



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza
dell'Informazione



Deliverable 4: Sviluppo applicazione

Gruppo T16:

-  Hossain Rafiu
-  Missagia Fabio
-  Posenato Alessandro

Sommario

Scopo del documento	4
1. User flows	5
1.1 Home Page	5
1.2 Pubblicazione annuncio	6
1.3 Lista annunci	6
1.4 Login	7
1.5 Registrazione	7
1.6 Profilo	8
1.7 Annunci salvati	8
1.8 Ricerche salvate	9
1.9 Messaggi	9
1.10 Annunci pubblicati	9
1.11 Modifica dati	10
1.12 Logout	10
1.13 Cancellazione account	10
2. Application Implementation and Documentation	12
2.1 Project Structure	13
Backend	14
Frontend	15
2.2 Project Dependencies	16
Backend	16
Frontend	17
2.3 Project Data or Database	18
Utente	18
Annuncio	19
Locale	19
Chat	19
Messaggi	20
Ricerca	20
Filtro	20
2.4 Project APIs	21
2.4.1 Resources Extraction from the Class Diagram	21
Utente	22
GET APIs:	23

POST APIs:	23
Annuncio	24
GET APIs:	25
POST APIs:	25
DELETE APIs:	25
Ricerca	26
POST APIs:	26
Chat	27
GET APIs:	27
Messaggio	28
POST APIs:	28
2.4.2 Resource Models	30
Utente	30
Login	30
Registrazione	31
Logout	32
Recupero Password	33
Modifica profilo	33
Elimina profilo	34
Annunci salvati	35
Annunci pubblicati	36
Ricerche Salvate	37
Visualizza Lista Chat	38
Annuncio	39
Visualizza lista annunci	39
Visualizza annuncio	40
Salva annuncio	41
Rimuovi Annuncio Salvato	42
Pubblica annuncio	42
Modifica annuncio	43
Elimina annuncio	44
Pagamento	45
Ricerca	46
Ricerca annunci	46
Salva Ricerca	46
Rimuovi Ricerca Salvata	47
Chat	48

Crea Chat	48
Apri Chat	49
Messaggio	50
Invia Messaggio	50
2.5 Sviluppo API	51
2.5.1 Token checker middleware	51
2.5.2 Login	52
2.5.3 Registrazione	53
2.5.4 Annunci pubblicati	55
2.5.5 Cancella account	56
2.5.6 Pubblicazione annuncio	57
2.5.7 Visualizza annuncio	59
2.5.8 Salva annuncio	60
2.5.9 Elimina annuncio	62
2.5.10 Rimozione annuncio salvato	63
2.5.11 Salva ricerca	64
2.5.12 Crea chat	66
2.5.13 Invio messaggio	67
3. Documentazione delle API	69
4. Implementazione frontend	73
4.1 Home	73
4.2 Registrazione	74
4.3 Login	74
4.4 Profilo	75
4.5 Pubblicazione annuncio	76
4.6 Annunci pubblicati	78
4.7 Annunci salvati	78
5. Repository GitHub e informazioni sul deployment	79
5.1 Struttura Github repository	79
5.2 Informazioni deployment	79
6. Testing	81
6.1 Tests	81
6.2 Coverage	84



Scopo del documento

Il presente documento fornisce tutte le informazioni necessarie per lo sviluppo dell'applicazione HouseFinder.

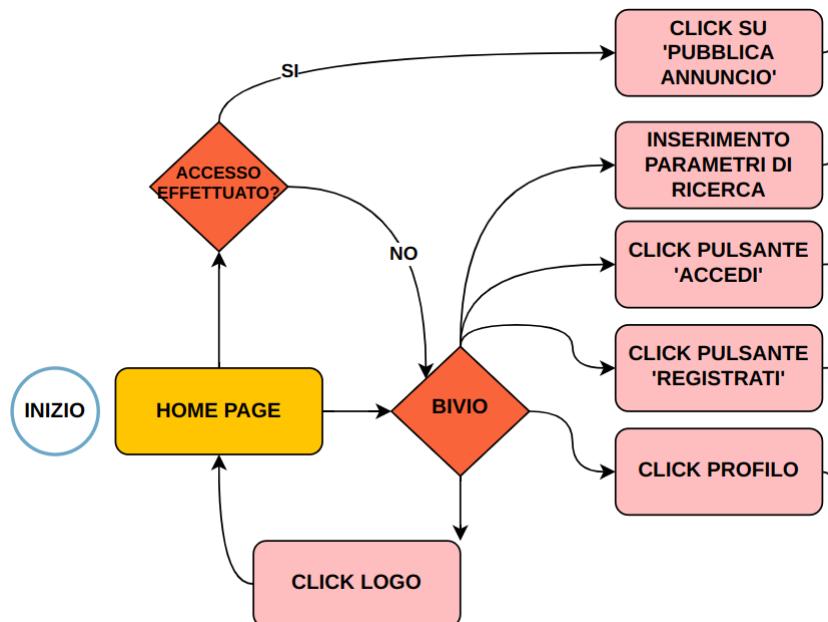
Partendo dalla descrizione dei flussi utente legati al ruolo dell'utente nell'applicazione, il documento prosegue con la presentazione delle API necessarie (tramite l'API Model e il modello delle risorse) per visualizzare, inserire e modificare gli annunci.

Per ogni API realizzata, oltre ad una descrizione delle funzionalità fornite, il documento include la sua documentazione e i test eseguiti. Infine, una sezione è dedicata alle informazioni del repository Git e al deployment dell'applicazione stessa.

1. User flows

In questa sezione del documento di sviluppo riportiamo lo “user flow” per l’utente del nostro sito. La figura descrive il flusso di azioni che l’utente può eseguire all’interno del sito. Di seguito si trova la spiegazione in dettaglio del diagramma che si può trovare per intero al seguente [link](#).

1.1 Home Page



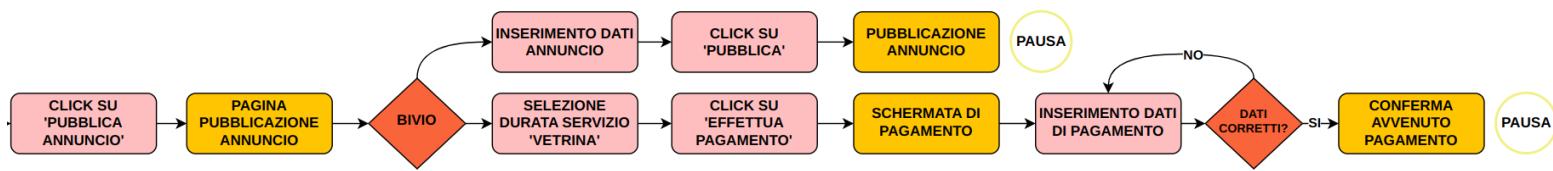
Dalla ‘Home Page’ si possono eseguire le seguenti operazioni:

- CLICK LOGO
- CLICK SU ‘PUBBLICA ANNUNCIO’
- INSERIMENTO PARAMETRI DI RICERCA
- CLICK PULSANTE ‘ACCEDI’
- CLICK PULSANTE ‘REGISTRATI’
- CLICK PROFILO

Cliccando sul logo si viene rimandati alla Home Page.

Il pulsante ‘Pubblica Annuncio’ sarà visibile solo quando l’utente avrà effettuato l’accesso, dunque è presente un controllo per vedere se l’utente può accedere a questa funzionalità. Le altre operazioni verranno spiegate in dettaglio nelle prossime sezioni.

1.2 Pubblicazione annuncio



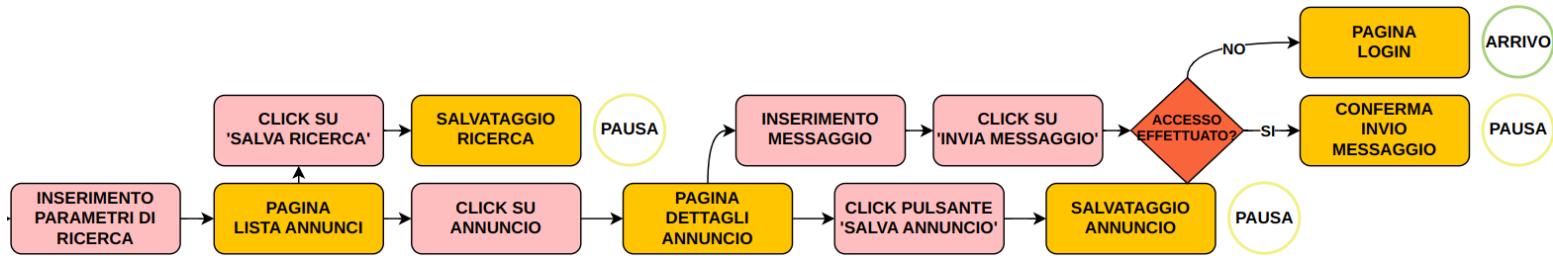
Dopo aver cliccato su 'Pubblica annuncio', l'utente viene rimandato alla pagina 'Pubblicazione Annuncio'.

Da qui avrà la possibilità di inserire i dati dell'annuncio e cliccando sul pulsante 'Pubblica' il sistema si occuperà della pubblicazione di esso.

Inoltre l'utente potrà decidere di acquistare il servizio 'Vetrina'. Per far ciò dovrà selezionare la durata del servizio e cliccando su 'Effettua Pagamento' verrà reindirizzato alla 'Schermata di Pagamento'.

Da qui potrà inserire i dati di pagamento e, in caso questi ultimi siano corretti, effettuare il pagamento. Una volta completata l'operazione, il sistema fornirà all'utente una notifica sull'esito positivo o negativo del pagamento.

1.3 Lista annunci



Inserendo i parametri di ricerca sulla barra di ricerca della 'Home page' si ottiene una pagina con la lista degli annunci che rispettano i parametri.

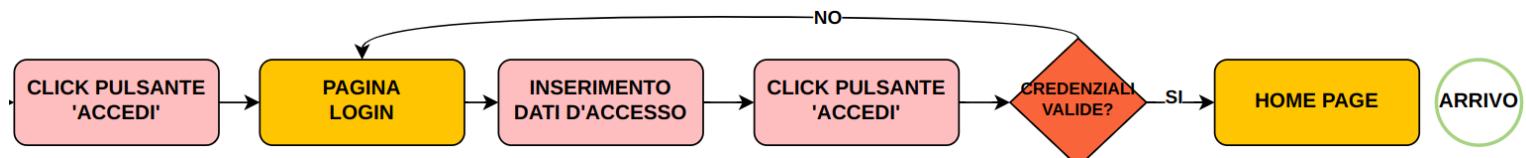
Dalla pagina si può premere il tasto 'Salva Ricerca' per salvare i parametri della ricerca appena effettuata.

È poi possibile cliccare su uno qualsiasi degli annunci per visualizzarne la pagina dei dettagli.

Dalla pagina 'Dettagli Annuncio' è possibile inserire ed inviare un messaggio al locatore.

Quando il tasto 'Invia Messaggio' viene premuto il messaggio viene inviato solo se l'utente ha effettuato l'accesso altrimenti si verrà rinviati alla pagina di login. Infine da questa pagina si può cliccare sul tasto 'Salva Annuncio' che metterà l'annuncio nella sezione 'Annunci Salvati' del profilo.

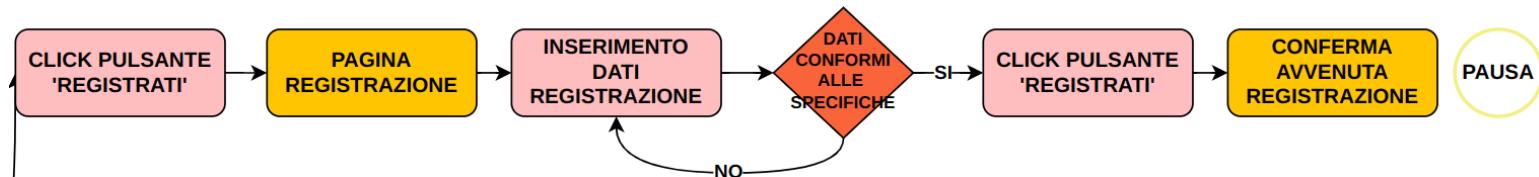
1.4 Login



Premendo sul tasto 'Accedi' si viene rinviai alla pagina di login. Da qui si potranno inserire i dati d'accesso e cliccando sul tasto 'Accedi' il sistema verificherà le credenziali.

Se le credenziali sono valide l'utente sarà reindirizzato alla 'Home Page', mentre se le credenziali non sono valide l'utente dovrà reinserirle correttamente.

1.5 Registrazione



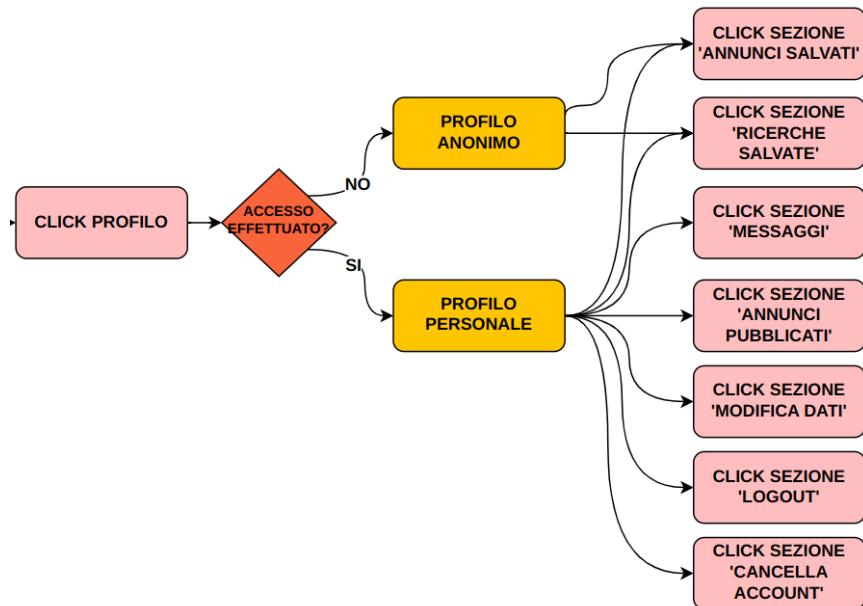
Premendo sul tasto 'Registrati' si accede alla pagina di registrazione. L'utente potrà inserire i dati di registrazione, i quali verranno controllati dal sistema per vedere se sono conformi alle specifiche.

I dati sono conformi se:

- tutti i campi sono stati compilati
- i campi nome e cognome contengono solo lettere
- il numero di telefono contiene solo cifre
- il campo e-mail contiene una e-mail valida
- il contenuto del campo password e quello di conferma password sono identici
- la casella 'Accetto i Termini e le condizioni' è spuntata
- la password è almeno lunga 8 caratteri, ha almeno un numero, ha almeno una maiuscola e ha almeno un carattere speciale.

Una volta che i dati sono conformi sarà possibile cliccare il tasto 'Registrati'. In seguito, il sistema confermerà il corretto avvenimento della registrazione.

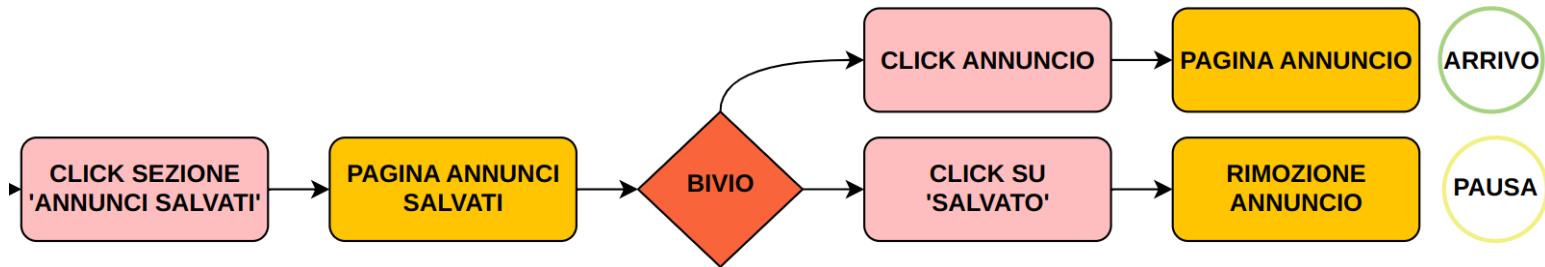
1.6 Profilo



Premendo sul profilo si accede all'area personale.

Se non si ha effettuato l'accesso le uniche sezioni disponibili saranno quelle di 'Annunci Salvati' e 'Ricerche Salvate', altrimenti l'utente potrà accedere a tutte le altre sezioni.

1.7 Annunci salvati

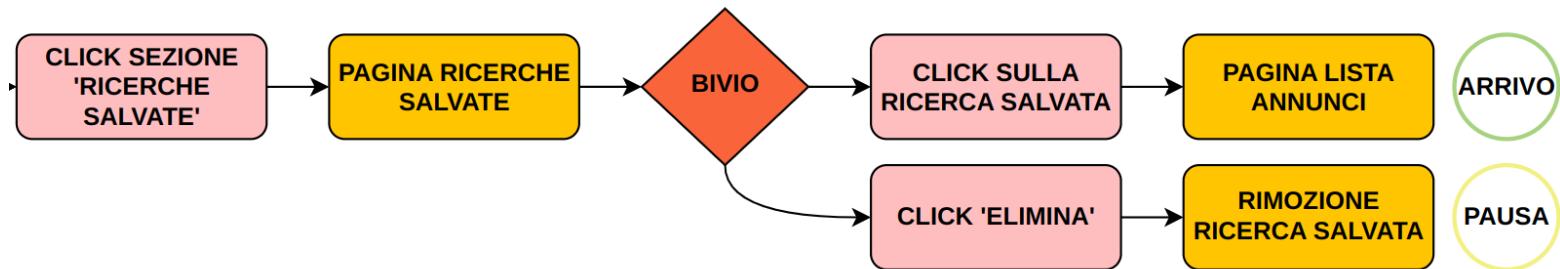


Premendo su 'Annunci Salvati' si accede alla pagina 'Annunci Salvati'. Qui si trova la lista degli annunci che l'utente ha deciso di salvare.

Cliccando su 'Salvato' è possibile rimuovere l'annuncio da questa pagina.

Cliccando su un qualsiasi annuncio si viene reindirizzati alla pagina dei dettagli dell'annuncio.

1.8 Ricerche salvate

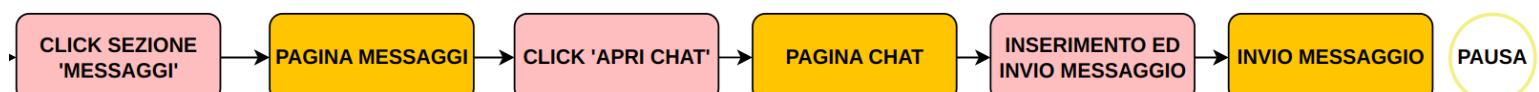


Premendo su 'Ricerche Salvate' si accede alla pagina 'Ricerche Salvate'. Qui si trova la lista delle ricerche che l'utente ha deciso di salvare.

Cliccando su 'Elimina' è possibile rimuovere la ricerca da questa pagina.

Cliccando su una qualsiasi ricerca viene effettuata una ricerca con gli stessi parametri della ricerca salvata.

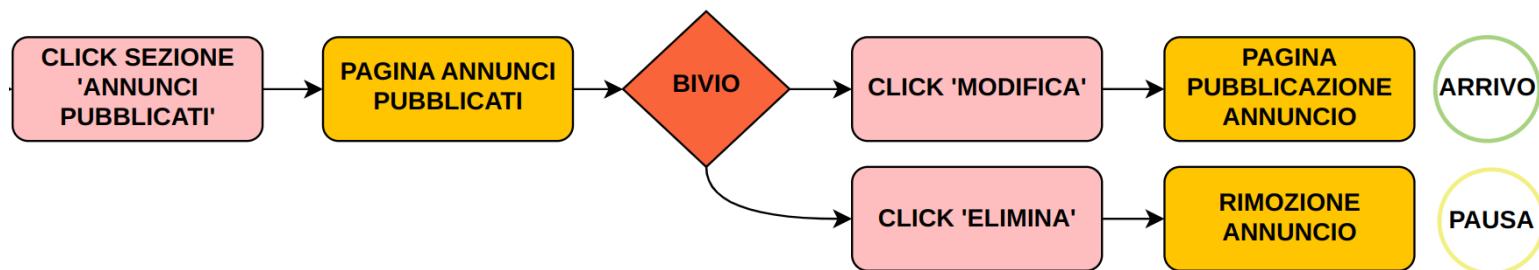
1.9 Messaggi



Premendo su 'Messaggi' si accede alla pagina dei messaggi. Qui si trovano le chat con gli altri utenti.

Cliccando su 'Apri Chat' si apre la conversazione con il corrispettivo utente. Da qui sarà possibile scrivere ed inviare messaggi.

1.10 Annunci pubblicati

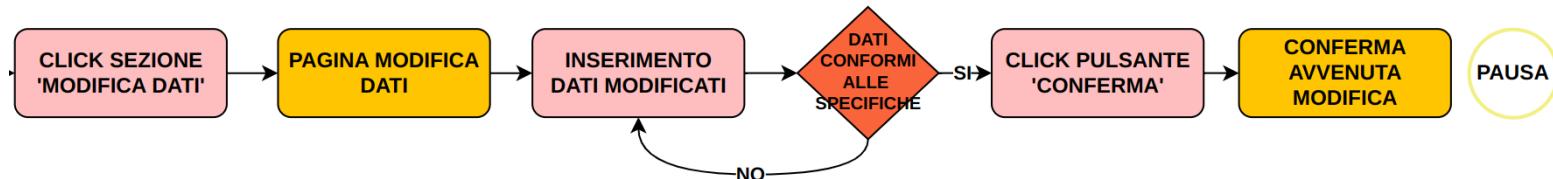


Premendo su 'Annunci Pubblicati' si accede alla pagina 'Annunci Pubblicati'. Qui si trovano tutti gli annunci degli immobili pubblicati dall'utente.

Cliccando su 'Elimina' è possibile rimuovere l'annuncio dal sito e da questa pagina.

