

Progetto:

Co.Inqui

Titolo del documento:

Documento di sviluppo dell'applicazione web

Document Info:

Doc. Name	D4_Co.Inqui_SviluppoApplicazioneWeb		
Doc. Number	D4	Group number	G30
Description	Documento di sviluppo dell'applicazione web.		

SCOPO DEL DOCUMENTO	4
USER FLOW	5
APPLICATION IMPLEMENTATION AND DOCUMENTATION	6
PROJECT STRUCTURE	6
PROJECT DEPENDENCIES	7
MODELLI NEL DATABASE	8
MODELLO USER	8
MODELLO SESSION	8
MODELLO APARTMENT	9
MODELLO EXPENSE	9
PROJECT APIs	9
DIAGRAMMA DI ESTRAZIONE DELLE RISORSE	10
DIAGRAMMA DI ESTRAZIONE DELLE RISORSE	11
DIAGRAMMA DELLE RISORSE	12
ENDPOINTS PER AUTENTICAZIONE E REGISTRAZIONE	12
ENDPOINTS PER GLI UTENTI	12
ENDPOINTS PER LE SPESE	13
ENDPOINTS PER GLI APPARTAMENTI	13
SCHEMA COMPLETO	14
SVILUPPO API	15
AUTENTICAZIONE E REGISTRAZIONE	15
REGISTER	15
LOGIN	15
LOGOUT	15
UTENTE	16
FINDALL	16
FINDBYID	16
DELETEBYID	16
PATCHBYID & PUTBYID	16
GETAPARTMENTMEMBERS	16
DELETEAPARTMENTMEMBER	16
SPESE	17
FINDALL	17
FINDBYID	17
DELETEBYID	17
PATCHBYID & PUTBYID	17
ADDAPARTMENTEXPENSE	17
GETAPARTMENTEXPENSE	17
REMOVEAPARTMENTEXPENSE	17
APPARTAMENTO	18
ADDAPARTMENT	18

FINDALL	18
FINDBYID	18
DELETEBYID	18
PATCHBYID & PUTBYID	18
GETAPARTMENTDEBITS	18
ADDAPARTMENTMEMBER	19
API DOCUMENTATION	20
FRONT-END IMPLEMENTATION	22
LOGIN	22
REGISTER	22
APPARTAMENTI	23
CREAZIONE NUOVO APPARTAMENTO	24
MODIFICA APPARTAMENTO	24
ELIMINA APPARTAMENTO	25
PAGINA PRINCIPALE APPARTAMENTO	25
PAGINA SPESE APPARTAMENTO	26
MODALE AGGIUNTA SPESA	26
MODALE ELIMINA SPESA	27
MODALE MODIFICA SPESA	27
PAGINA ACCOUNT	28
MODALE ELIMINA ACCOUNT	28
TESTING	29
RISULTATI DEL TESTING	31
GITHUB REPOSITORY E INFORMAZIONI SUL DEPLOYMENT	33
ESEGUIRE IL SERVER SULLA PROPRIA MACCHINA	34

SCOPO DEL DOCUMENTO

Lo scopo del presente documento è quello di esporre le informazioni necessarie per lo sviluppo di una parte dell'applicazione web Co.Inqui.

Nel primo capitolo si parla dello User Flow, ovvero la traduzione attraverso un diagramma delle azioni che un utente può fare con l'applicazione e delle conseguenze di queste azioni.

Successivamente si tratta dello sviluppo effettivo dell'applicazione, analizzando come sono organizzati i file nel progetto, descrivendo le API implementate e le tecnologie usate per arrivare al prodotto attuale.

Un'attenta descrizione delle API è effettuata grazie al diagramma delle risorse e al diagramma di estrazione delle risorse, con il quale si individuano le risorse estratte a partire dal diagramma delle classi presente nel precedente deliverable, il D3.

Nel quarto capitolo si spiega come è stata effettuata la documentazione delle API tramite swagger.

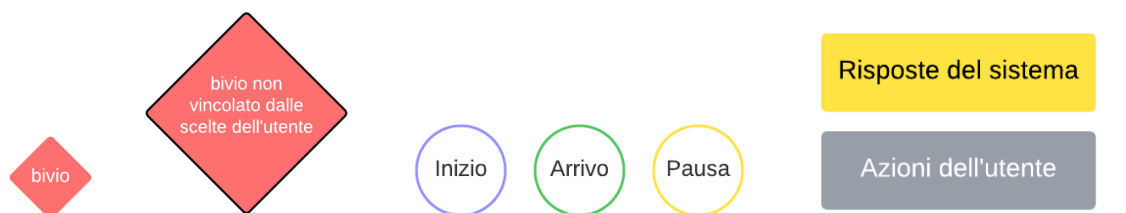
Infine, si parla di come sono stati scritti ed effettuati i test del codice e vengono mostrati i risultati di alcuni di essi.

USER FLOW

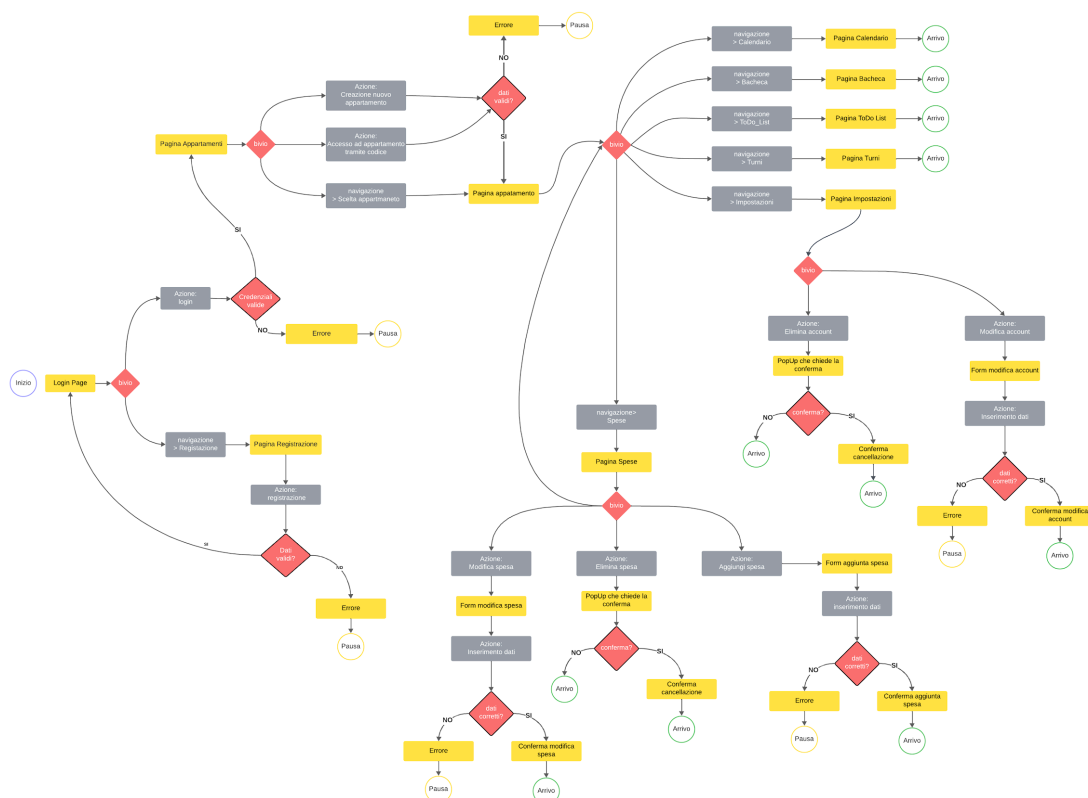
In questa prima parte del documento forniamo lo User Flow che abbiamo seguito per lo sviluppo di una parte della nostra applicazione.

In particolare abbiamo deciso di soffermarci su 3 funzionalità principali della nostra applicazione, ovvero l'autenticazione e la registrazione di un utente, l'accesso ad un appartamento e la creazione di un nuovo appartamento ed infine la pagina che si occupa della gestione delle spese e dei debiti tra coinquilini.

In seguito viene riportata una didascalia delle componenti utilizzate nel diagramma:



Con l'uso di questi componenti siamo giunti al seguente User Flow:



APPLICATION IMPLEMENTATION AND DOCUMENTATION

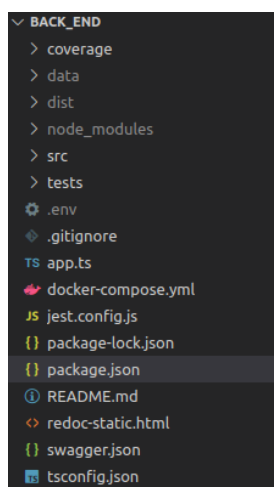
L'applicazione CO.Inqui è stata sviluppata usando il framework di React per la parte di front-end mentre è sono stati usati NodeJs e Express per la realizzazione della parte di back-end. Per il salvataggio dei dati abbiamo optato per un database non relazionale fornito come servizio da MongoDB, ovvero MongoDB Atlas.

Come si può vedere dallo UserFlow presente nella pagina precedente abbiamo implementato le seguenti funzionalità: la registrazione e il login degli utenti, la creazione degli appartamenti, la modifica degli appartamenti, l'eliminazione degli appartamenti, l'aggiunta di nuove spese, la modifica di spese già presenti nel database, l'eliminazione delle spese e il calcolo dei debiti tra gli utenti di uno stesso appartamento. Infine abbiamo implementato le funzionalità di logout, di modifica dell'account e di eliminazione di un account.

PROJECT STRUCTURE

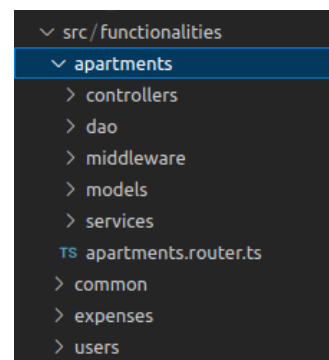
Per lo sviluppo dell'applicazione abbiamo optato per lo sviluppo di front-end e back-end come due moduli separati, quindi in questa sezione del documento offriamo una overview generale prima dell'una e poi dell'altra parte.

Partiamo dalla parte di back-end, in seguito una overview su come sono strutturati i files: Troviamo dalla parte superiore, in ordine:

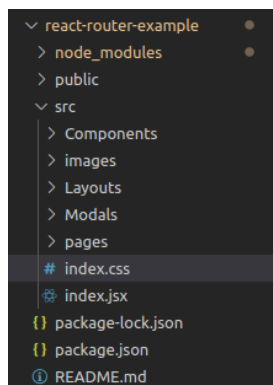


- coverage/*: cartella che contiene il report dei test
- data/*: cartella contenente i dati del database di test che viene usato con docker
- dist/*: folder contenente l'output della compilazione dei file di typescript
- node_modules/*: cartella contenente tutti i moduli node usati
- src/*: cartella contenente tutti i file typescript del back-end
- tests/*: cartella contenente i file di tests
- *.env* : file con le variabili d'ambiente
- *.gitignore* : file contenente i percorsi ignorati da git
- *app.ts*: file che rappresenta il punto di accesso del programma
- **.json* : file di configurazione di typescript e nodejs
- *swagger.json*: file che presenta la documentazione degli endpoints
- *jest.config.js*: file di configurazione di jest

Se approfondiamo l'analisi della cartella "src/" scopriamo altre quattro cartelle, una per gli appartamenti, una per gli utenti, una per le spese ed una per i componenti comuni, come la connessione al database. Le prime tre cartelle seguono tutte la stessa struttura, ovvero presentano un file che definisce i percorsi e diverse cartelle che rappresentano i vari livelli dell'API, quali middleware, controller, dao, models eservices. Questa scelta implementativa sarà spiegata in seguito in modo più approfondito nella sezione SVILUPPO API quindi in questo frangente viene solo indicata la struttura senza specificare le funzionalità.



Passiamo ora allo studio della parte di front-end della nostra applicazione, la quale presenta la seguente struttura:



- node_modules/*: cartella contenente tutti i moduli node usati
- public/*: folder che contiene files statici serviti direttamente al client
- src/*: cartella che contiene tutti i file di React che servono per la visualizzazione del front-end.
- src/components/*: cartella che contiene i componenti di react che possono essere usati in più pagine
- src/images/*: cartella che contiene le immagini usate nell'applicazione
- src/Layouts/*: cartella che contiene i file di layout dell'applicazione
- src/Modals/*: cartella contenente tutti i modali che appaiono nell'applicazione
- src/pages/*: questa è la cartella più grande in quanto contiene tutte le implementazioni delle varie pagine che l'utente può visitare durante l'utilizzo dell'applicazione.

