Università degli Studi di Napoli "Parthenope"
Facoltà di Scienze e Tecnologie
Corso di laurea in Informatica

Progetto di Reti di Calcolatori

PropertEase

Docente
Emanuel Di Nardo

Studenti e matricole
Amore Gabriel - 0124002585
Ben Baccar Alexandr - 0124002586
De Micco Carmine - 0124002630

Anno Accademico 2023-2024

Indice

Capitolo 1

Traccia progetto

Si vuole sviluppare un sistema software per per un'agenzia immobiliare che offre servizi di vendita di proprietà e consulenze da parte di professionisti.
Ogni immobile ha caratteristiche come **indirizzo, metratura, descrizione, ascensori, numero di vani, accessori, giardini e terrazzi.**

Le case sono suddivise in:

- **Appartamento:** inteso come un locale in un condominio;
- **Casa indipendente:** inteso come una villetta;
- **Garage:** inteso come spazio non adibito al domicilio.

Il sistema deve essere gestito in modalità **amministratore** e in modalità **cliente** (e.g. operazioni sul web).

L'**amministratore**, inteso come un *agente immobiliare*, può effettuare le seguenti operazioni:

- Visualizzare, inserire, modificare ed eliminare un immobile con le relative informazioni;
- Inserire le date in cui è disponibile per appuntamenti con i clienti;
- Visualizzare ed eliminare gli appuntamenti presi con i clienti.

Il **cliente** può effettuare le seguenti operazioni:

- Visualizzare gli immobili;

- Prendere appuntamenti con gli amministratori nel limite di quelle messe a disposizione da questi ultimi;
- Eliminare i propri appuntamenti presi in precedenza.

L'architettura del software è basata su quella **Client-Server**. A tal fine, sono stati richiesti tre server:

- **Gestore degli immobili:** gestisce tutte le interazioni da parte dei vari client che necessitano di operazioni sul database riguardanti gli immobili;
- **Gestore degli appuntamenti:** gestisce tutte le interazioni da parte dei vari client che necessitano di operazioni sul database riguardanti gli appuntamenti;
- **Main handler:** il suo compito principale è quello di smistare le richieste agli altri due server, tuttavia, svolge anche funzioni che non ricadono nelle altre due categorie, come registrazione e login degi client.

1.1 Tecnologie

La realizzazione del progetto ha coinvolto l'utilizzo di diverse tecnologie:

- Il *linguaggio di programmazione* scelto è **Java**, noto per la sua versatilità e portabilità.
- Per l'implementazione dell'*interfaccia grafica*, è stato utilizzato **JavaFX**, un framework flessibile che ha permesso la creazione di un'esperienza utente semplice e intuitiva.
- Per garantire la *persistenza dei dati*, è stato adottato il DBMS **SQLite**: scelto per la sua leggerezza e faciltà di integrazione, consentendo un'efficace archiviazione e recupero delle informazioni e garantendo un'ottima gestione dei dati.

Capitolo 2

Struttura

Per il corretto funzionamento dell'applicazione, è essenziale avviare gli applicativi server e client in sequenza.

Per avviare il server, la classe `StartServer`:

1. *Istanzia* il **server** principale e i due server secondari;
2. *Assegna* a ognuno di questi una **porta**, arbitrariamente 1926, 1927 e 1928;
3. *Istanzia* un **thread** per ognuno dei server.
   Ogni thread inizia automaticamente l'esecuzione del main-loop del server, ossia si pone in ascolto di connessioni e crea un ulteriore thread per ogni client che si connette.

Ogni *client* è gestito da un'appropriata *strategia*, implementate nelle classi:

- `AppointmentClientStrategy`: per il gestore degli appuntamenti;
- `HandlerClientStrategy`: per il gestore principale;
- `PosterClientStrategy`: per il gestore degli immobili.

La logica di base per la *comunicazione* con il client è implementata all'interno della classe `ClientManager`, che offre funzioni di utilità per leggere o scrivere un messaggio sullo stream della socket; questa classe inoltre fornisce un oggetto per la connessione e la gestione del database attraverso `DatabaseConnectionProxy`.