Cliccando su 'Modifica' si viene reindirizzati alla pagina 'Pubblicazione Annuncio', dove saranno già inseriti i dati attuali dell'annuncio per permettere all'utente di modificarli.

1.11 Modifica dati



Premendo su 'Modifica dati' si accede alla pagina 'Modifica Dati'. Da questa pagina è possibile modificare tutti i dati personali dell'utente.

Una volta inseriti i nuovi dati il sistema verificherà che essi siano conformi alle specifiche (le stesse della registrazione). Se i dati sono conformi allora si potrà cliccare il tasto 'Conferma'.

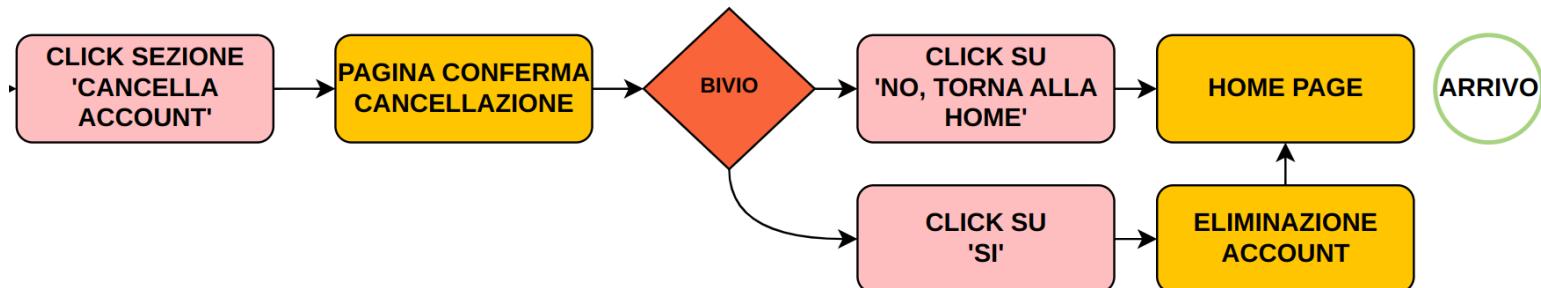
In seguito il sistema notificherà l'utente sull'esito positivo o negativo dell'operazione di modifica.

1.12 Logout



Cliccando su 'Logout' verrà terminata la sessione corrente dell'utente. Una volta terminata la sessione l'utente verrà reindirizzato alla 'Home Page'.

1.13 Cancellazione account



Cliccando su 'Cancella Account' è possibile eliminare il proprio account con tutti i dati annessi.

Per portare a termine la cancellazione è necessario confermare dalla pagina 'Conferma Cancellazione'. Qui verrà chiesto all'utente se è sicuro di eliminare l'account. Cliccando su 'NO' l'utente viene rinviato alla 'Home Page'. Cliccando su 'Sì' l'account verrà eliminato e l'utente viene rinviato alla 'Home Page'.

2. Application Implementation and Documentation

Nelle sezioni precedenti abbiamo individuato le funzionalità che devono essere implementate nella nostra applicazione, con un'idea di come l'utente sarebbe andato ad utilizzarle.

L'applicazione è stata sviluppata utilizzando NodeJS e usando VueJS come framework per il frontend. Le informazioni e i dati vengono salvati e gestiti nel database, usando MongoDB.

A causa del numero e della complessità delle funzionalità che verranno descritte nei paragrafi successivi, abbiamo deciso di non includere la funzionalità di messaggistica all'interno del sistema. Tuttavia, riteniamo che il nucleo principale del nostro sistema, che consiste nell'interazione con gli annunci, rimanga sufficientemente completo, permettendoci di applicare le conoscenze acquisite durante il corso.

Un'altra parte del sistema che abbiamo scelto di non sviluppare riguarda il processo di pagamento della vetrina negli annunci. Questa decisione è stata presa considerando il fatto che l'integrazione di sistemi di pagamento online (nel nostro caso, Stripe) richiederebbe una conoscenza approfondita delle complesse API fornite dal sistema.

2.1 Project Structure

Il progetto è stato strutturato in due repository: una per il backend ed una per il frontend.

Nei seguenti paragrafi descriveremo l'organizzazione di entrambe le parti utilizzando screenshot che mostrano la struttura delle cartelle.



Backend

Il repository del backend è strutturato come in figura. Sono presenti le seguenti cartelle:

- **controllers**

In questa cartella si trova la logica principale dell'applicazione, che comprende i metodi chiamati quando viene ricevuta una richiesta dalle API.

- **models**

In questa cartella vengono raggruppate le definizioni dei modelli utilizzati da mongoose per salvare informazioni su MongoDB.

- **routes**

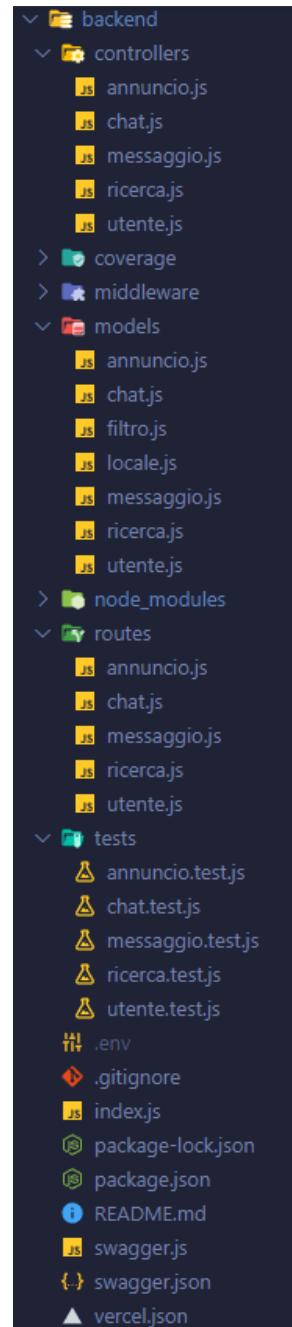
I file presenti in questa cartella si occupano di gestire le richieste e di associarle al controller appropriato.

- **middleware**

In questa cartella è contenuto il token checker, il quale verifica l'autenticità e i privilegi dell'utente attraverso l'analisi del token fornito con la richiesta. In questo modo, l'API distingue tra gli utenti autorizzati e quelli non autorizzati, consentendo solo agli utenti autenticati con i privilegi appropriati di accedere alle risorse protette.

- **tests**

In questa cartella sono contenuti i test automatizzati realizzati utilizzando il framework Jest. I test sono progettati per verificare il corretto funzionamento delle diverse componenti dell'applicazione.



Nella root sono inoltre presenti il file index.js, ossia il punto di avvio dell'applicazione, il file swagger.json, contenente le informazioni per visualizzare la pagina di documentazione, e altri file relativi alla gestione di Git, Node o dei suoi pacchetti.



Frontend

Il repository del frontend è strutturato come in figura. Sono presenti le seguenti cartelle:

- **src**

In questa cartella si trova la logica principale dell'applicazione Vue, divisa in varie sottocartelle.

- **assets**

In questa cartella sono contenute le immagini e i fogli di stile.

- **components**

In questa cartella sono contenuti i componenti Vue utilizzati nell'applicazione, ossia blocchi modulari di codice che rappresentano parti riutilizzabili dell'interfaccia utente.

- **router**

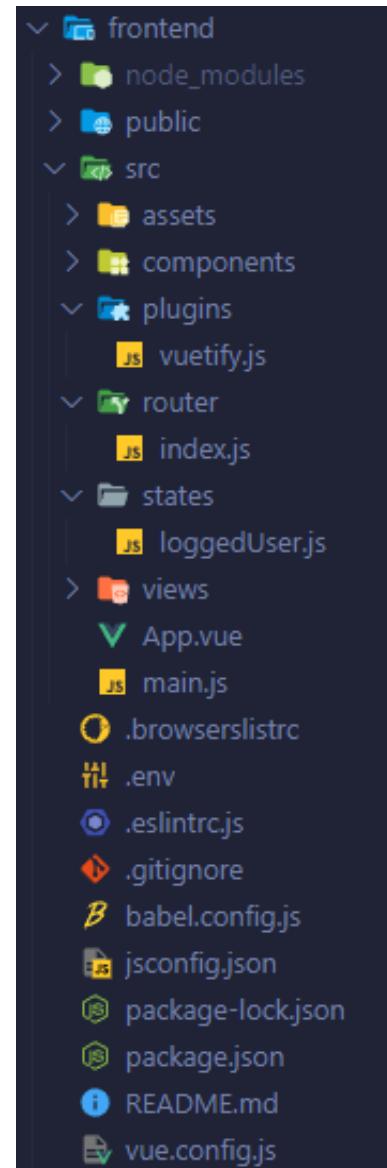
Questa cartella contiene la definizione del router dell'applicazione, che si occupa di gestire la navigazione all'interno dell'applicazione Vue. Attraverso il router, vengono definiti i percorsi (routes) che corrispondono a diverse viste dell'applicazione.

- **states**

In questa cartella è contenuto il file responsabile della gestione dello stato dell'utente durante il processo di accesso e autenticazione.

- **views**

In questa cartella sono contenute le diverse pagine dell'applicazione, ciascuna rappresentante una vista specifica e dedicata.



È poi presente il file App.vue, che contiene il codice da cui viene avviato il frontend, e altri file utilizzati per far funzionare il progetto Vue.

2.2 Project Dependencies

I seguenti moduli Node sono necessari per il funzionamento dell'applicazione e sono stati aggiunti ai file `package.json`.

Backend

Per il backend sono indispensabili:

- **cors**

Semplifica la gestione delle richieste CORS (Cross-Origin Resource Sharing) per consentire le richieste da origini diverse.

- **dotenv**

Carica le variabili d'ambiente da un file `.env`, consentendo di gestire facilmente le configurazioni specifiche dell'ambiente.

- **express**

Un framework web veloce, flessibile e minimalista per Node.js che semplifica la creazione di applicazioni web e API.

- **jsonwebtoken**

Una libreria per la creazione e la verifica dei JSON Web Token (JWT), che sono utilizzati per autenticare e autorizzare le richieste tra client e server.

- **mongoose**

Semplifica l'interazione con il database MongoDB, fornendo un'interfaccia semplice per la definizione di modelli, la gestione delle query e altro ancora.

- **mongoose-unique-validator**

Semplifica la gestione dei campi univoci nei modelli, fornendo controlli di validazione e messaggi di errore predefiniti.

- **swagger-ui-express**

Fornisce un'interfaccia utente generata automaticamente per visualizzare e interagire con la documentazione di API Swagger, integrabile con Express.js.

- **jest**

Un framework di testing avanzato per JavaScript che offre funzionalità complete per l'esecuzione di test.

- **supertest**

Un modulo di supporto per Jest che semplifica l'invio di richieste HTTP agli endpoint API e la verifica delle risposte durante i test di integrazione.

- **nodemon**

Monitora i file e riavvia automaticamente l'applicazione quando vengono apportate modifiche, semplificando il processo di sviluppo e il debugging.

Frontend

Per il frontend sono indispensabili:

- **core-js**

Fornisce una raccolta di funzionalità polyfill per il supporto di caratteristiche ECMAScript (ES) più recenti in ambienti JavaScript più vecchi o meno supportati.

- **dotenv**

- **jsonwebtoken**

- **vue**

Il framework principale, utilizzato per la creazione di interfacce utente (UI) reattive e interattive.

- **vue-router**

Un routing ufficiale per Vue.js che consente di gestire in modo dichiarativo la navigazione tra le diverse viste o pagine dell'applicazione.

- **vuetify**

Un framework di componenti UI ricco di funzionalità per Vue.js che consente di creare rapidamente interfacce accattivanti e reattive. Fornisce un'ampia gamma di componenti predefiniti e strumenti di personalizzazione per soddisfare le esigenze di design dell'applicazione.

2.3 Project Data or Database

Per la gestione dei dati sono state definite 7 principali strutture dati. In seguito vengono descritte brevemente le strutture dati e ne viene presentato un esempio.

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
annuncios	11	2.48KB	231B	36KB	1	36KB	36KB
chats	1	64B	64B	20KB	1	20KB	20KB
filtros	1	71B	71B	20KB	1	20KB	20KB
locales	1	118B	118B	20KB	1	20KB	20KB
messaggi	1	68B	68B	36KB	1	20KB	20KB
ricercas	1	79B	79B	20KB	1	20KB	20KB
utentes	4	1.15KB	296B	36KB	2	72KB	36KB

Utente

La collection dell'utente memorizza le informazioni personali dell'utente, compresa la password che viene sottoposta a hash per garantire una maggiore sicurezza.

```
_id: ObjectId('63dec8a0e427fc4cf15fce')
nome: "Mario"
cognome: "Rossi"
data_nascita: 2001-11-21T23:00:00.000+00:00
numero_tel: "3485562489"
email: "mario.rossi@gmail.com"
password: "03ac674216f3e15c761ee1a5e255f067953623c8b388b4459e13f978d7c846f4"
> annunci_salvati: Array
> annunci_pubblicati: Array
> ricerche_salvate: Array
```

Annuncio

La collection dell'annuncio memorizza tutte le informazioni relative ad un annuncio pubblicato.

Il parametro *vetrina* viene utilizzato per identificare gli annunci che sono stati messi in vetrina, conferendo loro una priorità superiore rispetto agli altri.

```
_id: ObjectId('646a8eab0d36594c17097d97')
creatore: ObjectId('646a8b22e879344af80902d8')
▶ foto: Array
superficie_tot: 340
numero_bagni: 2
numero_locali: 5
▶ locali: Array
prezzo: 400
classe_energetica: "A"
indirizzo: "Via Sommarive 5"
arredato: false
▶ vetrina: Object
```

Locale

Questa collection memorizza le informazioni su uno specifico locale.

```
_id: ObjectId('63ef98f1a07ce6a51758b847')
nome_locale: "Cucina"
superficie: 50
> arredamento: Array
```

Chat

Questa collection memorizza le conversazioni tra un utente ed un locatore, in relazione ad un annuncio specifico.

La raccolta *messaggi* contiene i messaggi scambiati tra i due utenti durante la comunicazione.

```
_id: ObjectId('63ef95e3a07ce6a51758b845')
▶ messaggi: Array
annuncio: ObjectId('647c88b73cf4be8789298884')
locatore: ObjectId('647c888e3cf4be8789298883')
utente: ObjectId('647c88723cf4be8789298882')
```

Messaggi

Questa collection memorizza tutte le informazioni relative ad un messaggio inviato.

```
_id: ObjectId('63ef99c7a07ce6a51758b848')
messaggio: "Ciao, come stai?"
data: 2023-02-16T18:02:44.373+00:00
mittente: ObjectId('647c8a943cf4be8789298885')
```

Ricerca

Questa collection memorizza le informazioni relative ad una ricerca effettuata dall'utente, utili nel caso un utente voglia salvare la ricerca.

```
_id: ObjectId('63ef9adba07ce6a51758b849')
testo: "Via Sommarive, 11, 38123 Povo TN"
> filtri: Object
```

Filtro

Questa collection memorizza le informazioni su un filtro specifico applicato dall'utente, utile per ottenere rapidamente gli annunci desiderati.

```
_id: ObjectId('63ef9883a07ce6a51758b846')
nome_filtro: "prezzo"
min: 50000
max: 110000
```

2.4 Project APIs

In questa sezione vengono esplicitate le risorse da realizzare tramite il diagramma di estrazione delle risorse e il diagramma del modello delle risorse.

2.4.1 Resources Extraction from the Class Diagram

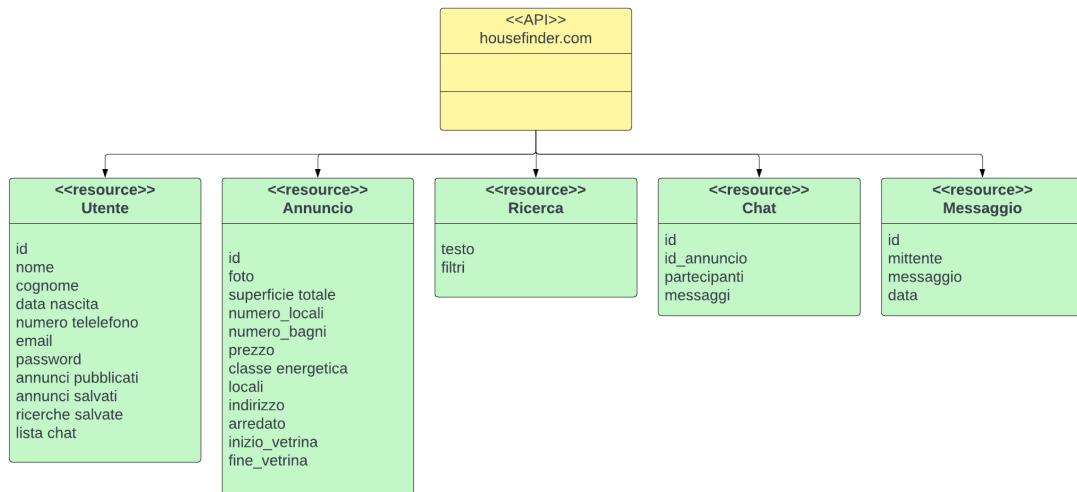
Di seguito è presente una rappresentazione visiva di tutte le risorse estratte dal diagramma delle classi, realizzato nel documento *D3: Documento di architettura*, identificando le classi che rappresentano le risorse e le relazioni tra tali classi.

Le risorse sono caratterizzate da un nome e da un metodo di utilizzo della risorsa, che può essere di tipo *GET*, *POST*, *PUT*, *PATCH* o *DELETE*.

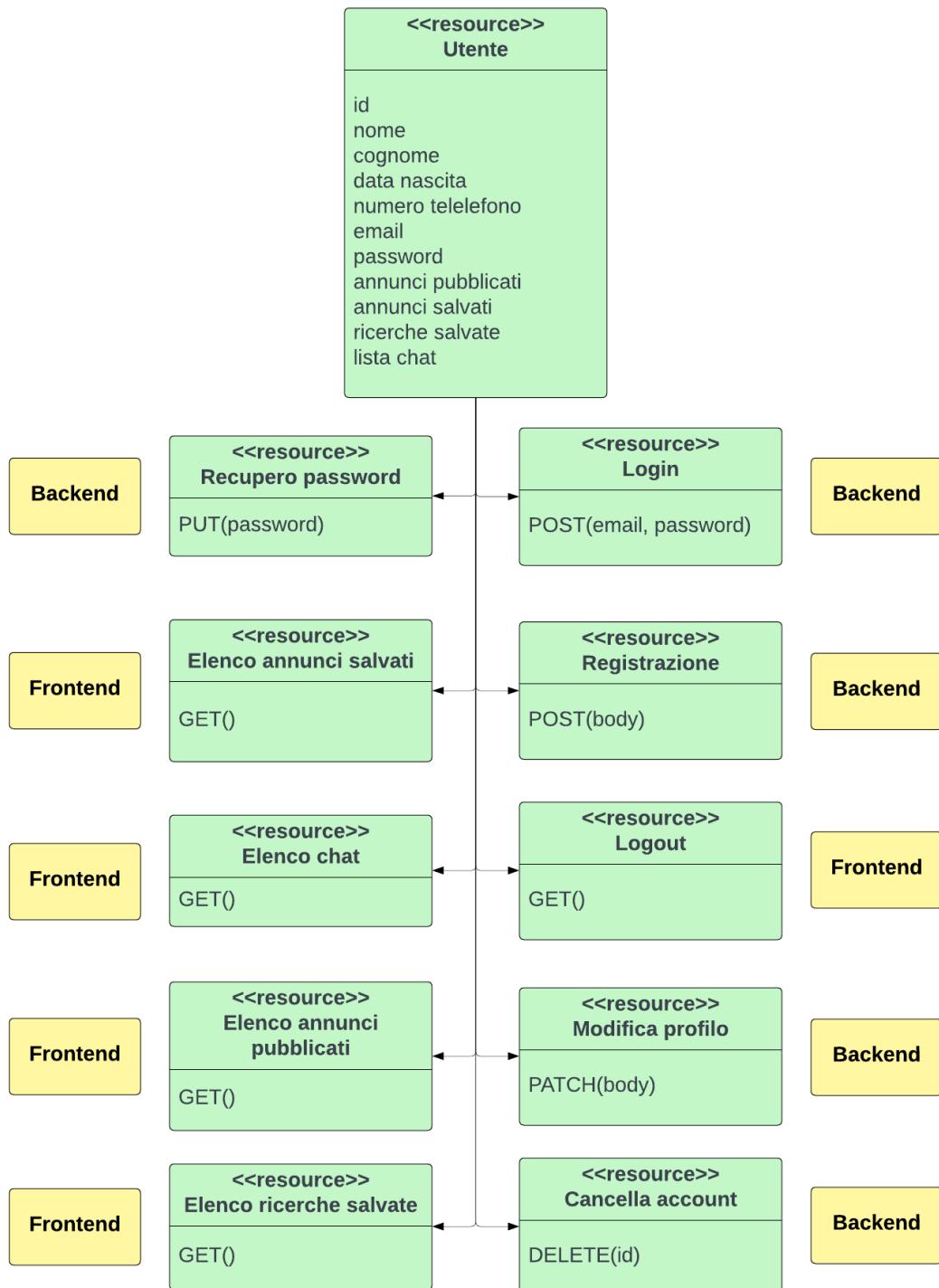
Inoltre vengono separate tra risorse di tipo backend, che quindi operano sui dati svolgendo operazioni interne al sistema, e frontend, nel caso in cui abbiano lo scopo di mostrare qualcosa all'utente.

Analizzando il diagramma delle classi, sono state individuate cinque risorse principali:

1. Utente
2. Annuncio
3. Ricerca
4. Chat
5. Messaggio



Utente



GET APIs:

- **logout**: questo metodo permette di eseguire il logout dell'utente e cancellare le informazioni di accesso nel browser.
- **elenco annunci pubblicati**: questo metodo restituisce la lista di id degli annunci pubblicati dall'utente.
- **elenco annunci salvati**: questo metodo restituisce la lista di id degli annunci salvati dall'utente.
- **elenco ricerche salvate**: questo metodo restituisce la lista di id delle ricerche salvate dall'utente.
- **elenco chat**: questo metodo restituisce la lista di id delle chat dell'utente.