PROJECT DEPENDENCIES

Per la parte di front-end i moduli di node utilizzati sono:

- *express*: framework che fornisce funzionalità per la creazione di RESTful API, e non solo
- *class-validator*: modulo che permette di validare i dati dell'istanza di una classe, lo abbiamo usato per il controllo dei parametri passati alle API
- *cookie-parser*: modulo che serve per fare il parsing dei cookies
- *cors*: modulo che permette all'applicazione di supportare il Cross-Origin Resource Sharing protocol
- *dotenv*: modulo che permette di sfruttare le variabili d'ambiente
- *mongoose*: modulo che permette di collegarsi ed interagire con un database mongoDB
- *shortid*: modulo che fornisce dei codici casuali, usato per assegnare gli id alle istanze nei database
- *jest*: modulo per svolgere i test delle API dell'applicazione
- *supertest*: modulo che permette di contattare un endpoint delle API durante i test

Passando alla parte di front-end i moduli usati sono:

- *axios*: modulo che permette di interagire con il back-end
- *tutti i moduli di react*: moduli di default che react usa per poter funzionare che vengono installati di default

MODELLI NEL DATABASE

Tutti i dati persistenti nell'applicazione sono salvati all'interno di un database, che come abbiamo visto è MongoDB, quindi abbiamo definito in partenza dei modelli sulla base delle definizioni presenti nel Deliverable 3. Per le funzionalità che abbiamo deciso di supportare nella nostra applicazione i modelli di cui abbiamo necessità sono 4 e verranno presentati di seguito.

MODELLO USER

```
//Schema definition for users
userSchema = new this.Schema({
  _id: String,
  salt: String,
  email: String,
  password: String,
  username: {type: String, unique: true},
  firstName: String,
  lastName: String,
  birthDate: Date,
  telephone: String
}, {id: false});
```

Il modello utente si rende necessario in quanto la registrazione di un utente al sistema è obbligatoria per poter usufruire delle sue funzionalità.

I dati principali che vogliamo salvare dell'utente sono anagrafici e di contatto, tra i primi rientrano il nome, il cognome e la data di nascita mentre tra i secondi l'indirizzo mail e il telefono.

Inoltre sono presenti altri campi relativi solo all'applicazione come lo username e l'id, che servono

per riconoscere in modo univoco l'utente e i campi password e salt che sono necessari per poter provvedere alla sua autenticazione. Il campo salt è necessario in quanto, per motivi di sicurezza non è possibile salvare in chiaro la password in chiaro, quindi usiamo un sistema di hashing + salt per mantenere al sicuro i nostri utenti.

Questo che segue è un esempio di un'istanza di utente salvata sul nostro database.

```
_id: "nRIeHSWJD"
salt: "FsWQJlN1KZ"
email: "mail"
password: "bce4417d811e350..."
username: "Cardo"
firstName: "Riccardo"
lastName: "Miolato"
birthDate: 2003-08-04T00:00:00Z
telephone: "1234567899"
__v: 0
```

MODELLO SESSION

Per garantire agli utenti di non dover inserire le loro credenziali ad ogni accesso abbiamo deciso di implementare le sessioni che servono a mantenere attivo l'accesso di un utente all'applicazione per un periodo di tempo di un'ora. Per garantire questa funzionalità salviamo due dati principali, ovvero l'id della sessione e lo username dell'utente al quale è associata, oltre all'id dell'istanza e alla data di scadenza della sessione. Grazie alle sessioni possiamo verificare la validità delle richieste che vengono fatte alle API che richiedono l'autenticazione dell'utente.

Ecco un esempio del modello e di un'istanza nel database:

```
sessionSchema = new this.Schema({
  _id: String,
  sessionId: String,
  userId: String,
  expireDate: Date
}, {id: false});
```

```
_id: "A8XgGv_bI"
sessionId: "9XcEOhhr4J"
userId: "username"
expireDate: 2024-05-02T15:00:00Z
__v: 0
```


MODELLO APARTMENT

Il modello "apartment" copre un ruolo fondamentale all'interno del contesto della nostra applicazione, in quanto è il componente sulla quale tutto il resto si fonda. Abbiamo identificato degli attributi per la descrizione di un appartamento, quali l'id, il nome e la descrizione, e degli attributi che servono come chiavi esterne che evidenziano le relazioni con oggetti di altre collections, quali l'admin di un appartamento, i suoi utenti e le spese relative all'appartamento. Le chiavi esterne relative agli utenti sfruttano lo username come chiave esterna mentre per le spese usiamo l'id come chiave esterna.

Ecco un esempio di come appare un appartamento salvato nel database:

```
_id: "7THzuAq0u"
name : "appartamento2"
description : "descrizione"
admin : "username"
▼ users : Array (1)
  0: "username"
▼ expenses : Array (2)
  0: "mOJ3mmFwU"
  1: "QjhsatScD"
__v : 0
```

```
apartmentSchema = new this.Schema({
  _id: String,
  name: String,
  description: String,
  admin: String,
  users: [String],
  expenses: [String]
}, {id: false});
```

MODELLO EXPENSE

L'ultimo modello che abbiamo deciso di implementare è quello delle spese, in quanto la nostra applicazione ha lo scopo principale quello della gestione delle finanze di un piccolo gruppo di persone. I dati che abbiamo ritenuto fondamentali da salvare in questo caso non sono tanti, ma sono significativi e sono l'importo della spesa, chi l'ha effettuata (creditore), chi è coinvolto (debitori) e la data nel quale è stata effettuata. Inoltre per poter garantire della personalizzazione abbiamo inserito i campi nome e descrizione in modo che gli utenti della nostra applicazione possano facilmente comprendere la natura delle spese svolte. Anche in questo caso, come per appartamento, abbiamo dovuto inserire delle chiavi esterne per poter risalire a tutti gli utenti coinvolti, sia il creditore che i debitori.

Sulla sinistra, al di sotto della definizione dello schema, si può vedere un esempio di spesa salvata sul database.

```
expenseSchema = new this.Schema({
  _id: String,
  name: String,
  description: String,
  import: Number,
  date: Date,
  creditor: String,
  debtors: [String]
});
```

```
_id: "mOJ3mmFwU"
name : "Spesa1"
description : "Spesa desc 1"
import : 20
date : 2024-05-02T00:00:00.00
creditor : "username"
debtors : Array (1)
  0: "username"
__v : 0
```

PROJECT APIs

In questa parte del documento vengono descritte le varie APIs implementate. I diagrammi usati per la descrizione delle API sono 2, ovvero il diagramma di estrazione delle risorse e il diagramma delle risorse. Entrambi sono sviluppati partendo dal diagramma delle classi del D3 concentrandosi esclusivamente sulle funzionalità effettivamente implementate.

DIAGRAMMA DI ESTRAZIONE DELLE RISORSE

Questo diagramma mostra come vengono estratte le varie risorse sviluppate nel sistema a partire dal class diagram. Le principali risorse individuate sono 3, ovvero gli utenti, gli appartamenti, le spese.

Per ogni risorsa abbiamo deciso di implementare i metodi principali delle API, ovvero:

- Estrazione di tutte le risorse della stessa tipologia (GET)
- Estrazione di solo una risorsa cercando in base all'id (GET)
- Inserimento di una nuova risorsa (POST)
- Eliminazione di una risorsa basandosi sull'id (DELETE)
- Aggiornamento di una risorsa in base all'id (PUT e PATCH)

Queste operazioni forniscono una base sulla quale fornire le funzionalità dell'applicazione, ma per semplicità sono state studiate delle API più specifiche per estrarre/aggiornare punti specifici di una risorsa. Per esempio, visto che la nostra applicazione si basa su tutte quelle risorse che sono associate ad un appartamento, abbiamo deciso di creare degli endpoint per:

- Estrazione di tutti i membri di un appartamento (GET)
- Estrazione di tutte le spese di un appartamento (GET)
- Inserimento di una spesa collegata ad un appartamento (POST)
- Inserimento di un nuovo membro in un appartamento (POST)
- Abbandono dell'appartamento da parte di un utente (DELETE)
- Eliminazione di una spesa di un appartamento (DELETE)
- Estrazione di tutti i saldi degli utenti (GET)

Abbiamo deciso di sviluppare degli endpoint per l'autenticazione degli utenti tra cui rientrano:

- Azione di login (POST)
- Azione di logout (POST)
- Azione di registrazione (POST)

Inoltre nel diagramma, a fianco di ogni risorsa, viene specificato dove questa ha rilevanza, se nel front-end, nel caso delle richieste GET, o nel back-end, nel caso di tutte le altre richieste. Da notare che per il passaggio dei parametri viene usato *body* per indicare che vengono passati tutti o quasi gli attributi della risorsa.

Nella pagina seguente vi è il diagramma di estrazione delle risorse.

