

UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Triennale in Ingegneria Informatica e dell'Automazione



TESI DI LAUREA

**Progettazione e implementazione di un'app in tecnologia SWIFT
per il supporto nella ricerca di abitazioni: la componente di gestione
degli utenti e delle loro interazioni**

**Design and implementation of an app in SWIFT technology for
housing search support: the component for managing users and
their interactions**

Relatore

Prof. Domenico Ursino

Correlatore

Dott. Enrico Corradini

Candidato

Andrea Marini

ANNO ACCADEMICO 2022-2023

Sommario

L'innovazione tecnologica e la crescente domanda di soluzioni digitali nel settore immobiliare hanno spinto la progettazione e l'implementazione di un'applicazione mobile avanzata, sviluppata in tecnologia Swift per dispositivi iOS. Questa tesi si concentra sulla componente di gestione degli utenti e delle loro interazioni all'interno di un'applicazione concepita per agevolare la ricerca di abitazioni.

Il processo di sviluppo inizia con la definizione chiara dei requisiti funzionali e l'analisi delle esigenze degli utenti nel contesto della ricerca immobiliare. La progettazione dell'interfaccia utente è guidata da principi di usabilità e accessibilità, assicurando che gli utenti possano navigare agevolmente tra le opzioni di ricerca, visualizzare dettagli immobiliari e gestire le loro interazioni con il sistema.

Gli utenti possono registrare i propri profili, autenticarsi in modo sicuro e gestire le loro preferenze di ricerca. Inoltre, l'applicazione offre funzionalità avanzate, come il salvataggio tra i preferiti, la ricerca e la visualizzazione del risultato su mappa e la gestione delle comunicazioni tra utenti con stesse abitazioni salvate tra i preferiti, in modo da trovare, in modo più semplice, eventuali coinquilini con cui dividere l'abitazione.

Swift è un linguaggio di programmazione moderno e altamente performante, progettato appositamente per dispositivi iOS. Esso si distingue per la sua sicurezza avanzata del tipo di dati, facilitando la prevenzione di errori comuni di programmazione. La scelta di usarlo come linguaggio principale di sviluppo garantisce una base solida per l'applicazione, consentendo una codifica efficiente e riducendo il rischio di bug critici. Inoltre, Swift è integrato con potenti framework iOS, come SwiftUI, e l'ambiente di sviluppo XCode, che ha semplificato notevolmente il processo di progettazione e implementazione dell'interfaccia utente. Ciò ha permesso una realizzazione più rapida e un controllo più preciso dell'aspetto e del comportamento dell'applicazione, migliorando l'esperienza utente finale.

Keyword: Ingegneria del software, Applicazione nativa, Architettura dell'applicazione, Swift, Programmazione asincrona, API, Postman

Introduzione	1
1 Introduzione a SWIFT	3
1.1 Swift	3
1.1.1 Un po' di storia	3
1.1.2 Caratteristiche generali	4
1.2 SwiftUI	8
1.2.1 Caratteristiche Generali	9
1.2.2 Live rendering	9
1.2.3 Property wrapper	9
1.3 XCode	12
2 Specifica e Analisi dei Requisiti	14
2.1 Che cos'è l'ingegneria del software	14
2.2 UML	15
2.3 Requisiti	16
2.3.1 Requisiti in linguaggio naturale	16
2.3.2 Requisiti funzionali	17
2.3.3 Requisiti non funzionali	18
2.4 Modellazione dei casi d'uso	18
2.4.1 Rappresentazione grafica dei casi d'uso	18
2.4.2 Casi d'uso in linguaggio naturale	18
3 Progettazione	23
3.1 Diagramma delle classi	23
3.2 Diagramma di sequenza	24
3.3 Diagramma di attività	28
3.4 Architettura dell'app	28
4 Implementazione	41
4.1 Frontend	41
4.1.1 Mockup	41
4.1.2 Utilizzo di SwiftUI nel codice	41
4.2 Backend	48
4.2.1 Google Maps	48

4.2.2	Firebase	49
4.2.3	Postman	50
4.2.4	Esempi significativi nel codice	50
5	Manuale Utente	56
5.1	Autenticazione	56
5.2	Utilizzo della mappa	56
5.3	Utilizzo della lista delle abitazioni	59
5.4	Utilizzo della chat	60
5.5	Utilizzo della sezione del profilo	63
6	Discussioni	66
6.1	Analisi SWOT del sistema realizzato	66
6.1.1	Punti di forza	66
6.1.2	Punti di debolezza	67
6.1.3	Opportunità	67
6.1.4	Minacce	67
6.2	Confronto con sistemi correlati	68
6.2.1	Immobiliare	68
6.2.2	Idealista	70
6.2.3	Confronto con la nostra applicazione	70
7	Conclusioni	73
	Bibliografia	74
	Ringraziamenti	76

Elenco delle figure

1.1	Il fondatore e il logo di Swift	4
1.2	Variabili case-sensitive	5
1.3	Funzionamento del type inference implicito	5
1.4	Tipi di dato comuni presenti in Swift	5
1.5	Definizione e chiamata di una funzione in Swift	6
1.6	Classi e Oggetti in Swift	7
1.7	Operatori per la gestione della Null Safety	7
1.8	Logo di SwiftUI	9
1.9	Approccio Imperativo vs Approccio Dichiarativo	10
1.10	Funzionamento di @Binding	12
1.11	Logo di XCode	13
2.1	Ciclo di vita di un progetto	15
2.2	Sintassi per esprimere un requisito	16
2.3	Esempio di utilizzo di «include»	19
2.4	Esempio di utilizzo di «extend»	19
2.5	Modello dei casi d'uso Autenticazione	20
2.6	Modello dei casi d'uso RicercaFiltrata	20
2.7	Modello dei casi d'uso Chat	20
2.8	Modello dei casi d'uso ModificaProfilo	21
3.1	Visibilità degli elementi di un package	24
3.2	I vari tipi di molteplicità delle associazioni	24
3.3	Esempio completo di associazione tra classi	24
3.4	Suddivisione in package dell'applicazione	25
3.5	Diagramma delle classi del package GestioneUtenti	25
3.6	Diagramma delle classi del package GestioneProprietà	26
3.7	Diagramma delle classi del package GestioneMappa	26
3.8	Diagramma delle classi del package GestioneChat	27
3.9	Rappresentazione della linea di vita	27
3.10	Rappresentazione della barra di attivazione	27
3.11	Tipologie di messaggio	28
3.12	Diagramma di sequenza di Registrazione	29
3.13	Diagramma di sequenza di Login	29
3.14	Diagramma di sequenza di RicercaFiltrata	30

3.15	Diagramma di sequenza di ModificaProfilo	31
3.16	Diagramma di sequenza di Chat	32
3.17	Diagramma di attività di Registrazione	33
3.18	Diagramma di attività di Login	34
3.19	Diagramma di attività di RicercaFiltrata	35
3.20	Diagramma di attività di Chat	36
3.21	Diagramma di attività di ModificaProfilo	37
3.22	Model di User	37
3.23	Esempio di ViewModel	38
3.24	Esempio di view	39
3.25	Suddivisione in file dell'applicazione	40
4.1	Mockup per l'autenticazione	42
4.2	Mockup della mappa	43
4.3	Mockup della lista di abitazioni come risultato della ricerca e dei filtri	43
4.4	Mockup dei preferiti e della schermata con tutte le informazioni	44
4.5	Mockup della lista delle chat e della schermata con la chat singola	44
4.6	Mockup del profilo e della schermata di modifica dell'email	45
4.7	Mockup della schermata di modifica della password e di modifica delle informazioni personali	45
4.8	Codice per la costruzione della schermata di Login	47
4.9	Esempio di utilizzo del metodo .task	48
4.10	Database di autenticazione	49
4.11	Database Firestore con le informazioni relative ad un utente	50
4.12	Chiamata API per l'ottenimento dell'accessToken	51
4.13	Uso di async e await	51
4.14	Esempio senza completion	52
4.15	Esempio con completion	52
4.16	Codice della funzione per ottenere l'accessToken	53
4.17	Codice della funzione per fare una chiamata API a Idealista	54
4.18	Utilizzo di completion con getAccessToken e postRequestIdealista	55
5.1	Schermata di login e di registrazione	57
5.2	Schermata con la mappa	57
5.3	Risultato della ricerca tramite barra di ricerca o click sulla mappa	58
5.4	Maggiori informazioni su un'abitazione	58
5.5	Schermate della lista di abitazioni	59
5.6	Schermata dei filtri	60
5.7	Schermata dei preferiti	61
5.8	Schermata con la lista degli utenti con cui chattare	61
5.9	Schermata della chat tra due utenti	62
5.10	Informazioni sull'utente con cui si sta chattando	62
5.11	Schermata relativa al proprio profilo	63
5.12	Schermata per la modifica dell'email	64
5.13	Schermata per la modifica della password	64
5.14	Schermata per aggiornare le informazioni del proprio profilo	65
6.1	Home Page di Immobiliare.it	69
6.2	Schermata con informazioni dettagliate di un'abitazione (Immobiliare)	69
6.3	Schermata per la pubblicazione di un annuncio (Immobiliare)	70
6.4	Home Page di Idealista.it	70