Per **avviare il programma** client, la classe `StartApplication` estende la classe `Application` e fornisce le istruzioni di configurazione per l'interfaccia e l'avvio.

La **creazione delle** `interfacce grafiche` è definita all'interno dei file `FXML`, dove vengono specificati gli elementi che le compongono e le relative proprietà. Ogni interfaccia è associata a una classe `Controller`, il cui compito è gestire sia l'aspetto grafico che la logica sottostante tramite codice.

Per la **gestione dei dati** degli:

- **Immobili:** è stata realizzata la classe `House`, i cui oggetti sono creati ed affidati a una gerarchia di classi che segue il pattern "Builder";
- **Appuntamenti:** è stata implementata la classe `Appointment`;
- **Utenti loggati:** è stata sviluppata la classe `User`.

2.1 Pattern

Per l'implementazione dei server, sono stati adottati due **design pattern**, uno comportamentale (`STRATEGY`) ed uno strutturale (`PROXY`).
Il primo implementa il caricamento on-demand del database per questioni di performance mentre l'altro, tramite scambio di messaggi in JSON, implementa la selezione degli algoritmi fini allo smistamento dei server ausiliari da parte dell'handler.

2.1.1 Proxy

È un pattern structural che consente di fornire un sostituto o un placeholder per un altro oggetto. Il proxy controlla l'accesso all'oggetto originale, consentendo di eseguire qualcosa prima o dopo che la richiesta sia arrivata all'oggetto originale.

Utilizzo pratico

Nel nostro caso, è stato utilizzato un "Virtual Proxy" per la connessione al database nei server secondari. Questo approccio ci permette di creare la connessione solo quando è effettivamente necessaria (on-demand), evitando così il costo di una connessione costante.

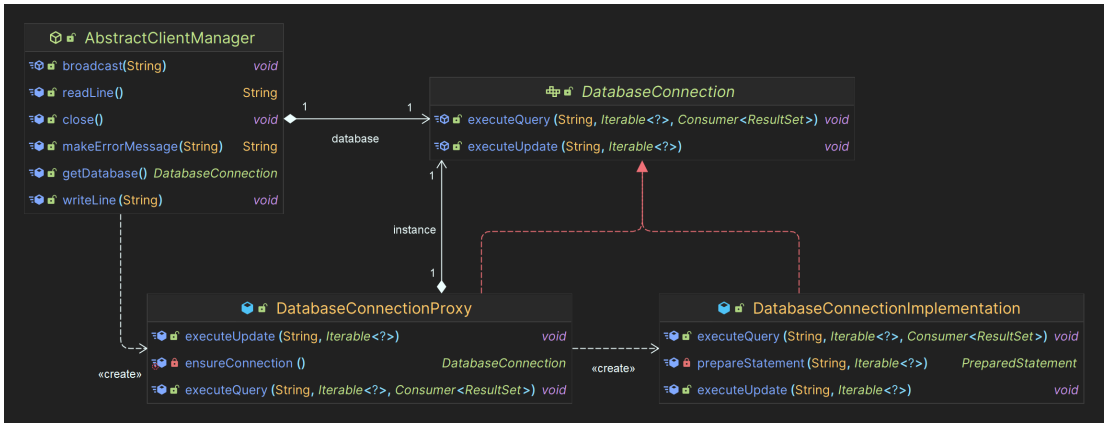

Figura 2.1: Struttura Proxy pattern implementato nel SW con metodi

2.1.2 Strategy

Questo pattern behavioural definisce una famiglia di algoritmi, incapsulandoli e rendendoli intercambiabili. Si usa quando si ha necessità di modificare dinamicamente gli algoritmi utilizzati dall'applicazione.
N.B.: L'algoritmo cambia indipendentemente dai client che lo usano

Utilizzo pratico

Nel nostro caso, questo pattern è stato utilizzato per implementare facilmente la gestione dei comportamenti dei vari server, questi forniscono funzionalità diverse e gestiscono messaggi diversi ma, alla base delle loro funzionalità c'è sempre lo stesso principio: lo scambio di messaggi.