DIAGRAMMA DI ESTRAZIONE DELLE RISORSE

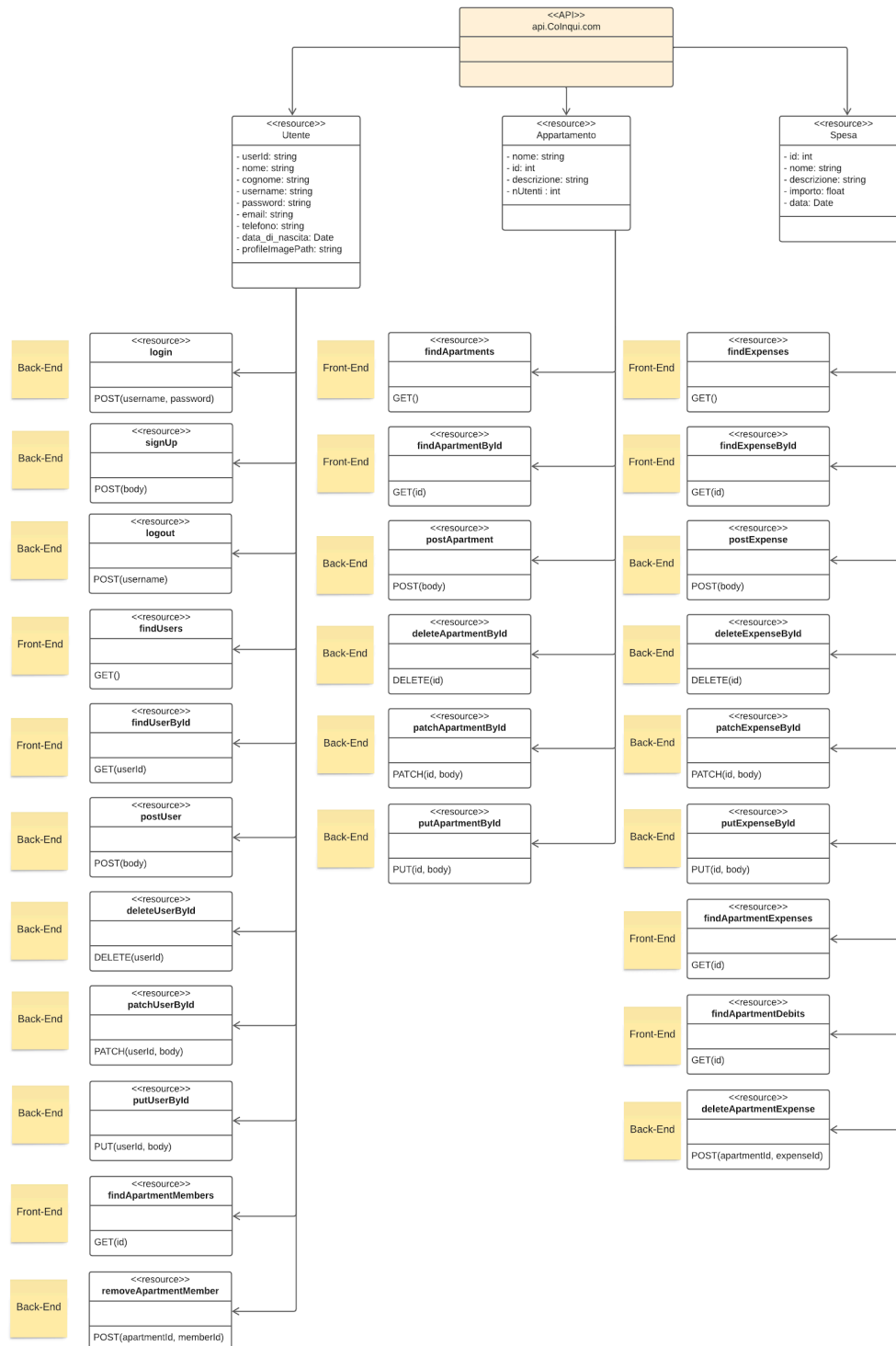


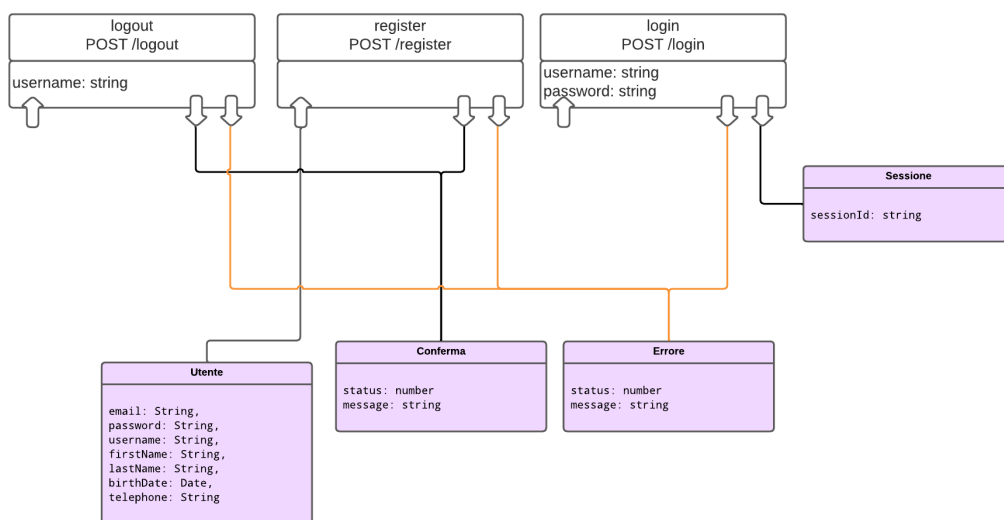
DIAGRAMMA DELLE RISORSE

Nel seguente diagramma delle risorse rappresentiamo le API sviluppate nel progetto in modo più approfondito rispetto al precedente diagramma di estrazione delle risorse.

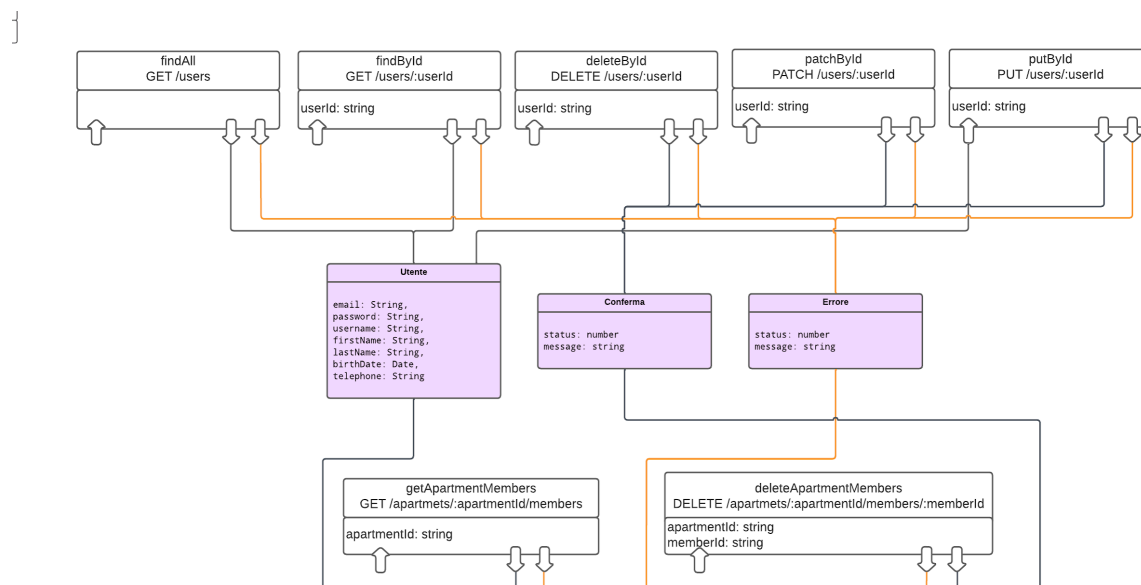
In questo diagramma sono specificati gli input e gli output di tutti gli endpoint della nostra API. Gli endpoint possono ritornare errori, nel caso ci fossero dei problemi, oppure potrebbero inviare le risorse richieste o un messaggio di conferma di avvenuta operazione.

Per comodità di lettura dividiamo il diagramma in vari punti.

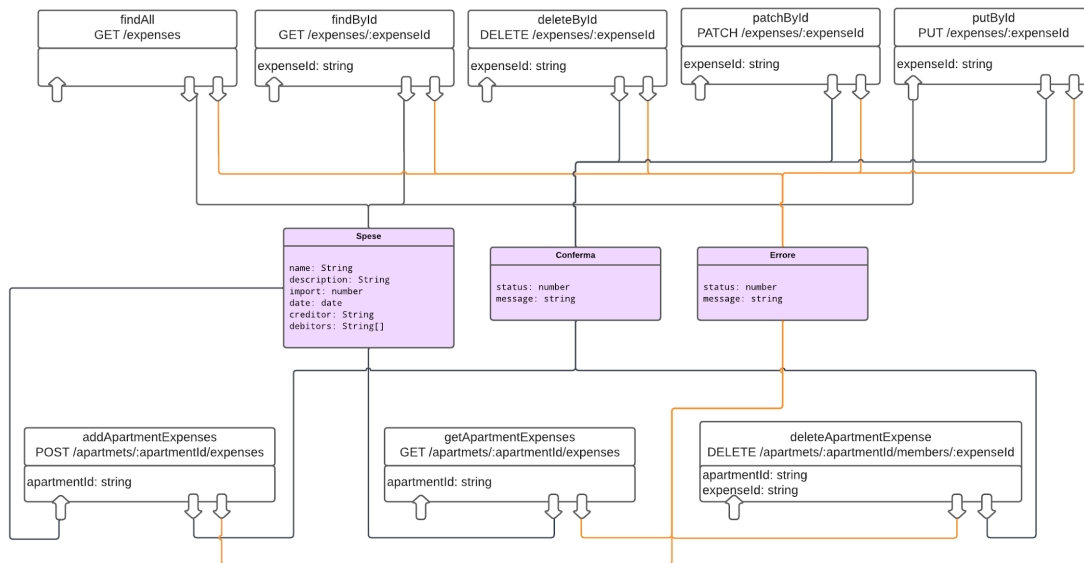
ENDPOINTS PER AUTENTICAZIONE E REGISTRAZIONE



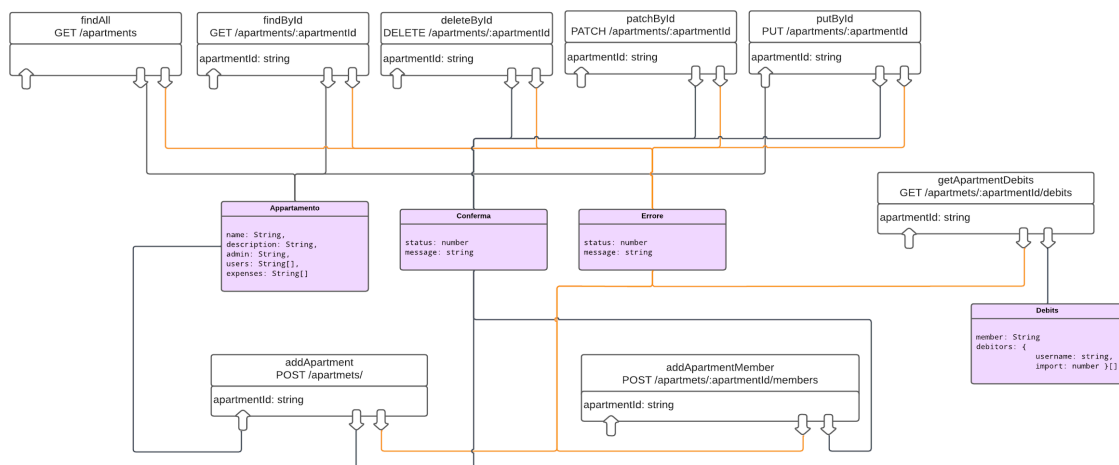
ENDPOINTS PER GLI UTENTI



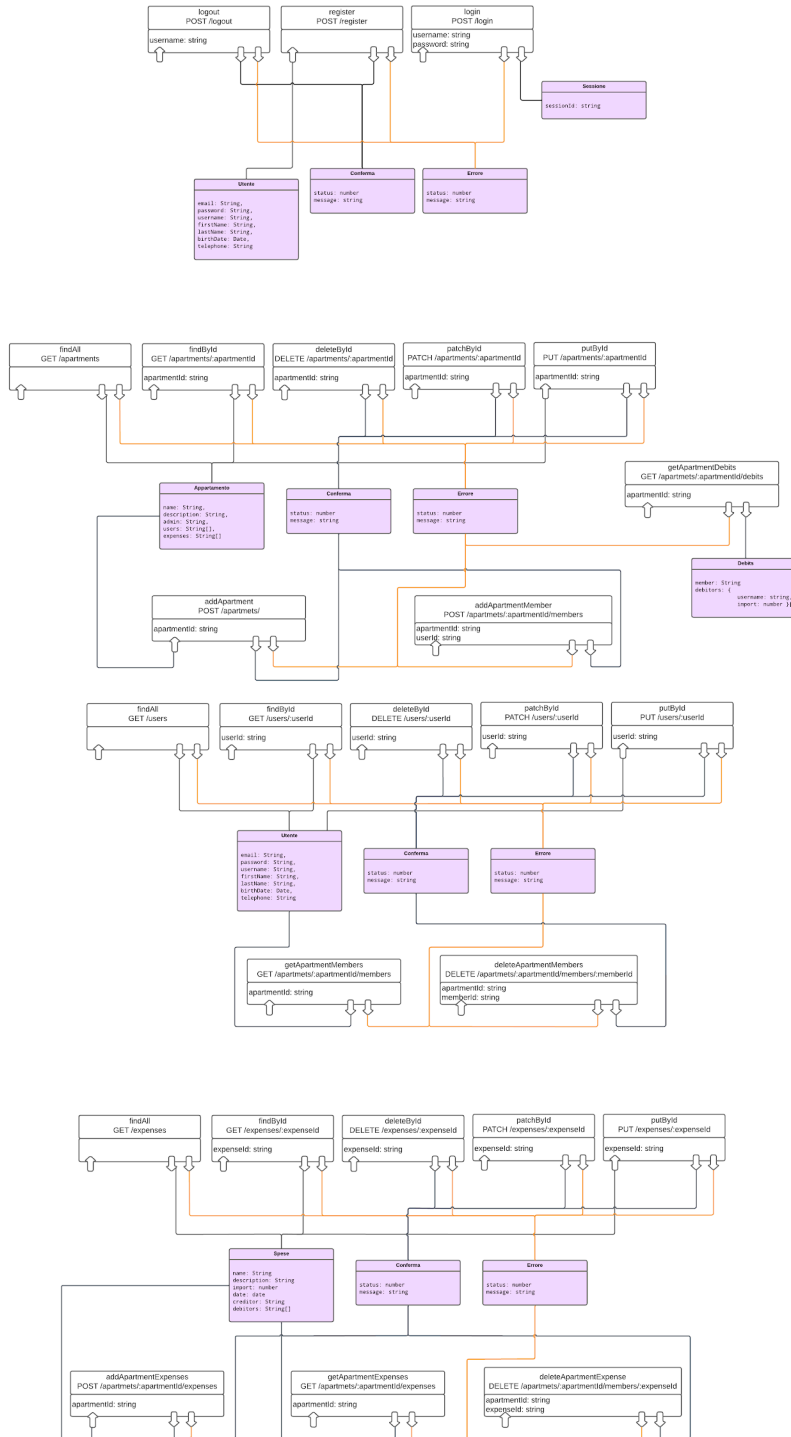
ENDPOINTS PER LE SPESE



ENDPOINTS PER GLI APPARTAMENTI



SCHEMA COMPLETO



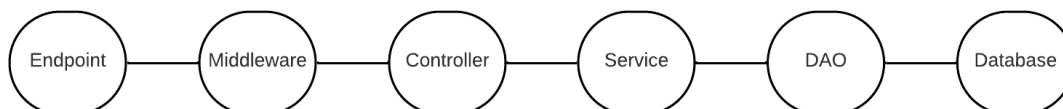
SVILUPPO API

In questa sezione del documento descriveremo lo scopo e il funzionamento dei vari endpoint dell'API sviluppati.

