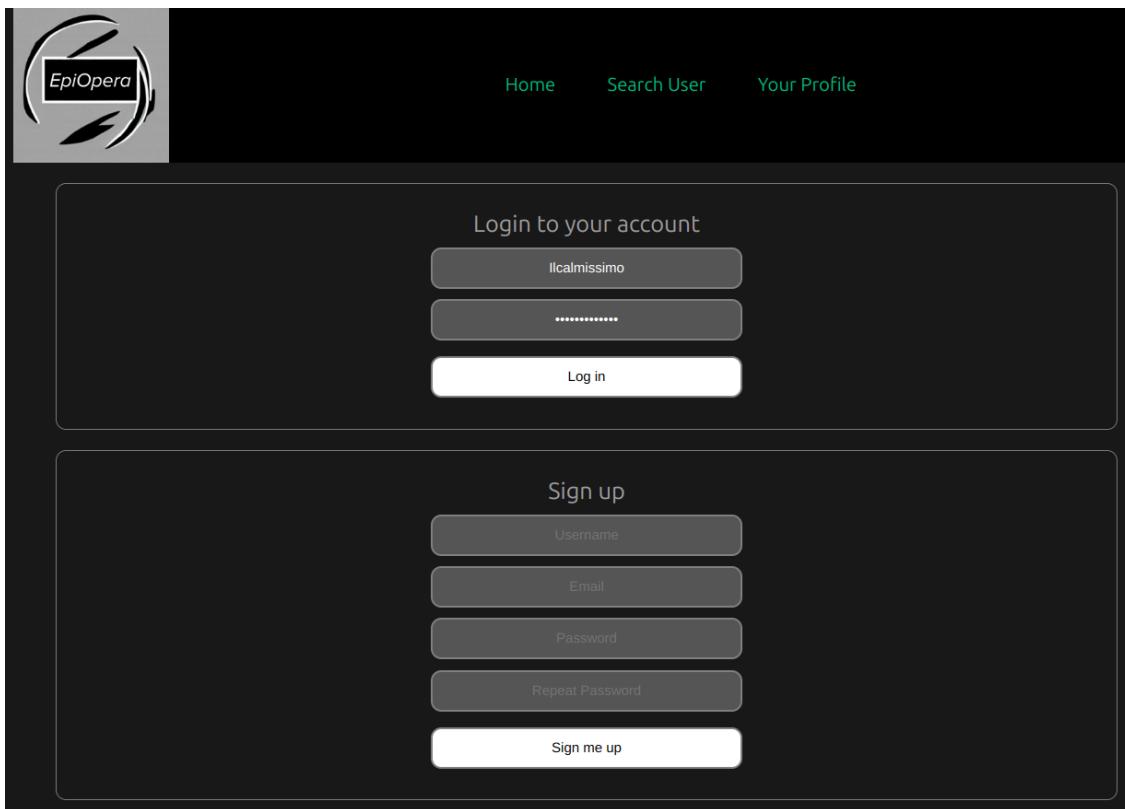


- Cercare utenti e visualizzarne le varie statistiche del profilo, i propri post ed interagire. Dovesse il profilo cercato non esistente l'utente verrà notificato tramite un apposito avvertimento



- Visualizzare il proprio profilo, le interazioni, le statistiche e accedere alle impostazioni riguardo ad esso, se il login non dovesse essere effettuato si avrà la possibilità di eseguirlo in questa pagina, oltre alla registrazione. I vari inserimenti testuali offrono controlli sulla correttezza del contenuto inserito, per esempio mostrano un avvertimento dovesse essere la password inserita non corretta. Ciò funziona anche per la creazione di un post la cui restrizione riguardo all'esclusività tra media e testo impone che l'utente venga notificato se dovesse provare a creare un post con sia media che testo.