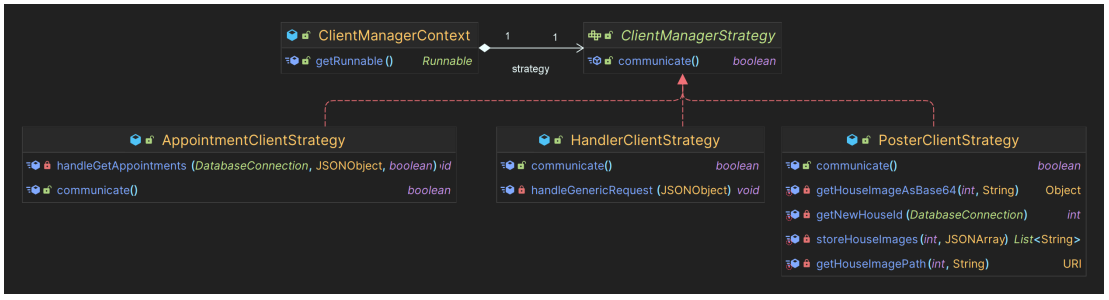

Figura 2.2: Struttura Strategy pattern implementato nel SW con metodi

Capitolo 3

Architettura

Il sistema impiega un'architettura di tipo **ibrido**, utilizzando sia caratteristiche del modello **client/server** chee quelle del modello **peer-to-peer**.
In dettaglio, la struttura prevede l'utilizzo di tre server:

- **Handler Server:** coordina la comunicazione tra il client e gli altri due server;
- **Poster Server:** svolge funzioni di gestione degli *annunci*;
- **Appointment Server:** svolge funzioni di gestione degli *appuntamenti*.

La connessione è effettuata tramite **Socket**, con protocollo **TCP/IP**, risultando quindi *connection-oriented* e *reliable*. Possiamo dividere l'architettura in quattro processi:

- **Main Client:** agisce puramente da client e si interfaccia esclusivamente con il processo "Handler Server", sia per l'autenticazione che per la comunicazione con gli altri due processi;
- **Handler Server:** agisce sia da server, verso il processo "Handler Server" che da client, verso di processi "Poster Server" e "Appointment Server".
  Si occupa principalmente di smistare i messaggi in arrivo dal processo "Main Client" e di gestire l'autenticazione con lo stesso;
- **Poster Server:** agisce puramente da server, verso il processo "Handler Server".
  Si occupa di gestire i comandi riguardanti gli immobili, ossia inserimento, cancellazione e selezione.;

- **Appointment Server:** agisce puramente da server, verso il processo "Handler Server".
  Si occupa di gestire i comandi riguardanti gli appuntamenti, ossia inserimento, cancellazione e selezione.

Capitolo 4

Comunicazione

La comunicazione tra le varie entità del sistema è eseguita tramite **primitive bloccanti** per `read` e per `write`, nel caso di Java, `writeLine` e `readLine`. Lo scambio di messaggi avviene in **locale** basandosi sul protocollo **TCP/IP** tramite l'utilizzo di socket **bidirezionali**; pertanto, sarà sufficiente utilizzare una singola socket per ogni client connesso.

Quando viene **inviato** un messaggio, questo viene posto in un **buffer interno**: qualora il buffer dovesse essere *pieno*, il thread è bloccato finché non si *svuota*.
Al fine di garantire questo comportamente, viene utilizzato il tipo `PrintWriter`, configurato con `autoFlush`.
N.B.: il messaggio viene processato prima di essere inviato, aggiungendo simboli di escape.

Quando viene **ricevuto** un messaggio, il thread è bloccato in attesa di un messaggio da leggere, una volta che il messaggio è presente sullo stream di input, viene processato rimuovendo i simboli di escape e inserito in una struttura dati `JSON`.

Per design, è assunto che il contenuto e la struttra dei messaggi in JSON siano validi e NON è per tanto necessario eseguire operazioni di controllo.
Verranno però validati i parametri inviati dal client.

4.1 Main Client ⇔ Handler Server

Il client principale si interfaccia con l'Handler Server per autenticarsi, e per la comunicazione con gli altri server.
L'istante in cui un client si connette all'Handler Server, questo genera automaticamente altre due connessioni ai server ausiliari; a questo punto i tre server si mettono in attesa di messaggi.
La comunicazione diretta per l'autenticazione viene effettuata specificando nel campo `type` della richiesta "generic".