6.5	Schermata con informazioni dettagliate di un'abitazione (Idealista)	71
6.6	Schermata per la pubblicazione di un annuncio (Idealista)	71
6.7	Tipologie di immobile in Immobiliare	72

A fronte di un'esigenza personale, io ed un mio collega abbiamo deciso di sviluppare un'applicazione che permettesse agli studenti universitari di cercare (o comunque semplificare ed ottimizzare le metodologie di ricerca) un appartamento in affitto in una città di interesse. A causa, però, della scarsità di risorse reperibili e messe a disposizione da siti esterni, il prodotto finale è un'applicazione che fornisce la visualizzazione, ed altre funzioni, di appartamenti affittabili, ma non la possibilità, ad esempio, di inserire annunci. Le nostre informazioni provengono dal sito Idealista, uno dei principali portali del mercato immobiliare, che ci ha gentilmente fornito l'accesso alle sue risorse.

La scelta della realizzazione di un'applicazione deriva da più fattori. Innanzitutto, al giorno d'oggi, la stragrande maggioranza delle persone ha a disposizione uno smartphone e, quindi, risulta estremamente comodo accedere direttamente a delle funzionalità di interesse. Oltre a ciò, entrando nello specifico, le applicazioni mobili sono generalmente più veloci da caricare rispetto, ad esempio, ai siti web, in quanto possono archiviare dati e risorse direttamente sul dispositivo dell'utente. Esse risultano spesso più coinvolgenti per il fatto che possono sfruttare le interazioni tattili e gestuali del touchscreen. Infine, l'esperienza dell'utente è migliorata anche da un'altissima personalizzazione delle schermate, dall'invio di notifiche, che possono aiutare a mantenerlo informato e impegnato con l'applicazione, e da sicurezza elevata, grazie a misure di crittografia avanzate e controllo degli accessi.

A supporto di questo nostro progetto, abbiamo utilizzato la nuovissima tecnologia Swift, che permette la realizzazione di applicazioni native per dispositivi Apple. Esistono diversi linguaggi di programmazione che permettono di creare applicazioni ibride, ovvero utilizzabili da sistemi operativi diversi, ma quelle native risultano sempre più veloci ed efficienti. Swift è diventato un linguaggio di spicco per gli sviluppatori iOS grazie alla combinazione di prestazioni, sicurezza, facilità d'uso e supporto da parte di Apple; è un linguaggio versatile che rende più semplice la creazione di app iOS di alta qualità.

In questo elaborato sono stati analizzati molti tra gli aspetti più importanti dello sviluppo di un'applicazione. In una prima fase viene indicata tutta la progettazione, a partire dalla raccolta dei requisiti, che esprimono cosa l'applicazione dovrà fare e i suoi vincoli, e dalla realizzazione dei casi d'uso, che, invece, esprimono come l'applicazione dovrà svolgere le varie funzionalità, fino ad arrivare all'implementazione vera e propria, in cui verranno analizzati gli aspetti di logica che stanno dietro il funzionamento dell'applicazione. Successivamente, verranno riportati un manuale per l'utilizzo dell'applicazione, e l'analisi SWOT del sistema realizzato, per la valutazione delle condizioni interne ed esterne all'applicazione.

La presente tesi è composta da sette capitoli strutturati come di seguito specificato:

- Nel Capitolo 1 verrà effettuata un'introduzione generale sul linguaggio di programmazione utilizzato, ovvero Swift, sui suoi framework e sull'IDE di sviluppo XCode.
- Nel Capitolo 2 si entrerà nella parte progettuale dell'applicazione, partendo con la raccolta dei requisiti e la realizzazione dei casi d'uso.
- Nel Capitolo 3 si proseguirà con ciò che è iniziato nel Capitolo 2, focalizzandoci, in particolar modo, sull'interazione dell'utente con l'applicazione.
- Nel Capitolo 4 verranno riportati frammenti di codice, in modo da indicare esempi di realizzazione e di particolare significato.
- Nel Capitolo 5 verrà realizzato un manuale per gli utenti, in modo da permettere a qualsiasi utente di sfruttare al meglio l'applicazione.
- Nel Capitolo 6 verrà effettuata un'analisi del sistema realizzato, evidenziando i punti di forza, di debolezza, le opportunità e le minacce.
- Infine, nel Capitolo 7, verrà realizzato un resoconto di ciò che è stato trattato nei capitoli precedenti e verranno delineati dei possibili sviluppi futuri dell'applicazione.

In questo capitolo iniziale si cerca di dare una visione generale della tecnologia Swift. Di seguito sono riportati principalmente gli aspetti base del linguaggio, cercando di spiegarli nel modo più semplice ed esaustivo possibile. Oltre al linguaggio di programmazione in sè, viene anche trattato il nuovissimo framework SwiftUI, che rende ancora più semplice lo sviluppo delle interfacce utente di applicazioni per dispositivi Apple. Nella parte finale del capitolo, è presente una breve descrizione dell'ambiente di sviluppo utilizzato, XCode, indicandone le caratteristiche principali.

1.1 Swift

Swift è un linguaggio di programmazione orientato agli oggetti per sistemi iOS, iPadOS, macOS, watchOS, tvOS e Linux, presentato dalla casa produttrice Apple. Permette, quindi, di sviluppare applicazioni *native* per dispositivi dell'azienda di Cupertino¹. Un'applicazione viene detta nativa quando viene realizzata specificamente per un sistema operativo², come può essere, ad esempio, iOS, ovvero l'OS³ di un iPhone. I vantaggi che porta la programmazione nativa sono molteplici, a partire dalla maggiore velocità e fluidità dell'applicazione stessa fino ad arrivare all'estrema facilità di accesso alla maggior parte delle componenti, come fotocamera e microfono, e funzionalità del nostro dispositivo.

1.1.1 Un po' di storia

Swift è un linguaggio di programmazione giovane. Infatti il suo sviluppo inizia nel 2010, guidato da Chris Lattner, un programmatore informatico della Apple, e viene ufficialmente pubblicato il 9 settembre 2014 (Figura 1.1). Il nome originale è, in realtà, Shiny, dall'inglese "brillante", con il quale il suo creatore voleva indicare qualcosa che sarebbe stata tanto nuovo quanto brillante. Swift nasce con il proposito di essere più intuitivo e "rapido"⁴, prendendo vari aspetti da altri linguaggi già esistenti sul mercato, come Objective-C e Python. I fattori, che rendono possibili maggiori prestazioni in termini di velocità, sono peculiarità, quali la gestione *ottimale ed automatica* della memoria del dispositivo; una memoria piena è, infatti, spesso causa di un rallentamento generale di tutto il dispositivo.

¹Apple è un'azienda con sede a Cupertino, in California.

²In parole povere, un sistema operativo è ciò che sta alla base dell'operatività di un calcolatore.

³Acronimo che sta per Operative System (Sistema Operativo).

⁴La traduzione in italiano di "swift" è "rapido".



(a) Chris Lattner



(b) Logo Swift

Figura 1.1: Il fondatore e il logo di Swift

Swift è un linguaggio in continua evoluzione: l'ultima versione è stata rilasciata il 30 marzo 2023. Per rendere più efficiente lo sviluppo di applicazioni e per far sì che sia soggetto a continuo miglioramento, Swift è stato reso open-source⁵ dal 2015.

1.1.2 Caratteristiche generali

Swift è un linguaggio di programmazione general-purpose⁶ creato utilizzando un approccio moderno alla sicurezza, alle prestazioni e ai modelli di progettazione del software. Una delle caratteristiche di spicco è, però, la semplificazione della scrittura e della manutenibilità dei programmi per lo sviluppatore. Per raggiungere questo obiettivo, Swift si basa sui concetti di sicurezza, chiarezza ed espressività.

Variabili

Una variabile è un contenitore di dati situato in una porzione di memoria destinata a contenere valori. In Swift le variabili possono essere dichiarate in due modi:

- `var`: serve per dichiarare una variabile *mutable*, ovvero il suo valore può cambiare durante la fase di compilazione del codice;
- `let`: serve per dichiarare una variabile *immutable*, ovvero il suo valore non può cambiare durante la fase di compilazione del codice. In pratica serve per dichiarare una costante.