Per lo sviluppo della nostra API abbiamo deciso di seguire uno schema ben strutturato che gestisce il flusso dei dati dalla richiesta dell'utente alla risposta del sistema.

In particolare abbiamo implementato i seguenti moduli:

- **Middleware:** serve per effettuare dei controlli preliminari sui parametri passati dall'utente.
- **Controller:** responsabile della gestione delle richieste in arrivo e della produzione di una risposta appropriata.
- **Service:** I services incapsulano la business logic della nostra API, astraggono i dettagli della manipolazione dei dati, della validazione e di altre operazioni.
- **DAO:** Il Data Access Object è il modulo responsabile di fornire i servizi esposti dal modulo service attraverso la comunicazione diretta con il database.



A seguire la descrizione delle API divise per risorsa di interesse.

AUTENTICAZIONE E REGISTRAZIONE

REGISTER

L'endpoint register ha lo scopo di gestire la registrazione di un nuovo utente.

Riceve dall'utente un'istanza di utente che per prima cosa, viene passata al middleware, la funzione `validateBodyFields` che controlla che tutti i campi siano valutati.

Dopo questo primo controllo se ne fa un altro per verificare l'unicità dello username attraverso la funzionalità `validateUsernameDoesntExists` e se anche questo controllo viene superato allora il nuovo utente viene registrato con successo.

LOGIN

L'endpoint login valida le credenziali di un utente che prova ad accedere al sistema. Nel caso queste credenziali risultassero corrette, attraverso la funzione del middleware `validateLoginData` l'API restituisce all'utente il codice di una sessione la quale ha la durata di un'ora che permette all'utente di effettuare tutte le operazioni all'interno dell'applicazione.

LOGOUT

Questo endpoint ha lo scopo di invalidare la sessione attiva dell'utente.

Come prima cosa il parametro username di input viene passato per il middleware che controlla che effettivamente sia valutato con la funzione `validateLogoutData` e in seguito effettua l'accesso al database per eliminare la sessione relativa allo username dell'utente.

UTENTE

Da qui in poi tutti gli endpoint per funzionare hanno il bisogno di assicurarsi che l'utente sia autenticato, quindi prima di ogni operazione viene chiamata la funzione del middleware *"validateUserSession"*.

FINDALL

Endpoint che restituisce tutte le istanze di utente presenti all'interno del database.
Si tratta di una funzione che non viene mai usata direttamente dal front-end, ma risulta molto utile per la gestione di operazioni nel back-end.

FINDBYID

Endpoint che ricerca nel database l'utente con lo *userId* passato come parametro.
Attraverso la funzione del middleware *"validateUserId"* si verifica in primo luogo l'esistenza dell'utente cercato, se non esiste viene ritornato un codice di errore 404, altrimenti viene ritornata l'istanza dell'utente ricercato.

DELETEBYID

Endpoint che ha la funzione di eliminare un'istanza di un utente dal database.
Come nell'endpoint descritto sopra si verifica la presenza dell'utente nel database tramite la funzione del middleware *"validateUserId"* e nel caso non ci fossero problemi cerca ed elimina l'istanza dal database.

PATCHBYID & PUTBYID

Endpoint che viene utilizzato per apportare delle modifiche ad un utente identificato da un *id*.
Sfrutta sempre la funzione del middleware per il controllo della validità dell'*id* e se tutto risulta corretto va ad effettuare l'aggiornamento dell'istanza dell'utente nel database.
La differenza tra la richiesta *PATCH* e *PUT* è che la prima non richiede l'intera risorsa dell'utente per modificare uno o più attributi mentre la seconda sì.

GETAPARTMENTMEMBERS

Endpoint che ha lo scopo di restituire tutti i membri di un appartamento. Preso in input l'*id* dell'appartamento si sfrutta il middleware, con la funzione *"validateApartmentId"*, per validare l'esistenza dell'appartamento.
L'istanza di ogni appartamento possiede dei riferimenti agli utenti membri, possiamo vederli come chiavi esterne, e grazie a questi possiamo recuperare nel database tutti gli utenti a cui questi fanno riferimento.

DELETEAPARTMENTMEMBER

Endpoint che ha lo scopo di rimuovere un utente dall'appartamento di riferimento. In questo caso gli *id* di input sono 2, ovvero l'*id* dell'appartamento e quello dell'utente. Grazie alle funzioni del middleware controllo l'esistenza dell'appartamento nel database e del *memberId* all'interno dello specifico campo dell'appartamento.
Da notare che questo endpoint non rimuove l'utente dal database, ma elimina solo il collegamento tra l'appartamento e l'utente.

SPESE

FINDALL

Endpoint che restituisce tutte le istanze delle spese presenti all'interno del database.

Si tratta di una funzione che non viene mai usata direttamente dal front-end, ma risulta molto utile per la gestione di operazioni nel back-end.

FINDBYID

Endpoint che ricerca nel database la spesa con l'expenseId passato come parametro.

Attraverso la funzione del middleware "*validateExpenseId*" si verifica in primo luogo l'esistenza della spesa ricercata, se non esiste viene ritornato un codice di errore 404, altrimenti viene ritornata l'istanza della spesa.

DELETEBYID

Endpoint che ha la funzione di eliminare un'istanza di una spesa dal database.

Come nell'endpoint descritto sopra si verifica la presenza dell'utente nel database tramite la funzione del middleware "*validateExpenseId*" e nel caso non ci fossero problemi cerca ed elimina l'istanza della spesa dal database.

PATCHBYID & PUTBYID

Endpoint che viene utilizzato per apportare delle modifiche ad una spesa identificata da un id. Sfrutta sempre la funzione del middleware per il controllo della validità dell'id e se tutto risulta corretto va ad effettuare l'aggiornamento dell'istanza della spesa nel database.

La differenza tra la richiesta PATCH e PUT è che la prima non richiede l'intera risorsa della spesa per modificare uno o più attributi mentre la seconda sì.

ADDAPARTMENTEXPENSE

Endpoint che ha il compito di aggiungere una nuova spesa ad un appartamento specificato.

L'input è un'istanza di una spesa, la quale viene aggiunta tramite una funzione interna, non visibile dall'esterno, che rispecchia la POST con cui si registra l'utente. Questa funzione ritorna l'id della nuova spesa il quale viene inserito all'interno dell'array delle spese dell'appartamento come chiave esterna.

GETAPARTMENTEXPENSE

Endpoint che ha lo scopo di fornire agli utenti di un appartamento tutte le spese che li riguardano. Prende in input l'id dell'appartamento, il quale viene valutato dal middleware attraverso la funzione "*validateApartmentId*", e grazie alle chiavi delle spese presenti nell'istanza dell'appartamento può effettuare una ricerca nel database di tutte le spese referenziate. L'operazione avviene tramite il metodo GET di HTTP.

REMOVEAPARTMENTEXPENSE

Endpoint che rimuove una spesa da un appartamento. A differenza di quello che rimuove l'utente non abbiamo interesse a mantenere salvata l'istanza della spesa non collegata ad un appartamento, quindi non eliminiamo solo il riferimento ma anche l'oggetto stesso.

Anche in questo caso si usa il middleware per valutare l'id dell'appartamento, ma in più si aggiunge la funzione per il controllo dell'id della spesa, ovvero "*validateApartmentExpenseId*". L'endpoint è acceduto tramite metodo DELETE.

APPARTAMENTO

ADDAPARTMENT

Endpoint che gestisce la creazione di un nuovo appartamento.

Il middleware in questo caso, tramite la funzione "*validateApartmentBody*", verifica la completezza dell'istanza passata come parametro e nel caso sia tutto corretto allora conferma la registrazione di un nuovo appartamento.

FINDALL

Endpoint che restituisce tutti gli appartamenti di cui l'utente è partecipe.

L'utente in questione viene riconosciuto grazie alla sessione attiva. In questo caso la funzione "findall" viene sfruttata anche dal front-end.

FINDBYID

Endpoint che dato in input l'id di un appartamento ne restituisce la relativa istanza salvata nel database. Il compito del middleware in questo caso è quello di validare l'esistenza dell'appartamento che possiede l'id uguale a quello passato come input tramite la funzione "*validateApartmentId*".

DELETEBYID

Endpoint che si occupa dell'eliminazione di un appartamento dal database.

Per eseguire questa operazione bisogna prima stabilire che l'utente che la esegue corrisponda con l'admin dell'appartamento, quindi questo controllo rientra nei compiti del middleware che lo svolge con la funzione "*validateIsAdmin*".

Se colui che svolge la funzione viene riconosciuto come admin dell'appartamento allora l'operazione viene validata e l'appartamento eliminato.

PATCHBYID & PUTBYID

Endpoint che viene utilizzato per apportare delle modifiche ad un appartamento identificato da un id.

Sfrutta due funzionalità del middleware ovvero "*validateIsAdmin*", come nell'endpoint precedente chi può apportare modifiche deve essere l'admin, e "*validateApartmentId*" che controlla l'esistenza di un appartamento con id uguale a quello passato in input.

La differenza tra la richiesta PATCH e PUT è che la prima non richiede l'intera risorsa della spesa per modificare uno o più attributi mentre la seconda sì.

GETAPARTMENTDEBITS

Endpoint che ritorna tutte le informazioni rispetto ai debiti degli utenti di un appartamento.

Una volta validato l'id dell'appartamento attraverso il middleware con la funzione "*validateApartmentId*" la funzione dell'endpoint è quella di recuperare le spese dell'appartamento e calcolare per ogni utente quanti soldi gli sono dovuti da ogni altro membro.

ADDAPARTMENTMEMBER

Questo endpoint ha la funzione di allegare nuovi membri ad un appartamento.

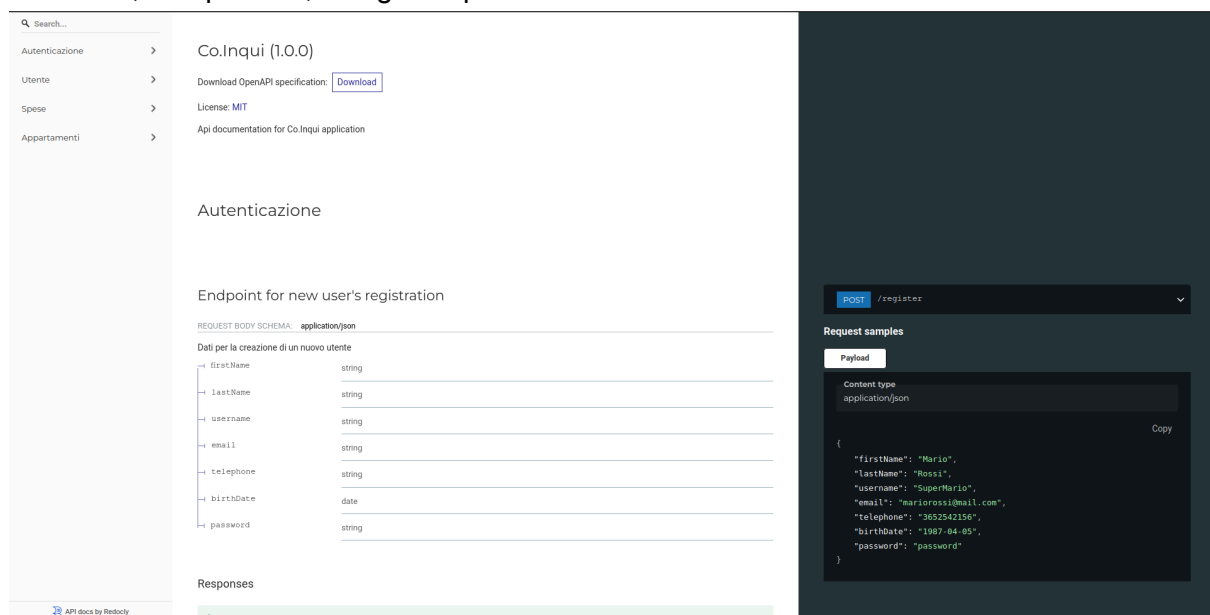
A differenza di “addApartmentExpense” che gestisce anche il salvataggio di una nuova spesa, questo endpoint gestisce solo l’aggiunta di un riferimento ad un utente già esistente. L’utente che vuole aggiungersi all’appartamento viene riconosciuto attraverso la sessione, che viene usata per estrarre l’id dell’utente da linkare all’appartamento.