POST APIs:

- **login**: questo metodo permette all'utente di eseguire l'accesso al sistema.
- **registrazione**: questo metodo permette all'utente di creare il proprio account su HouseFinder. Se le informazioni fornite sono valide, allora il sistema accetterà i dati richiesti, e invierà un messaggio di posta elettronica per verificare l'indirizzo email.

PUT APIs:

- **recupero password**: questo metodo permette di eseguire il reset della password dell'utente, sostituendola con una nuova, nel caso abbia smarrito quella attuale.

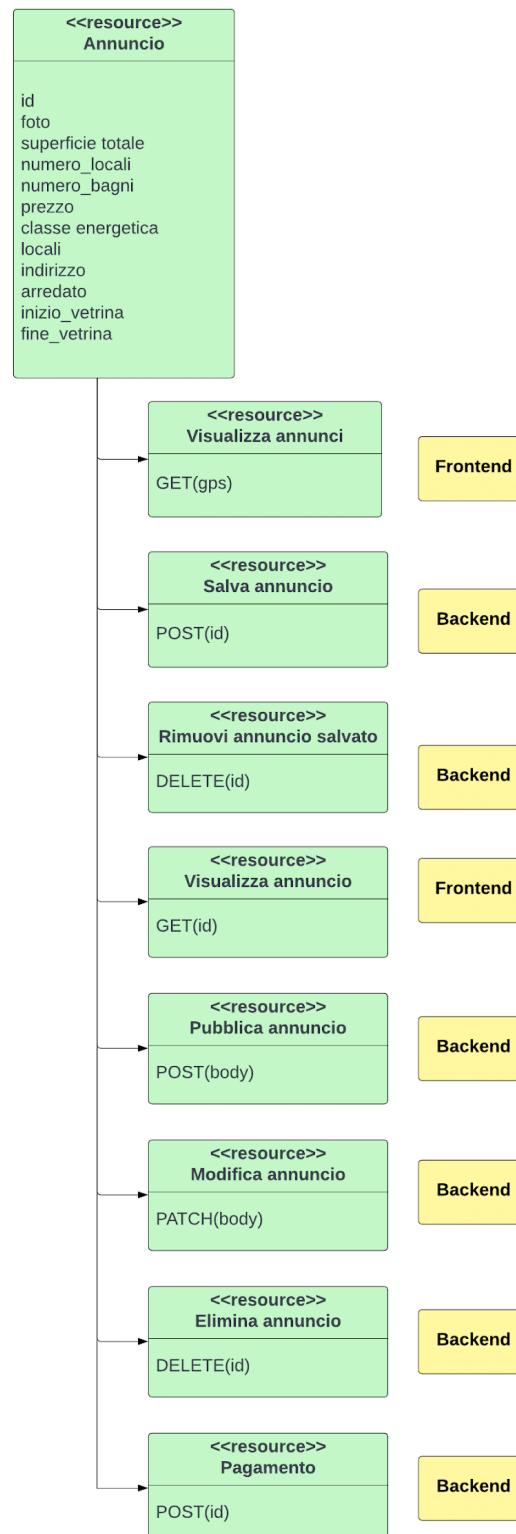
PATCH APIs:

- **modifica profilo**: questo metodo permette di aggiornare i dati dell'utente. Nel caso l'utente modifichi anche l'email o la password, riceverà in seguito una email di notifica.

DELETE APIs:

- **cancella account**: questo metodo permette di cancellare le informazioni dell'utente dal sistema.

Annuncio



GET APIs:

- **visualizza annunci:** questo metodo permette di ottenere la lista di id degli annunci da visualizzare
- **visualizza annuncio:** questo metodo permette di ottenere tutte le informazioni relative ad uno specifico annuncio

POST APIs:

- **salva annuncio:** questo metodo permette di salvare l'id di un annuncio per visualizzarlo in un secondo momento
- **pubblica annuncio:** questo metodo permette ad un utente di pubblicare un annuncio
- **pagamento:** questo metodo permette ad un utente di effettuare un pagamento per la vetrina

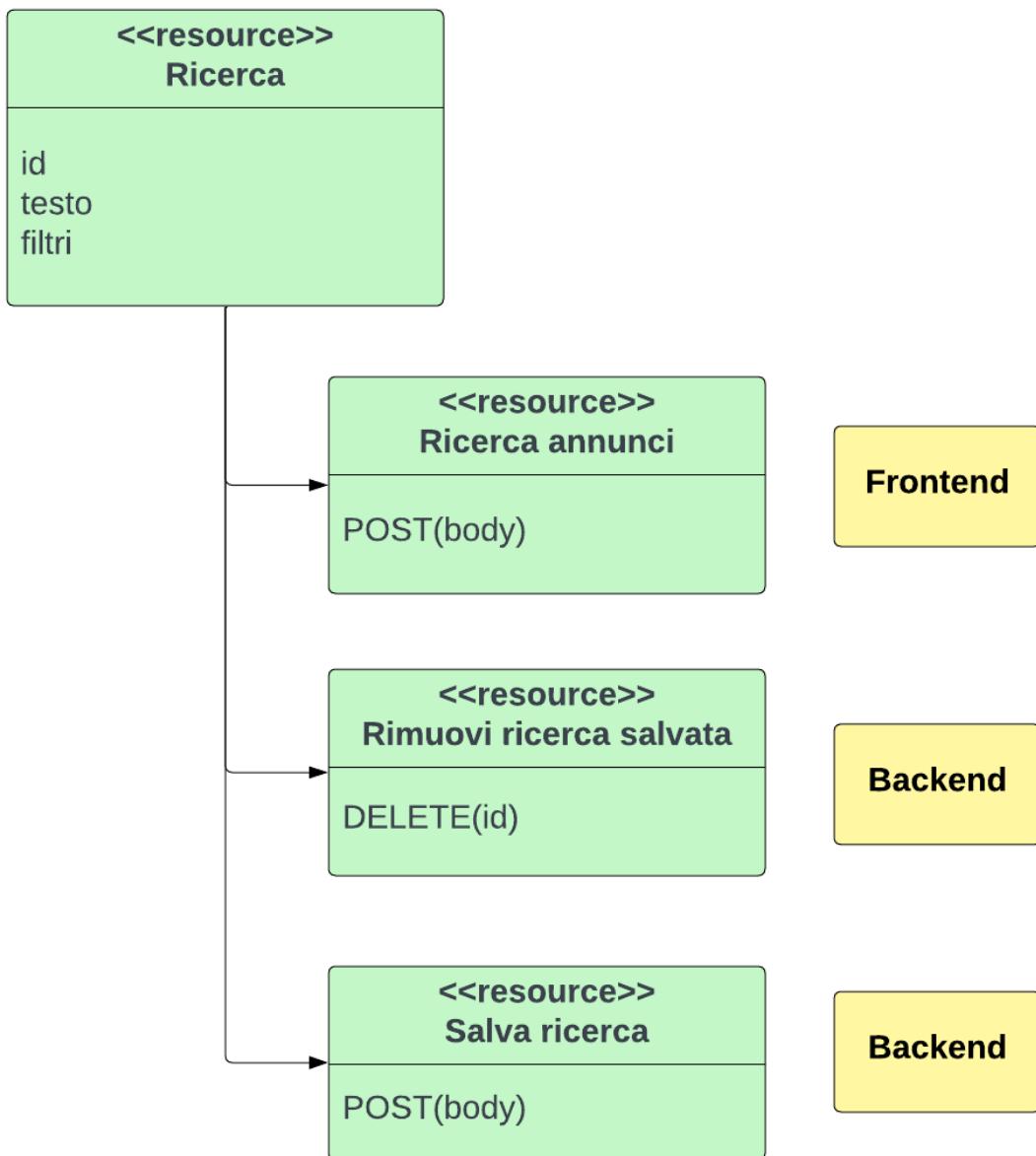
PUT APIs:

- **modifica annuncio:** questo metodo permette di modificare un annuncio

DELETE APIs:

- **elimina annuncio:** questo metodo permette di eliminare definitivamente un annuncio pubblicato
- **rimuovi annuncio salvato:** questo metodo permette di rimuovere un annuncio dagli annunci salvati

Ricerca



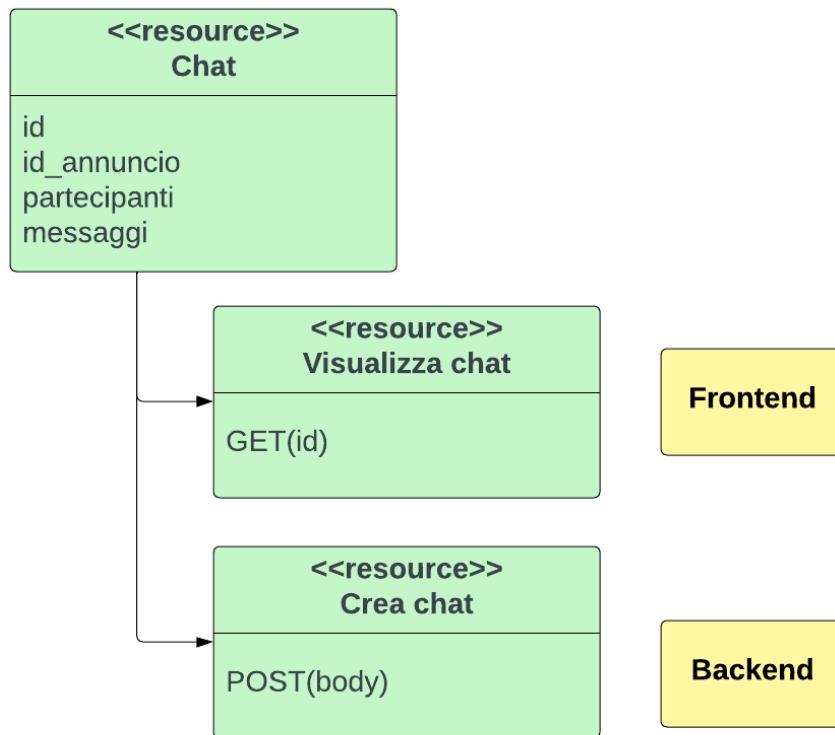
POST APIs:

- **ricerca annunci**: questo metodo permette di eseguire una ricerca, restituendo gli annunci
- **salva ricerca**: permette all'utente di salvare i parametri di ricerca, per poi poter eseguire una ricerca con gli stessi parametri

DELETE APIs:

- **rimuovi ricerca salvata:** questo metodo permette di rimuovere una ricerca dalle ricerche salvate

Chat



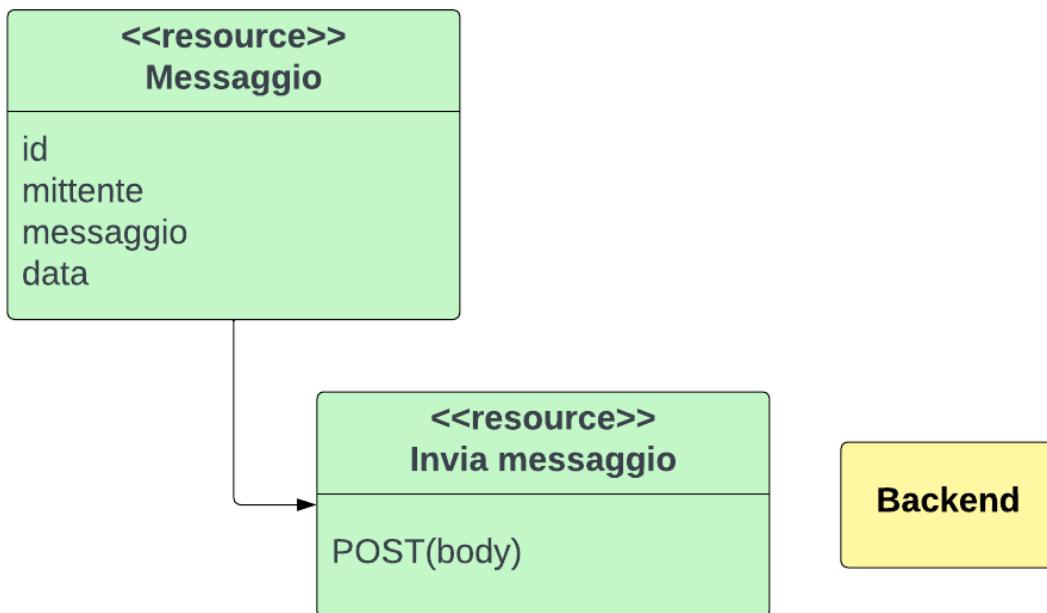
GET APIs:

- **visualizza chat:** questo metodo permette di aprire una specifica conversazione, per vederne i messaggi oppure per inviarne di nuovi.

POST APIs:

- **crea chat:** questo metodo permette di creare una chat tra l'acquirente e il locatore per un annuncio specifico.

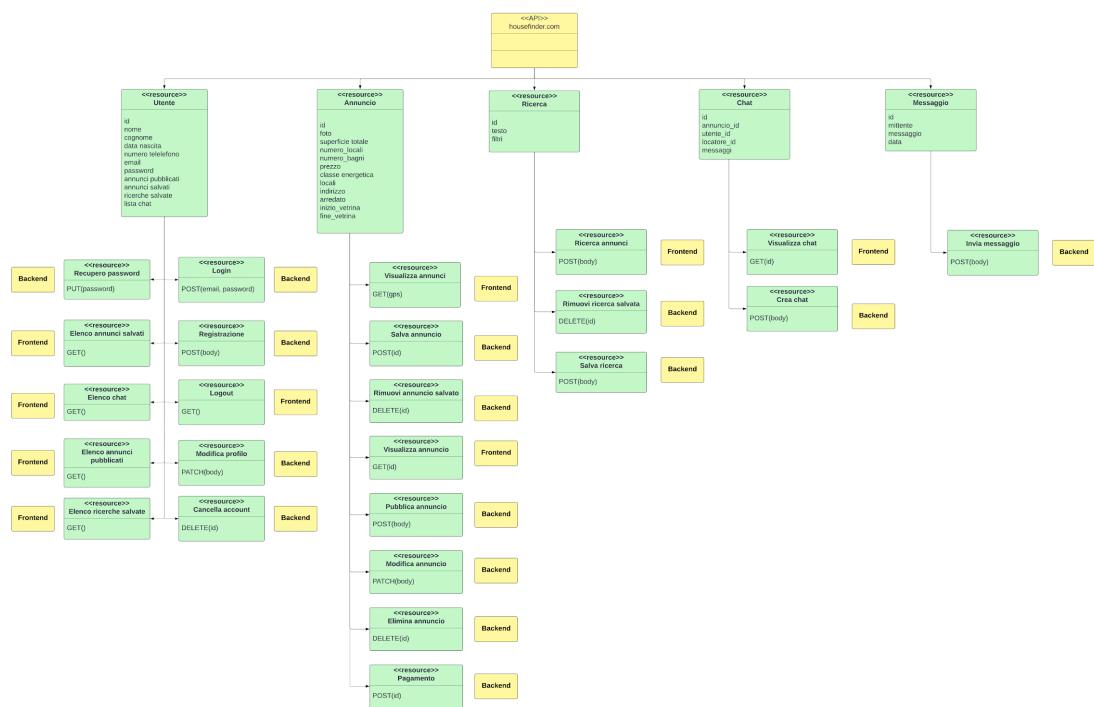
Messaggio



POST APIs:

- **invia messaggio:** permette all'utente di inviare un messaggio. Prende in input, il mittente, il destinatario e il contenuto da inviare.

Nella seguente figura viene riportato il diagramma di estrazione delle risorse completo.



2.4.2 Resource Models

Le risorse individuate nella fase di Resource Extraction possono essere ulteriormente approfondite attraverso l'uso dei Resource Model. Per ogni API, vengono specificati il nome, il metodo e l'URL con cui potervi interagire.

Per poter utilizzare le API che necessitano di autenticazione, deve essere incluso il token relativo all'account negli header della richiesta, nel parametro `x-access-token`.

Utente

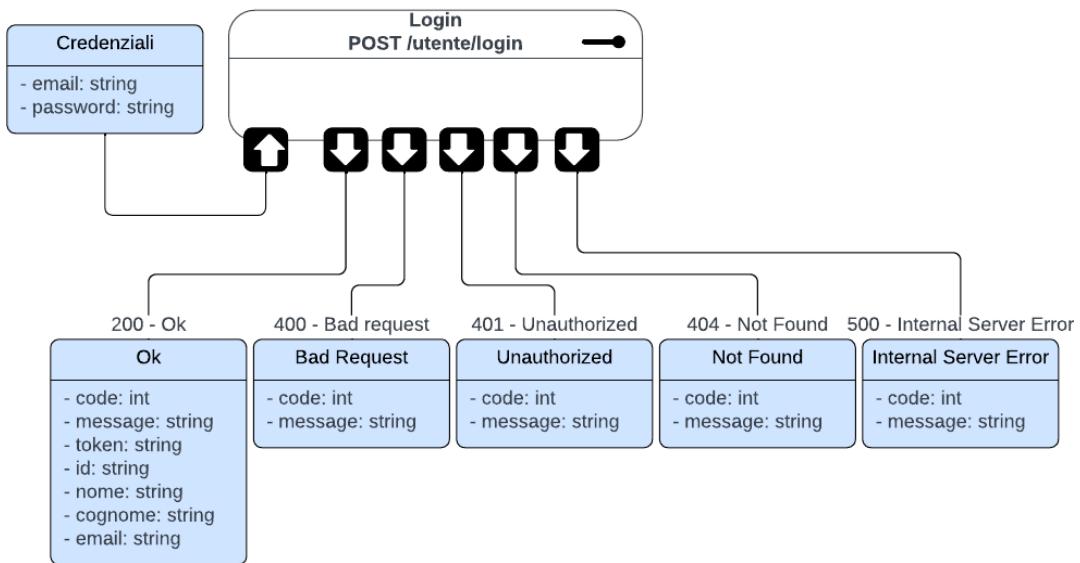
Login

- Metodo: **POST**
- URI: **/utente/login**

Questo endpoint è dedicato all'autenticazione di un utente nel sistema. Quando viene richiamato, specificando l'email e la password nel body della richiesta, viene generato un token JWT che consente l'accesso a tutte le altre risorse richieste. Questo token viene restituito, insieme alle informazioni dell'account che poi verranno gestite per la visualizzazione delle pagine.

Di seguito sono elencati i codici restituiti dall'API:

- 200 - Ok: l'accesso è avvenuto correttamente
- 400 - Bad request: alcuni parametri sono assenti
- 401 - Unauthorized: password errata
- 404 - Not found: l'email dell'utente non è presente nel sistema
- 500 - Internal server error: nel caso in cui si verifica un qualsiasi errore al lato server



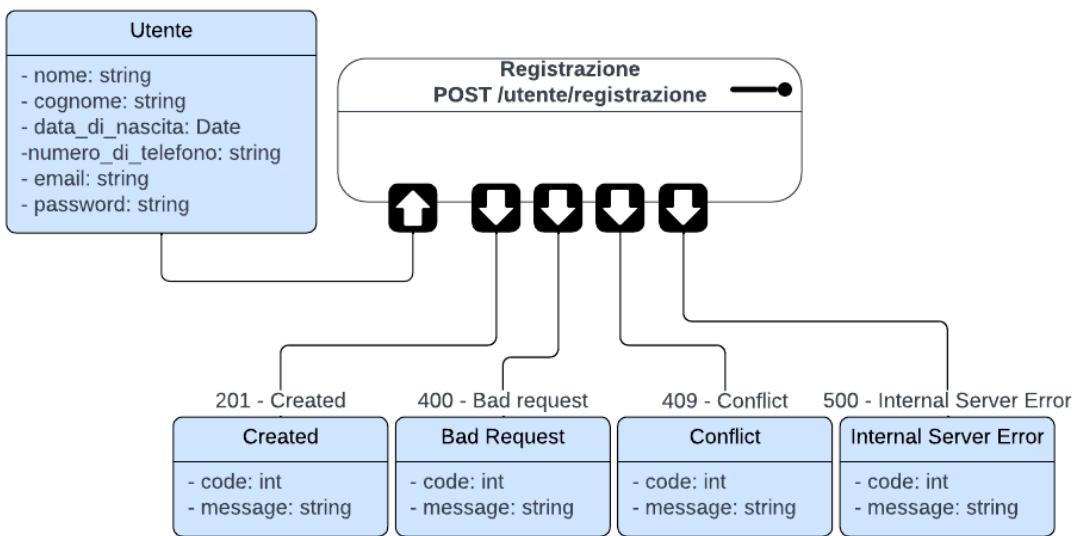
Registrazione

- Metodo: **POST**
- URI: **/utente/registrazione**

Questo endpoint è dedicato alla registrazione di un utente nel sistema. Deve essere chiamato specificando nel body della richiesta i dati sensibili dell'utente, come nome, cognome, data di nascita, numero di telefono, un indirizzo email e una password. Il sistema immagazzinerà questi dati nel database, creando quindi un account.