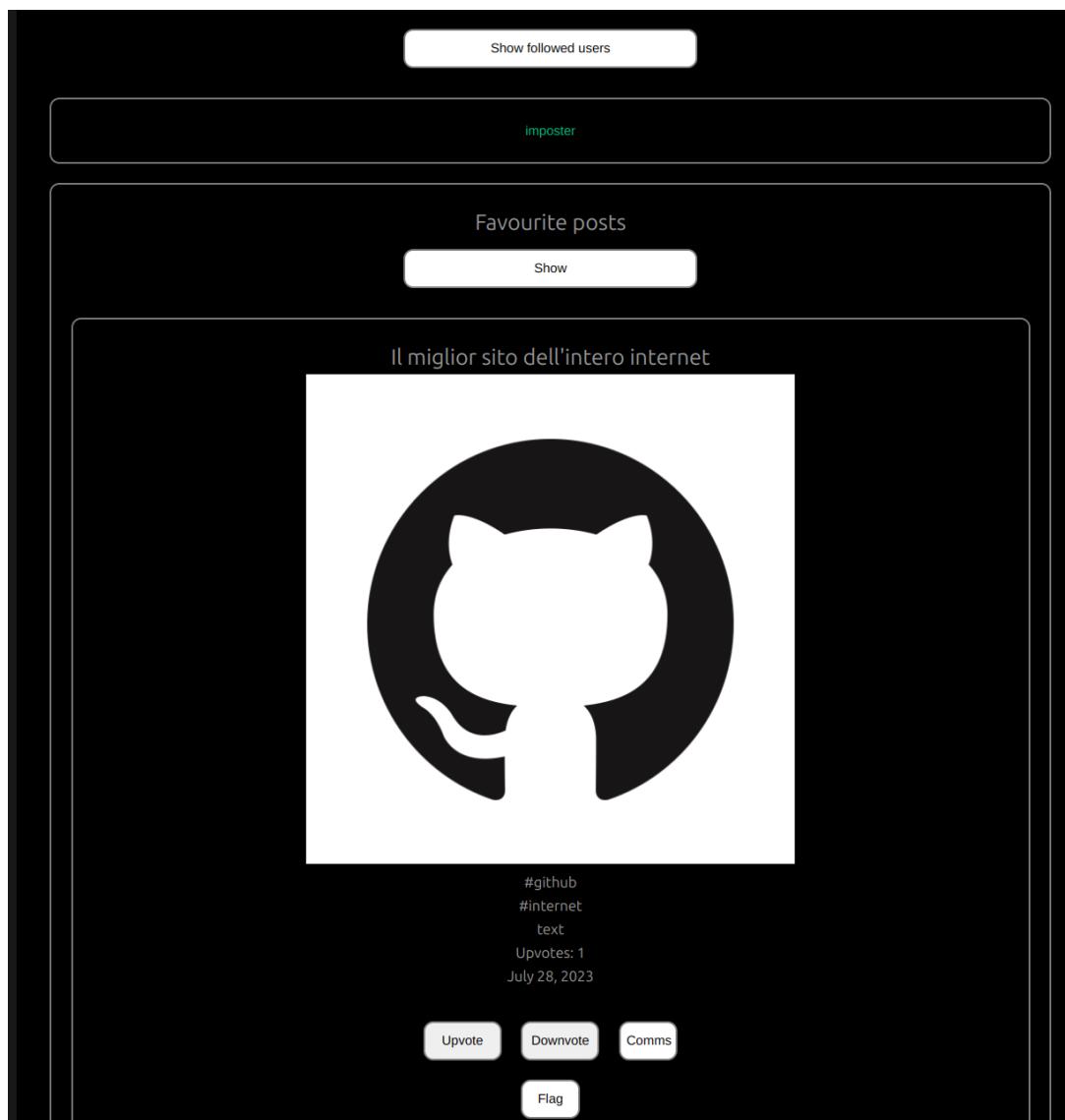


The screenshot shows the EpiOpera user interface. At the top left is the EpiOpera logo. The top navigation bar includes links for "Home", "Search User", and "Your Profile". Below this is a "Login to your account" section with fields for "Username" (containing "Ilcalmissimo") and "Password" (containing "*****"), followed by a "Log in" button. To the right is a "Sign up" section with fields for "Username", "Email", "Password", and "Repeat Password", followed by a "Sign me up" button.