API DOCUMENTATION

Gli endpoints dell'applicazione Co.Inqui sono documentati con il modulo di NodeJs Swagger, il quale fornisce una visualizzazione degli endpoints chiara e intuitiva anche per i non addetti ai lavori grazie all'utilizzo dello standard OpenAPI.

Per poter visualizzare la documentazione del progetto bisogna, una volta scaricati i file dalla repository del back-end, accedere al file chiamato *“redoc-static.html”*

Una volta aperto il link della documentazione ci si trova di fronte alla seguente schermata, nella quale sulla sinistra si trovano le categorie degli endpoints, mentre nella parte centrale si trovano, uno per uno, tutti gli endpoints documentati.



Co.Inqui (1.0.0)

Download OpenAPI specification: [Download](#)

License: MIT

Api documentation for Co.Inqui application

Autenticazione

Endpoint for new user's registration

REQUEST BODY SCHEMA: application/json

Dati per la creazione di un nuovo utente

firstName	string
lastName	string
username	string
email	string
telephone	string
birthDate	date
password	string

Responses

Request samples

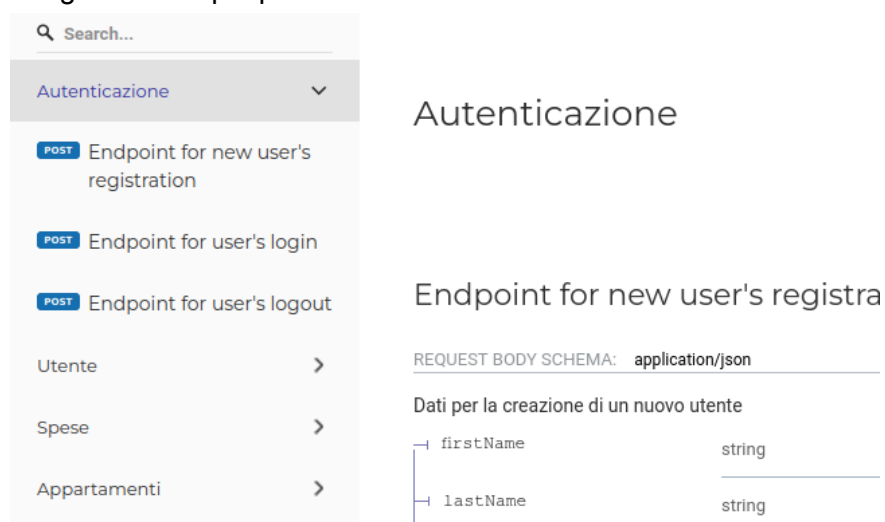
POST /register

Payload

Content type: application/json

```
{
  "firstName": "Mario",
  "lastName": "Rossi",
  "username": "SuperMario",
  "email": "mariorossi@mail.com",
  "telephone": "3652542156",
  "birthDate": "1987-04-05",
  "password": "password"
}
```

Per cercare un end-point, senza scorrere tutta la pagina, è sufficiente espandere il menu sulla sinistra relativo alla categoria desiderata e scegliere l'endpoint che si vuole visualizzare. In figura l'esempio per *“Autenticazione”*.



Autenticazione

Endpoint for new user's registration

Endpoint for user's login

Endpoint for user's logout

Utente

Spese

Appartamenti

Autenticazione

Endpoint for new user's registra

REQUEST BODY SCHEMA: application/json

Dati per la creazione di un nuovo utente

firstName	string
lastName	string

Gli endpoint che si possono consultare sono di 5 tipologie diverse, ovvero:

- GET: tipo di endpoint che restituisce dei dati
- POST: tipo di endpoint che carica dei dati
- PUT e PATCH: tipo di endpoint che serve per apportare modifiche ai dati
- DELETE: tipo di endpoint utilizzato per la rimozione dei dati

Una volta scelto un endpoint questo si presenta nel seguente modo:

Endpoint for new user's registration

REQUEST BODY SCHEMA: application/json

Dati per la creazione di un nuovo utente

firstName	string
lastName	string
username	string
email	string
telephone	string
birthDate	date
password	string

Responses

> 201

OK

— 400

The format of the user is not correct

— 409

Username already in use

POST /register

Request samples

Payload

Content type
application/json

Copy

```
{
  "firstName": "Mario",
  "lastName": "Rossi",
  "username": "SuperMario",
  "email": "matorossi@mail.com",
  "telephone": "3652542156",
  "birthDate": "1987-04-05",
  "password": "password"
}
```

Sulla parte sinistra viene evidenziato il formato del payload che l'endpoint si aspetta di ricevere, mentre sulla parte sinistra si trova il percorso relativo, con il quale si accede all'endpoint, e un esempio di payload con dei dati reali.

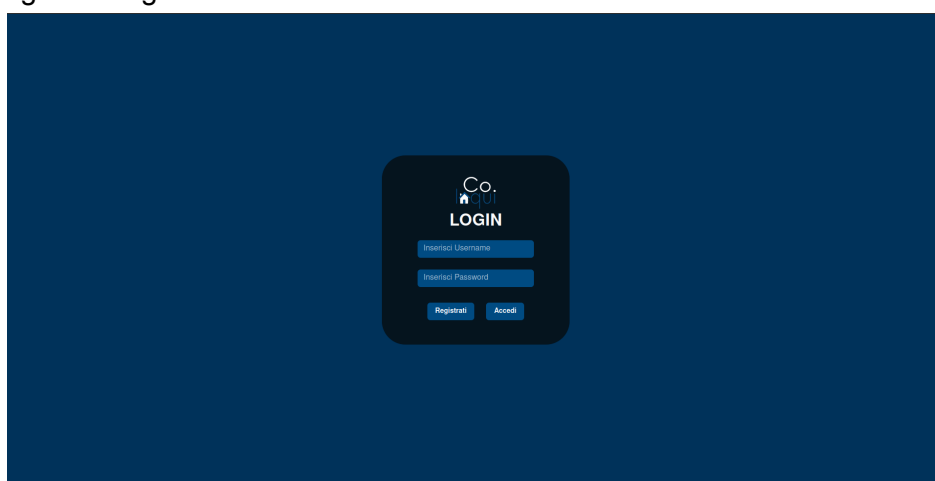
Nella parte inferiore si trovano i vari codici HTTP di stato che vengono ritornati dall'endpoint, in questo caso 201 se tutto va come previsto, altrimenti 400 o 409 se si presentano degli errori nell'operazione.

FRONT-END IMPLEMENTATION

In questa sezione vengono spiegate accuratamente le varie schermate che si possono trovare nel front-end e le azioni che si possono compiere interagendo con esse.

LOGIN

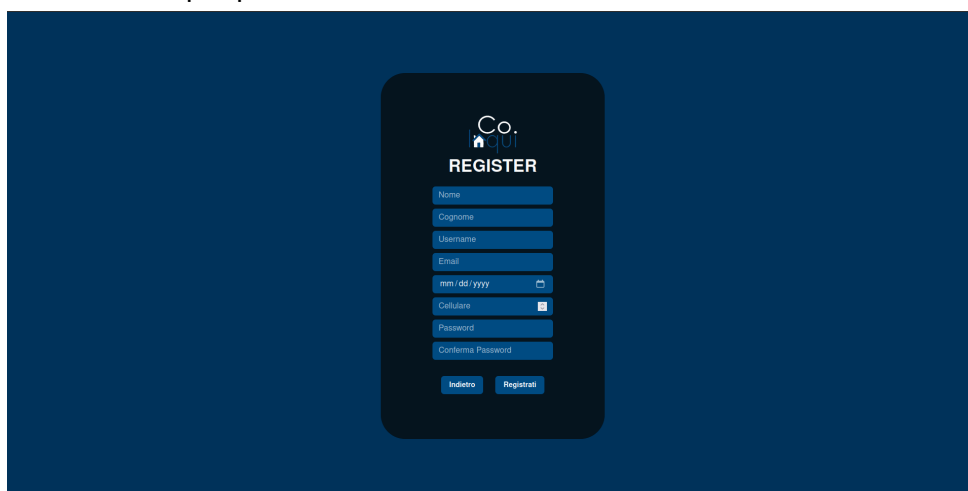
La pagina di Login è la schermata più importante e la prima con cui si viene in contatto in quanto se non si è loggati non si può accedere a tutte le altre funzionalità. E' possibile provare ad effettuare l'accesso tramite il pulsante "Accedi" che nel caso in cui i dati di accesso fossero sbagliati farà comparire un messaggio di errore. In caso non si avesse un account è possibile procedere alla registrazione tramite l'apposito pulsante "Registrati" che aprirà la pagina di registrazione.



REGISTER

La schermata di registrazione permette ai nuovi utenti di iscriversi inserendo i dati richiesti. Se non si vuole registrare un nuovo utente tramite il pulsante "Indietro" è possibile tornare alla pagina di login.

Tramite il pulsante "Registrati" è possibile provare a registrarsi e in caso di errore comparirà un messaggio rosso invitandoci a ricontrollare i dati inseriti, mentre se la registrazione avviene con successo l'utente viene portato alla pagina di login dove potrà inserire le proprie credenziali di accesso per poter usufruire del servizio.



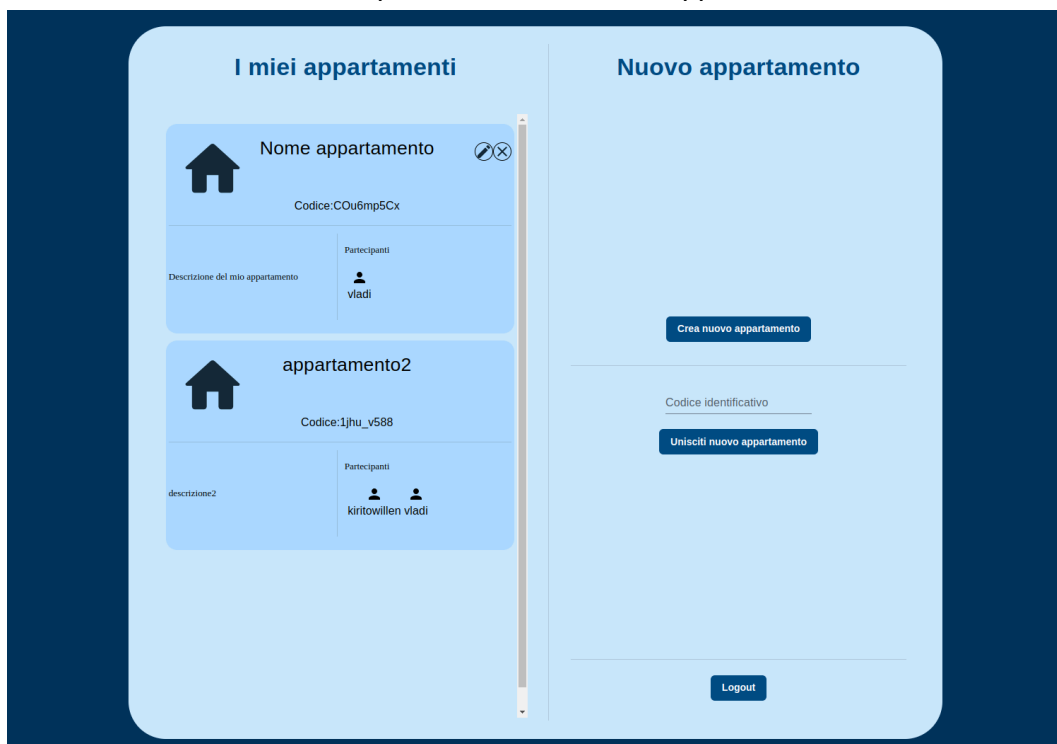
APPARTAMENTI

Questa è la pagina principale dell'applicazione ed è divisa in due: una parte con la lista di tutti gli appartamenti di cui l'utente fa già parte ed una parte che permette di creare nuovi appartamenti oppure di accedere ad uno già esistente tramite il codice identificativo.

Nella sezione "Nuovo appartamento" è possibile sia unirsi ad un appartamento tramite il suo codice univoco, inserendolo nel campo di input e usando l'apposito pulsante "Unisciti nuovo appartamento", sia creare un nuovo appartamento tramite l'apposito pulsante "Crea nuovo appartamento".