Di seguito elencati i codici restituiti dall'API:

- **201 - Created:** la registrazione è avvenuta correttamente
- **400 - Bad request:** l'input non è formattato correttamente
- **409 - Conflict:** l'indirizzo email è già presente nel sistema
- **500 - Internal server error:** nel caso in cui si verifichi un qualsiasi errore al lato server



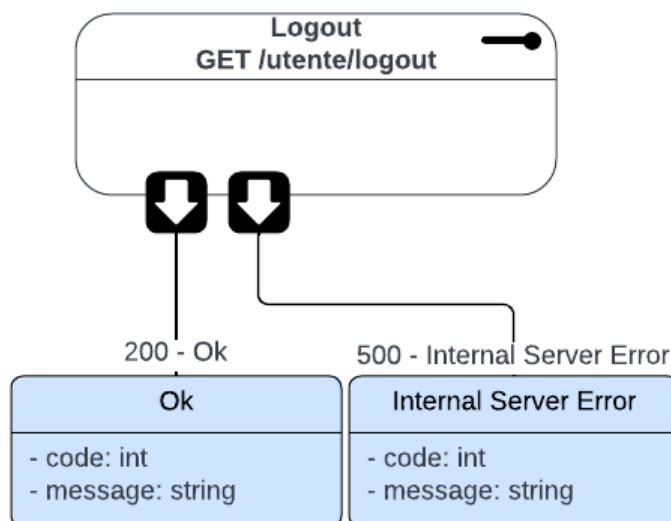
Logout

- Metodo: **GET**
- URI: **/utente/logout**

Questo endpoint è dedicato al logout dell'utente.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: il logout è avvenuto correttamente
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



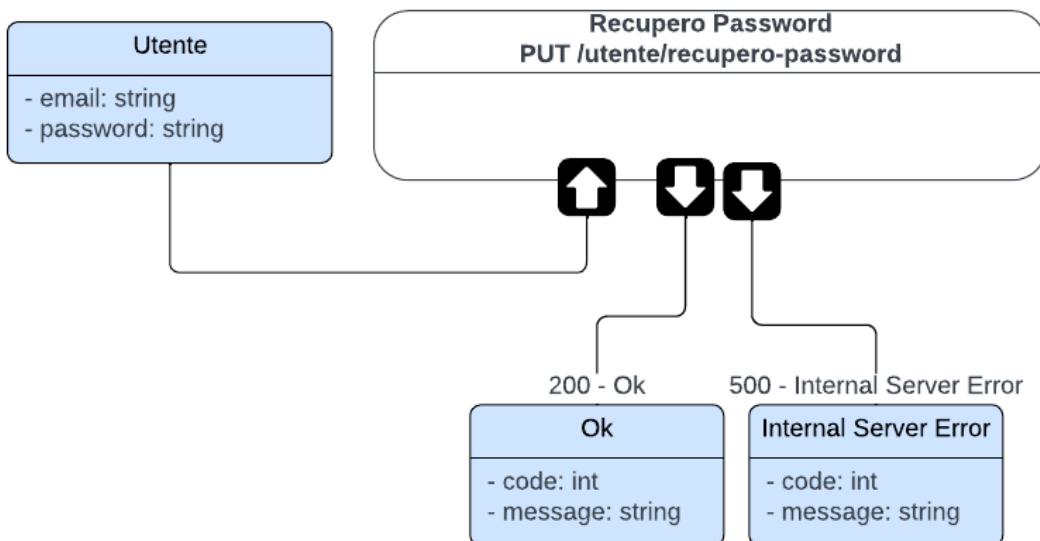
Recupero Password

- Metodo: **POST**
- URI: **/utente/recupero-password**

Questo endpoint è dedicato al recupero della password dell'utente, nel caso se la sia dimenticata. Prende come input l'email dell'account, a cui verrà inviata la procedura di recupero della password, e la nuova password da assegnare.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: il recupero è avvenuto con successo
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



Modifica profilo

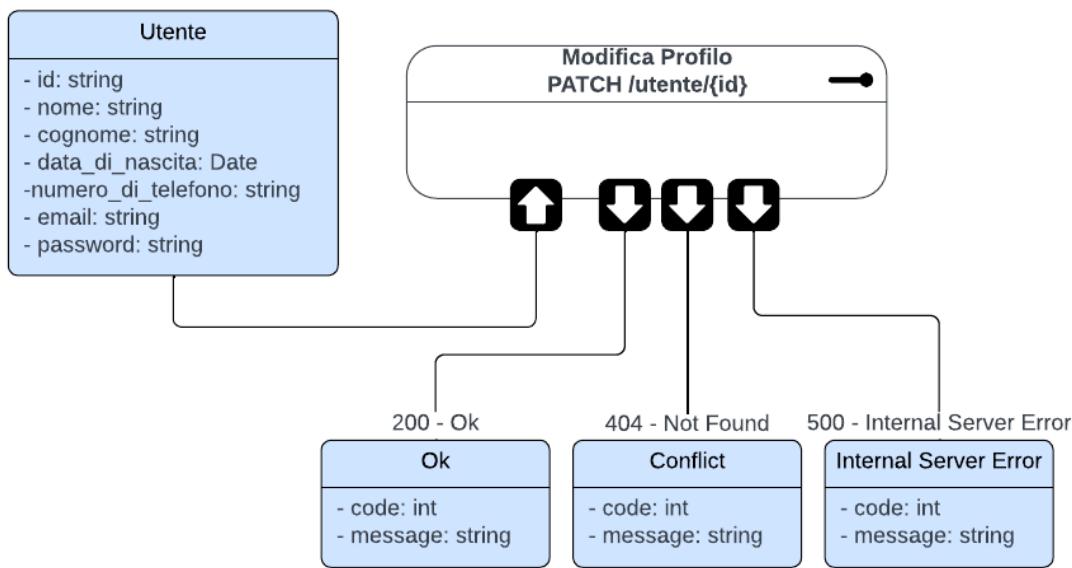
- Metodo: **PATCH**
- URI: **/annuncio/modifica_profilo**

Questo endpoint è dedicato alla modifica delle informazioni dell'account. Prende come input i dati sensibili da modificare, e ignora i campi che hanno un valore nullo.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: le modifiche sono state applicate correttamente
- 404 - Not found: utente non trovato

- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



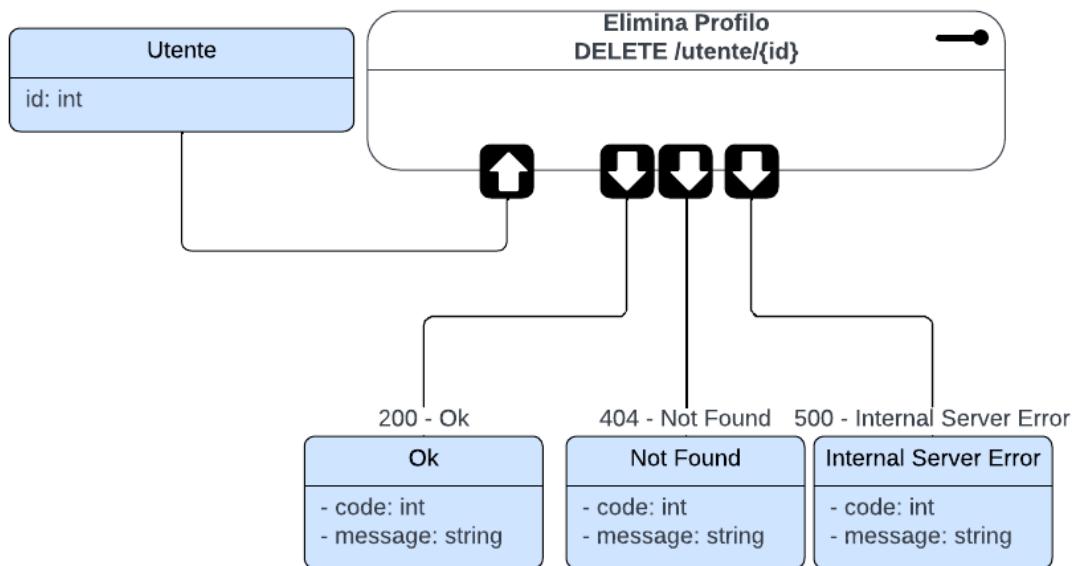
Elimina profilo

- Metodo: **DELETE**
- URI: **/utente/{id}**

Questo endpoint è dedicato alla eliminazione dell'account del sistema.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: il profilo è stato eliminato con successo
- 404 - Not found: nel caso in cui la pagina non venga trovata
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



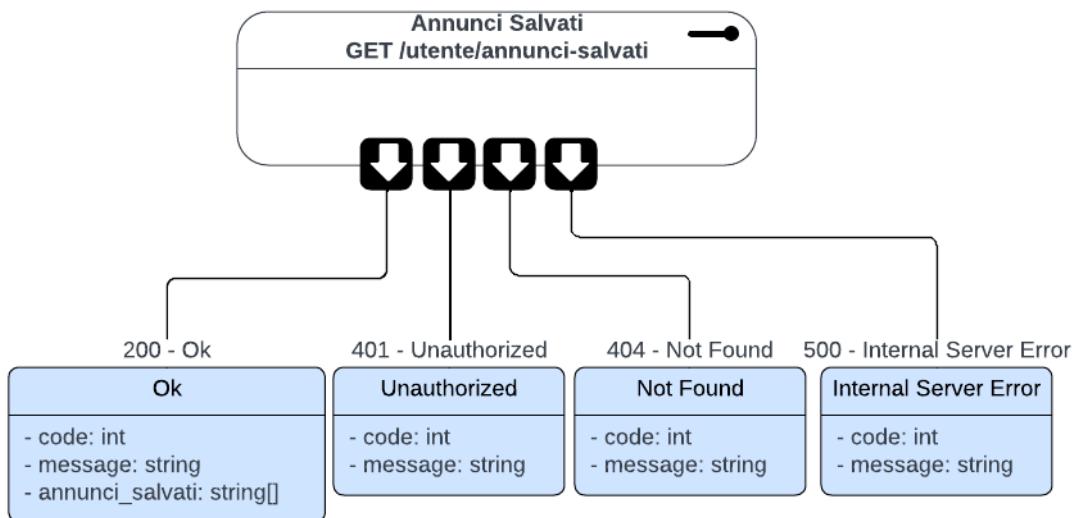
Annunci salvati

- Metodo: **GET**
- URI: **/utente/annunci-salvati**

Questo endpoint è dedicato alla visualizzazione della lista degli annunci salvati da un utente. Restituisce la lista degli id degli annunci salvati.

Di seguito elencati i codici restituiti dall'API:

- **200 - Ok**: la lista di annunci salvati viene restituita con successo
- **401 - Unauthorized**: accesso non autorizzato
- **404 - Not found**: nel caso in cui la pagina non venga trovata
- **500 - Internal server error**: nel caso in cui si verifichi un qualsiasi errore al lato server



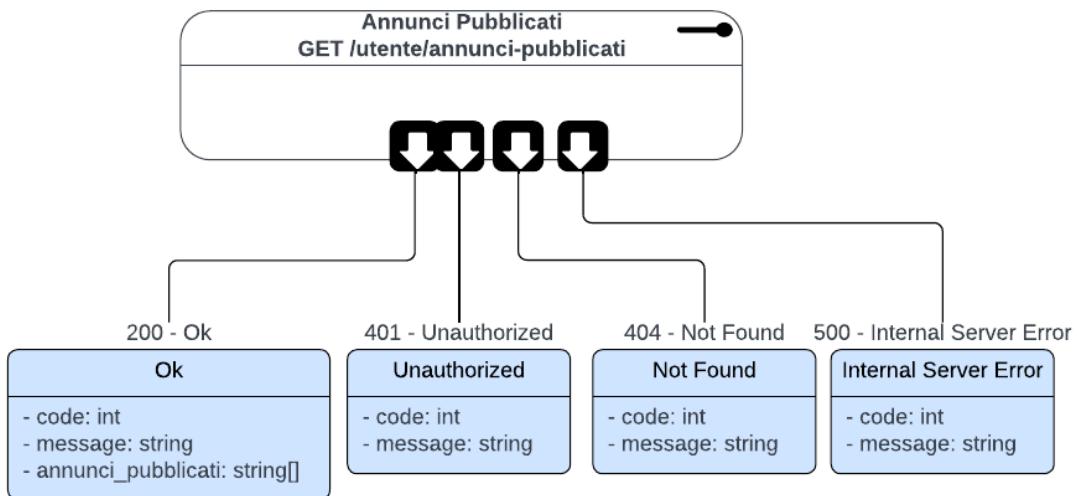
Annunci pubblicati

- Metodo: **GET**
- URI: **/utente/annunci-pubblicati**

Questo endpoint è dedicato alla visualizzazione della lista degli annunci pubblicati da un utente dall'utente. Restituisce la lista degli id degli annunci pubblicati.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: la lista degli annunci pubblicati viene restituita con successo
- 401 - Unauthorized: accesso non autorizzato
- 404 - Not found: nel caso in cui la pagina non venga trovata
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



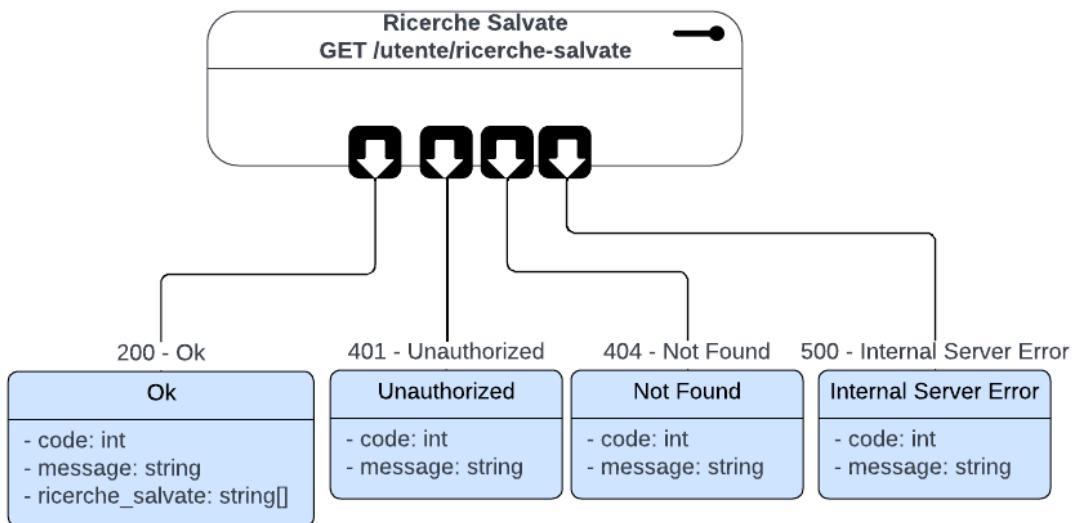
Ricerche Salvate

- Metodo: **GET**
- URI: **/utente/ricerche-salvate**

Questo endpoint è dedicato alla visualizzazione della lista delle ricerche salvate. Ritorna la lista degli id delle ricerche salvate dall'utente.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: la lista delle ricerche salvate viene restituita con successo
- 401 - Unauthorized: accesso non autorizzato
- 404 - Not found: nel caso in cui la pagina non venga trovata
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



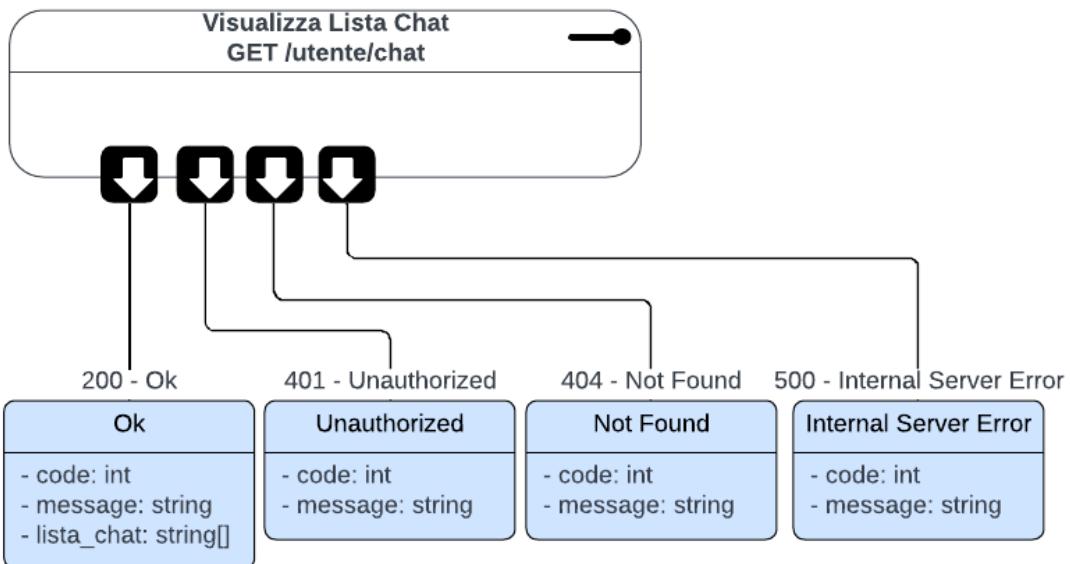
Visualizza Lista Chat

- Metodo: **GET**
- URI: **/utente/chat**

Questo endpoint è dedicato alla visualizzazione della lista delle chat dell'utente. Restituisce la lista degli id delle chat.

Di seguito elencati i codici restituiti dall'API:

- **200 - Ok**: la lista delle chat viene restituita con successo
- **401 - Unauthorized**: accesso non autorizzato
- **404 - Not found**: nel caso in cui la pagina non venga trovata
- **500 - Internal server error**: nel caso in cui si verifichi un qualsiasi errore al lato server



Annuncio

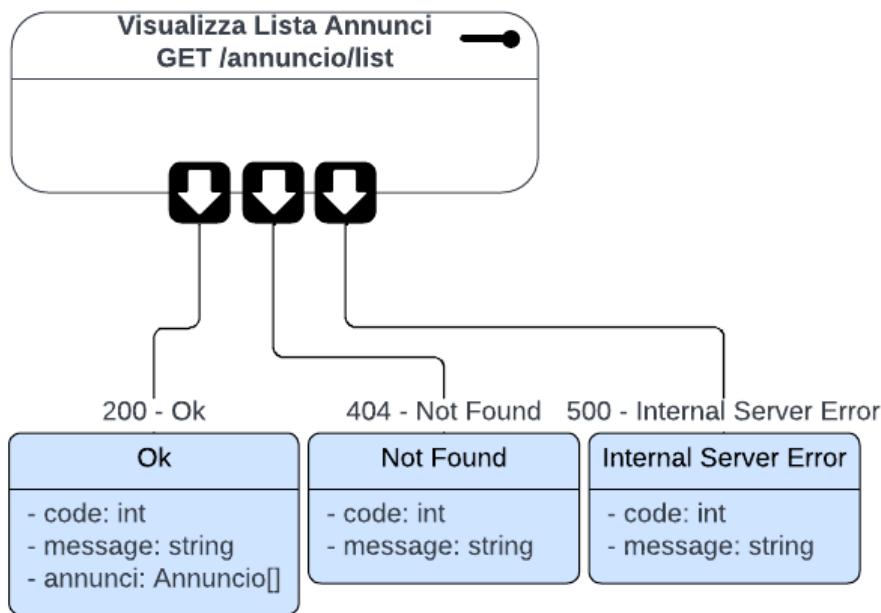
Visualizza lista annunci

- Metodo: **GET**
- URI: **/annuncio/list**

Questo endpoint è dedicato alla visualizzazione degli annunci sulla Homepage dell'utente. Ritorna la lista degli annunci presenti nel sistema.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: la richiesta è stata eseguita con successo
- 404 - Not found: nel caso in cui la pagina non venga trovata
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



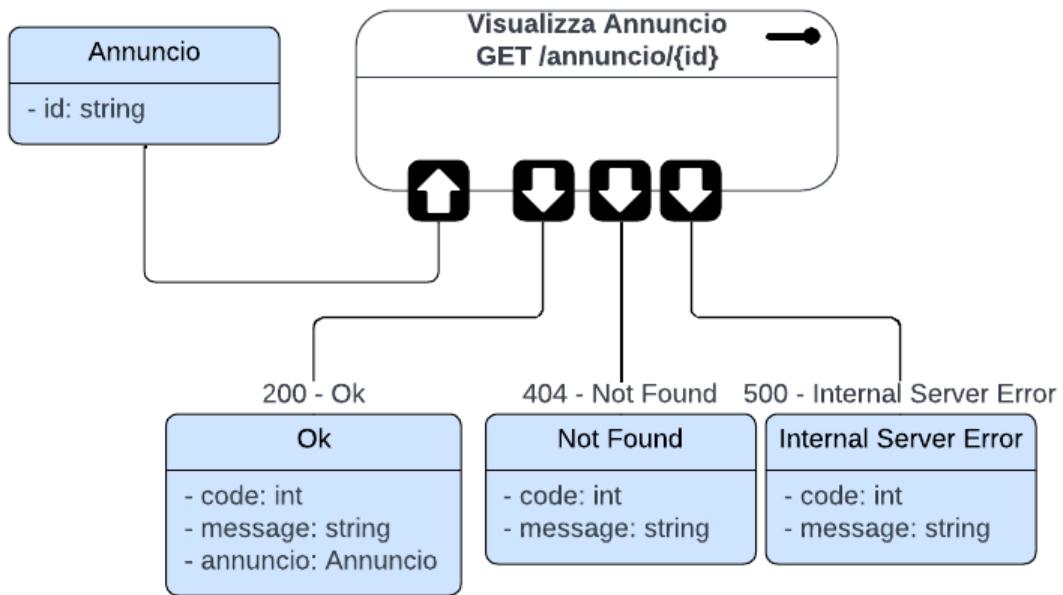
Visualizza annuncio

- Metodo: **GET**
- URI: **/annuncio/{id}**

Questo endpoint è dedicato alla visualizzazione di uno specifico annuncio, dato il suo id. Restituisce tutte le informazioni relative all'annuncio.