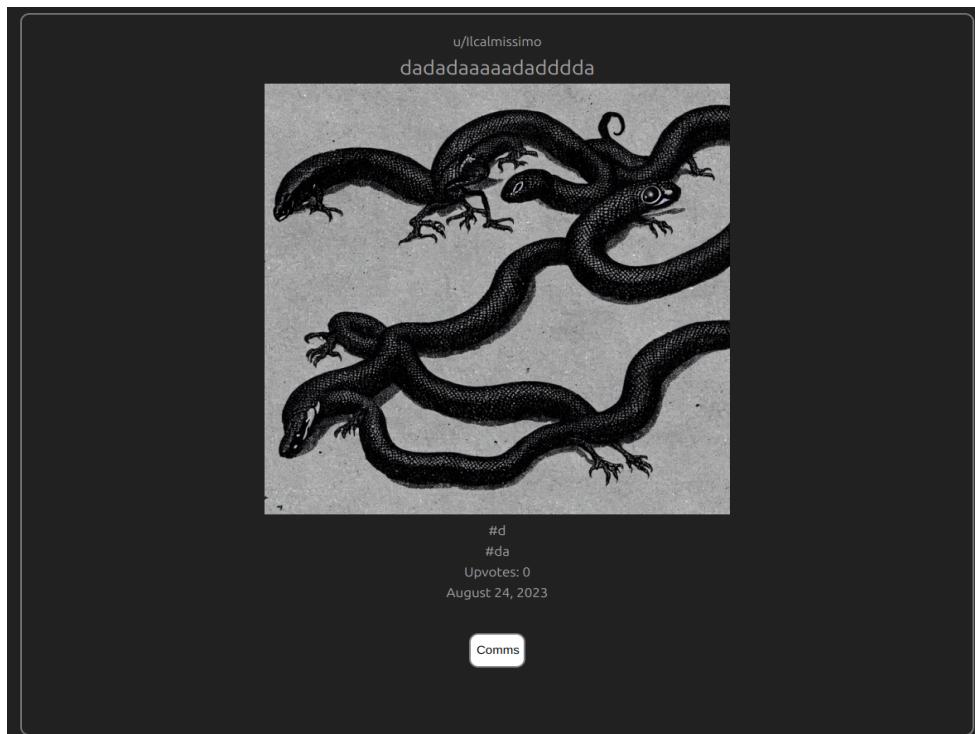
The screenshot shows the "Settings" page. It features three input fields with "Submit" buttons: "New Email address", "New Password", and "Repeat Password". Below these is a dropdown menu labeled "NSFW:" with a "Submit" button next to it. At the bottom is a large "Delete this account" button.

La pagina offre anche una sezione “settings” in cui modificare le impostazioni del proprio profilo ed eventualmente eliminare l’account.