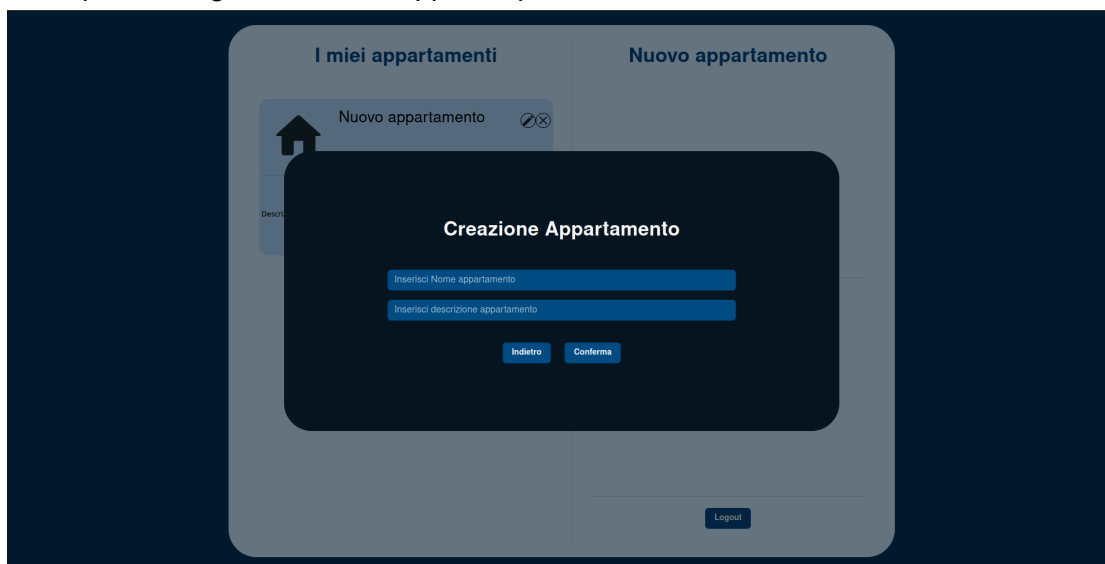
Nella sezione "I miei appartamenti" è possibile vedere la lista degli appartamenti a cui si ha accesso. Per poter accedere all'interno della sezione dell'applicazione relativa all'appartamento è sufficiente cliccare su quello che si vuole visitare. L'utente che crea un appartamento è identificato come l'admin dell'appartamento e questo gli consente di poter svolgere delle funzioni ulteriori rispetto a dei semplici coinquilini, tramite i due pulsanti in alto a destra può modificare un appartamento oppure eliminarlo direttamente.

E' inoltre possibile utilizzare il pulsante "Logout" per uscire dal proprio account in caso si voglia accedere ad un altro o semplicemente uscire dall'applicazione.



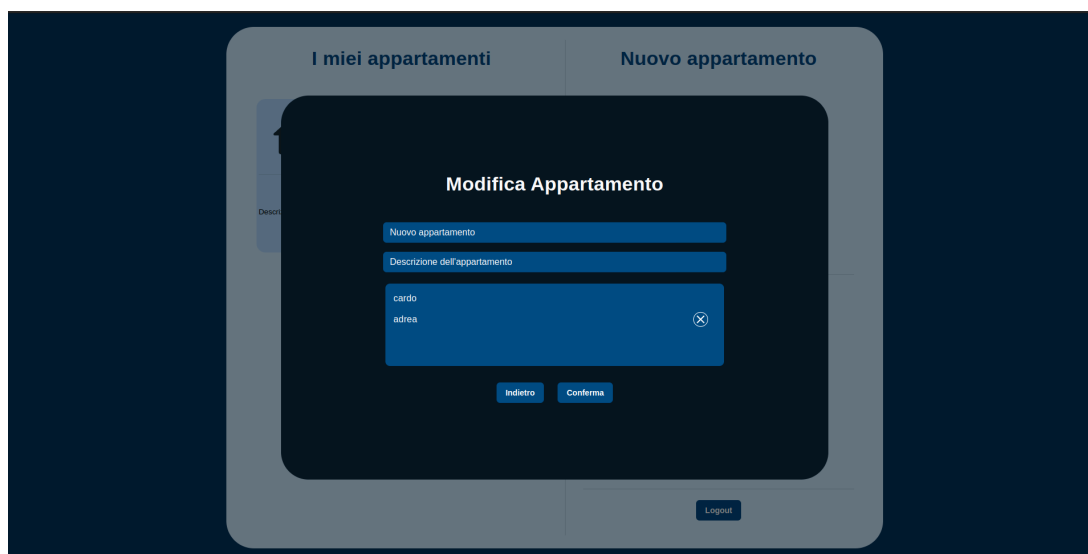
CREAZIONE NUOVO APPARTAMENTO

Premendo sul pulsante “Crea nuovo appartamento” viene aperto un modale che richiede l’inserimento di un nome e di una descrizione per creare l’appartamento dopodichè si può creare l’appartamento tramite l’apposito pulsante “Conferma” oppure tornare indietro in caso si sia nel posto sbagliato tramite l’apposito pulsante “Indietro”.



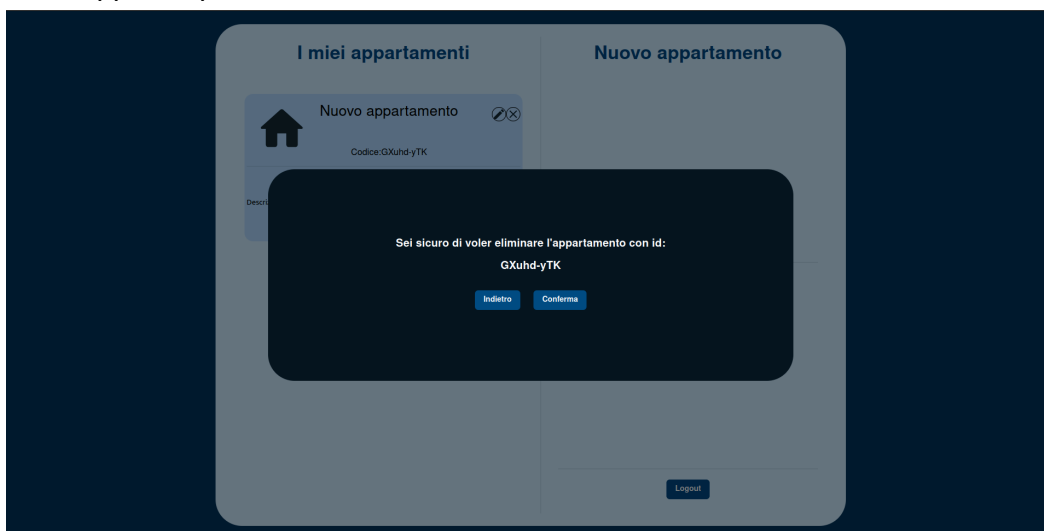
MODIFICA APPARTAMENTO

Se si è l'admin di un appartamento si può accedere alla pagina di modifica dove è possibile modificare le informazioni come il nome o la descrizione dell'appartamento e gestire gli utenti presenti, avendo la possibilità di espellerne uno tramite la 'X' presente sulla destra. Le modifiche svolte vengono salvate se si clicca sul pulsante “Conferma”, altrimenti non si fosse convinti delle modifiche effettuate è possibile annullare l'operazione tramite il pulsante “Indietro”.



ELIMINA APPARTAMENTO

L'admin di un appartamento può accedere al modale di eliminazione dove gli viene chiesto se è sicuro di voler eliminare l'appartamento identificato dal suo particolare id e dove può confermare le proprie intenzioni tramite il pulsante "Conferma" oppure annullare l'operazione attraverso l'apposito pulsante "Indietro".

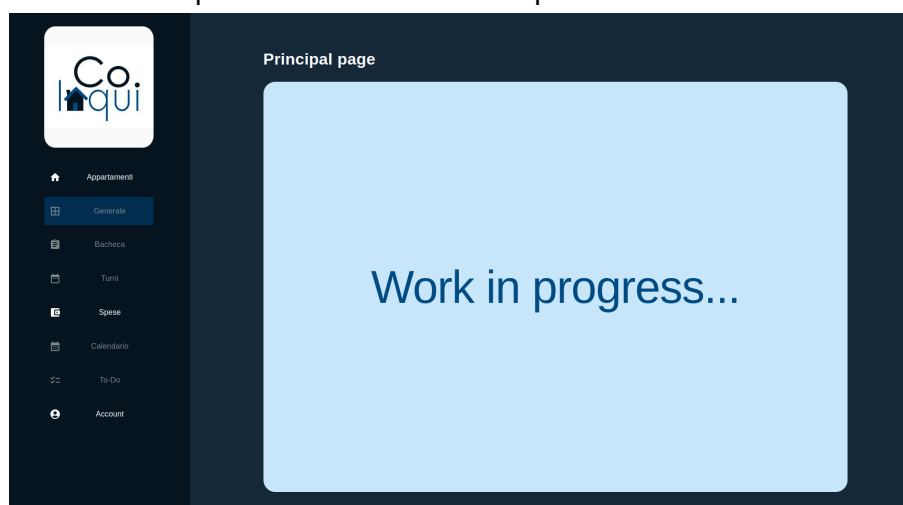


PAGINA PRINCIPALE APPARTAMENTO

Quando l'utente entra in un appartamento viene portato in questa schermata, dove sulla sinistra appare la barra di navigazione che contiene le varie sezioni dell'applicazione, mentre sulla destra contiene le informazioni relative alla pagina cercata.

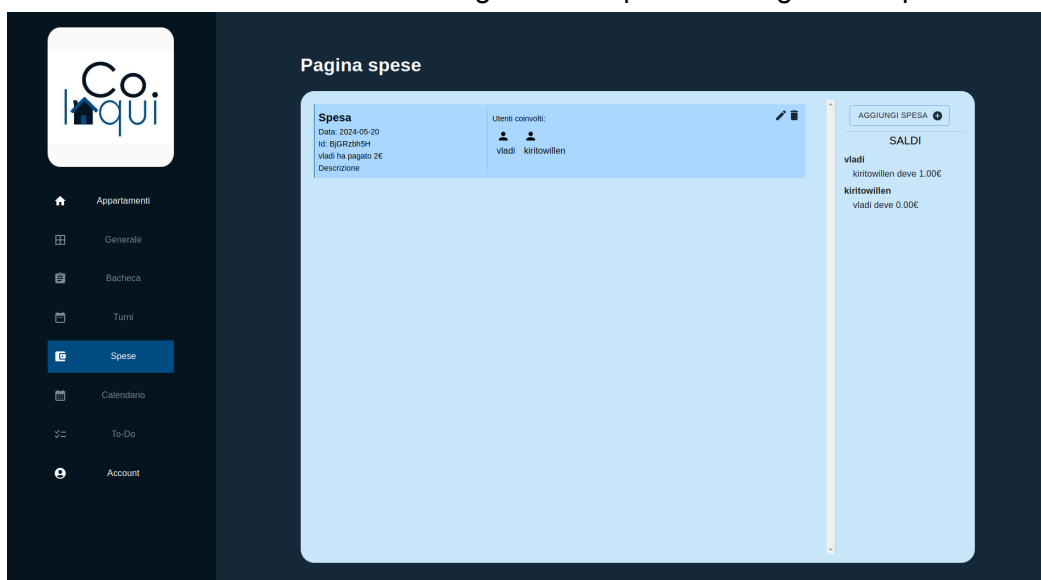
Per tornare alla lista degli appartamenti è sufficiente premere il pulsante "Appartamenti".

Poiché non tutte le pagine navigabili sono state implementate, abbiamo deciso di indicare queste pagine attraverso un placeholder "Work in progress...". Le pagine implementate si distinguono dalle altre in quanto nella barra laterale presentano una colorazione più chiara.