4.2 Autenticazione - `handleSignInRequest`

All'avvio del client, viene mostrata una finestra di login dove l'utente potrà inserire il suo username e la sua password, una volta confermati i dati, l'applicazione costruirà una richiesta `JSON` con i parametri necessari al login e verrà inviata immediatamente al server.
Il client si mette poi in attesa di una risposta dal server, se questa contiene i privilegi di accesso allora il login avrà avuto successo, altrimenti la risposta conterrà un errore con una breve spiegazione del problema.

4.3 Registrazione - `handleSignUpRequest`

Dalla schermata di login, viene mostrato un bottone che permette di registrarsi, la registrazione richiede all'utente di inserire:

- Username;
- Password;
- Nome;
- Cognome.

Una volta confermati i dati, l'applicazione costruisce una richiesta `JSON` con i parametri necessari per la registrazione e la invia immediatamente al server.
A questo punto, il client si mette in attesa di una risposta dal server. Se la risposta contiene un `JSON` vuoto, l'utente viene registrato nel sistema. In caso contrario, la risposta conterrà un errore con una breve spiegazione del problema (es. l'username scelto è già in uso).

4.4 Main Client ⇔ Handler Server

L'Handler Server gestisce la comunicazione con il Poster Server internamente e in maniera trasparente all'utente. I messaggi aventi il campo type impostato a "poster" vengono inoltrati automaticamente a questo server.
Le richieste supportate dal Poster Server sono:

- `getHouses`;
- `insertHouse`;
- `deleteHouse`.

4.5 Richiesta di Immobili – `getHouses`

Questa richiesta viene effettuata immediatamente quando `MainViewController`, ossia la pagina principale dell'applicazione, viene mostrata; oppure quando si cambia pagina verso `MainViewController`.
Questa richiesta non ha bisogno di parametri e si limita a restituire l'intera collezione di immobili disponibili nel database come un array `JSON`.
Non essendo previsti errori per questa richiesta, possiamo assumere che andrà sempre a buon fine.

4.6 Inserimento di Immobili – `insertHouse`

La richiesta di inserimento di un immobile gestisce sia l'inserimento che la modifica degli stessi. Viene originata da `AddHouseController`, ossia la pagina di inserimento e modifica immobili.

La richiesta dovrà contenere tutti i parametri necessari per l'inserimento di un immobile, con eventuali parametri opzionali come le immagini da inserire e l'id per l'immobile.
Quest'ultimo risulta opzionale in quanto, quando la richiesta contiene l'id, viene interpretata come una modifica.
N.B.: Qualora un immobile non abbia una o più immagini, ne verrà utilizzata una placeholder.
Le immagini sono trasmesse dal server e al client in formato **base64**.

Non essendo previsti errori per questa richiesta, possiamo assumere che andrà sempre a buon fine.

4.7 Cancellazione di Immobili – `deleteHouse`

La richiesta di cancellazione di un immobile è originata da `AddHouseController` e necessita solo di un parametro, l'id dell'immobile da eliminare.
Poiché l'amministratore non specifica mai manualmente l'id (in quanto automaticamente determinato da `AddHouseController`), si può assumere che questo sia sempre valido, pertanto, tale funzione non restituisce nessun errore.

4.8 Handler Server ⇔ Appointment Server

L'Handler Server gestisce la comunicazione con l'Appointment Server internamente e in maniera *trasparente* all'utente. I messaggi che hanno il campo `type` impostato ad "appointment" vengono inoltrati automaticamente a questo server.

4.9 Richiesta Appuntamenti – `getAppointmentsForUser / Agent`

Questa richiesta acquisisce gli appuntamenti per l'utente specificato, il quale può essere sia un cliente che un agente immobiliare. La richiesta è originata da `MainViewController` automaticamente.
In base al tipo di utente attualmente loggato, viene richiamata la funzione `getAppointmentsForUser` o `getAppointmentsForAgent`.
Sarà ritornata una lista di appuntamenti presi dall'utente oppure la lista di appuntamenti a cui partecipa l'agente sottoforma di array `JSON`. Si può inoltre assumere che non saranno ritornati errori.