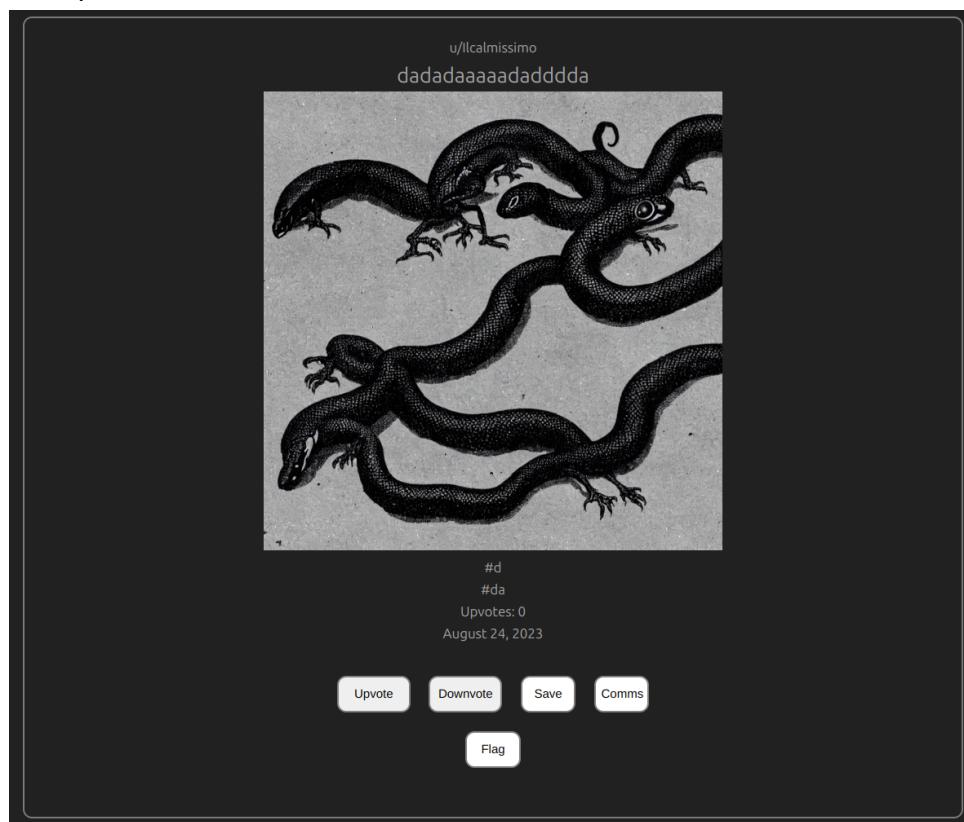


In ultimis la pagina offre due sezioni: “Followed users” dove possono essere visualizzati i nomi degli utenti seguiti e, cliccando, essere reindirizzati alla pagina del profilo seguito; “Favourite posts” dove vengono listati i post che sono stati marcati come preferiti.

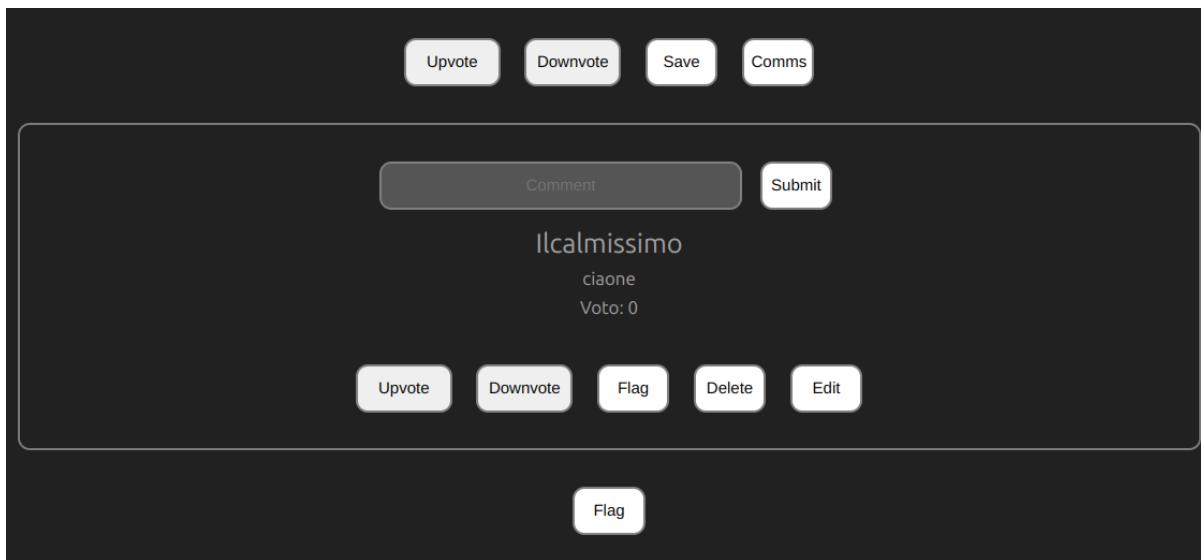
L'elemento fondamentale del sito è il "post" che può essere visualizzato sia da un utente registrato che da un visitatore non autenticato.



In questo primo caso il post è visualizzato da un visitatore non autenticato, notare che l'unica interazione disponibile è visualizzare i commenti.



In questo secondo caso il post è visualizzato da un utente autenticato, ora le interazioni disponibili sono numerose.



Come per il post, la schermata di visualizzazione dei commenti renderà disponibile molte interazioni all'utente registrato, mentre consentirà solo la visualizzazione all'utente visitatore.

GitHub Repository e Info sul Deployment

L'intero codice del sito è consultabile alla repository:

<https://github.com/SE2022-2023-groupT18/SE2022-2023-Marostica-Masellis-Marchini.git>

Il gruppo ha scelto di posizionare l'intero progetto in una sola repository vista la comodità riscontrata nel successivo sviluppo derivante dalla centralità dello sviluppo.