Lo scope di una variabile è statico, quindi, una volta assegnata la parola `var` o `let`, questa non può essere cambiata. Una buona norma è quella di preferire l'utilizzo di costanti alle variabili, in quanto questo rende l'intento dello sviluppatore più chiaro ed il compilatore Swift sarà in grado di effettuare ulteriori ottimizzazioni in fase di compilazione.

Un'ulteriore caratteristica delle variabili in Swift è la *case-sensitivity*, ovvero la sensibilità alle maiuscole. Questo significa che due variabili, dichiarate con lo stesso nome ma con caratteri minuscoli/maiuscoli diversi, sono differenti (Figura 1.2).

⁵Non protetto da copyright e liberamente modificabile dagli utenti.

⁶Caratterizzato da una certa versatilità, adatti a molti impieghi e non specializzati per particolari esigenze.



Figura 1.2: Variabili case-sensitive

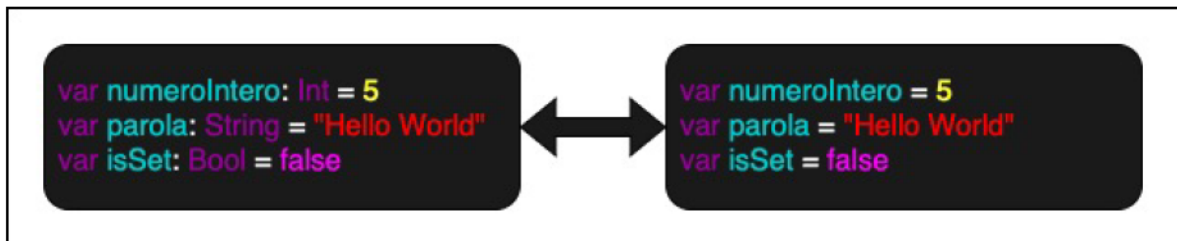


Figura 1.3: Funzionamento del type inference implicito

Tipi di dato

Swift è un linguaggio fortemente tipizzato: il compilatore deve essere sempre in grado di determinare, sia esplicitamente che implicitamente, con il meccanismo di *type inference* (Figura 1.3), il tipo di dato di variabili, costanti, argomenti e risultati di funzione in ogni istante, pena il fallimento della compilazione. Il type inference implicito consiste nell'assegnamento di tipo alla variabile tramite il valore con cui viene inizializzata. Swift è, dunque, un linguaggio *type safe*, caratteristica che permette di prevenire tutta una serie di errori di esecuzione già durante la compilazione.

I tipi di dato presenti in Swift sono quelli classici e vengono riportati nella Figura 1.4.

Funzioni

Una funzione è una sequenza di istruzioni raggruppate in un unico blocco, espletando così una specifica operazione sui dati del programma stesso in modo tale che, a partire da determinati input, restituisca determinati output.

In Swift una funzione viene definita tramite la parola chiave `func`, seguita dal nome della funzione, dai parametri, dal tipo del valore di ritorno e dal corpo della funzione. Dato che Swift è fortemente tipizzato, per ogni parametro della funzione bisogna specificarne il tipo.

Dato	Tipo di Dato
Numero	Int : numeri interi (sia positivi che negativi) Float : numeri in virgola mobile (32 bit) Double : numeri in virgola mobile (64 bit)
Testo (carattere, parola, insieme di parole)	Character : solo carattere singolo String : sia carattere singolo, sia parola, sia testo
Predicato Logico	Bool : può assumere valore true o false

Figura 1.4: Tipi di dato comuni presenti in Swift

Definizione di una funzione:

```
func nome_funzione (nomeParametro1: tipoParametro1...) -> tipo_valore_di_ritorno {  
    // corpo della funzione  
    // return valore_di_ritorno  
}
```

Chiamata di una funzione:

```
nome_funzione (nomeParametro1: valoreParametro1)
```

Figura 1.5: Definizione e chiamata di una funzione in Swift

Per invocare una funzione, basta usarne il nome con lo stesso numero e tipo di parametri della sua definizione (Figura 1.5).

Classi ed oggetti

Swift è un linguaggio basato sul paradigma di programmazione orientata agli oggetti. Una classe, nella OOP⁷, è un costrutto usato come modello per creare oggetti e comprende attributi e metodi che saranno condivisi da tutti gli oggetti creati (istanze) a partire dalla classe. Un oggetto è, di fatto, un'istanza di una classe. Un metodo speciale presente all'interno di ogni classe è il costruttore, indicato con `init`, esso è incaricato di inizializzare correttamente le proprietà degli oggetti di una classe. In generale una classe permette di definire un nuovo tipo di dato (In Figura 1.6 *Persona* diventa un nuovo tipo di dato).

Null Safety

Una delle caratteristiche di Swift è il fatto di essere un linguaggio sicuro, fornendo opportuni meccanismi per evitare l'esecuzione di codice non corretto, ed il verificarsi di effetti collaterali e inaspettati. Oltre alla forte tipizzazione, un altro meccanismo che rende così sicuro Swift è la *null-safety* tramite le variabili *optional*. Un tipo *optional* permette di indicare che una variabile potrebbe non avere valore (in questo caso il suo contenuto sarà `nil`). Normalmente non può accadere che una variabile contenga valore `nil`; se così fosse, durante la fase di compilazione, si andrebbe incontro alla *Null Pointer Exception (NPE)*, che porterebbe ad un crash dell'applicazione, in quanto si sta tentando di accedere ad una variabile con valore nullo.

Per definire un tipo *optional*, bisogna aggiungere un `?` al nome del tipo che la variabile conterrà nel caso ci sia invece un valore. È come se dicessimo al compilatore: "questa variabile o contiene un valore di questo tipo, oppure non ha valore `nil`". Altri operatori utili per la gestione della Null Safety sono `!` e `??`. Il primo serve per indicare che la variabile su cui viene aggiunto sicuramente avrà valore diverso da `nil` (nel caso in cui lo fosse, sarebbe lanciata la NPE); il secondo serve per specificare un valore alternativo nel caso in cui la variabile assuma valore nullo (Figura 1.7).

Librerie

In informatica, una libreria è un insieme di funzioni o strutture dati predefinite e predisposte per essere collegate ad un'applicazione attraverso un opportuno collegamento, che viene introdotto grazie alla parola chiave `'import'`. Lo scopo delle librerie software è

⁷Acronimo che sta per Object Oriented Programming (Programmazione orientata agli oggetti).

Definizione di una classe:

```
class Persona {  
  
    //Attributi della classe  
    let nome: String  
    let cognome: String  
  
    //Metodo della classe  
    func nomeCompleto() {  
        return self.nome + " " + self.cognome  
    }  
  
    //Inizializzatore della classe  
    init(nome: String, cognome: String) {  
        self.nome = nome  
        self.cognome = cognome  
    }  
}
```

Creazione di oggetti:

```
let p1 = Persona(nome: "Antonio", cognome: "Calanducci")  
var p2 = Persona(nome: "Steve", cognome: "Jobs")
```

Figura 1.6: Classi e Oggetti in Swift**Dichiarazione di una variabile optional:**

```
var year: Int?
```

Unwrap di un optional:

```
print (year!)
```

Nil Coalescing:

```
var year: Int ?? 2001
```

Figura 1.7: Operatori per la gestione della Null Safety

fornire una collezione di *entità di base pronte per l'uso*, ovvero riuso di codice, evitando al programmatore di dover riscrivere ogni volta le stesse funzioni o strutture dati e facilitando, così, le operazioni di sviluppo e manutenzione.

Swift mette a disposizione un elevatissimo numero di librerie. Quella principale è la libreria `Foundation` che consente di importare tutta una serie di funzionalità alla base della maggior parte delle applicazioni. Ad esempio, essa permette l'uso delle classi primitive (per intenderci, i tipi di dato) oppure, parlando in un contesto più ampio, di interagire tramite URL e comunicare con altri server usando i protocolli standard di comunicazione.

Interoperabilità con il linguaggio C

Abbiamo precedentemente detto che Swift è stato rilasciato il 9 settembre 2014, ma l'azienda Apple produce i suoi dispositivi da molti più anni. *Objective-C* è rimasto a lungo il linguaggio di programmazione di riferimento per Apple e lo sviluppo di Swift è stato ufficializzato con la nascita di un team dedicato guidato da Chris Lattner.

Come lo stesso nome suggerisce, *Objective C* è un'estensione a oggetti del linguaggio C, mantenendo la completa compatibilità col C. Per evitare problemi di compatibilità con vecchi dispositivi, Swift è stato progettato per coesistere con il linguaggio *Objective-C*. In particolare, è possibile l'uso di codice *Objective C*, *Objective C++* e Swift in un singolo programma. Al momento della pubblicazione di Swift, per sostenere questa interoperabilità tra linguaggi, uno dei lavoratori del team di sviluppo sostenne che Swift poteva essere immaginato come "Object C, ma senza il C". Ovviamente, essendo un linguaggio di programmazione più nuovo e progettato ad hoc, il codice Swift risulta più veloce ed efficiente del suo predecessore.