4.10 Richiesta Appuntamenti – insertAgentAvailability

Questa richiesta è originata da `InsertAvailabilityController` ed inserisce un range di disponibilità in base ai parametri `start_date` e `end_date`.

Le date saranno inserite una ad una nel database e, nel caso una data sia già presente, sarà semplicemente saltata.
Poiché il controllo `end_date > start_date` è effettuato dal client, si può assumere che non saranno ritornati errori.

4.11 Richiesta Appuntamenti – `bookAppointment`

Questa richiesta è originata da `HouseDetailsController`, in questo caso, l'utente loggato, ossia il cliente, vuole riservare un appuntamento per un immobile scegliendo il primo agente disponibile.
La funzione ritorna due errori che devono essere obbligatoriamente gestiti dal client; in entrambi i casi l'appuntamento NON è prenotato e la disponibilità dell'agente rimane invariata. Gli errori sono:

- `notAvailable:` Se l'utente sceglie una data dove nessun agente ha dato disponibilità;
- `alreadyBooked:` Se l'utente sceglie una data per cui ha già preso un appuntamento.

4.12 Richiesta Appuntamenti – removeAppointment

Questa richiesta è originata da `MainViewController` quando l'utente, ossia il cliente, vuole cancellare un appuntamento. La funzione richiede un solo parametro: l'id dell'appuntamento da rimuovere.
Poiché l'utente può solo cancellare un appuntamento che appare nella lista di appuntamenti, si può assumere che l'id sarà sempre valido, pertanto la richiesta non ritorna errori.

Capitolo 5

Database

Come sopracitato, è stato utilizzato SQLite in quanto la natura embedded permette di incorporarlo direttamente nell'applicazione, rendendo la gestione dei dati più agevole e contribuendo a ridurre la complessità della configurazione del sistema. Questo è particolarmente vantaggioso per un'applicazione di medie dimensioni come la nostra.
Segue il diagramma relazionale del database impiegato come fondamento per la memorizzazione a lungo termine dei dati del software.