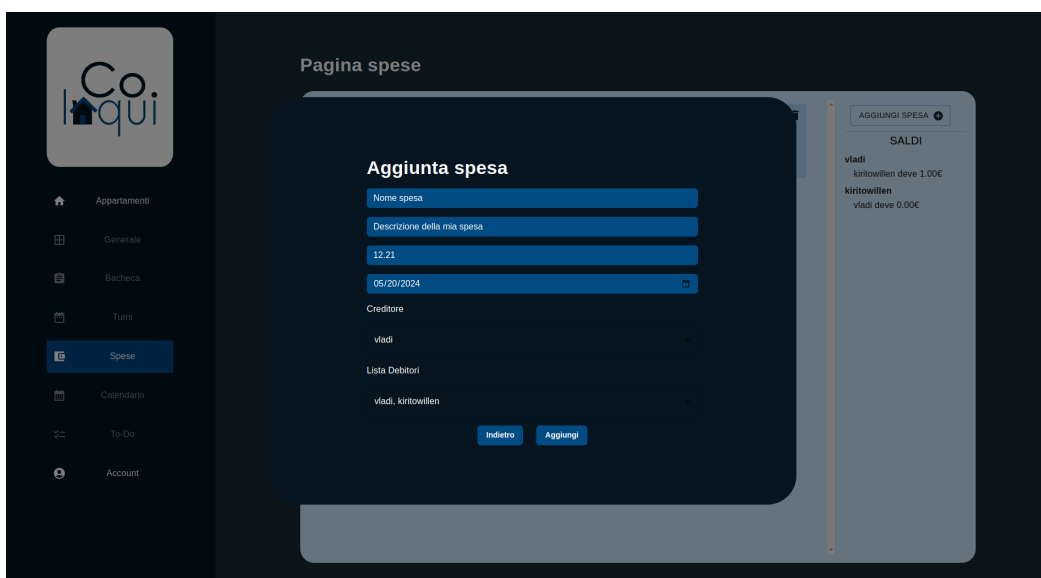
PAGINA SPESE APPARTAMENTO

In questa pagina è possibile trovare a sinistra una lista di tutte le spese presenti nell'appartamento con i loro dati di riferimento come il nome, la data, la descrizione, l'importo pagato dal creditore e la lista dei debitori. Tramite gli appositi pulsanti di "Modifica" e di "Elimina" è inoltre possibile operare sulle singole spese. Nella parte a destra è presente il pulsante "Aggiungi Spesa" che apre il modale per permettere l'aggiunta di una nuova spesa oltre ad una lista contenente i saldi di tutti gli utenti rispetto a tutti gli altri inquilini.



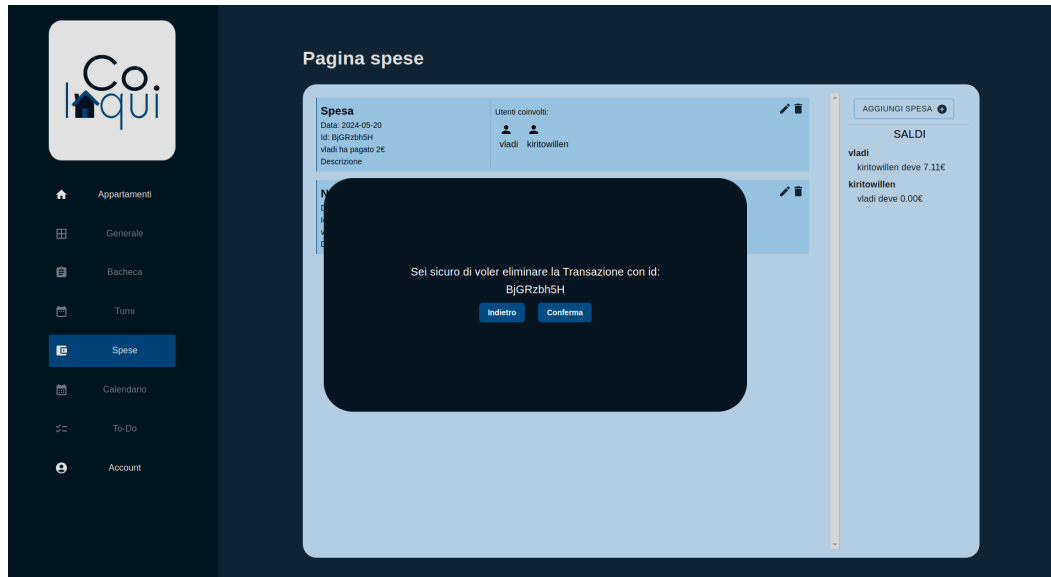
MODALE AGGIUNTA SPESA

Il modale di aggiunta spesa permette di creare nuove spese dopo aver inserito tutti i dati necessari come il nome, la descrizione, l'importo, la data, il creditore e la lista di debitori tramite l'apposito pulsante "Aggiungi" che in caso manchino dati provvederà a farlo notare. In caso si sia aperto il modale per sbaglio si può sempre tornare indietro tramite l'apposito pulsante "Indietro".



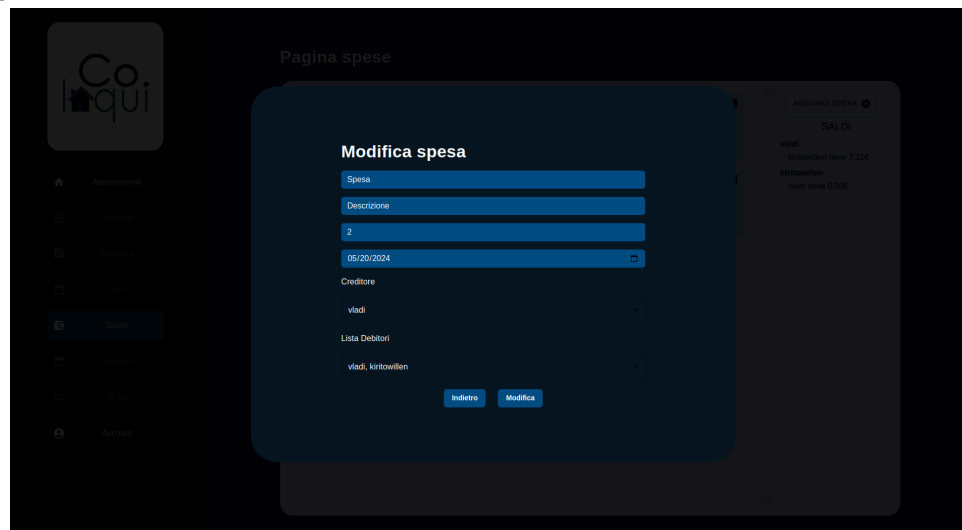
MODALE ELIMINA SPESA

Questo modale permette di eliminare una spesa chiedendo prima conferma e permettendoci di verificare che l'id della transazione sia effettivamente quello della spesa che si vuole eliminare. E' poi possibile confermare la propria scelta tramite il pulsante "Conferma" o tornare indietro in caso si abbia cambiato idea tramite il pulsante "Indietro".



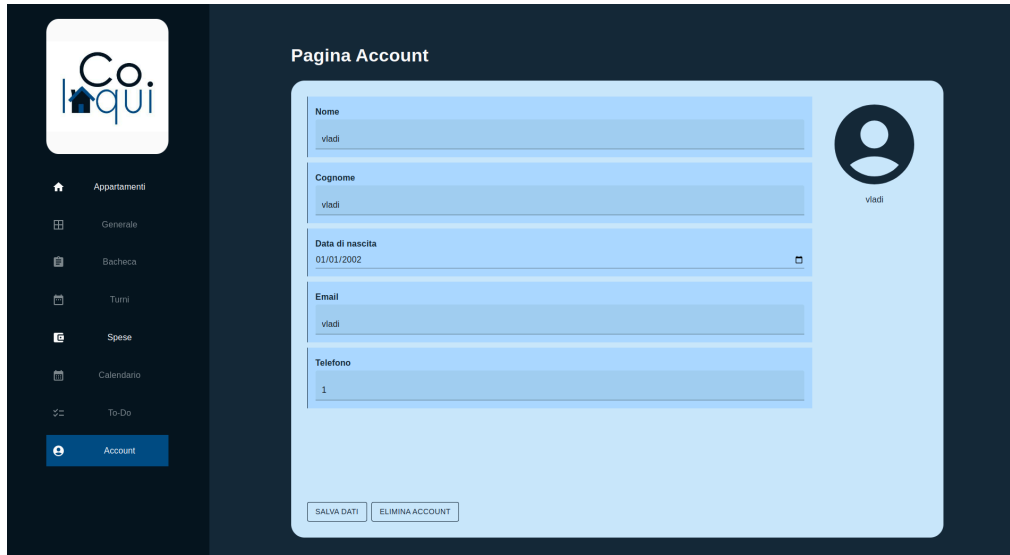
MODALE MODIFICA SPESA

Questo modale permette la modifica delle spese permettendo di cambiare tutti i valori inseriti alla creazione confermando poi le proprie scelte tramite il pulsante "Conferma" o tornando indietro grazie al pulsante "indietro".



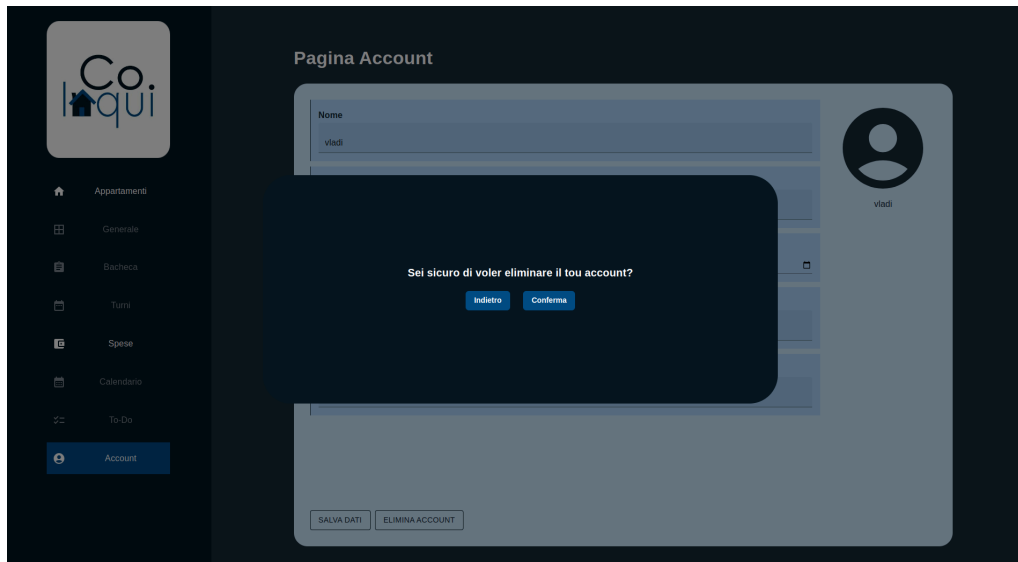
PAGINA ACCOUNT

In questa pagina è possibile vedere e modificare tutti i propri dati come nome, cognome, data di nascita, email e numero di telefono. Lo username non è modificabile in quanto usato come identificativo univoco e può essere solo visualizzato. Se vengono effettuate delle modifiche è possibile salvarle tramite il pulsante “Salva dati”. È inoltre possibile eliminare il proprio account tramite l'apposito pulsante “Elimina account”.



MODALE ELIMINA ACCOUNT

Se si vuole eliminare il proprio account appare questo modale che di annullare o confermare la propria scelta grazie ai appositi pulsanti di “Indietro” e “Conferma”



TESTING

Per effettuare il testing del back_end dell'applicazione abbiamo usato la libreria chiamata Jest e il modulo *supertest* che serve per poter svolgere delle chiamate a delle API.

I test della nostra applicazione sono stati suddivisi in tre categorie principali, ovvero *Appartamenti*, *Utenti* e *Spese*.

I file di test si possono trovare, nella folder del back-end, all'interno della cartella /tests.

Per lo svolgimento dei test abbiamo deciso di utilizzare un database diverso da quello di 'produzione' in modo che eventuali errori, o modifiche sgradite, non vadano a modificare il rendimento dell'applicazione nel suo insieme.

Vediamo ora la struttura dei test:

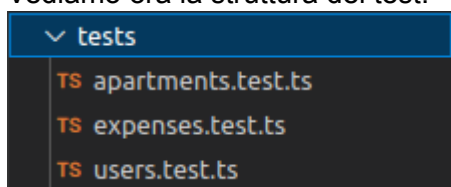


Fig1. Struttura della folder /tests

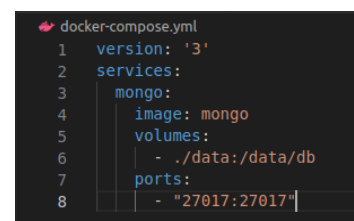


Fig2. Docker compose file

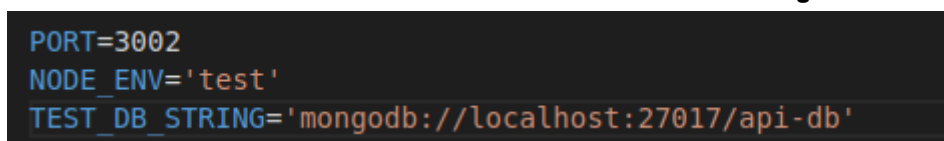


Fig3. File .env con le variabili d'ambiente

Vediamo ora la struttura dei file di test, prendiamo come esempio il file che controlla gli endpoint riguardanti gli appartamenti.