Di seguito elencati i codici restituiti dall'API:

- **200 - Ok**: l'annuncio è stato restituito con successo
- **404 - Not found**: nel caso in cui la pagina non venga trovata
- **500 - Internal server error**: nel caso in cui si verifichi un qualsiasi errore al lato server



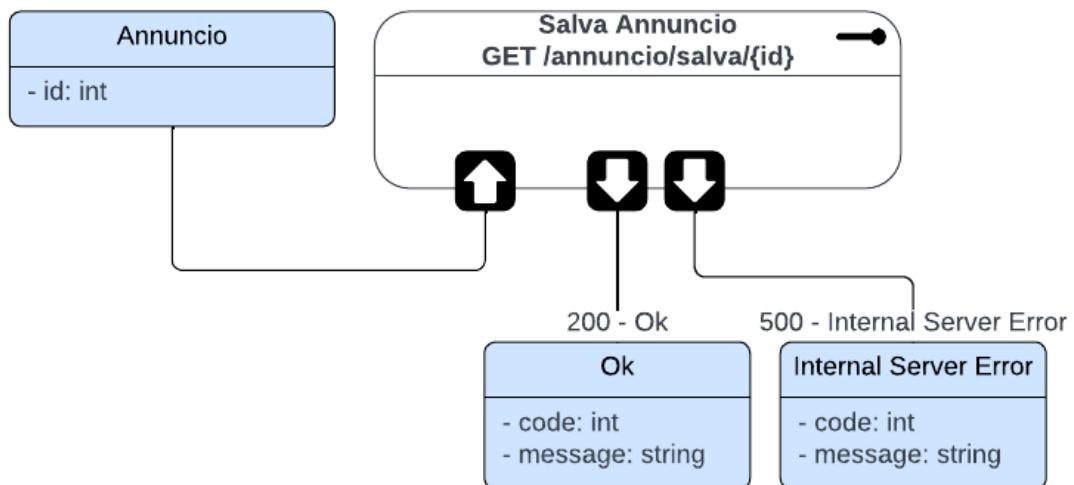
Salva annuncio

- Metodo: **GET**
- URI: **/annuncio/salva/{id}**

Questo endpoint è dedicato al salvataggio di un annuncio nella lista degli annunci salvati.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: l'annuncio è stato salvato con successo
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



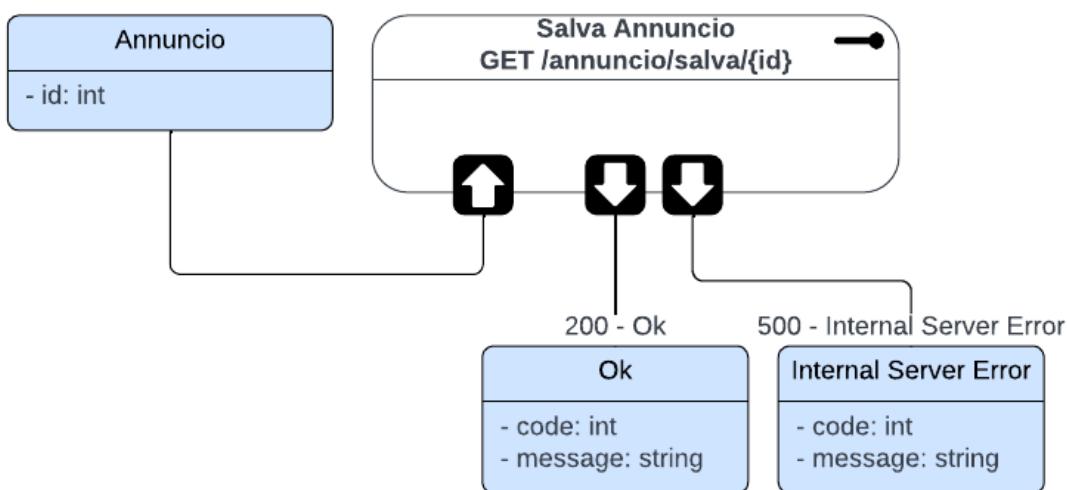
Rimuovi Annuncio Salvato

- Metodo: **DELETE**
- URI: **/annuncio/{id}**

Questo endpoint è dedicato alla rimozione di un annuncio dalla lista degli annunci salvati.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: l'annuncio è stato rimosso con successo
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



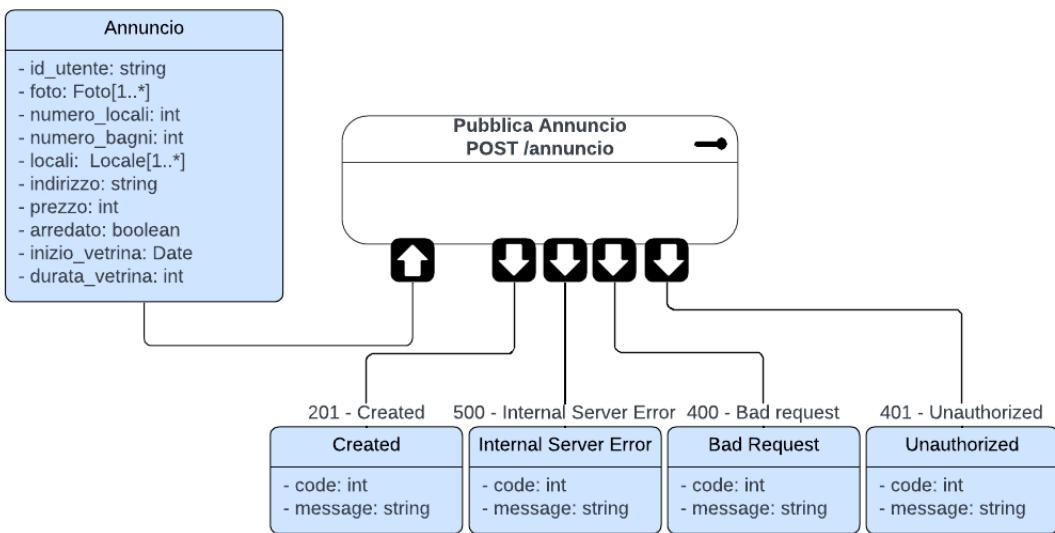
Pubblica annuncio

- Metodo: **POST**
- URI: **/annuncio/pubblica**

Questo endpoint è dedicato alla pubblicazione di un nuovo annuncio da parte di un nuovo utente. Devono essere specificate nel body le informazioni riguardanti il nuovo annuncio. Alcuni parametri sono obbligatori e non possono essere omessi.

Di seguito elencati i codici restituiti dall'API:

- 201 - Created: l'annuncio viene pubblicato con successo
- 400 - Bad Request: alcuni parametri sono assenti
- 401 - Unauthorized: utente non autorizzato
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



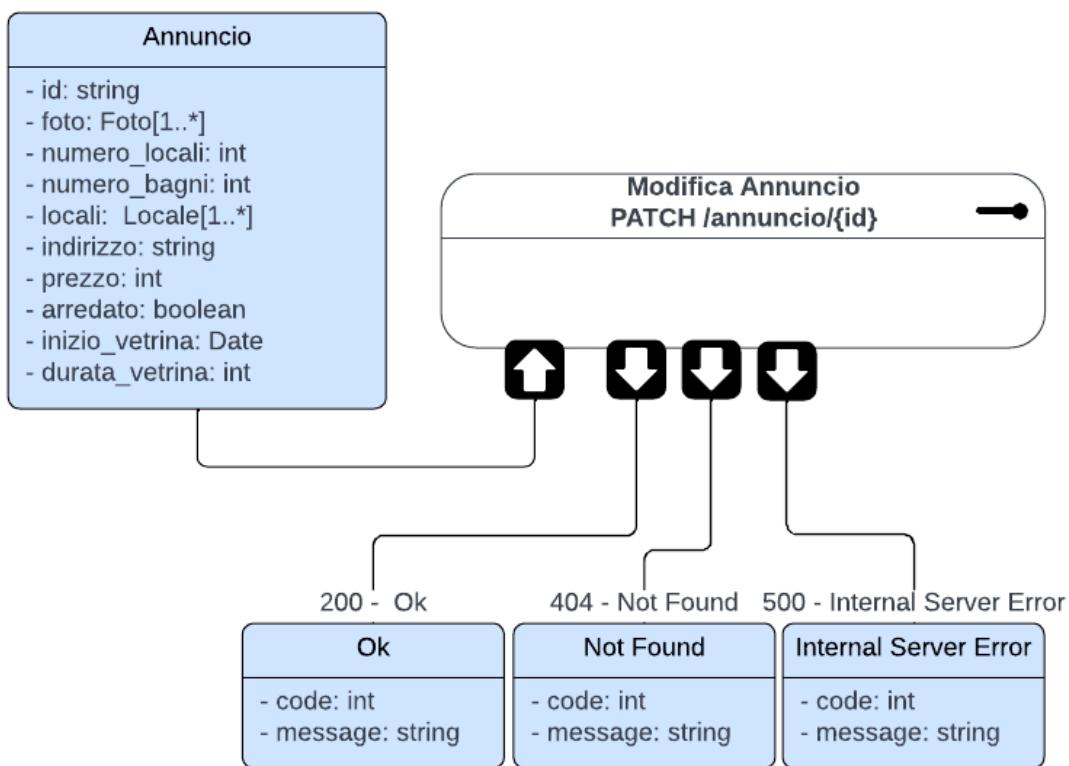
Modifica annuncio

- Metodo: **PATCH**
- URI **/annuncio/{id}**

Questo endpoint è dedicato alla modifica di un annuncio pubblicato in precedenza. Devono essere specificate nel body le informazioni aggiornate dell'annuncio. I campi che sono nulli, non verranno cambiati.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: l'annuncio viene modificato con successo
- 404 - Not found: nel caso in cui la pagina non venga trovata
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



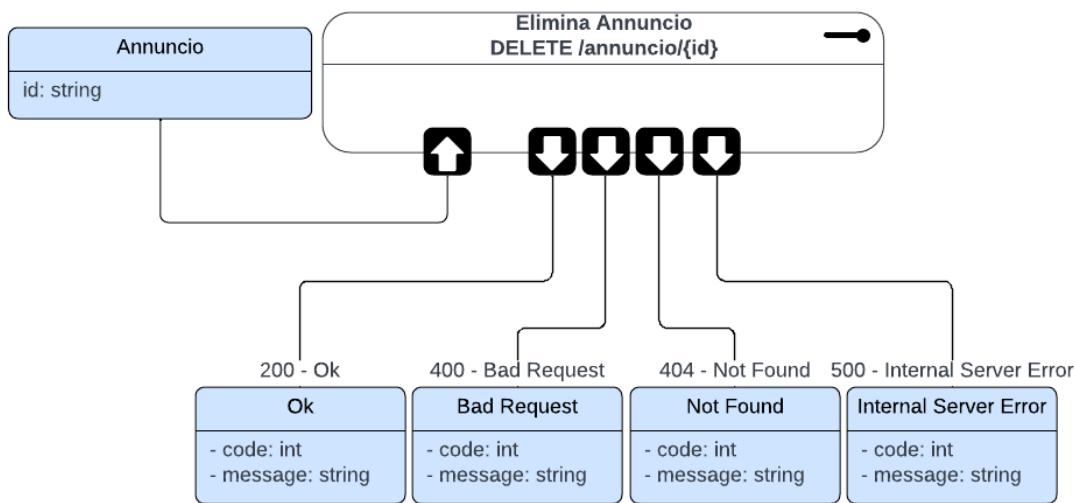
Elimina annuncio

- Metodo: **DELETE**
- URI: **/annuncio/{id}**

Questo endpoint è dedicato alla eliminazione di un annuncio pubblicato dall'utente.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: l'annuncio viene eliminato con successo
- 400 - Bad request: l'input non è formattato correttamente
- 404 - Not found: nel caso in cui la pagina non venga trovata
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



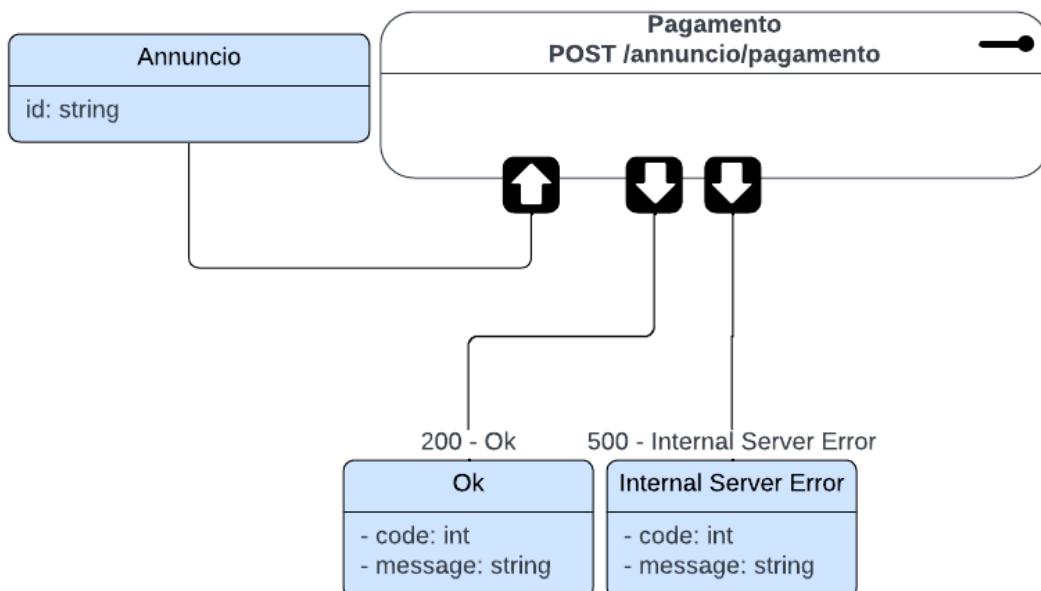
Pagamento

- Metodo: **POST**
- URI: **/annuncio/pagamento**

Questo endpoint è dedicato al pagamento della vetrina da parte di un utente. Prende come input l'id dell'annuncio da mettere in vetrina.

Di seguito elencati i codici restituiti dall'API:

- [200 - Ok](#): pagamento effettuato
- [500 - Internal server error](#): nel caso in cui si verifichi un qualsiasi errore al lato server



Ricerca

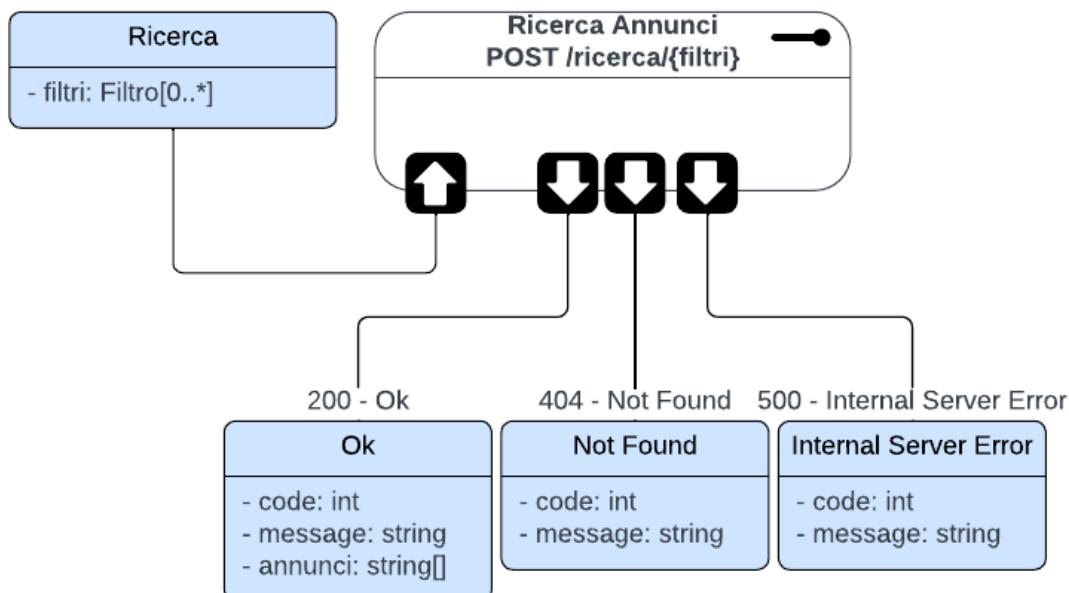
Ricerca annunci

- Metodo: **GET**
- URI: **/ricerca/{filtr}**

Questo endpoint è dedicato alla ricerca di annunci presenti nel sistema. Prende come input la lista dei filtri che gli annunci devono soddisfare. Ritorna la lista degli id degli annunci che soddisfano le condizioni dei filtri.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: lista degli annunci restituita con successo
- 404 - Not found: nel caso in cui la pagina non venga trovata
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



Salva Ricerca

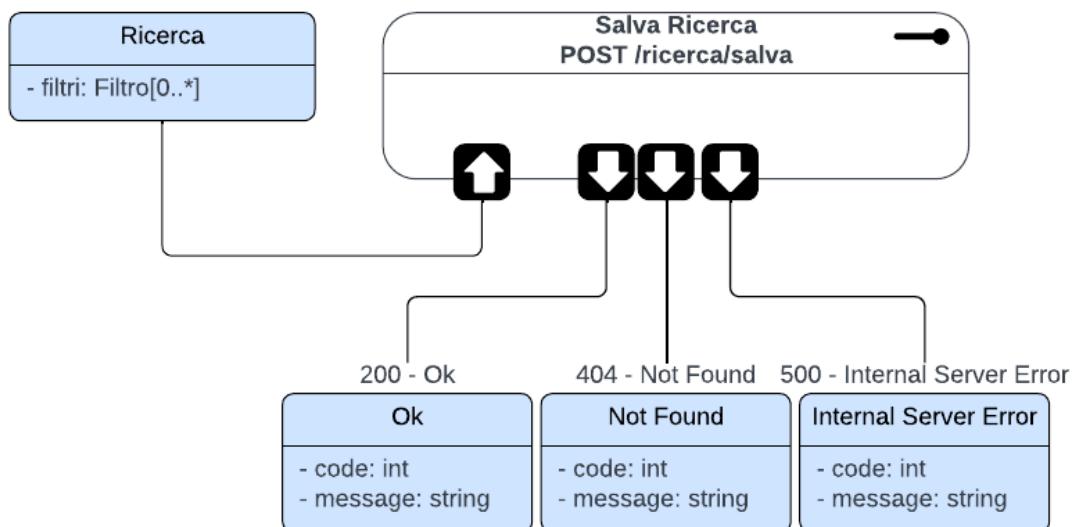
- Metodo: **POST**
- URI: **/ricerca/salva**

Questo endpoint è dedicato al salvataggio dei parametri di una determinata ricerca. Prende come input i filtri della ricerca e crea un oggetto "ricerca" che verrà salvato nel database.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: ricerca salvata con successo

- 404 - Not found: nel caso in cui la pagina non venga trovata
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



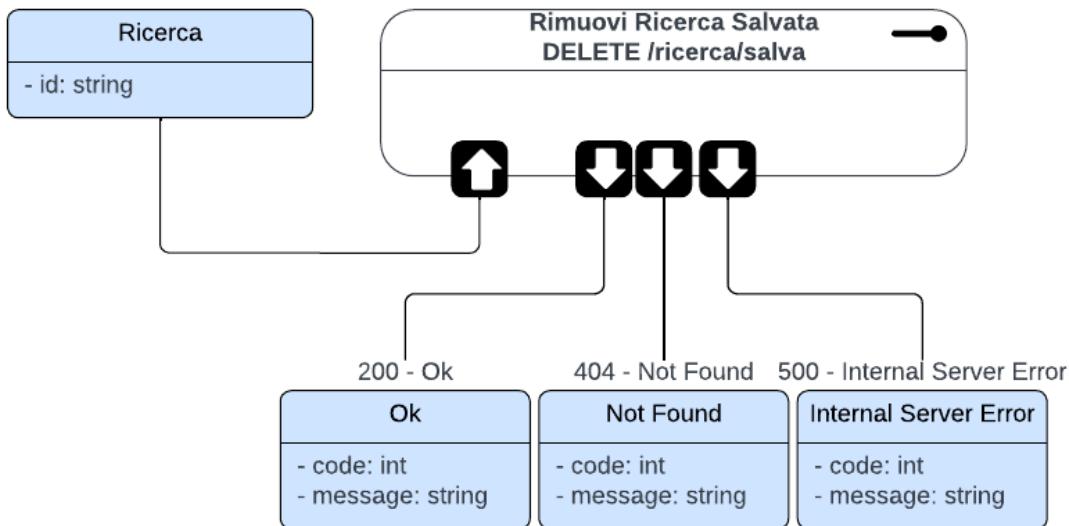
Rimuovi Ricerca Salvata

- Metodo: **DELETE**
- URI: **/ricerca/salva/{id}**

Questo endpoint è dedicato alla rimozione di una ricerca salvata da un utente. Prende come input l'id della ricerca, e la rimuove dalla lista.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: ricerca rimossa con successo
- 404 - Not found: nel caso in cui la pagina non venga trovata
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



Chat

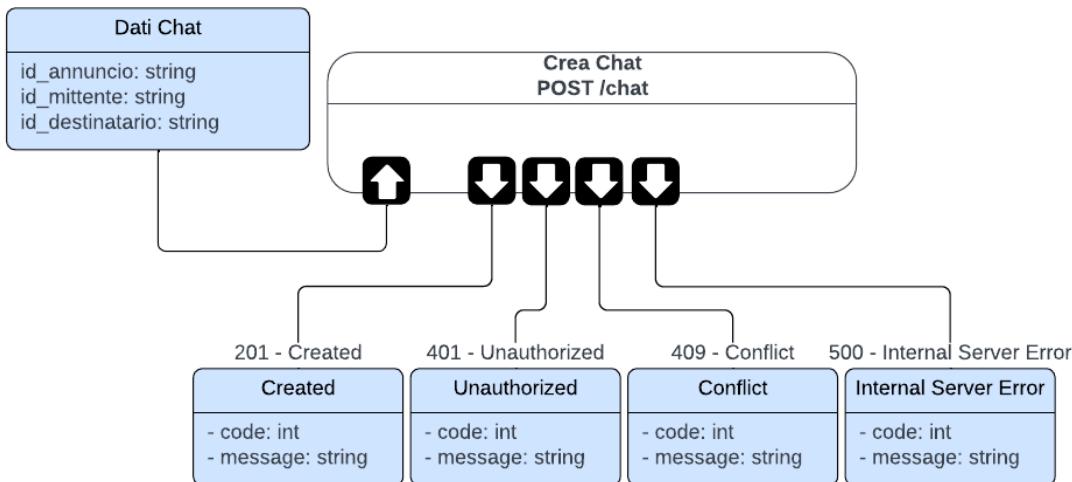
Crea Chat

- Metodo: **POST**
- URI: **/chat**

Questo endpoint è dedicato alla creazione di una chat nel sistema. Prende come input l'id dell'annuncio a cui è riferita la chat, l'id del mittente e quello del destinatario.