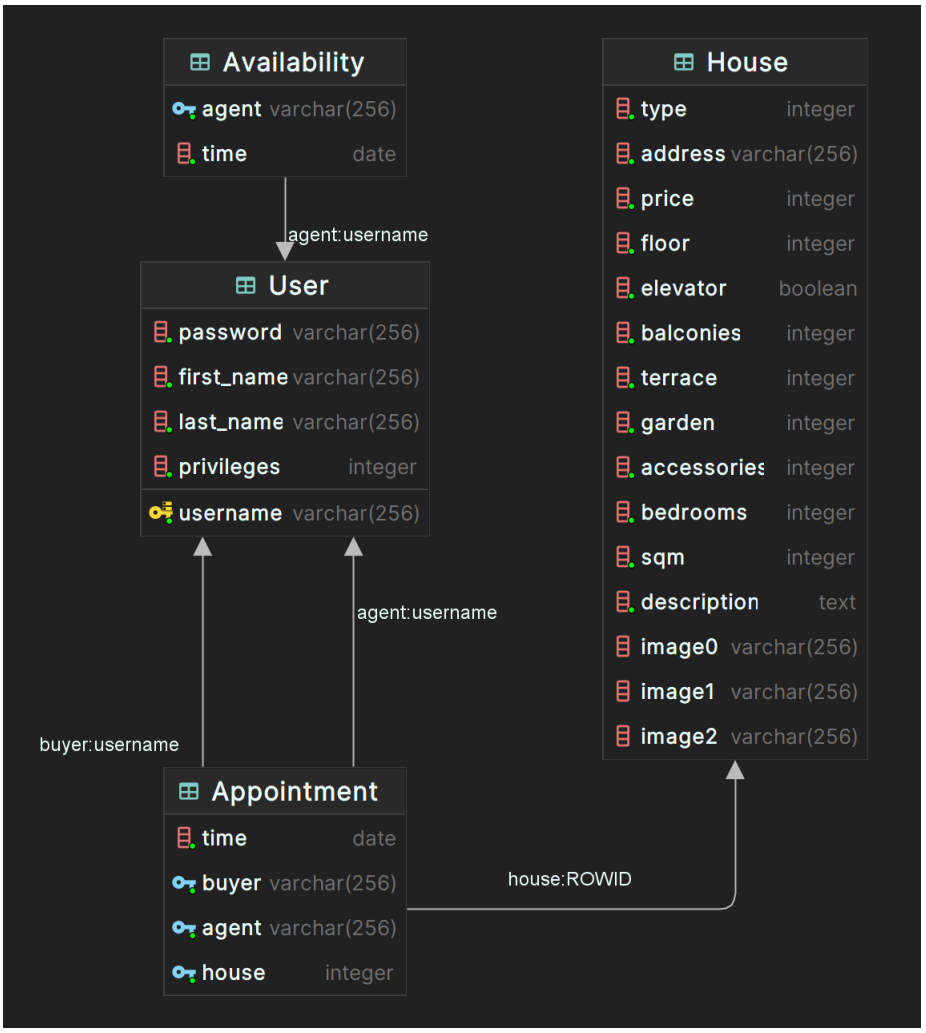

Figura 5.1: Diagramma relazionale del database

Capitolo 6

Guida all'utente

6.1 Avvio dell'applicazine

Per poter utilizzare l'applicativo in maniera corretta bisogna

1. Avviare il processo server "`StartServer`";
2. Avviare il processo client "`StartApplication`".

6.2 Accesso

Per assegnare il ruolo di "Amministratore" a un utente, è necessario effettuare l'inserimento direttamente nel database per garantire un adeguato controllo dei privilegi e una buona gestione degli accessi.

Per utilizzare l'applicazione, sia il cliente che l'amministratore (agente immobiliare) devono effettuare l'accesso. Tale procedura richiede l'inserimento di un nome utente e di una password.
Al momento del tentativo di accesso, il sistema, tramite il `Main Handler` interroga il database per verificare se l'utente è registrato e se le credenziali inserite sono corrette.

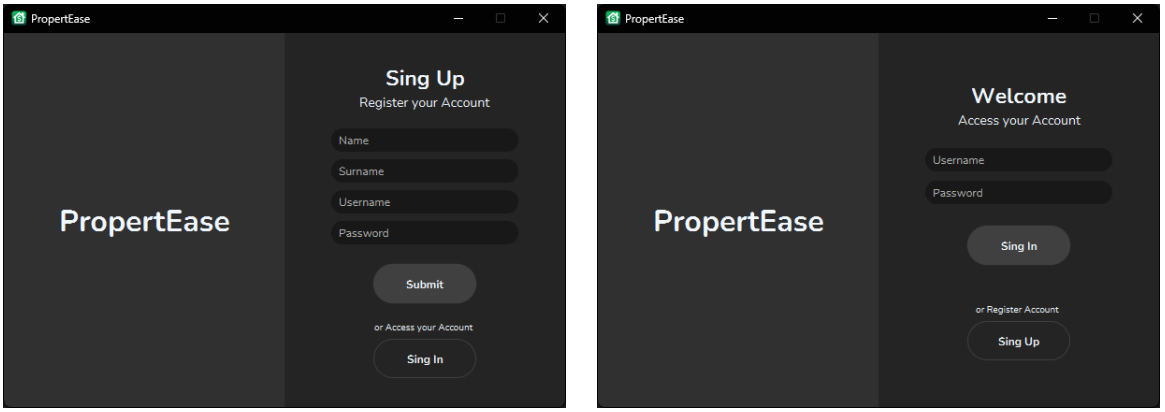

Figura 6.1: (sx): registrazione (dx): accesso

Nel caso in cui un utente non risulti registrato, è possibile effettuare la registrazione tramite la sezione dedicata denominata "Sign Up". In questa sezione, vengono richiesti il nome, il cognome, il nome utente e la password.
N.B.: Sono implementati controlli sull'unicità del nome utente. Se i dati inseriti sono corretti, le credenziali vengono memorizzate nel database.

6.3 Utilizzo applicazione

6.3.1 Utente

L'interfaccia utente post-login è molto intuitiva, con le card cliccabili per ogni immobile ed un tab sulla sinistra con i suoi appuntamenti.