Confronto con Java

Java, insieme a suoi successori come Kotlin, è un linguaggio di programmazione utilizzato nella programmazione mobile per sistemi operativi Android. I linguaggi di programmazione possono essere fondamentalmente di tipo compilato o di tipo interpretato. La differenza di base fra i due è che, mentre nei linguaggi compilati il programma è rappresentato da un file già scritto in binario e, una volta lanciato, viene letto direttamente ed eseguito, nei linguaggi interpretati il programma è rappresentato dal codice sorgente che ogni volta viene prima tradotto in un codice intermedio e poi compilato, con un conseguente calo di prestazioni.

Swift, rispetto a Java, non necessita di essere interpretato. Questa differenza fondamentale tra i due linguaggi di programmazione ha una notevole ripercussione sulla RAM (memoria). Infatti il sistema Android ne risulta svantaggiato rispetto al rivale iOS.

1.2 SwiftUI

SwiftUI (Figura 1.8) è un framework di sviluppo realizzato da Apple e scritto interamente in Swift. È stato pubblicato il 3 giugno 2019 ed il suo scopo è quello di facilitare la costruzione di interfacce utente. Mette a disposizione viste⁸, strutture di layout e molto altro, come ad esempio gestori di eventi per rilevare tocchi, gesti e altri tipi di input, oppure strumenti per gestire il flusso di dati dai modelli dell'app fino alle viste e ai controller⁹ che gli utenti vedono e con cui interagiscono. Dal suo rilascio in poi, SwiftUI ha guadagnato una notevole popolarità tra gli sviluppatori grazie alla sua semplicità, flessibilità e integrazione perfetta con il linguaggio di programmazione Swift.

⁸Interfacce per gli utenti.

⁹Componente di un'applicazione che gestisce la sua logica.



Figura 1.8: Logo di SwiftUI

1.2.1 Caratteristiche Generali

Prima di SwiftUI per la costruzione di schermate veniva usata la libreria UIKit, interamente scritta in Objective-C; questa libreria obbligava a scrivere un gran numero di linee di codice solo per far funzionare la componente di interfaccia, prescindendo dalle funzionalità. SwiftUI mette a disposizione delle componenti utilizzabili e direttamente funzionanti. Inoltre, questo nuovo framework utilizza un approccio dichiarativo piuttosto che uno imperativo. La programmazione imperativa è un paradigma di programmazione che utilizza istruzioni che modificano lo *stato* del programma e si concentra su *come* quest'ultimo dovrebbe raggiungere il risultato; invece quella dichiarativa esprime la logica di un calcolo senza descriverne il flusso di controllo e si concentra su *cosa* il programma dovrebbe realizzare (Figura 1.9). Nell'immagine relativa all'approccio dichiarativo (Figura 1.9), abbiamo consigliato alla `list` di mostrare gli elementi presenti dentro `model.list`. Ad ogni elemento chiediamo di creare una row con un'Image ed un elemento Text.

Un'altra peculiarità che ha reso SwiftUI così amato dai suoi sviluppatori è Live Rendering, che verrà approfondito nella prossima sezione.

1.2.2 Live rendering

Il live rendering è quel processo che permette di compilare ed iniettare un cambiamento in maniera pressoché istantanea nell'app in esecuzione.

È, sicuramente, una delle funzionalità più comode di SwiftUI, in quanto fornisce un'anteprima *in tempo reale* dell'interfaccia utente durante la codifica. Gli sviluppatori possono vedere le modifiche apportate al codice riflesse nell'anteprima, facilitando la prototipazione rapida, senza dover ogni volta compilare tutto il codice. La tecnologia che rende possibile tutto questo è una feature del linguaggio Swift che si chiama *Dynamic Replacement*.

1.2.3 Property wrapper

Prima di parlare dei property wrapper, è utile dare la definizione di *stato* di un'applicazione. Lo stato di un programma è definito come la sua condizione corrente relativa agli input memorizzati. Per fare un esempio più familiare, prendiamo in considerazione la chimica: lo stato di un oggetto, come un gas, liquido o solido, mostra la sua attuale composizione fisica. Allo stesso modo, lo stato di un programma mostra la sua condizione corrente rispetto a valori o contenuti; gli input memorizzati in un'applicazione sono salvati all'interno di variabili o costanti.

Nell'analizzare lo stato di un programma, gli sviluppatori possono esaminare e *osservare* i valori memorizzati in questi input. SwiftUI utilizza un design pattern¹⁰, chiamato "*Observer*", che definisce una dipendenza tra oggetti, in modo tale che, se un oggetto cambia stato, tutti gli oggetti da esso dipendenti vengono notificati e aggiornati automaticamente.

¹⁰Un pattern può essere definito "una soluzione progettuale generale ad un problema ricorrente".

```
var list: [User]
list = getList()
if (list.isEmpty) {
    showErrorAlert()
} else {
    refreshTableView() // chiama il cellForRow
}
func cellForRowAt(...) {
    let model = list[indexPath.row]
    cell.label.text = model.name
    cell.label.textColor = UIColor.black
    cell.image = model.image // lo stato dell'oggetto cell è mutabile
    //
    //
    return cell
}
```

(a) Approccio Imperativo

```
@State var model = Users.listModel
List(model.list) { item in
    Image(item.image)
    Text(item.name)
        .fontWeight(.light)
        .color(.black) // l'oggetto Text è immutabile, viene ridefinito ad ogni modifica
}
```

(b) Approccio Dichiarativo

Figura 1.9: Approccio Imperativo vs Approccio Dichiarativo

Arrivati a questo punto, possiamo dare la definizione di property wrapper. Un property wrapper è una struttura che definisce come una proprietà viene memorizzata o calcolata durante la lettura. Un property wrapper viene introdotto tramite l'uso del simbolo @.

@State

SwiftUI utilizza @State per consentire di modificare i valori all'interno di una struct¹¹ (modificandone lo stato), cosa che normalmente non sarebbe consentita perché le struct, a differenza delle classi, sono value type, ovvero, durante un'operazione di assegnamento, viene copiato l'intero blocco di memoria occupato dalle proprietà dell'istanza, in una nuova area di memoria assegnata alla nuova variabile. Le classi, invece, sono reference type e, quindi, ogni operazione di assegnamento avrà ripercussione sull'istanza originale, in quanto tutte le istanze condividono lo stesso indirizzo di memoria. Mentre le viste sono volatili, lo stato è un archivio persistente creato da SwiftUI per conto della vista. SwiftUI può distruggere e ricreare una vista ogni volta che è necessario senza perdere lo stato memorizzato dalla proprietà. Quando il valore dello stato cambia, la vista ne invalida l'aspetto e ricalcola il corpo.

@Binding

Il property wrapper @Binding consente di connettere una variabile @State tra la vista principale e una vista secondaria. Ciò può tornare utile per qualsiasi situazione in cui desideriamo aggiornare una vista principale dall'interno della vista secondaria. Per creare un'associazione di questo tipo occorre passare una proprietà @State tramite il prefisso \$; in questo modo passiamo il riferimento all'indirizzo di memoria della variabile. Nella Figura 1.10 riportiamo un esempio di funzionamento del binding: la binding wrapped property value viene passata al valore dello slider in modo che questa proprietà venga aggiornata quando lo slider cambia il suo valore; in questo modo MySlider mantiene un riferimento del valore dello slider. Nella vista padre (ContentView) abbiamo una proprietà state sliderValue per mantenere memoria del valore dello slider al di fuori della vista figlio. Quando viene creato lo Slider creiamo un'associazione passando la proprietà sliderValue con il prefisso \$ e creando così un riferimento alla proprietà State. MySlider View non si limita a leggere il valore di sliderValue, ma ne scrive anche il valore. MySlider può leggere e scrivere il valore della proprietà sliderValue di ContentView, ma non può osservare le modifiche utilizzando questa associazione (a differenza di @State). Non appena MySlider modifica il valore della proprietà sliderValue, SwiftUI ricreerà ContentView e quindi MySlider come figlio. Questo è ciò che viene chiamato *two-way binding*, perché qualsiasi modifica al valore di sliderValue aggiornerà lo slider nella vista figlio e qualsiasi modifica allo slider aggiornerà sliderValue nella vista genitore.