Di seguito elencati i codici restituiti dall'API:

- [201 - Created](#): chat creata con successo
- [401 - Unauthorized](#): utente non autorizzato
- [409 - Conflict](#): chat già esistente
- [500 - Internal server error](#): nel caso in cui si verifichi un qualsiasi errore al lato server



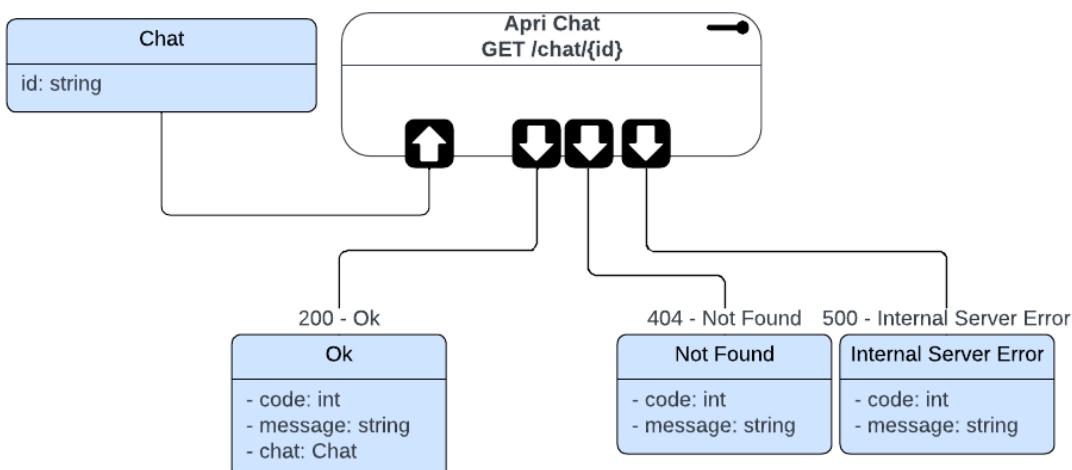
Apri Chat

- Metodo: **GET**
- URI: **/chat/{id}**

Questo endpoint è dedicato alla visualizzazione di una determinata chat. Prende come input l'id della chat. Restituisce i dati relativi alla chat per poi essere visualizzata.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: chat ottenuta correttamente
- 404 - Not Found: chat non trovata
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



Messaggio

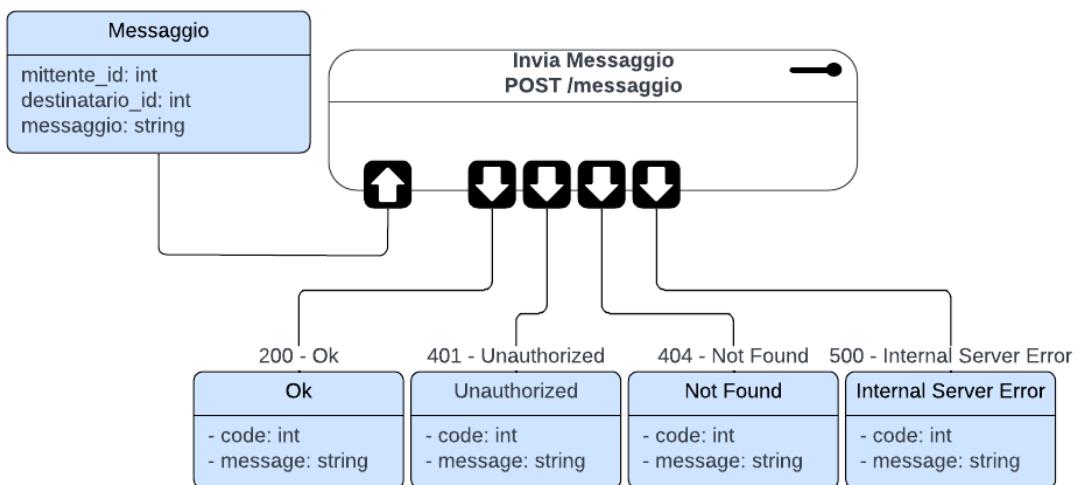
Invia Messaggio

- Metodo: **POST**
- URI: **/messaggio**

Questo endpoint è dedicato all'invio di un messaggio. Prende come input il messaggio e l'id della chat in cui deve essere spedito.

Di seguito elencati i codici restituiti dall'API:

- 200 - Ok: messaggio inviato con successo
- 401 - Unauthorized: utente non autorizzato
- 404 - Not Found: chat inesistente
- 500 - Internal server error: nel caso in cui si verifichi un qualsiasi errore al lato server



2.5 Sviluppo API

In questo capitolo viene approfondito lo sviluppo e il funzionamento delle API precedentemente illustrate. Tuttavia, nonostante siano state implementate tutte le API menzionate, abbiamo deciso di riportare quelle ritenute più significative per il funzionamento del sistema al fine di evitare ripetizioni e codici eccessivamente lunghi.

Utente

2.5.1 Token checker middleware

```
const jwt = require('jsonwebtoken');

const tokenChecker = async (req, res, next) => {
    // header or url parameters or post parameters
    var token = req.body.token || req.query.token || req.headers['x-access-token'];
    if (!token)
        return res.status(400).json({
            code: 400,
            message: 'No token provided.'
        });
    // decode token, verifies secret and checks expiration
    jwt.verify(token, process.env.SUPER_SECRET, function (err, decoded) {
        if (err)
            return res.status(401).json({
                code: 401,
                message: 'Token not valid'
            });
        else {
            // if everything is good, save in req object for use in other routes
            req.loggedUser = decoded.data;
            next();
        }
    });
};

module.exports = tokenChecker
```

Per permettere l'autenticazione da parte degli utenti, è stato necessario implementare un token checker middleware. Un middleware è un componente che viene eseguito durante il processo di routing, ovvero durante il processo di inoltro delle richieste HTTP a specifiche funzioni o endpoint. Nel caso del token checker middleware, questo componente è stato utilizzato per verificare che ogni richiesta HTTP fosse corredata da un token di autenticazione valido.

Come si vede in figura, per implementare il controllo del token, è stata utilizzata la libreria jsonwebtoken. Questa libreria offre funzionalità per

codificare e decodificare i token basati sullo standard JWT (JSON Web Token).

In particolare, è stata utilizzata la funzione "verify" di jsonwebtoken per verificare la validità di un token fornito in una richiesta HTTP. Se il token è valido, il middleware consente il passaggio della richiesta al relativo endpoint. Altrimenti, il middleware blocca la richiesta e restituisce un errore di autenticazione.

2.5.2 Login

```
router.post('/utente/login', utenteController.login);
```

```
const login = async (req, res) => {
    //salvo nella variabile email la email che mi ha passato l'utente nella
    //richiesta POST
    //salvo nella variabile password la password che mi ha passato l'utente
    //nella richiesta POST
    const { email, password } = req.body;

    //se la email o la password sono vuote
    if (!email || !password)
        return res.status(400).json({
            code: 400,
            message: 'Alcuni parametri sono assenti.'
        });

    let user = await Utente.findOne({ email: email }).exec().catch((err) => {
        return res.status(500).json({
            code: 500,
            message: 'Internal server error.'
        });
    });
    //se non ho trovato l'utente nel database
    if (!user)
        return res.status(404).json({
            code: 404,
            message: 'Email non presente nel sistema.'
        });

    //se la password hashata dell'utente è diversa da hash(password inserita)
    if (user.password != hash(password))
        return res.status(401).json({
            code: 401,
            message: 'Password errata.'
        });

    var payload = { id: user._id }
    var token = getToken(payload);

    return res.status(200).json({
        code: 200,
        message: 'L\'accesso è avvenuto correttamente.',
        token: token,
        id: user.id,
        nome: user.nome,
        cognome: user.cognome,
        email: user.email
    });
}
```

L'API del login è un endpoint del backend che gestisce le richieste di autenticazione degli utenti. Quando un utente tenta di effettuare il login, invia una richiesta HTTP al backend contenente il proprio username e password.

All'inizio del processo di autenticazione, l'API effettua dei controlli preliminari per verificare che i campi richiesti non siano vuoti. Successivamente, l'API esegue i controlli di esistenza nel database per verificare che la combinazione di email e password forniti sia valida.

Se i controlli hanno esito positivo, viene generato un token di autenticazione e restituito al client insieme alla risposta della richiesta HTTP. Il token di autenticazione può quindi essere utilizzato dal client per autenticare le successive richieste HTTP inviate al backend. Se i controlli di esistenza nel database hanno esito negativo, l'API del login restituisce un errore di autenticazione al client.

2.5.3 Registrazione

```
router.post('/utente/registrazione', utenteController.registrazione);
```

```
const registrazione = async (req, res) => {
    //questa dicitura permette di salvare ogni campo nella rispettiva
    variabile
    const {
        nome,
        cognome,
        data_nascita,
        numero_tel,
        email,
        password
    } = req.body;

    if (!nome || !cognome || !data_nascita || !numero_tel || !email ||
    !password)
        return res.status(400).json({
            code: 400,
            message: 'L\'input non è formattato correttamente.'
        });

    let user = await Utente.findOne({ email: email }).exec().catch((err) => {
        return res.status(500).json({
            code: 500,
            message: 'Internal server error.'
        });
    });
}
```

```
//se l'utente è già registrato
if (user)
    return res.status(409).json({
        code: 409,
        message: 'Indirizzo email già presente nel sistema.'
    });

// creo un nuovo utente
const nuovoUtente = new Utente({
    nome: nome,
    cognome: cognome,
    data_nascita: data_nascita,
    numero_tel: numero_tel,
    email: email,
    password: hash(password),
    annunci_salvati: [],
    annunci_pubblicati: [],
    ricerche_salvate: []
});

nuovoUtente.save((err) => {
    if (err)
        return res.status(500).json({
            code: 500,
            message: 'Internal server error.'
        });

    return res.status(201).json({
        code: 201,
        message: 'La registrazione è avvenuta con successo.'
    });
});
```

L'API della registrazione è un endpoint del backend che gestisce le richieste di registrazione da parte degli utenti.

All'inizio del processo di registrazione, l'API effettua dei controlli preliminari per verificare che i campi richiesti non siano vuoti e che non esista già un utente con l'email fornita. Se i controlli hanno esito positivo, viene creato l'utente nel database utilizzando i dati forniti e viene restituita una risposta di successo al client. Se i controlli hanno esito negativo, ovvero se l'utente esiste già nel database, viene restituito un errore al client.

È importante notare che la password non viene memorizzata in chiaro, ma ne viene calcolato l'hash tramite algoritmo sha256.

2.5.4 Annunci pubblicati

```
router.get('/utente/annunci-pubblicati', tokenChecker,
  utenteController.get_annunci_pubblicati);
```

```
const get_annunci_pubblicati = async (req, res) => {
  const userId = req.loggedUser.id;
  if (!userId)
    return res.status(401).json({
      code: 401,
      message: 'Utente non autorizzato.'
    });

  let user = await Utente.findOne({ _id: userId }).exec().catch((err) => {
    return res.status(500).json({
      code: 500,
      message: 'Internal server error.'
    });
  });

  if (!user)
    return res.status(404).json({
      code: 404,
      message: 'Utente non trovato.'
    });

  const annunci_pubblicati = await Annuncio.find({ _id: { $in: user.annunci_pubblicati } });

  return res.status(200).json({
    code: 200,
    message: 'Elenco degli annunci pubblicati ottenuto correttamente',
    annunci_pubblicati: annunci_pubblicati
  });
}
```

L'API degli annunci pubblicati è un endpoint del backend che restituisce l'elenco di annunci pubblicati da un determinato utente.

Dopo che l'API ha ricevuto la richiesta, effettua un controllo per verificare che l'utente è registrato e quindi autorizzato ad eseguire l'API.

Se i controlli hanno esito positivo, vengono restituiti all'utente tutti gli annunci da lui pubblicati. Se i controlli hanno esito negativo, ovvero se l'utente non è autorizzato, viene restituito un errore.

2.5.5 Cancella account

```
router.delete('/utente/:id', tokenChecker, utenteController.cancella_account);
```

```
const cancella_account = async (req, res) => {
    const userId = req.loggedUser.id;
    let user = await Utente.findOne({ _id: userId }).exec().catch((err) => {
        return res.status(500).json({
            code: 500,
            message: 'Internal server error.'
        });
    });

    //controllo se l'utente esiste nel database
    if(!user)
        return res.status(404).json({
            code: 404,
            message: "L'utente richiesto non esiste."
        });

    //prima di cancellare un utente devo cancellare tutti gli annunci che ha pubblicato
    Annuncio.deleteMany({ _id: { $in: user.annunci_pubblicati } }, (err) => {
        if (err)
            return res.status(500).json({
                message: 'Internal server error.'
            });
    });

    Utente.findOneAndDelete({ _id: userId }, (err, data) => {
        if (err)
            return res.status(500).json({
                code: 500,
                message: "Internal server error."
            });
        else
            return res.status(200).json({
                code: 200,
                message: "Utente cancellato correttamente."
            });
    });
}
```

L'API per la cancellazione dell'account è un endpoint del backend che permette di cancellare un utente dal sistema.

Dopo che l'API ha ricevuto la richiesta, effettua un controllo per verificare che l'utente sia registrato nel sito.

Se i controlli hanno esito positivo, vengono prima cancellati tutti gli annunci che aveva pubblicato l'utente e successivamente cancellato l'utente stesso dal sistema. Se i controlli hanno esito negativo viene restituito un errore.

Annuncio

2.5.6 Pubblicazione annuncio

```
router.post('/annuncio', tokenChecker, annuncioController.pubblica_annuncio);
```

```
const pubblica_annuncio = async (req, res) => {
  const {
    userId,
    foto,
    numero_bagni,
    numero_locali,
    locali,
    superficie_tot,
    prezzo,
    classe_energetica,
    indirizzo,
    arredato,
    durata_vetrina
  } = req.body;
  var scadenza_vetrina = calcola_vetrina(durata_vetrina);

  if (!(numero_bagni &&
        numero_locali &&
        superficie_tot &&
        prezzo &&
        classe_energetica &&
        indirizzo))
    return res.status(400).json({
      code: 400,
      message: 'Alcuni parametri sono assenti.'
    });
  if (!userId)
    return res.status(401).json({
      code: 401,
      message: 'Utente non autorizzato.'
    });
  const nuovoAnnuncio = new Annuncio({
    creatore: userId,
    foto: foto,
    numero_bagni: numero_bagni,
    numero_locali: numero_locali,
    locali: locali,
    superficie_tot: superficie_tot,
    prezzo: prezzo,
    classe_energetica: classe_energetica,
```

```
    indirizzo: indirizzo,
    arredato: arredato,
    vetrina: {
      data_inizio: new Date(),
      data_fine: scadenza_vetrina
    }
  });

//salviamo l'annuncio nel database
nuovoAnnuncio.save((err) => {
  if (err)
    return res.status(500).json({
      code: 500,
      message: 'Internal server error.'
    });
});

//lo inseriamo nella lista degli annunci salvati dell'utente
const user = await Utente.findById(userId);
user.annunci_pubblicati.push(nuovoAnnuncio._id);
await user.save();
return res.status(201).json({
  code: 201,
  message: 'Annuncio creato con successo.'
});
}
```

L'API per la creazione di un annuncio è un endpoint del backend che gestisce le richieste di creazione di nuovi annunci da parte dell'utente. Quando un utente tenta di pubblicare un nuovo annuncio, invia una richiesta HTTP al backend contenente tutte le informazioni dell'annuncio. In seguito l'API effettua dei controlli preliminari per verificare che i campi non siano vuoti e che l'utente sia registrato nel sistema. Infine l'id dell'annuncio viene salvato nell'elenco degli annunci pubblicati dell'utente.

2.5.7 Visualizza annuncio

```
router.get('/annuncio/:id', annuncioController.get_annuncio);
```

```
const get_annuncio = async (req, res) => {
    Annuncio.findOne({ _id: req.params.id }, (err, data) => {
        if (err)
            return res.status(500).json({
                code: 500,
                message: 'Internal server error.'
            });
        else if (!data)
            return res.status(404).json({
                code: 404,
                message: "Annuncio non trovato."
            });
        else
            return res.status(200).json({
                code: 200,
                message: "Annuncio ottenuto correttamente.",
                annuncio: data
            });
    });
}
```

L'API per ottenere un annuncio è un endpoint del backend che restituisce un annuncio dato il suo id.

Quando un utente clicca su un annuncio, invia una richiesta HTTP al backend contenente l'id dell'annuncio.

In seguito l'API controlla se l'annuncio esiste e, in caso di successo, restituisce tutte le informazioni relative all'annuncio. Se l'annuncio richiesto non esiste viene restituito un errore.

2.5.8 Salva annuncio

```
router
  .get('/annuncio/salva/:id', tokenChecker, annuncioController.salva_annuncio);
```

```
const salva_annuncio = async (req, res) => {
  Utente.findOne({ _id: req.loggedUser.id }, (err, user) => {
    if (err)
      return res.status(500).json({
        code: 500,
        message: 'Internal server error.'
      });
    else if (!user)
      return res.status(404).json({
        code: 404,
        message: 'Utente non trovato.'
      });

    if (user.annunci_salvati.includes(req.params.id))
      return res.status(409).json({
        code: 409,
        message: 'Annuncio già presente negli annunci salvati.'
      });
    Annuncio.findOne({ _id: req.params.id }, (err, data) => {
      if(err) {
        return res.status(500).json({
          code: 500,
          message: 'Internal server error.'
        });
      }
      if(!data) {
        return res.status(404).json({
          code: 404,
          message: 'Annuncio non trovato.'
        });
      }
    });
    Utente.findOneAndUpdate({ _id: req.loggedUser.id }, { $push: { annunci_salvati: req.params.id } }, (err, data) => {
      if (err)
        return res.status(500).json({
          code: 500,
          message: 'Internal server error.'
        });
      else if (!data)
```

```
        return res.status(404).json({
            code: 404,
            message: 'Annuncio non trovato.'
        });
    else
        return res.status(200).json({
            code: 200,
            message: "Annuncio salvato con successo."
        });
    });
}
}
```

L'API per salvare un annuncio è un endpoint del backend che permette di inserire un annuncio tra gli annunci salvati di un determinato utente.

Quando un utente clicca sul pulsante per salvare un annuncio, invia una richiesta HTTP al backend contenente l'id dell'annuncio e dell'utente.

Dopo che l'API ha ricevuto la richiesta, effettua un controllo per verificare che l'annuncio esista e che non fosse già stato salvato in precedenza dall'utente. Se i controlli hanno esito positivo, viene confermato all'utente che l'annuncio è stato salvato con successo e l'id dell'annuncio viene salvato nell'elenco degli annunci salvati dall'utente. In caso contrario, viene restituito un errore.

2.5.9 Elimina annuncio

```
router
.delete('/annuncio/:id', tokenChecker, annuncioController.elimina_annuncio);
```

```
const elimina_annuncio = (req, res) => {
    const id = req.params.id;
    Annuncio.findOneAndDelete({ _id: id }, (err, data) => {
        if (err)
            return res.status(500).json({
                code: 500,
                message: 'Internal server error.'
            });
        else if (!data)
            return res.status(404).json({
                code: 404,
                message: 'Annuncio non trovato.'
            });
        else {
            // rimuoviamo l'id dell'annuncio dalla lista di annunci pubblicati dall'utente
            Utente.findOneAndUpdate({ _id: req.loggedUser.id }, { $pull: { annunci_pubblicati: req.params.id } }, (err, data) => {
                if (err)
                    return res.status(500).json({
                        code: 500,
                        message: 'Internal server error.'
                    });
                return res.status(200).json({
                    code: 200,
                    message: 'Annuncio eliminato con successo.'
                });
            });
        }
    });
}
```

L'API per la cancellazione di un annuncio è un endpoint del backend che permette di cancellare un annuncio dal sistema.

Dopo che l'API ha ricevuto la richiesta, effettua un controllo per verificare che l'annuncio esista nel sistema.

Se i controlli hanno esito positivo, l'annuncio viene cancellato con successo, altrimenti viene restituito un errore.

2.5.10 Rimozione annuncio salvato

```
router
.delete('/annuncio/salva/:id', tokenChecker, annuncioController.rimuovi_annuncio_salvato);
```

```
const rimuovi_annuncio_salvato = async (req, res) => {
  const { id } = req.params;
  Utente.findOneAndUpdate({ _id: req.loggedUser.id }, { $pull: { annunci_salvati: id } }, (err, data) => {
    if (err)
      return res.status(500).json({
        code: 500,
        message: 'Internal server error.'
      });
    else if (!data)
      return res.status(404).json({
        code: 404,
        message: 'Utente non trovato.'
      });
    else
      return res.status(200).json({
        code: 200,
        message: "Annuncio rimosso con successo."
      });
  });
}
```

L'API per la rimozione di un annuncio dagli annunci salvati è un endpoint del backend che consente di eliminare un annuncio specifico dalla lista degli annunci salvati dall'utente.

Quando un utente clicca sul pulsante per rimuovere un annuncio dagli annunci salvati, invia una richiesta HTTP al backend contenente l'id dell'annuncio e dell'utente stesso.

Dopo che l'API ha ricevuto la richiesta, effettua un controllo per verificare che l'utente esista. In caso di successo, viene confermato all'utente che l'annuncio è stato rimosso con successo e l'id dell'annuncio viene rimosso dall'elenco degli annunci salvati dall'utente. Se i controlli hanno esito negativo, viene restituito un errore.

Ricerca

2.5.11 Salva ricerca

```
router.post('/ricerca/salva', tokenChecker, ricercaController.salva_ricerca);
```

```
const salva_ricerca = async (req, res) => {
    const id = req.loggedUser.id;
    const {
        superficie_tot, numero_bagni, numero_locali,
        prezzo, classe_energetica, indirizzo, arredato
    } = req.query;

    const filtri = {
        superficie_tot, numero_bagni, numero_locali, prezzo,
        classe_energetica, arredato
    };

    // Rimuovi i parametri nulli o vuoti
    Object.keys(filtri).forEach(key => filtri[key] = null && delete filtri[key]);

    const nuovaRicerca = new Ricerca({
        testo: indirizzo,
        filtri: filtri
    });
    //salviamo l'annuncio nel database
    nuovaRicerca.save((err) => {
        if (err)
            return res.status(500).json({
                code: 500,
                message: 'Internal server error.'
            });
    });

    Utente.findOneAndUpdate({ _id: id }, { $push: { ricerche_salvate: nuovaRicerca._id } }, (err, data) => {
        if (err)
            return res.status(500).json({
                code: 500,
                message: 'Internal server error.'
            });
        else if (!data)
            return res.status(404).json({
                code: 404, message: 'Utente non trovato.'
            });
        else
            return res.status(200).json({
                code: 200,
                message: "Ricerca salvata con successo."
            });
    });
}
```

L'API per salvare una ricerca è un endpoint del backend che permette di inserire una ricerca tra le ricerche salvate da un determinato utente.