Nella parte superiore dei file di test troviamo, oltre agli import necessari per l'esecuzione dei test, due funzioni, ovvero:

- *beforeAll*: avvia il server dove verranno effettuati i test
- *afterAll*: chiude il server aperto in precedenza

Infine troviamo la "describe" che è un contenitore che contiene tutti i test che abbiamo implementato.

```

15 describe("Checking Apartments API endpoints", () => {
16 >   const user1: IUser = {--
25   };
26
27 >   const user2: IUser = {--
35   };
36
37 >   const apartment1: Partial<IApartment> = {--
40   };
41
42   let userId1: string, userId2: string,
43       sessionId1: string, sessionId2: string;
44
45 >   beforeAll(async () => {--
89   });
90
91 >   afterAll(async () => {--
99   })
100

```

Una volta avviato il server i nostri test possono iniziare, però per prima cosa dobbiamo creare dei dati che ci torneranno poi utili per lo svolgimento dei test, in questo caso si possono vedere due utenti ed un appartamento. Nel metodo `beforeAll` questi utenti vengono inseriti nel sistema e poi ci si logga con uno dei due in modo da poter avere una sessione valida con la quale fare delle chiamate alle API. Con `afterAll` invece si pulisce il database in modo da non lasciare dati che possano andare a compromettere i risultati di altri gruppi di test.

Infine nella parte finale ci sono tutte le funzioni di test vere e proprie, ognuna delle quali va a verificare un comportamento specifico per ogni endpoint. Si può notare attraverso i commenti che sono in ordine di funzionalità, quindi in alto si testa l'inserimento (POST), seguito dalla GET, PUT e PATCH e infine la DELETE.

```

// TEST SUL POST DEGLI APPARTAMENTI
let apartmentId: string;
it('Should register a new apartment', async () => {--
});

it('Should not insert a new apartment', async () => {--
});

// GET di un appartamento
it('Should return 200 and the asked apartment, where user1 has to be the admin', async () => {--
});

it('Should return 404 because apartment searched does not exists', async () => {--
});

it('Should return all the apartments in which the user is member', async () => {--
});

// PUT
it('Should return 204 and modify the apartment name', async () => {--
});

it('Should deny modify to non admin user', async () => {--
});

it('Should return 404 due to apartment not found', async () => {--
});

it('Should return code 400 due to wrong parameters composition', async () => {--
});

// PATCH
it('Should return 204 and modify the apartment name', async () => {--
});

```

```

// GET di un appartamento
it('Should return 200 and the asked apartment, where user1 has to be the admin', async () => {
  const res = await supertest(app)
    .get(`/apartments/${apartmentId}`)
    .set('Cookie', `session=${sessionId1}`);

  const expected = {--
  };

  const {__v, ...received} = res.body;

  expect(res.status).toBe(200);
  expect(received).toEqual(expected);
});

```

Come ultima cosa riportiamo l'esempio di un test, in questo caso una GET di un appartamento. Si può vedere nella parte superiore che attraverso la libreria *supertest* andiamo a richiamare l'endpoint che vogliamo testare e nella parte inferiore andiamo a verificare che i valori ritornati siano uguali a quelli che ci aspettiamo. Se tutto va come deve il test viene considerato superato, altrimenti, anche se sono una "expect" fallisce il test viene considerato fallito.

RISULTATI DEL TESTING

Per effettuare i test, una volta aver installato la repo “back_end” da github, bisogna seguire i seguenti passaggi:

1. Avviare il container di docker: “*sudo docker compose up -d*”
2. Nel caso ci siano istanze nel database queste vanno rimosse per garantire un corretto funzionamento dei test:
 - a. Comando “*sudo docker exec -it <nome container> mongosh*”
 - b. Poi si seleziona il database “api-db” con “*use api-db*”
 - c. Infine eliminare tutte le istanze nelle collections:
 - i. db.users.deleteMany({})
 - ii. db.apartments.deleteMany({})
 - iii. db.expenses.deleteMany({})
 - iv. db.sessions.deleteMany({})
3. Infine lanciare i test con il comando “*npm test*”

Questi sono i risultati dei test svolti nella nostra applicazione. Come si può notare le test suites sono 3, ovvero i 3 files indicati nella parte precedente dove abbiamo elencato la struttura della folder “/tests”, e sono tutte passate con successo insieme ai loro 55 test singoli.

```
Test Suites: 3 passed, 3 total
Tests: 55 passed, 55 total
Snapshots: 0 total
Time: 4.97 s, estimated 8 s
Ran all test suites.
```

Questa appena presentata è un’analisi molto superficiale, quindi grazie all’analisi della “code coverage” proveremo a portare un’analisi più completa riguardo all’esecuzione dei test.

Viene ora riportata la tabella con tutti i dati relativi alla copertura dei test, ovvero un report che jest fa in automatico e che presenta sotto forma di pagina html.

Per accedere a questa schermata basta accedere, dalla cartella radice del progetto, al seguente percorso “/coverage/lcov-report/index.html”.

File	Statements	Branches	Functions	Lines
back_end	86.36%	19/22	33.33%	1/3
back_end/src/functionalities/apartments	100%	17/17	100%	0/0
back_end/src/functionalities/apartments/controllers	94.44%	68/72	78.57%	22/28
back_end/src/functionalities/apartments/dao	90.24%	37/41	42.85%	3/7
back_end/src/functionalities/apartments/middleware	100%	30/30	100%	13/13
back_end/src/functionalities/apartments/models	100%	9/9	90%	9/10
back_end/src/functionalities/apartments/services	93.33%	14/15	100%	0/0
back_end/src/functionalities/common	80%	4/5	100%	0/0
back_end/src/functionalities/common/services	80%	12/15	50%	2/4
back_end/src/functionalities/expenses	100%	12/12	100%	0/0
back_end/src/functionalities/expenses/controllers	85.71%	12/14	100%	0/0
back_end/src/functionalities/expenses/dao	100%	16/16	0%	0/2
back_end/src/functionalities/expenses/middleware	100%	39/39	100%	11/11
back_end/src/functionalities/expenses/model	100%	14/14	70%	7/10
back_end/src/functionalities/expenses/services	100%	8/8	100%	0/0
back_end/src/functionalities/users	100%	14/14	100%	0/0
back_end/src/functionalities/users/controllers	100%	24/24	100%	2/2
back_end/src/functionalities/users/dao	75%	60/80	52.63%	10/19
back_end/src/functionalities/users/middleware	97.14%	34/35	93.75%	15/16
back_end/src/functionalities/users/models	100%	16/16	57.14%	8/14
back_end/src/functionalities/users/services	100%	9/9	100%	0/0

Come si può notare la tabella riporta, per ogni percorso del progetto, i dati relativi alla percentuale di righe di codice eseguite, il numero di branch testati, ovvero quanti if-statement sono stati testati e infine il numero di funzioni provate.

Nella tabella sono presenti molti percorsi, ma come visto nella sezione SVILUPPO API, molti di questi non sono direttamente chiamabili dall'esterno, ma sono ausiliari per il corretto funzionamento dei nostri endpoints. Proprio per questo i nostri test vanno a fare delle chiamate solo agli endpoints, ma come si può ben vedere dalla code coverage una buona percentuale di tutte le funzioni ausiliarie sono utilizzate.

Nonostante tutti i file abbiano una percentuale di copertura molto alta, risalta il risultato del percorso *“back_end/src/functionalities/users/dao”* quindi proviamo a vedere come mai si presenta questo risultato e se è necessario preoccuparsi di questo risultato o meno.

Grazie al report di jest possiamo cliccare sul percorso indicato per poter osservare i file all'interno della cartella, come si evince dalla figura seguente l'anomalia si presenta all'interno del file users.dao.

sessions.dao.ts	<div></div>
users.dao.ts	<div></div>

Se clicchiamo il file possiamo vedere il codice ed andare alla ricerca delle linee evidenziate, ovvero di quelle che non sono proprio state eseguite.

```

105 //DELETE requests
106 async removeById(userId: string){
107   6x const user = await this.getUserById(userId);
108   6x const username = user.username;
109
110   if(username){
111     6x const apartments = await apartmentsDao.listUserApartments(25, 0, username);
112
113     6x apartments.map(async (apartment) => {
114       1 if(apartment._id){
115         apartment.expenses.map(async expenseId => {
116           const expense = await expensesService.readById(expenseId);
117
118           1 if(expense){
119             //Se l'utente ha pagato la spesa la spesa viene direttamente rimossa
120             1 if(expense.creditor && expense.creditor === username){
121               await expensesDao.removeExpenseById(expenseId);
122             }
123             //Altrimenti solo il riferimento all'utente viene rimosso
124             const debtors = expense.debtors;
125
126             let debtorIndex = debtors.indexOf(username);
127
128             if(debtorIndex >= 0){
129               debtors.splice(debtorIndex, 1);
130               await expensesService.patchById(expenseId, { debtors: debtors });
131             }
132           });
133         });
134       }
135     });
136     const users = apartment.users;
137
138     let index = users.indexOf(username);
139     users.splice(index, 1);
140
141     await apartmentService.patchById(apartment._id, {users: users});
142   }
143 }
144
145 }
146
147 6x await this.User.deleteOne({_id: userId}).exec();
148
149 }

```

Dall'analisi del codice risulta che la maggior parte delle linee di codice che non sono eseguite si trovano in questa funzione, ovvero quella che si occupa di eliminare l'account di un utente. Se si fa un'analisi approfondita del codice si può notare che le parti evidenziate in rosso servono per rimuovere tutti i riferimenti verso l'utente che si vuole eliminare che altri oggetti nel database possono avere. Questa funzionalità risulta non testata poiché aggiunta di recente, ma in ogni caso tutte le funzioni che chiamano elementi esterni, quali expenseService, expenseDao e apartmentService, sono già testati con altri test presenti nelle suites delle spese e degli appartamenti.

GITHUB REPOSITORY E INFORMAZIONI SUL DEPLOYMENT

Tutto lo sviluppo della web app è stato svolto tenendo i progressi su github, in seguito forniamo una lista di URL contenenti diverse parti del nostro progetto:

- <https://github.com/G30-unitn-IngegneriaDelSoftware/progetto>
- https://github.com/G30-unitn-IngegneriaDelSoftware/back_end
- <https://github.com/G30-unitn-IngegneriaDelSoftware/Deliverables>

Nel primo link si trovano i file relativi allo sviluppo del frontend, nel secondo i file relativi allo sviluppo del back_end e infine si trova una repo dove abbiamo caricato tutti i Deliverables che abbiamo scritto durante lo sviluppo del progetto.

Per quanto riguarda il deployment dell'applicazione ci siamo affidati al servizio di vercel, sia per il front-end che per il back-end. Per quanto riguarda il back-end il link di collegamento è il seguente, <https://is-2024-backend.vercel.app>, e lo riportiamo giusto per dovere di cronaca, ma diciamo che in questa forma non è utilizzabile se non con programmi che permettano di interagire tramite il protocollo HTTP come Postman. Per quanto riguarda il front-end, invece, il link è navigabile senza problemi ed è utilizzabile come se si navigasse su un qualsiasi sito internet. Il link è il seguente:

<https://is-2024-front-end.vercel.app>

ESEGUIRE IL SERVER SULLA PROPRIA MACCHINA

Nel caso si verificassero dei problemi con il link sopra indicato è possibile utilizzare l'applicazione in locale seguendo i seguenti passaggi.

Come prima cosa è necessario scaricare da github le due repo relative al front-end e al back-end che si possono trovare nei link sopra elencati.

Per entrambe le repo bisogna eseguire il comando “**npm install**” per scaricare i pacchetti necessari per far andare l'applicazione. Da notare che per quanto riguarda la parte di back-end per garantire il suo funzionamento bisogna anche creare un file .env come segue:

=====

```
PORT=3002
TEST_DB_STRING='.....'
DB_STRING='.....'
```

=====

Il primo parametro è la porta dove si vuole far partire il server, gli altri due sono le stringhe che permettono di collegarsi al database, il primo per l'ambiente di test mentre il secondo per l'ambiente di produzione.

Se non si ha a disposizione un database online e si vuole usare docker si può usare il seguente comando per avviare un database in locale:

```
sudo docker compose up -d
```

L'esecuzione di questo comando crea un database che risponde alla stringa:

```
mongodb://localhost:27017/api-db
```

Per quanto riguarda la parte di front-end se si vuole usare il back-end in locale, bisogna fare un piccolo cambiamento. Navigare in **src > index.jsx** e cambiare il valore della variabile defaultPath, che si trova alla riga 30, con localhost:Porta.

Infine, per avviare i due progetti, si eseguono i seguenti comandi:

- **npm start:** nella cartella del backend
- **npm run start:** per il front-end

Se non ci sono problemi, accedendo al link “localhost:3000” si dovrebbe poter accedere alla web app.