@StateObject e @ObservedObject

Abbiamo precedentemente parlato dell'annotazione @State, ma questa ha un grande limite: funziona soltanto con i tipi di dato di base (per intenderci Int, String, Bool...). Introducendo le classi, abbiamo, infatti, detto che queste permettono di definire nuovi tipi di dati. Fortunatamente SwiftUI fornisce le annotazioni @StateObject e @ObservedObject per validare il meccanismo che sta dietro a @State anche per altri tipi di dato. In particolare:

- Per utilizzare @ObservedObject, è, innanzitutto, necessario che la classe di riferimento sia conforme al protocollo @ObservableObject (deve essere dichiarato nell'in-

¹¹Struttura dati simile ad una classe.

```

struct ContentView: View {
    @State private var sliderValue: CGFloat = 50
    private var minSliderValue: CGFloat = 10
    private var maxSliderValue: CGFloat = 100
    var body: some View {
        VStack {
            Text("Slider value: \(Int(sliderValue))")
            MySlider(value: $sliderValue,
                    minValue: minSliderValue,
                    maxValue: maxSliderValue)
        }
    }
}

```

(a) Parent View

```

struct MySlider : View {
    @Binding var value: CGFloat
    var minValue: CGFloat
    var maxValue: CGFloat
    var body: some View {
        HStack {
            Text("\(Int(minValue))")
                .foregroundColor(.blue)
            Slider(value: $value, in: minValue...maxValue, step: 1)
            Text("\(Int(maxValue))")
                .foregroundColor(.blue)
        }
        .padding()
    }
}

```

(b) Child View

Figura 1.10: Funzionamento di @Binding

testazione della classe). All'interno della classe le proprietà che devono essere osservate vanno affiancate dall'annotazione `@Published`. All'esterno della classe, invece, l'istanza che deve essere osservata è preceduta dall'annotazione `@ObservedObject`.

- `@StateObject` è una versione specializzata di `@ObservedObject` e funziona quasi esattamente allo stesso modo; è necessario conformarsi al protocollo `@ObservableObject`, si può utilizzare `@Published` per contrassegnare le proprietà come causa di notifiche di modifica e tutte le viste che contengono un `@StateObject` aggiorneranno il loro corpo quando l'oggetto cambia. La differenza tra `@StateObject` e `@ObservedObject` è che mentre il primo immagazzina i dati all'interno della view, divenendo parte integrante di essa, il secondo crea una nuova istanza ogni volta che la view viene ricostruita. Questo significa che con uno `@StateObject` la vista non perde i dati ogni volta che viene distrutta e poi ricostruita, mentre con un `@ObservedObject` i dati vengono riportati al loro valore iniziale.

@EnvironmentObject

L'annotazione `@EnvironmentObject` serve a condividere dati, e, quindi, variabili, tra tutte le viste dell'applicazione. Ogni vista può, infatti, accedere a variabili precedute dall'annotazione `@EnvironmentObject` semplicemente riferendosi ad esse. Poiché tutte le viste puntano alla stessa variabile, se una vista la modifica, tutte le viste si aggiornano immediatamente; in questo modo non c'è il rischio di sincronizzare dati in modo diverso in parti diverse dell'app.

1.3 XCode

XCode (Figura 1.11) è l'IDE ufficiale della Apple. Con IDE¹² si intende un *ambiente di sviluppo integrato*, ovvero un software che, in fase di programmazione, supporta i programmatori nello sviluppo e nel debugging del codice sorgente di un programma. Generalmente un IDE mette a disposizione:

- un editor di codice sorgente che, generalmente, fornisce la syntax highlighting, rendendo più semplice la visualizzazione delle strutture ed il completamento automatico del codice;
- un compilatore e/o un interprete (in questo caso, un compilatore);

¹²Integrated development environment.



Figura 1.11: Logo di XCode

- un tool di building automatico;
- un debugger.

Oltre a questo, XCode fornisce anche strumenti per la fase di testing di un'applicazione, un canvas, con cui è possibile visualizzare il Live Rendering, strumenti per la distribuzione della propria applicazione su tutte le piattaforme Apple, e molto altro. E se il canvas non fosse abbastanza, su XCode è possibile anche impostare dei simulatori che riproducono virtualmente un qualsiasi dispositivo Apple, ed il comportamento dell'applicazione visualizzato sul simulatore sarà identico a quello su un dispositivo vero e proprio. Inoltre, XCode dialoga direttamente con GitHub¹³ e, quindi, è possibile collegarsi ai propri repository¹⁴ remoti, condividendo il lavoro svolto tenendo traccia del versioning¹⁵ dell'applicazione.

¹³Piattaforma basata su cloud per lo sviluppo di software e controllo della versione tramite Git, che consente agli sviluppatori di archiviare e gestire il proprio codice.

¹⁴Luogo centrale in cui vengono archiviati i dati.

¹⁵Gestione di versioni multiple di una stessa informazione.

Specifica e Analisi dei Requisiti

In questo capitolo iniziamo ad entrare in un ambito più tecnico e progettuale della nostra applicazione. Il capitolo si apre spiegando cos'è l'ingegneria del software (la nostra applicazione è un software) e dando la definizione di ciclo di vita di un progetto. Tutto ciò sarà importante in quanto il capitolo, e tutto il documento a seguire, approfondiranno le prime due fasi del ciclo di vita della nostra applicazione. In particolare, in questo capitolo, ci soffermeremo di più sulla parte di avvio e sull'inizio della parte di elaborazione, che comprendono la parte di analisi dei requisiti e la modellazione dei casi d'uso.

2.1 Che cos'è l'ingegneria del software

L'ingegneria del software è quella disciplina informatica che si occupa dei *processi produttivi e delle metodologie di sviluppo* finalizzati alla realizzazione di sistemi software. Un software è l'insieme delle istruzioni di un programma codificate in linguaggio macchina o in linguaggio di programmazione sotto forma di codice eseguibile. L'ingegneria del software si propone una serie di obiettivi legati all'evoluzione dello sviluppo del software sia da un punto di vista tecnologico che metodologico. Ovviamente, dietro ad ogni realizzazione di software, è presente tutta una fase progettuale. Concentrandoci in particolar modo sull'aspetto metodologico della realizzazione del software, è utile definire il ciclo di vita (Figura 2.1) di un progetto, ovvero un processo che segue l'intero sviluppo del progetto, dall'inizio fino al suo completamento. Tale ciclo di vita si compone di quattro fasi:

- *Avvio*, con il compito di stabilire la fattibilità del progetto e fissare i requisiti essenziali;
- *Elaborazione*, con il compito di creare una baseline del progetto¹, fissare la maggior parte dei requisiti funzionali, dei requisiti non funzionali e dei casi d'uso (vedremo in seguito cosa sono);
- *Costruzione*, con il compito di completare tutti i requisiti e i casi d'uso, nonchè costruire un'architettura di progetto stabile e completa;
- *Transizione*, con il compito di eliminare difetti scoperti in seguito ad un beta test² del progetto e di eseguire un riepilogo di fine progetto. Dopo questa fase si ha il rilascio ufficiale dell'applicazione agli utenti.

¹Linea guida del progetto.

²Versione di un software non definitiva, ma già testata dagli esperti, che viene messa a disposizione di un numero maggiore di utenti, confidando proprio nelle loro azioni imprevedibili che potrebbero portare alla luce nuovi bug o incompatibilità del software stesso.



Figura 2.1: Ciclo di vita di un progetto

In particolare, noi analizzeremo la prima e la seconda fase; quest'ultima, a sua volta, si divide in:

- *Requisiti*, che serve per portare alla luce tutti i requisiti mancanti.
- *Analisi*, che serve per ultimare il modello di analisi.
- *Progettazione*, che serve per ultimare il modello di progettazione con dei diagrammi (che approfondiremo in seguito).
- *Implementazione*, che serve per costruire la capacità operativa iniziale (codice eseguibile).
- *Test*, che serve per verificare le capacità operative iniziali.

2.2 UML

UML è un linguaggio visuale di modellazione dei sistemi di uso generico. In genere lo si associa a modellazione di sistemi software Object Oriented. È stato studiato per incorporare tutte le migliori pratiche di modellazione e di ingegneria del software. UML non fornisce alcuna metodologia di modellazione, ma esclusivamente la sintassi da utilizzare per la costruzione di modelli. UML ha due aspetti:

- *Struttura statica*, che riguarda i tipi di oggetti necessari per modellare il sistema e come sono tra loro correlati;
- *Struttura dinamica*, che riguarda il ciclo di vita di questi oggetti e come collaborano per fornire le funzionalità richieste al sistema.