Cliccando su una della card, si è reindirizzati alla pagina in cui sono mostrati i dettagli degli immobili e dove è possibile prenotare un appuntamento.

La navbar in alto permette la navigazione delle pagine.

6.3.2 Amministratore

Oltre a quanto sopracitato per l'utente, l'amministratore avrà le varie funzioni sempre in pulsanti locati in alto a destra della GUI.
Tramite questi bottoni, egli potrà inserire, rimuovere e aggiornare gli immobili

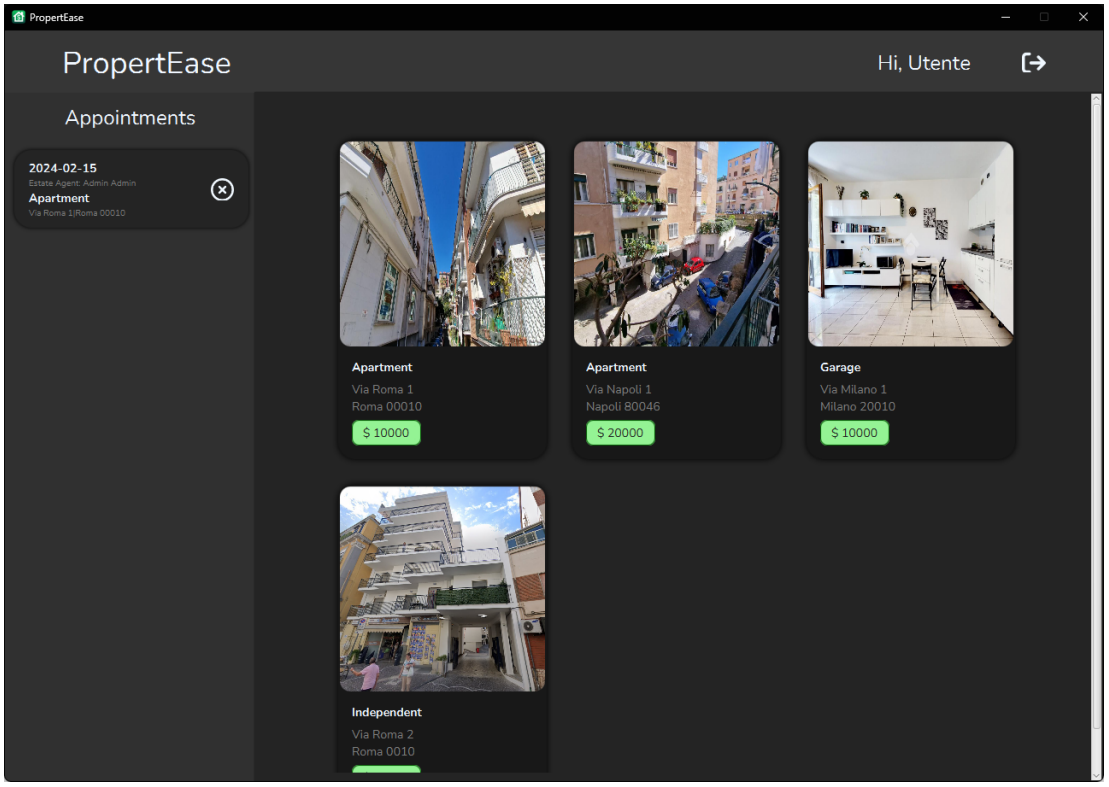