Quando un utente clicca sul pulsante per salvare una ricerca, invia una richiesta HTTP al backend contenente l'id dell'utente.

Dopo che l'API ha ricevuto la richiesta, effettua un controllo per verificare che l'utente esista e per rimuovere i campi che l'utente ha lasciato vuoti.

Se i controlli hanno esito positivo, viene confermato all'utente che la ricerca è stata salvata con successo, viene creata, e il suo id viene salvato nell'elenco delle ricerche salvate dall'utente. Altrimenti viene restituito un errore.

Chat

2.5.12 Crea chat

```
router.post('/ricerca', ricercaController.ricerca_annunci);
```

```
const crea_chat = async (req, res) => {
    const { id_annuncio, id_mittente, id_destinatario } = req.body;
    const userId = req.loggedUser.id;

    if (!userId)
        return res.status(401).json({
            code: 401,
            message: 'Utente non autorizzato.'
        });

    if (!id_annuncio || !id_mittente || !id_destinatario) {
        return res.status(500).json({
            code: 500,
            message: "Internal server Error."
        });
    }
    let chat = await Chat.findOne({ annuncio: id_annuncio, utente: id_mittente, locatore: id_destinatario });

    if(chat)
        return res.status(409).json({
            code: 409,
            message: "Chat già esistente."
        });
    else {
        const nuovaChat = new Chat({
            utente: id_mittente,
            locatore: id_destinatario,
            annuncio: id_annuncio
        });
        //salviamo la chat nel database
        nuovaChat.save((err) => {
            if (err)
                return res.status(500).json({
                    code: 500,
                    message: 'Internal server error.'
                });
        });

        //inseriamo l'id della chat nella lista_chat dell'utente
        const mittente = await Utente.findById(id_mittente);
        mittente.lista_chat.push(nuovaChat._id);
        await mittente.save();
        const locatore = await Utente.findById(id_destinatario);
        locatore.lista_chat.push(nuovaChat._id);
        await locatore.save();
        return res.status(201).json({
            code: 201,
            message: 'Chat creata con successo.'
        });
    }
}
```

```
    }
}
```

L'API per la creazione di una chat è un endpoint del backend che permette di avviare una conversazione tra un locatore e un utente, in relazione ad un annuncio specifico.

Quando un utente clicca sul pulsante per inviare un messaggio al locatore, viene verificata l'esistenza della chat e, in caso non esista, viene chiamata questa API. Prima di tutto viene effettuato un controllo per verificare che l'utente sia registrato e che la chat non esista. Se i controlli hanno esito positivo, la chat viene creata con successo e il suo id viene salvato nell'elenco delle chat dell'utente e del locatore. In caso contrario, viene restituito un errore.

Messaggio

2.5.13 Invio messaggio

```
router.post('/messaggio', tokenChecker, messaggioController.invia_messaggio);
```

```
const invia_messaggio = async (req, res) => {
  const { id_chat, messaggio } = req.body;

  const id_mittente = req.loggedUser.id;

  let data = new Date();

  if (!id_mittente)
    return res.status(401).json({
      code: 401,
      message: 'Utente non autorizzato.'
    });
  const nuovoMessaggio = new Messaggio({
    messaggio: messaggio,
    data: data,
    mittente: id_mittente
  });

  //salviamo il messaggio nel database
  nuovoMessaggio.save((err) => {
    if (err)
      return res.status(500).json({
        code: 500,
```

```
        message: 'Internal server error.'
    });
}

//lo inseriamo nella chat dell'utente
const chat = await Chat.findById(id_chat);
if(!chat) {
    return res.status(404).json({
        code: 404,
        message: 'Chat inesistente.'
    });
}
if(!chat.messaggi) {
    chat.messaggi = [];
}
chat.messaggi.push(nuovoMessaggio._id);
await chat.save();
return res.status(200).json({
    code: 200,
    message: 'Messaggio inviato con successo.'
});
}
```

L'API per l'invio di un messaggio è un endpoint del backend che permette di inviare un messaggio ad un locatore, in relazione ad un annuncio specifico.

Quando un utente clicca sul pulsante per inviare un messaggio al locatore, viene verificata l'esistenza della chat e, in caso di successo, viene chiamata questa API. Prima di tutto viene effettuato un controllo per verificare che l'utente sia registrato e che la chat sia già esistente. Se i controlli hanno esito positivo, il messaggio viene inviato con successo e l'id del messaggio viene salvato nella chat, altrimenti viene restituito un errore.



3. Documentazione delle API

Le API fornite dal lato backend dell'applicazione sono state documentate utilizzando Swagger UI. Grazie al modulo swagger-ui-express per NodeJS è stato possibile descrivere e documentare ogni endpoint, rendendo il tutto accessibile in una pagina web.

La pagina presenta un elenco con tutte le API e le relative descrizioni. Inoltre, cliccando su una API, si aprirà un menù a tendina con informazioni aggiuntive, come il formato dell'input e i formati e i codici degli output.

L'endpoint da invocare per raggiungere la seguente documentazione è:

<https://housefinder-backend.vercel.app/api-docs/>

The screenshot shows the Swagger UI interface for the HouseFinder API. At the top, it displays the title "HouseFinder API 1.0.0" and the base URL "[Base URL: housefinder-backend.vercel.app/]". Below this, a descriptive text states: "Lista degli endpoint dell'applicazione HouseFinder e informazioni sul loro utilizzo." The main content area is titled "utente" and contains a list of API endpoints for user management. Each endpoint is shown with its method, path, and a brief description. Some endpoints are highlighted with different colors (red, orange, blue) and icons (lock, padlock). The "Schemes" dropdown is set to "HTTPS" and the "Authorize" button is visible.

Method	Path	Description	Status
POST	/utente/login	Accesso account	🔒
POST	/utente/registrazione	Registrazione	🔒
PATCH	/utente/{id}	Modifica profilo	🔒
DELETE	/utente/{id}	Elimina profilo	🔒
PUT	/utente/recupero-password	Recupero password	🔒
GET	/utente/logout	Logout	🔒
GET	/utente/annunci-pubblicati	Annunci pubblicati	🔒
GET	/utente/annunci-salvati	Annunci salvati	🔒
GET	/utente/ricerche-salvate	Ricerche Salvate	🔒
GET	/utente/chat	Lista chat	🔒

**annuncio** API per la gestione degli annunci

GET	/annuncio/list	Lista annunci	✓	🔒
GET	/annuncio/{id}	Visualizza annuncio	✓	🔒
PATCH	/annuncio/{id}	Modifica annuncio	✓	🔒
DELETE	/annuncio/{id}	Elimina annuncio	✓	🔒
POST	/annuncio	Pubblica annuncio	✓	🔒
GET	/annuncio/salva/{id}	Salva annuncio	✓	🔒
DELETE	/annuncio/salva/{id}	Rimuovi annuncio salvato	✓	🔒
POST	/annuncio/pagamento	Pagamento	✓	🔒

ricerca API per la gestione della ricerca

POST	/ricerca	Ricerca annunci	✓	🔒
POST	/ricerca/salva	Salva ricerca	✓	🔒
DELETE	/ricerca/salva/{id}	Rimuovi ricerca salvata	✓	🔒

chat API per la gestione della chat

POST	/chat	Crea chat	✓	🔒
GET	/chat/{id}	Apri chat	✓	🔒

messaggio API per la gestione dei messaggi

POST	/messaggio	Invia messaggio	✓	🔒
-------------	------------	-----------------	---	---

È possibile provare ogni endpoint premendo sull'apposito pulsante "Try it out". Comparirà una casella di testo in cui sarà possibile digitare l'input, se necessario, e premendo sul pulsante "Execute" si potrà mandare la richiesta al server.

utente API per la gestione dell'utente

POST /utente/login Accesso account

Effettua il login

Parameters

Try it out

Name	Description
credentials * required object (body)	Credenziali necessarie Example Value Model

```
{
  "email": "string",
  "password": "string"
}
```

Parameters

Cancel

Name	Description
credentials * required object (body)	Credenziali necessarie Edit Value Model

```
{
  "email": "prova@prova.com",
  "password": "Qwerty1234#"
}
```

Cancel

Parameter content type

application/json

Execute

Dunque, il server risponderà e la pagina riceverà e rappresenterà l'output con i propri codici di stato.



Responses

Response content type: application/json

Curl

```
curl -X 'POST' \
  'https://housefinder-backend.vercel.app/utente/login' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "email": "prova@prova.com",
    "password": "Qwerty1234#"
}'
```

Request URL

<https://housefinder-backend.vercel.app/utente/login>

Server response

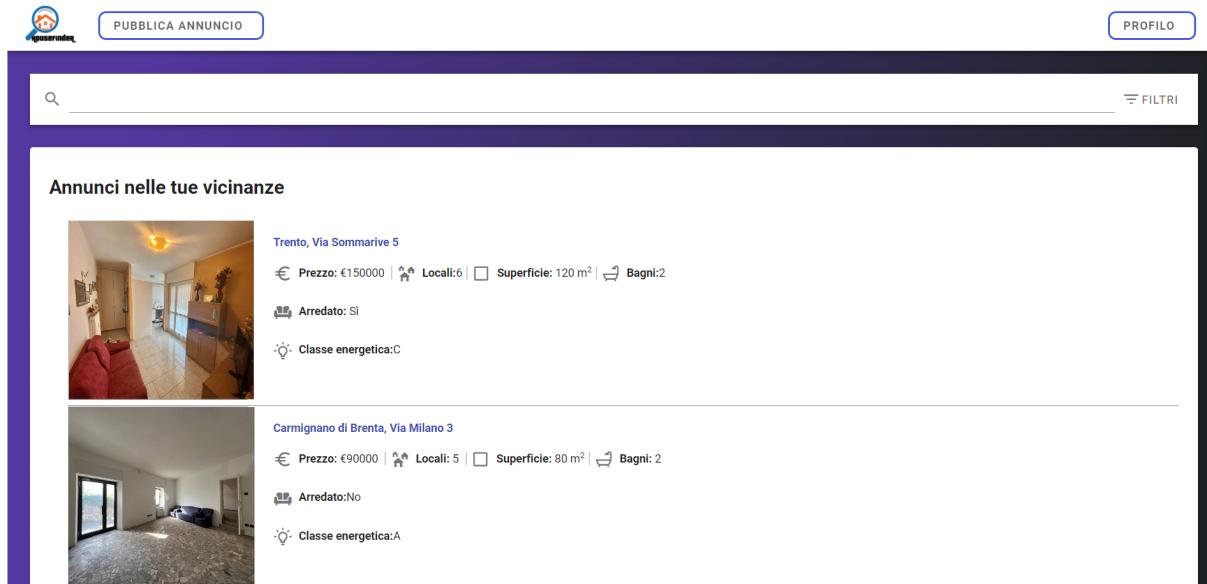
Code	Details
200	<p>Response body</p> <pre>{ "code": 200, "message": "L'accesso è avvenuto correttamente.", "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjp7ImlkIjoiNjQ4MmRmMDJmMGMyODIjODc3MDY0YjQ0In0sImhdCI6MTY4NjMwMTEyNywiZXhwIjoxNjg2Mzg3NTI3fQ.h9J09IDQtGhy8u91JBy_FfgPXYIoPTZ1_Ah6x4FZiv0", "id": "6482df02f0c089c877064b44", "nome": "Prova", "cognome": "Prova", "email": "prova@prova.com" }</pre> <p>Copy Download</p> <p>Response headers</p> <pre>access-control-allow-origin: * cache-control: public,max-age=0,must-revalidate content-length: 346 content-type: application/json; charset=utf-8 date: Fri, 09 Jun 2023 08:58:47 GMT etag: W/"15a-PcjTjKu7gbSzctqWITz/oy3fVo" server: Vercel strict-transport-security: max-age=63072000; includeSubDomains; preload x-powered-by: Express x-vercel-cache: MISS x-vercel-id: fra1::iad1::fhpk7-1686301123801-ab2c94b0b56a</pre>

4. Implementazione frontend

4.1 Home

Nell'header della pagina principale (presente in ogni pagina) si trova il logo del sito insieme a tre pulsanti: 'Registrati', 'Accedi' e 'Profilo'.

Nella parte inferiore della pagina, si trova una lista di annunci in evidenza, personalizzati in base alla posizione dell'utente.

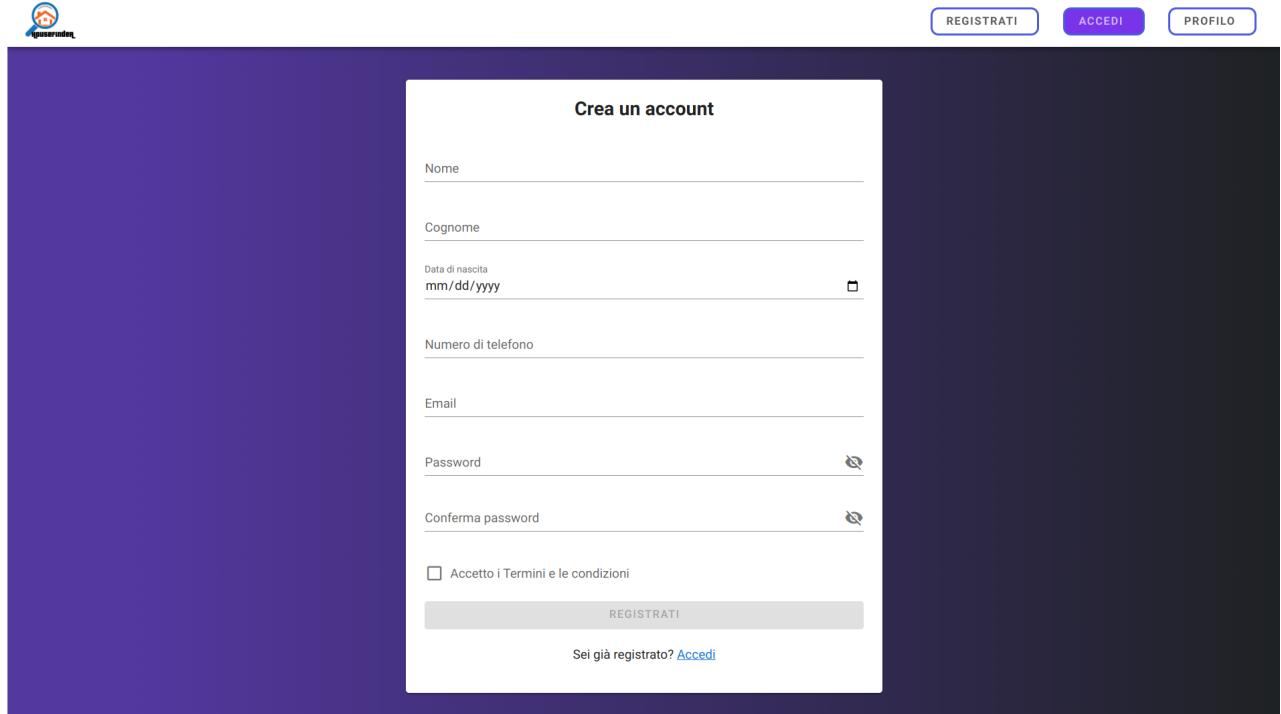


The screenshot shows the main interface of the Housefinder app. At the top, there is a dark header bar with a search bar containing a magnifying glass icon, a 'PUBBLICA ANNUNCIO' button, and a 'PROFILO' button. Below the header, there is a purple sidebar on the left and a white main content area. The main content area is titled 'Annunci nelle tue vicinanze' (Announcements in your vicinity). It displays two property listings:

- Trento, Via Sommarive 5**
€ Prezzo: €150000 | Locali: 6 | Superficie: 120 m² | Bagni: 2
Arredato: Sì
Classe energetica: C
- Carmignano di Brenta, Via Milano 3**
€ Prezzo: €90000 | Locali: 5 | Superficie: 80 m² | Bagni: 2
Arredato: No
Classe energetica: A

4.2 Registrazione

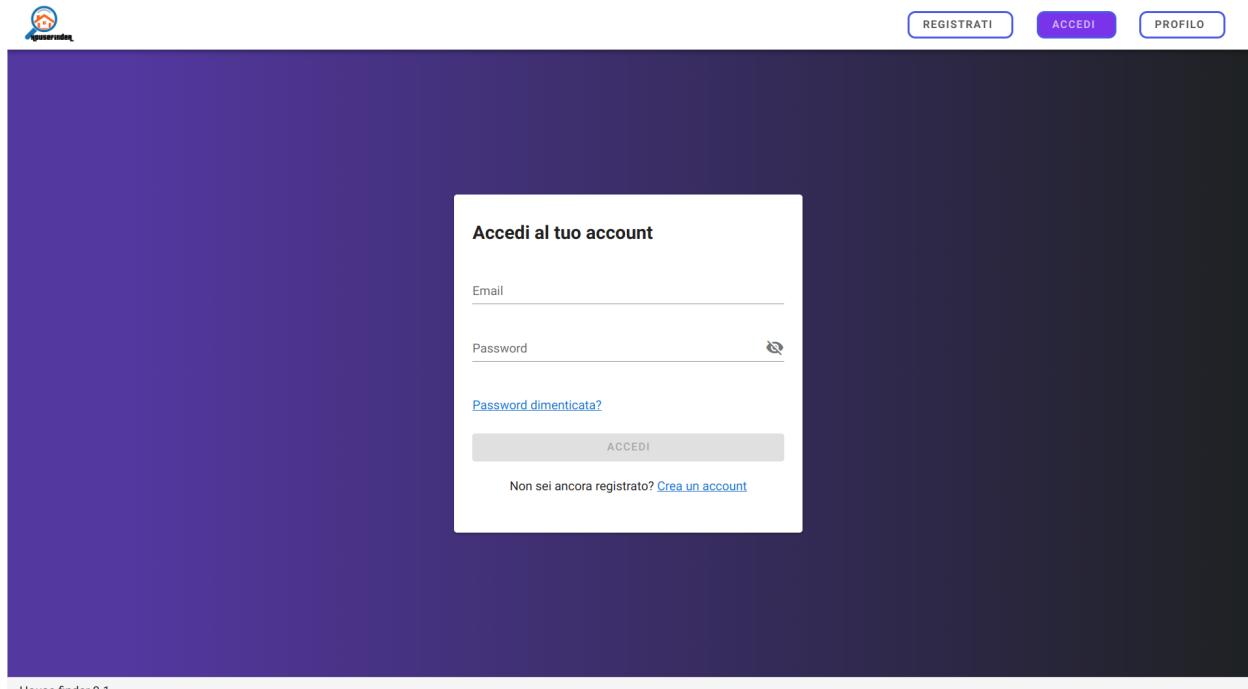
In questa pagina si trova un modulo di registrazione completo, che include tutti i campi necessari per inserire i dati richiesti per registrarsi al sito.



The screenshot shows a registration form titled "Crea un account". It includes fields for Nome, Cognome, Data di nascita (mm/dd/yyyy), Numero di telefono, Email, Password, and Conferma password. There is also a checkbox for accepting terms and conditions. At the bottom, there is a "REGISTRATI" button and a link "Sei già registrato? Accedi".

4.3 Login

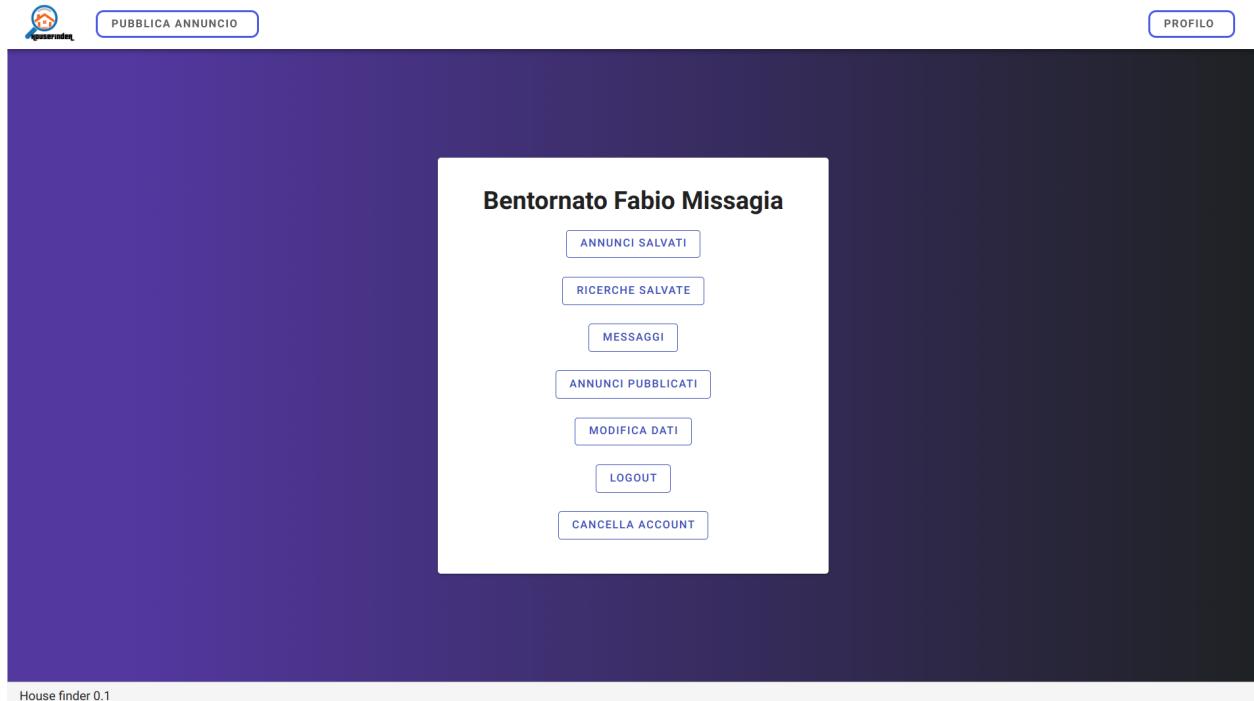
In questa pagina si trova un modulo di login, che include i campi 'email' e 'password' dove inserire i dati necessari per effettuare l'accesso al sito.