La struttura di UML è composta da:

- *Costituenti fondamentali*: entità, relazioni e diagrammi di base di UML;
- *Meccanismi comuni*: tecniche per raggiungere specifici obiettivi con UML
- *Architettura*: modo in cui UML esprime l'architettura del sistema



Figura 2.2: Sintassi per esprimere un requisito

2.3 Requisiti

L'ingegneria dei requisiti è la disciplina che si occupa della raccolta, della documentazione e del mantenimento dei requisiti. Un requisito può essere definito come una specifica che dovrebbe essere implementata; è, sostanzialmente, un'affermazione di *cosa* il sistema dovrebbe fare e non di come lo deve fare. UML non fornisce alcuna indicazione su come scrivere i requisiti, ma, per la loro formulazione, si usa la convenzione riportata in Figura 2.2. Essendo la fase di raccolta dei requisiti frutto di un'intervista ai committenti dell'applicazione, è buona prassi, prima di passare alla schematizzazione, fornire una descrizione in linguaggio naturale degli stessi.

2.3.1 Requisiti in linguaggio naturale

Il progetto consiste nella realizzazione di un'applicazione che permetta agli studenti universitari di trovare un appartamento in affitto in una città indicata. Ogni utente, per utilizzare per la prima volta l'applicazione, è tenuto a registrarsi, inserendo una e-mail e una password valida. Questa operazione deve essere svolta ogni volta che si apre l'applicazione. Invece, nel caso in cui l'utente sia già registrato, dovrà autenticarsi, sempre tramite email e password, in modo da entrare nell'applicazione con il proprio profilo e recuperare le tutte le informazioni ad esso associate. Le informazioni personali dell'utente, quali email, password, nome, cognome, numero di telefono e una breve descrizione personale, possono essere aggiunte, rimosse o modificate.

Una volta all'interno dell'applicazione, l'utente potrà scegliere, tramite barra di ricerca o click sulla mappa, la città di interesse per la ricerca delle case. La ricerca del luogo e degli appartamenti verrà effettuata utilizzando le risorse messe a disposizione rispettivamente da Google Maps e da Idealista. Al termine della ricerca, oltre che sulla mappa, gli appartamenti, insieme a tutte le informazioni ad essi relativi, saranno visualizzabili dall'utente in una lista.

Nel caso in cui l'utente abbia delle particolari esigenze, la ricerca può essere filtrata mediante parametri inseriti dallo stesso, in modo da limitare i risultati secondo le preferenze dell'utente. Se questo trova un'abitazione di proprio interesse, può salvarla tra i preferiti. L'utente potrà visualizzare una lista con tutte le abitazioni salvate e con tutte le informazioni ad esse relative, recuperabili cliccando sull'abitazione in questione.

Infine, l'utente potrà chattare con altri utenti che hanno salvato la stessa abitazione tra i preferiti. Inoltre l'utente può visualizzare quali abitazioni ha in comune con l'utente con cui sta chattando.

2.3.2 Requisiti funzionali

Come già detto in precedenza, un requisito funzionale specifica cosa farà il sistema. Nel caso della nostra applicazione avremo i seguenti requisiti funzionali:

1. L'applicazione dovrà consentire all'utente di autenticarsi all'avvio: quando avvia l'app, l'utente deve accedere con il proprio account, tramite l'inserimento di email e password, o registrarsi, tramite l'inserimento di email e password.
2. L'applicazione dovrà verificare che l'email sia corretta: ogni utente ha delle credenziali di accesso univoche, che devono essere inserite correttamente.
3. L'applicazione dovrà verificare che la password sia corretta: ogni utente ha delle credenziali di accesso univoche, che devono essere inserite correttamente.
4. L'applicazione dovrà consentire la gestione delle informazioni personali di un utente: ogni utente può aggiungere, modificare e rimuovere le proprie informazioni personali.
5. L'applicazione dovrà permettere all'utente di inserire il centro di ricerca delle abitazioni sulla mappa: l'utente può indicare la città in cui avverrà la ricerca delle abitazioni.
6. L'applicazione dovrà permettere all'utente di inserire dei parametri di ricerca: l'utente può decidere di inserire dei filtri che limitano la ricerca secondo le sue preferenze.
7. L'applicazione dovrà permettere all'utente di visualizzare il risultato della ricerca su mappa: l'utente, dopo aver effettuato una ricerca, deve poterne visualizzare il risultato sulla mappa.
8. L'applicazione dovrà permettere all'utente di visualizzare il risultato della ricerca all'interno di una lista paginata: l'utente, dopo aver effettuato una ricerca, deve poterne visualizzare il risultato sotto forma di una lista di abitazioni.
9. L'applicazione dovrà permettere all'utente di memorizzare le abitazioni di interesse tra i preferiti: l'utente può salvare delle abitazioni che saranno visualizzabili in un'apposita lista.
10. L'applicazione dovrà consentire all'utente di gestire la lista dei preferiti: l'utente può visualizzare ed eliminare abitazioni dalla sua lista dei preferiti.
11. L'applicazione dovrà consentire all'utente di visualizzare, in modo dettagliato, le informazioni relative ad una abitazione: l'utente può visualizzare nel dettaglio le informazioni relative ad ogni abitazione.
12. L'applicazione dovrà consentire all'utente di chattare con altri utenti del sito: l'utente può chattare con altri utenti purchè questi abbiano almeno una abitazione tra i preferiti in comune.
13. L'applicazione dovrà permettere all'utente di visualizzare le abitazioni in comune con gli altri utenti: l'utente potrà visualizzare le abitazioni tra i preferiti in comune con altri utenti.
14. L'applicazione dovrà consentire all'utente di visualizzare le informazioni relative ad un altro utente: l'utente potrà visualizzare le informazioni personali di un altro utente con cui sta chattando.

2.3.3 Requisiti non funzionali

Un requisito non funzionale è un vincolo che il sistema deve rispettare. I requisiti non funzionali specificano, o limitano, come il sistema debba essere implementato. Nel caso della nostra applicazione avremo i seguenti requisiti non funzionali:

1. L'applicazione potrà essere eseguita su dispositivi con una versione di iOS maggiore o uguale alle 13.0: i dispositivi con una versione del sistema operativo inferiore alla 13.0 non possono utilizzare l'applicazione.
2. L'applicazione dovrà essere implementata in Swift 5: il linguaggio di programmazione utilizzato per lo sviluppo dell'app è la quinta versione di Swift.
3. L'applicazione dovrà usare come database Firebase³: tutti i dati di un utente, i messaggi e le abitazioni nei preferiti saranno salvate su Firebase.

2.4 Modellazione dei casi d'uso

La modellazione dei casi d'uso è una tecnica complementare per raccogliere e documentare i requisiti. Infatti, i casi d'uso specificano *come* il sistema raggiunge un certo obiettivo. Un caso d'uso può essere definito come la specifica di una sequenza di azioni, incluse eventuali sequenze alternative, che un sistema può eseguire interagendo con attori esterni. Il modello è costituito da tre elementi:

- *Attore*, ovvero il ruolo assunto da una persona o da una cosa che utilizzano il sistema.
- *Caso d'uso*, ovvero quello che un attore può fare con il sistema.
- *Relazione*, ovvero le relazioni significative tra attori e casi d'uso.

Entriamo un po' più nel dettaglio. Un attore identifica il ruolo che un'entità esterna assume quando interagisce direttamente con il sistema. Gli attori sono sempre esterni al sistema. Quindi, un caso d'uso è qualcosa che un attore vuole che il sistema faccia. Un caso d'uso viene sempre avviato da un attore e descritto dal suo punto di vista. Esistono due tipi di attori: l'attore primario e l'attore secondario. La differenza tra i due è che soltanto l'attore primario può dare inizio al caso d'uso.

Le relazioni sono dei legami che intercorrono tra casi d'uso. La relazione *include* consente di includere il comportamento di un caso d'uso in un punto della sequenza di eventi di un altro caso d'uso (Figura 2.3). La relazione *extend* fornisce, invece, un metodo per aggiungere comportamento ad un caso d'uso esistente (Figura 2.4). La differenza tra le due relazioni è che, mentre nel primo caso il caso d'uso di base è incompleto senza i casi d'uso *inclusi*, nel secondo caso il caso d'uso di base è del tutto completo anche senza le sue *estensioni*.

2.4.1 Rappresentazione grafica dei casi d'uso

Nelle figure 2.5, 2.6, 2.7, 2.8 saranno riportati alcuni diagrammi di casi d'uso relativi alla nostra applicazione.

2.4.2 Casi d'uso in linguaggio naturale

Per ogni caso d'uso riportato nelle Figure 2.5-2.8, viene riportata una descrizione testuale che sottolinei la sequenzialità delle azioni.

³Firebase è una piattaforma online che permette di usufruire di vari servizi per la gestione di dati.