Figura 6.2: Schermata principale dell'applicazione

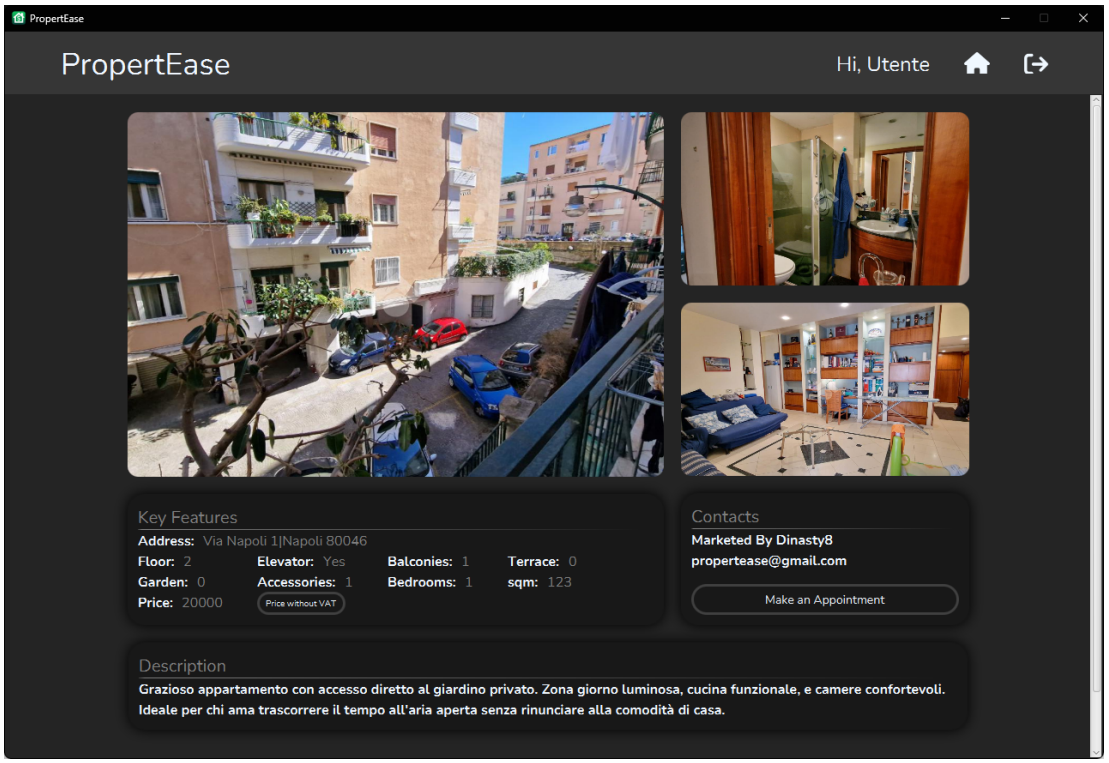

Figura 6.3: Schermata dei dettagli di un immobile

del database.

6.3.3 Errori

per ogni operazione di modifica, inserimento ed eliminazione su database, è stato inserito, dal punto di vista di interfaccia, un pop-up di conferma o metodi di ripristino dei dati.

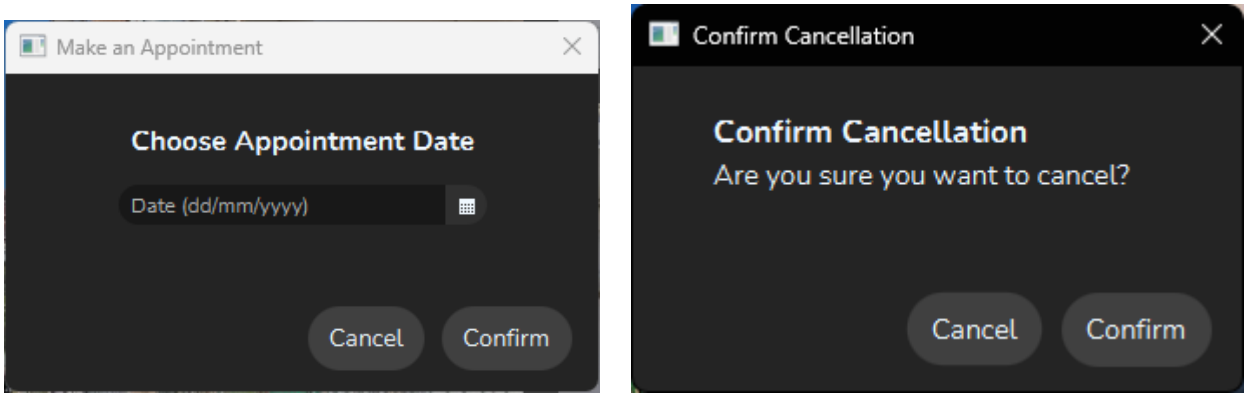

Figura 6.4: conferma di presa (sx) e cancellazione (dx) di un appuntamento