Il progetto è diviso in queste directory:

- coverage: i risultati del processo di testing
- dist: il prodotto del processo di building della parte di front end
- media: i vari contenuti multimediali tenuti dal sito (idealemente questa directory dovrebbe essere sostituita da un database apposito per i media)
- public: vari file pubblici
- server: il back end dell'applicazione, contiene il file server.js, il file principale del back end
- src: il front end dell'applicazione prima del building
- testing_proof: alcune immagini contenenti i risultati del testing

Il deployment è stato eseguito con Heroku in un unica applicazione che contiene sia back end che front end. Per gestire questo fatto il server usa le linee di codice sotto riportate per attivare istantaneamente e gestire il front end.

```
app.use(express.static(path.join(__dirname, '../dist')));
app.use('/media', express.static(path.join(__dirname, '../media')));

app.get('*', (req, res) => {
  res.sendFile(path.join(__dirname, '../dist', 'index.html'));
})
```



Tramite questo link si può accedere al sito distribuito:

<https://epiopera-4f1c76fdd577.herokuapp.com/>

Testing

Il testing è stato effettuato tramite le librerie Jest e supertest, i casi di test delle API sono espressi nei vari file nella directory /server/tests.

I risultati sono mostrati nelle seguenti immagini.

All files

92.25% Statements 655/710 78.66% Branches 188/239 96.66% Functions 87/90 92.79% Lines 644/694

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements	Branches	Functions	Lines
server	93.75%	30/32	100%	66.66% 2/3 93.75% 30/32
server/controllers	90.29%	465/515	75.24%	152/202 98.73% 78/79 91.03% 457/502
server/middleware	96.42%	27/28	100%	4/4 100% 2/2 96.29% 26/27
server/models	100%	16/16	100%	0/0 100% 0/0 100% 16/16
server/routes	100%	64/64	100%	0/0 100% 0/0 100% 64/64
server/util	96.36%	53/55	96.96%	32/33 83.33% 5/6 96.22% 51/53

All files server/controllers

90.29% Statements 465/515 75.24% Branches 152/202 98.73% Functions 78/79 91.03% Lines 457/502

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements	Branches	Functions	Lines
commento_post.js	87.9%	109/124	71.42%	35/49 95.23% 20/21 88.52% 108/122
commento_profilo.js	86.48%	96/111	69.38%	34/49 100% 19/19 87.96% 95/108
post.js	91.36%	127/139	73.68%	42/57 100% 19/19 91.85% 124/135
upload.js	84.61%	11/13	75%	3/4 100% 2/2 84.61% 11/13
utente.js	95.31%	122/128	88.37%	38/43 100% 18/18 95.96% 119/124

All files server/middleware

96.42% Statements 27/28 100% Branches 4/4 100% Functions 2/2 96.29% Lines 26/27

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements	Branches	Functions	Lines
auth.js	100%	14/14	100%	2/2 100% 1/1 100% 13/13
refreshToken.js	92.85%	13/14	100%	2/2 100% 1/1 92.85% 13/14

**All files server/models**

100% Statements 16/16 100% Branches 0/0 100% Functions 0/0 100% Lines 16/16

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.Filter:

File ▲	Statements	Branches	Functions	Lines
commento_post.js	100%	4/4	100%	100%
commento_profilo.js	100%	4/4	100%	100%
post.js	100%	4/4	100%	100%
utente.js	100%	4/4	100%	100%

All files server/routes

100% Statements 64/64 100% Branches 0/0 100% Functions 0/0 100% Lines 64/64

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.Filter:

File ▲	Statements	Branches	Functions	Lines
commento_post.js	100%	13/13	100%	100%
commento_profilo.js	100%	13/13	100%	100%
post.js	100%	15/15	100%	100%
upload.js	100%	6/6	100%	100%
utente.js	100%	17/17	100%	100%

All files server

93.75% Statements 30/32 100% Branches 0/0 66.66% Functions 2/3 93.75% Lines 30/32

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.Filter:

File ▲	Statements	Branches	Functions	Lines
server.js	93.75%	30/32	66.66%	93.75%

All files server/util

96.36% Statements 53/55 96.96% Branches 32/33 83.33% Functions 5/6 96.22% Lines 51/53

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.Filter:

File ▲	Statements	Branches	Functions	Lines
textRequirements.js	95.83%	46/48	80%	95.65%
token.js	100%	7/7	100%	100%

I casi in cui non è stato eseguito il 100% del codice sono casi di controllo del corretto funzionamento del database, fallimenti in questa fase non sono riproducibili se non tramite

un effettivo malfunzionamento del database MongoDB. Tutto il resto del codice è stato correttamente eseguito e testato.

Inoltre questi stessi risultati mostrati sono visualizzabili accedendo al file index.html nella directory /coverage/lcov-report.