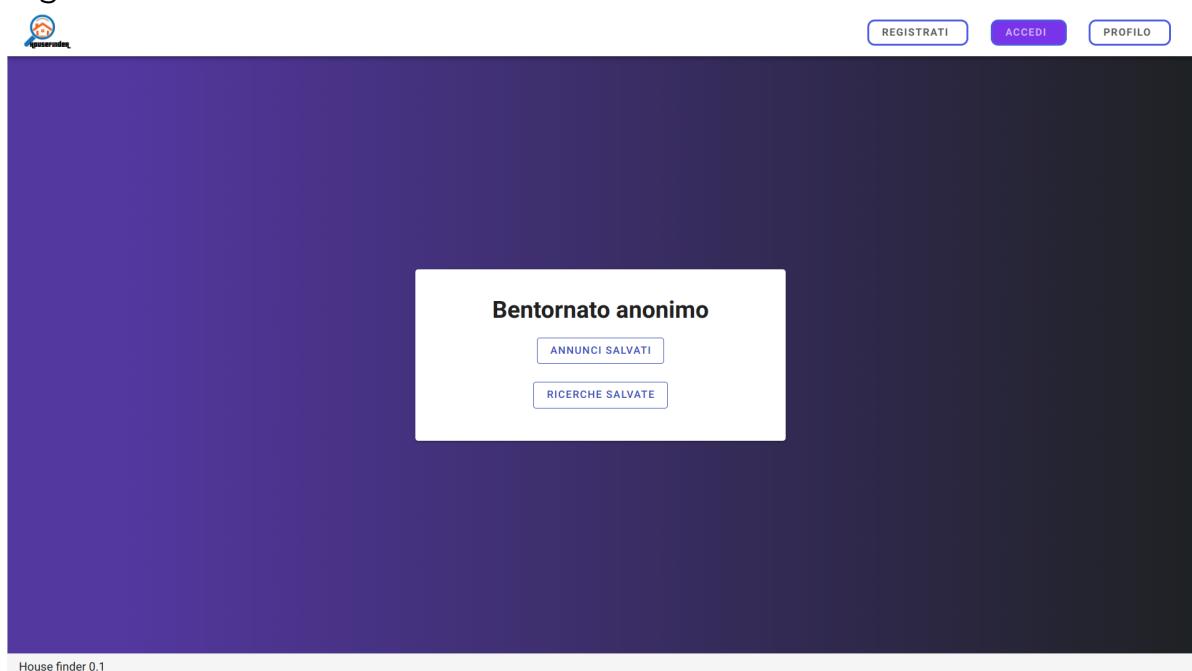
The screenshot shows a login form titled "Accedi al tuo account". It includes fields for Email and Password. Below the fields is a link "Password dimenticata?". At the bottom, there is an "ACCEDI" button and a link "Non sei ancora registrato? Crea un account".

4.4 Profilo

Una volta effettuato l'accesso nell'header sarà presente il pulsante 'Pubblica annuncio'. Nella pagina 'Profilo' verrà visualizzato un messaggio di benvenuto con il nome dell'utente. Sotto poi si trovano 7 sezioni accessibili dall'utente. 5 di queste sezioni portano ad altre pagine, le ultime 2 permettono di effettuare il logout e la cancellazione dell'account.



Se l'utente non ha effettuato l'accesso, la pagina 'Profilo' apparirà nel seguente modo:



4.5 Pubblicazione annuncio

In questa pagina si trova un modulo per la pubblicazione di un annuncio. Nella prima sezione si possono inserire le informazioni di base come via, prezzo, superficie ecc. Nella seconda sezione si possono inserire i vari locali che compongono l'immobile ed i mobili e/o elettrodomestici al loro interno.

 **Registrati**
 **Accedi**
 **Profilo**

Pubblica Annuncio

Superficie
120

Numero dei bagni
2

Prezzo
150000

Numero dei locali
6

Via
Trento, Via Sommarive 5

Classe energetica
C

Arredato Non arredato

Carica foto
11 files

Aggiungi i vari locali:

Inserisci il nome dei locali e i mobili al loro interno

Nome del locale Cucina	RIMUOVI LOCALE
Mobili al suo interno Lavastoviglie	
RIMUOVI	
Mobili al suo interno Microonde	
RIMUOVI	
AGGIUNGI MOBILE	
Nome del locale Bagno	RIMUOVI LOCALE
Mobili al suo interno Lavatrice	
RIMUOVI	
Mobili al suo interno Doccia	
RIMUOVI	
AGGIUNGI MOBILE	
AGGIUNGI LOCALE	

Annuncio in vetrina
Se si vuole mettere l'annuncio in vetrina,
segliere per quanto attivare il servizio:

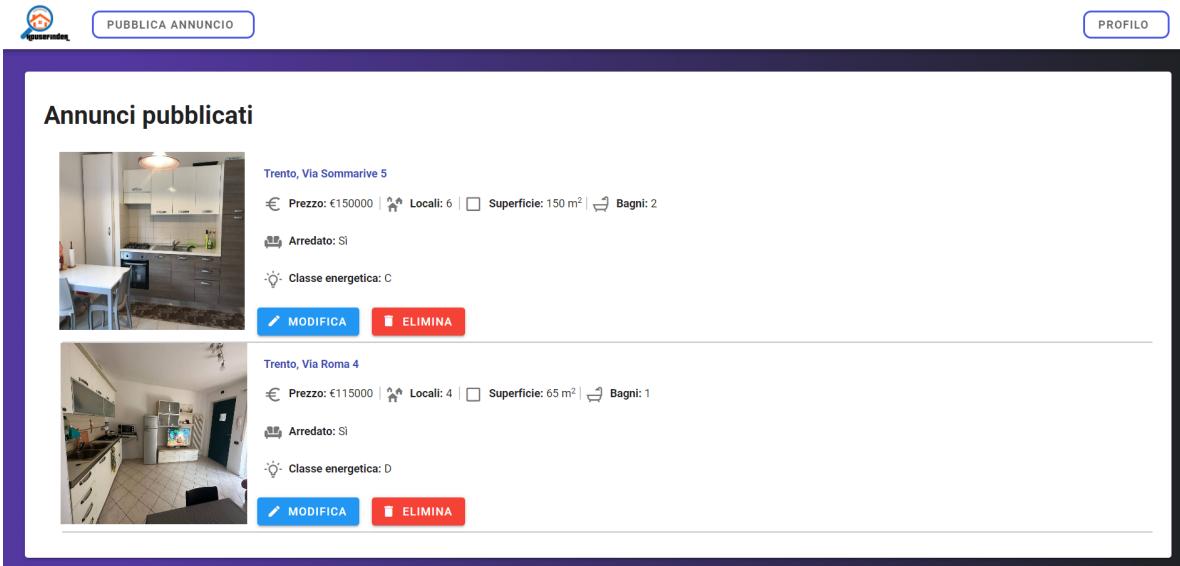
1 Giorno - €0,99
 7 Giorni - €4,99
 1 Mese - €19,99
 2 Mesi - €29,99

EFFETTUAR PAGAMENTO

PUBBLICA ANNUNCIO

4.6 Annunci pubblicati

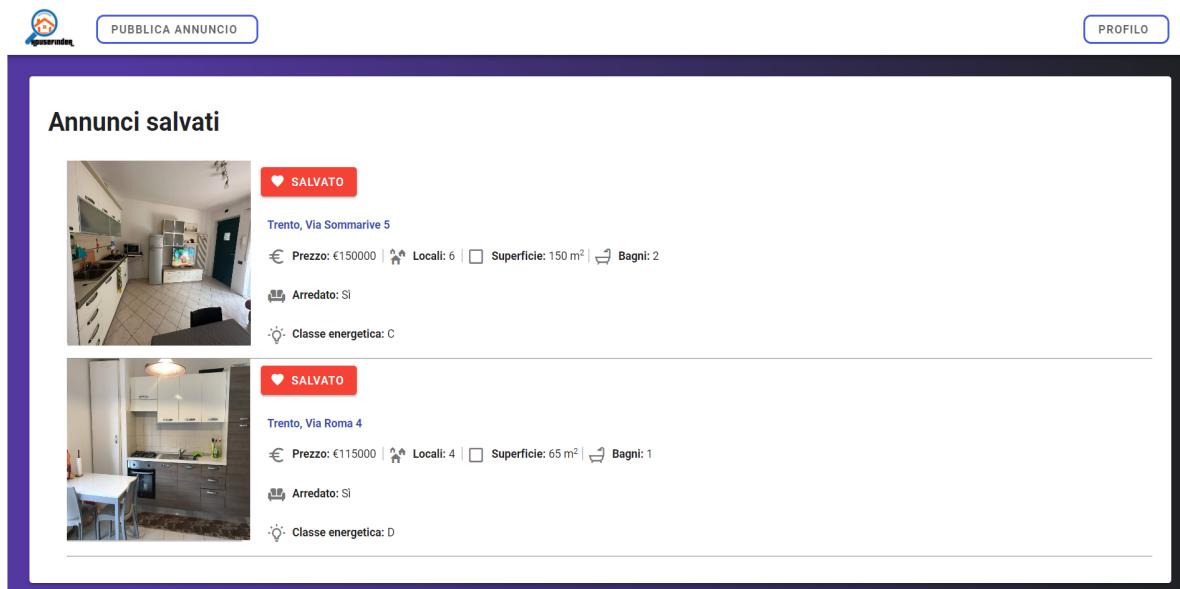
All'interno di questa pagina, accessibile attraverso il profilo dell'utente, si trova la lista degli annunci pubblicati. Per ciascun annuncio, vengono mostrate le informazioni di base, e sono presenti due pulsanti che consentono di modificarlo o eliminarlo.



Indirizzo	Prezzo	Locali	Superficie	Bagni	Arredato	Classe energetica
Trento, Via Sommarive 5	€150000	6	150 m ²	2	Sì	C
Trento, Via Roma 4	€115000	4	65 m ²	1	Sì	D

4.7 Annunci salvati

In questa pagina, accessibile attraverso il profilo dell'utente (anche non autenticato), si trova la lista degli annunci salvati. Per ogni annuncio, vengono mostrate le informazioni di base, ed è presente un pulsante 'salvato' che permette di rimuovere l'annuncio.



Indirizzo	Prezzo	Locali	Superficie	Bagni	Arredato	Classe energetica
Trento, Via Sommarive 5	€150000	6	150 m ²	2	Sì	C
Trento, Via Roma 4	€115000	4	65 m ²	1	Sì	D

5. Repository GitHub e informazioni sul deployment

5.1 Struttura Github repository

Per organizzare al meglio il progetto, è stato suddiviso su più repository secondo le varie funzioni che vanno a svolgere.

In totale sono state create 3 repository:

1. [Deliverables](#): contiene i documenti del progetto.
2. [Backend](#): contiene l'implementazione delle API, i file per il testing e tutto ciò che permette il funzionamento del sito.
3. [Frontend](#): contiene tutto ciò che compone l'interfaccia del sito, ossia pagine, fogli di stile, immagini ecc.

5.2 Informazioni deployment

Per il deployment abbiamo voluto evitare la complicazione e macchinosità del deployment in locale, abbiamo quindi optato per un deployment tramite piattaforma cloud.

Il deployment è stato effettuato tramite la piattaforma gratuita <https://vercel.com>, configurando l'automatic deployment sia per il frontend sia per il backend.

In particolare, per il backend è stato configurato il file `vercel.json` nel seguente modo:



```
{  
  "version": 2,  
  "builds": [  
    {  
      "src": "*.js",  
      "use": "@vercel/node"  
    }  
  ],  
  "routes": [  
    {  
      "src": "/(.*)",  
      "dest": "/"  
    }  
  ]  
}
```

Il sito è raggiungibile tramite il seguente link:

<https://housefinder-project.vercel.app>

Il backend invece, è raggiungibile tramite il seguente link:

<https://housefinder-backend.vercel.app>

6. Testing

Per il testing dell'applicazione è stato utilizzato il framework Jest. In particolare sono state create 5 testing suites per ogni modulo dell'applicazione: utente, annuncio, ricerca, chat, messaggio.

Per ogni test, si fa una richiesta http all'endpoint interessato, con allegati eventuali campi body o headers, e si verifica che il codice di stato restituito dal sistema sia quello aspettato.

Nei test in cui viene in qualche modo modificato il database, è presente una fase in cui viene ripristinato allo stato precedente al test. Per esempio, se si verifica la corretta creazione di un account utente, appena verificata la risposta del server, l'account viene cancellato.

Questo meccanismo è stato implementato per permettere la ripetibilità dei test.

Quando viene eseguito il comando “**npm run test**” nella cartella backend, viene eseguito lo script di test (contenuto nel file `package.json`).

Si possono notare i flag aggiuntivi “**-inBand --forceExit**”, che sono necessari per l'esecuzione corretta dei test.

“**-inBand**” fa sì che le testing suites vengano eseguite sequenzialmente piuttosto che in parallelo. Questo è necessario poiché se si dovessero eseguire in parallelo, sarebbero presenti casi di accesso simultaneo al database e l'esito dei test diviene quindi non deterministico.

“**--forceExit**” fa in modo di chiudere l'applicazione appena una testing suite si conclude. In questo modo non rimangono processi superflui in ascolto, occupando risorse e porte.

6.1 Tests

In totale sono stati programmati 54 tests organizzati in 5 testing suites. Eseguendo il comando “**npm run test**” nella cartella backend si può notare che ogni test viene passato.

```
PASS tests/annuncio.test.js (5.416 s)
POST /annuncio/list
  ✓ POST /annuncio/list con token deve ritornare 200 (940 ms)
GET /annuncio/{id}
  ✓ GET /annuncio/{id} con token deve ritornare 200 (78 ms)
  ✓ GET /annuncio/{id} con token, se l'annuncio non esiste, deve ritornare 404 (88 ms)
DELETE /annuncio/{id}
  ✓ DELETE /annuncio/{id} con token, se l'id esiste, deve ritornare 200 (353 ms)
  ✓ DELETE /annuncio/{id} con token, se l'id non esiste, deve ritornare 404 (114 ms)
PATCH /annuncio/{id}
  ✓ PATCH /annuncio/{id} con token, se l'id esiste, deve ritornare 200 (380 ms)
POST /annuncio
  ✓ POST /annuncio con tutti i parametri deve ritornare 200 (311 ms)
  ✓ POST /annuncio con parametri mancanti deve ritornare 400 (40 ms)
GET /annuncio/salva/{id}
  ✓ GET /annuncio/salva con token, se l'id esiste, deve ritornare 200 (188 ms)
  ✓ GET /annuncio/salva con token, se l'id non esiste, deve ritornare 404 (183 ms)
  ✓ GET /annuncio/salva con token, se l'id è già salvato, deve ritornare 409 (267 ms)
DELETE /annuncio/salva/{id}
  ✓ DELETE /annuncio/salva con token deve ritornare 200 (150 ms)
POST /annuncio/pagamento
  ✓ POST /annuncio/pagamento con token, con l'id dell'annuncio deve ritornare 200 (336 ms)
  ✓ POST /annuncio/pagamento con token, senza parametri, deve ritornare 500 (41 ms)

PASS tests/utente.test.js
POST /utente/registrazione
  ✓ POST /utente/registrazione senza parametri deve ritornare 400 (11 ms)
  ✓ POST /utente/registrazione con parametri mancanti deve ritornare 400 (9 ms)
  ✓ POST /utente/registrazione con una email già esistente deve ritornare 409 (619 ms)
  ✓ POST /utente/registrazione con parametri corretti deve ritornare 201 (189 ms)
PATCH /utente
  ✓ PATCH /utente senza parametri deve ritornare 404 (3 ms)
  ✓ PATCH /utente con parametri validi deve ritornare 200 (189 ms)
  ✓ POST /utente/registrazione con parametri corretti deve ritornare 201 (179 ms)
DELETE /utente
  ✓ DELETE /utente senza parametri deve ritornare 404 (3 ms)
  ✓ DELETE /utente senza parametri deve ritornare 404 (3 ms)
  ✓ DELETE /utente con parametri validi deve ritornare 200 (185 ms)
POST /utente/login
  ✓ POST /utente/login con parametri mancanti deve ritornare 400 (2 ms)
  ✓ POST /utente/login con parametri email non esistente deve ritornare 404 (36 ms)
  ✓ POST /utente/login con password errata deve ritornare 401 (37 ms)
  ✓ POST /utente/login con parametri corretti deve ritornare 200 (37 ms)
GET /utente/logout
  ✓ GET /utente/logout deve ritornare 200 (4 ms)
GET /utente/annunci-pubblicati
  ✓ GET /utente/annunci-pubblicati con utente anonimo deve ritornare 401 (3 ms)
  ✓ GET /utente/annunci-pubblicati con utente inesistente deve ritornare 404 (36 ms)
  ✓ GET /utente/annunci-pubblicati con utente loggato deve ritornare 200 (108 ms)
GET /utente/annunci-salvati
  ✓ GET /utente/annunci-salvati con utente anonimo deve ritornare 401 (2 ms)
  ✓ GET /utente/annunci-salvati con utente inesistente deve ritornare 404 (38 ms)
  ✓ GET /utente/annunci-salvati con utente loggato deve ritornare 200 (108 ms)
GET /utente/ricerche-salvate
  ✓ GET /utente/ricerche-salvate con utente anonimo deve ritornare 401 (4 ms)
  ✓ GET /utente/ricerche-salvate con utente inesistente deve ritornare 404 (37 ms)
  ✓ GET /utente/ricerche-salvate con utente loggato deve ritornare 200 (108 ms)
GET /utente/chat
  ✓ GET /utente/chat con utente anonimo deve ritornare 401 (3 ms)
  ✓ GET /utente/chat con utente inesistente deve ritornare 404 (38 ms)
  ✓ GET /utente/chat con utente loggato deve ritornare 200 (109 ms)
PUT /utente/recupero-password
  ✓ PUT /utente/recupero-password deve ritornare 500 (2 ms)

PASS tests/ricerca.test.js
POST /ricerca
  ✓ POST /ricerca senza parametri ritorna 200 (677 ms)
  ✓ POST /ricerca con parametri validi ritorna 200 (90 ms)
POST /ricerca/salva
  ✓ POST /ricerca/salva senza o con parametri ritorna 200 (256 ms)
DELETE /ricerca/salva/:id
  ✓ DELETE /ricerca/salva/:id con id valido ritorna 200 (245 ms)
  ✓ DELETE /ricerca/salva/:id con id non valido ritorna 500 (197 ms)
```

```
PASS tests/messaggio.test.js
POST /messaggio
✓ POST /messaggio con parametri corretti deve ritornare 200 (720 ms)
✓ POST /messaggio con chat inesistente deve ritornare 404 (67 ms)

PASS tests/chat.test.js
POST /chat
✓ POST /chat con parametri corretti deve ritornare 201 (827 ms)
✓ POST /chat che tenta di creare una chat esistente ritorna 409 (85 ms)
✓ POST /chat senza parametri deve ritornare 500 (41 ms)
GET /chat/{id}
✓ GET /chat/{id} con id valido deve ritornare 200 (78 ms)
✓ GET /chat/{id} con id inesistente deve ritornare 404 (76 ms)
```

Di seguito, alcuni esempi di test.

Login

```
test('POST /utente/login con parametri corretti deve ritornare 200',
async () => {
  const response = await request(app).post('/utente/login').send({
    email: "test@gmail.com",
    password: "Qwerty1234#"
  });
  expect(response.statusCode).toBe(200);
});
```

Nei test in cui è richiesto un utente autenticato, viene eseguito il login con un account di test, e viene immagazzinato il token di autenticazione restituito dall'endpoint del login. Un esempio è il seguente test.

Annunci pubblicati

```
test('GET /utente/annunci-pubblicati con utente loggato deve ritornare 200',
async () => {
  const login_response = await request(app).post('/utente/login').send({
    email: "test@gmail.com",
    password: "Qwerty1234#"
  });

  var token;
  if (login_response.statusCode === 200)
    token = login_response.body.token;

  const response = await
request(app).get('/utente/annunci-pubblicati').set('x-access-token', token);
  expect(response.statusCode).toBe(200);
});
```

6.2 Coverage

Eseguendo il comando “**npm run test**” nella cartella backend si ottiene il seguente risultato.

File	%Stmts	%Branch	%Funcs	%Lines	Uncovered Line #s
All files	87.31	70.7	85.71	87.55	
backend	89.09	41.66	85.71	92.3	
db.js	80	58	75	91.66	6
index.js	96.77	25	100	96.77	37
swagger.js	77.77	58	100	77.77	8-9
backend/controllers	83.38	72.06	85	83.22	
annuncio.js	83.83	76.47	95.23	83.83	8,13,30,52,57,70,84,89,136,161,201,228,240,253,271,276
chat.js	90.62	88	100	90.62	10,39,65
messaggio.js	85.71	62.5	100	85.71	12,26,41
ricerca.js	74.5	59.37	100	74	23,30,37,44,51,63,70,75,122,130,135,152,157
utente.js	84.54	73.21	68	84.25	24,76,104,129,135,160,166,191,197,222,228,264,287,297,305,313,320
backend/middleware	92.3	85.71	100	91.66	
tokenChecker.js	92.3	85.71	100	91.66	9
backend/models	100	100	100	100	
annuncio.js	100	100	100	100	
chat.js	100	100	100	100	
filtro.js	100	100	100	100	
locale.js	100	100	100	100	
messaggio.js	100	100	100	100	
ricerca.js	100	100	100	100	
utente.js	100	100	100	100	
backend/routes	100	100	100	100	
annuncio.js	100	100	100	100	
chat.js	100	100	100	100	
messaggio.js	100	100	100	100	
ricerca.js	100	100	100	100	
utente.js	100	100	100	100	

Si può notare che gran parte del codice è coperto. Le righe non coperte riguardano alcune funzioni non implementate e parti di codice che vengono eseguite solo in modalità deploy.