Figura 2.3: Esempio di utilizzo di «include»

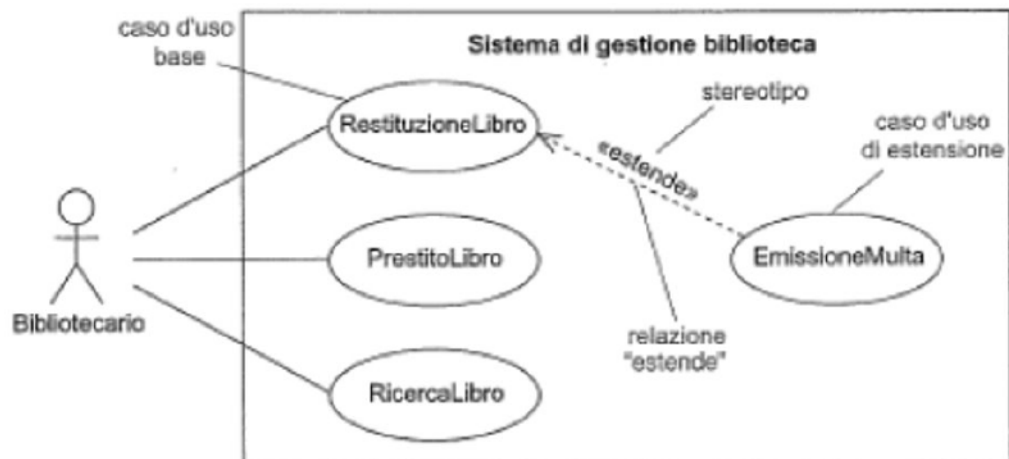


Figura 2.4: Esempio di utilizzo di «extend»

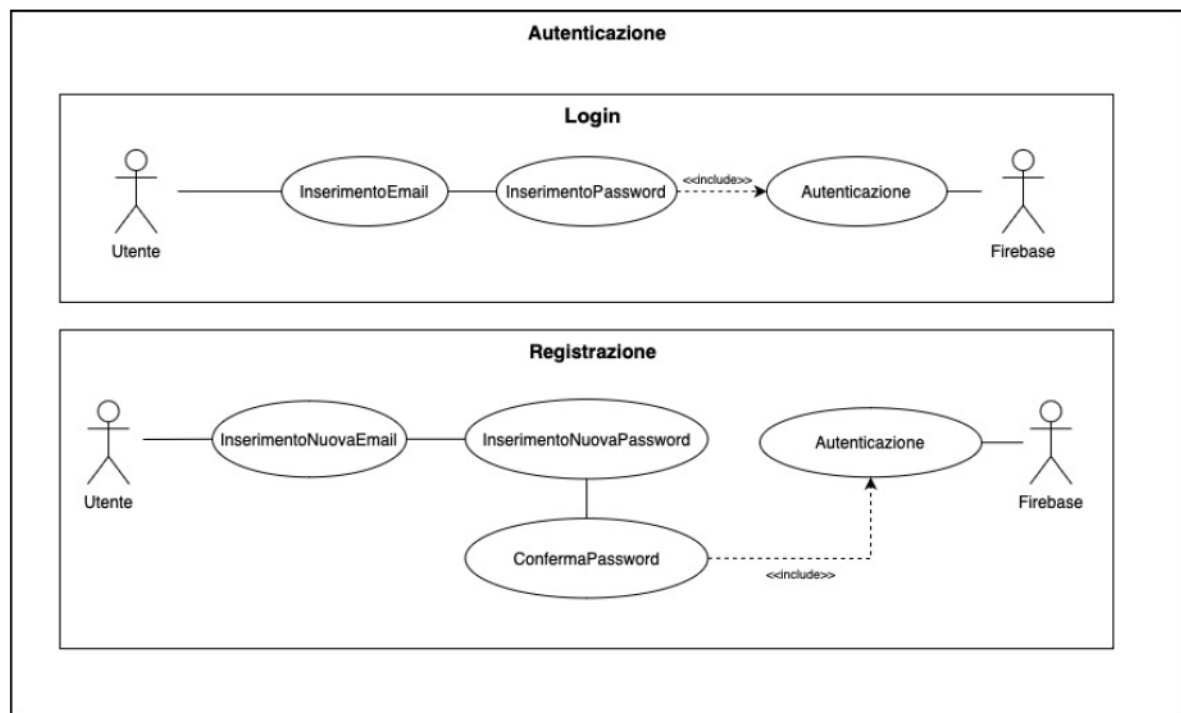


Figura 2.5: Modello dei casi d'uso Autenticazione

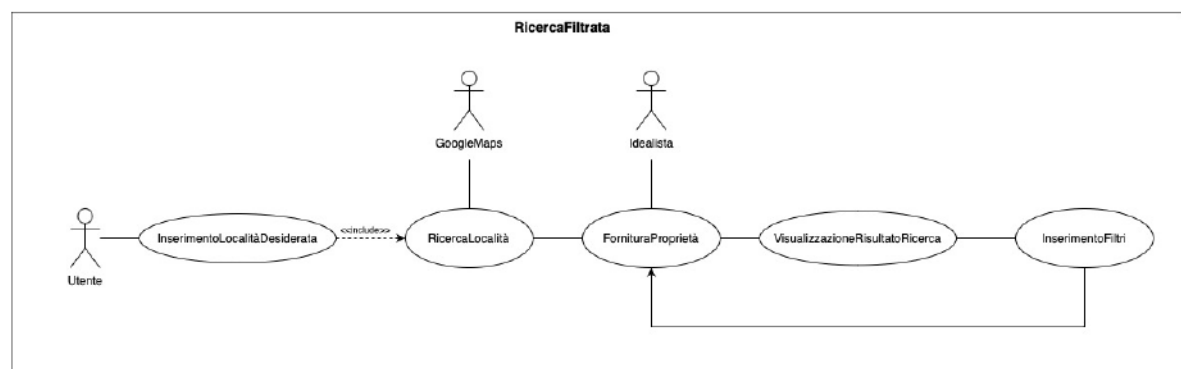


Figura 2.6: Modello dei casi d'uso Ricerca Filtrata

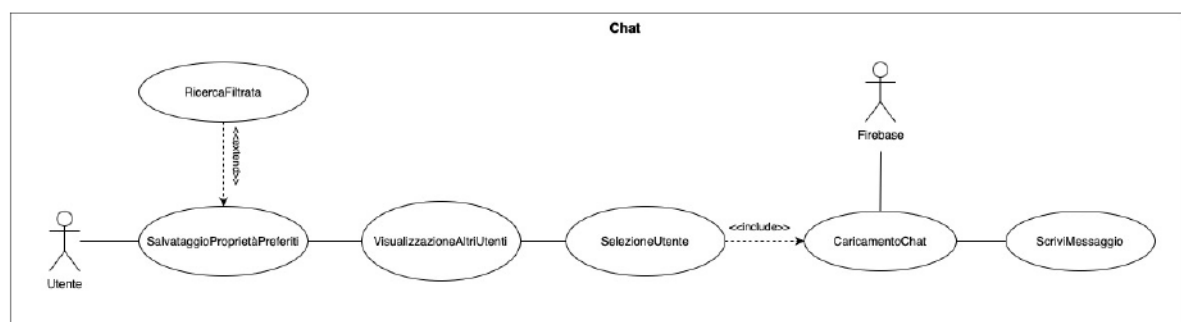


Figura 2.7: Modello dei casi d'uso Chat

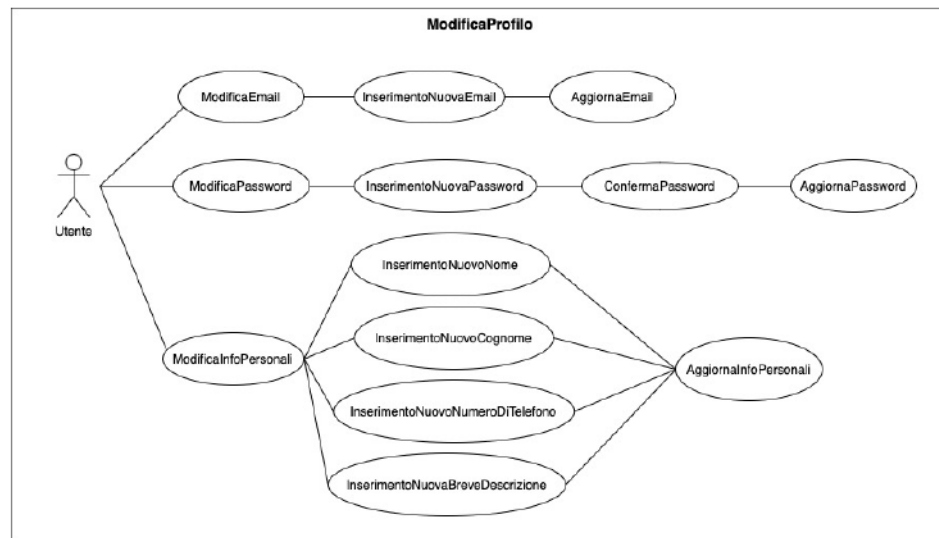


Figura 2.8: Modello dei casi d'uso ModificaProfilo

Autenticazione

Login:

1. Il caso d'uso inizia quando l'utente avvia l'applicazione
2. L'utente inserisce l'email
3. L'utente inserisce la password
4. Se le informazioni sono corrette l'utente viene autenticato (include)

Registrazione:

1. Il caso d'uso inizia quando l'utente avvia l'applicazione per la prima volta
2. L'utente inserisce l'email
3. L'utente inserisce la password
4. L'utente inserisce nuovamente la password per la conferma
5. Se i dati inseriti sono validi, l'utente viene registrato e autenticato (include)

Attore primario: Utente

Attore secondario: Firebase

RicercaFiltrata

1. Il caso d'uso inizia quando l'utente ricerca un appartamento in una città di interesse
2. GoogleMaps fornisce le funzionalità di ricerca della località (include)
3. Idealista fornisce i dati per gli appartamenti
4. L'utente visualizza il risultato della ricerca
5. Se l'utente ha delle esigenze particolari, può inserire dei filtri che limitano la ricerca in base alle sue preferenze

6. Idealista fornisce i dati delle abitazioni in base ai valori dei parametri di filtraggio

Attore primario: Utente

Attori secondari: GoogleMaps, Idealista

Precondizioni: L'utente deve essere autenticato

Chat

1. Il caso d'uso inizia quando l'utente, in seguito ad una ricerca, salva un'abitazione tra i preferiti
2. RicercaFiltrata (extend)
3. L'utente può visualizzare altri utenti che hanno le stesse abitazioni tra i preferiti
4. L'utente può selezionare un altro utente con cui chattare
5. Firebase carica la chat (include)
6. L'utente può iniziare a chattare

Attore primario: Utente

Attori secondari: Firebase

Precondizioni: L'utente deve essere autenticato

ModificaProfilo

1. Il caso d'uso inizia quando un utente vuole modificare le credenziali di accesso oppure delle informazioni relative al proprio profilo
2. Se l'utente vuole modificare
 - L'email
 - 2.1 L'utente deve inserire la nuova email
 - 2.2 Se l'email è valida, l'email viene aggiornata
 - La password
 - 2.1 L'utente deve inserire la nuova password
 - 2.2 L'utente deve inserire nuovamente la password per la conferma
 - 2.3 Se le password sono valide e coincidono, la password viene aggiornata
 - Le informazioni personali
 - 2.1 L'utente inserisce
 - Il nuovo nome
 - Il nuovo cognome
 - Il nuovo numero di telefono
 - La nuova descrizione
 - 2.2 Le informazioni personali vengono aggiornate

Attore primario: Utente

Precondizioni: L'utente deve essere autenticato

Mentre nel capitolo precedente abbiamo parlato di una fase progettuale separata dall'implementazione vera e propria, rimanendo su un piano concettuale, in modo da riuscire ad astrarre in modo corretto la realtà da modellare, in questo capitolo parleremo della fase di progettazione, in cui si iniziano ad improntare le varie componenti secondo la logica implementativa, come potremo già notare dal diagramma delle classi, in cui riporteremo attributi e metodi che saranno effettivamente presenti a livello di codice. Nella parte finale del capitolo, verrà trattata l'architettura scelta per lo sviluppo della nostra applicazione.

3.1 Diagramma delle classi

Il diagramma delle classi contiene un insieme di classi di progettazione. Una classe di progettazione è una classe le cui specifiche sono talmente complete che possono essere effettivamente implementate. Proprio come definito nel Capitolo 1, le classi contengono un insieme di attributi, indicati da nome e tipo, ed un insieme di metodi, che includono il nome, l'elenco dei parametri e il tipo di valore restituito. Una regola da tenere a mente è la seguente: *ogni classe deve modellare un unico concetto astratto e deve avere un insieme di metodi a supporto di questo suo unico compito*. La corretta definizione di attributi e metodi permette di rispettare questa regola. Prima di mostrare le classi di progettazione della nostra applicazione, è utile introdurre la definizione di package e di associazione tra classi.

Un package, in UML, è un'entità di raggruppamento che lega elementi con una forte correlazione semantica. Ogni package possiede un suo spazio dei nomi al cui interno i nomi degli elementi devono essere univoci ed ogni elemento del modello appartiene ad un unico package. Quando si elencano gli elementi di un package, si può anche specificare per ognuno la *visibilità*, che indica se l'elemento è visibile o meno agli elementi esterni al package (Figura 3.1). Si può sfruttare la visibilità degli elementi per limitare il grado di interdipendenza tra package.

Un'associazione tra classi si ha quando un'istanza di classe scambia messaggi con un'altra classe. Ogni associazione è costituita da:

- un nome, che non è obbligatorio;
- un ruolo, anch'esso non obbligatorio e definito per ciascuna classe coinvolta, che serve per chiarire il ruolo giocato da una classe all'interno di un'associazione;
- una molteplicità, che può essere di vario tipo ed indica se la partecipazione all'associazione è obbligatoria oppure facoltativa, ed il numero minimo e massimo di oggetti di una classe che possono essere relazionati ad un oggetto dell'altra classe (Figura 3.2).

Simbolo	Visibilità	Significato
+	pubblica	Gli elementi con visibilità pubblica sono visibili agli elementi esterni al package; essi possono essere esportati dal package.
-	privata	Gli elementi con visibilità privata sono del tutto nascosti all'interno del package in cui risiedono.

Figura 3.1: Visibilità degli elementi di un package

Indicatore	Significato
0..1	Zero o uno
1	Esattamente uno
0..*	Zero o più
1..*	Uno o più
n	Esattamente n (con $n > 1$)
0..n	Da zero a n (con $n > 1$)
1..n	Da uno a n (con $n > 1$)

Figura 3.2: I vari tipi di molteplicità delle associazioni

Un esempio completo di associazione può essere osservato nella Figura 3.3. L'immagine può essere tradotta nel modo seguente: almeno un'istanza della classe *persona* (partecipazione obbligatoria) può *lavorare per* una singola *azienda* (non obbligatoriamente). La persona è il dipendente, mentre l'azienda è il datore di lavoro.

Nella Figura 3.4 e nelle Figure 3.5-3.8, riportiamo, rispettivamente, la suddivisione in package ed i diagrammi delle classi della nostra applicazione.

3.2 Diagramma di sequenza

Un diagramma di sequenza è un diagramma di interazione, ovvero un diagramma usato per modellare qualsiasi tipo di interazione tra le parti all'interno di un sistema. Il diagramma di sequenza, in particolare, modella le interazioni tra gli oggetti in un unico caso d'uso. Essi illustrano come le diverse parti di un sistema interagiscono tra loro per svolgere una funzione, e l'ordine in cui le interazioni avvengono quando un particolare caso d'uso viene eseguito. Un diagramma di sequenza è strutturato in modo tale da rappresentare una linea temporale che inizia dall'alto e scende gradualmente per segnare la sequenza delle interazioni. Ogni oggetto



Figura 3.3: Esempio completo di associazione tra classi

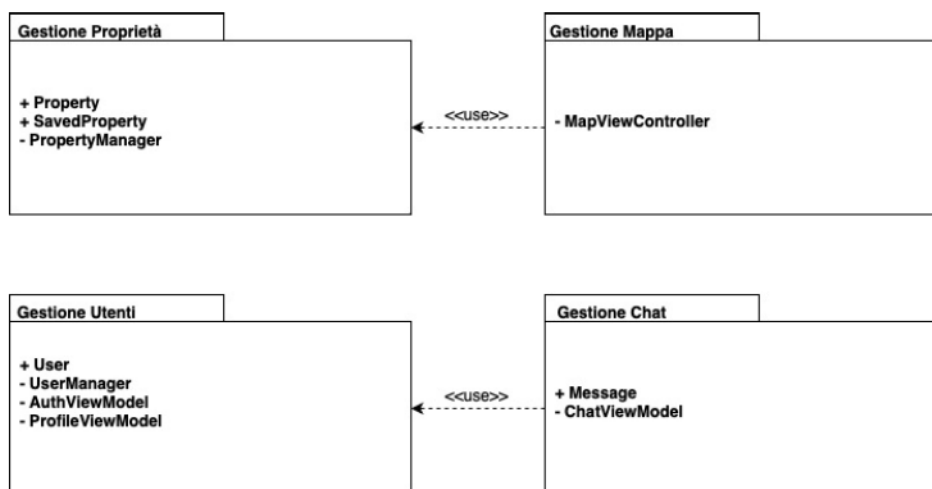


Figura 3.4: Suddivisione in package dell'applicazione

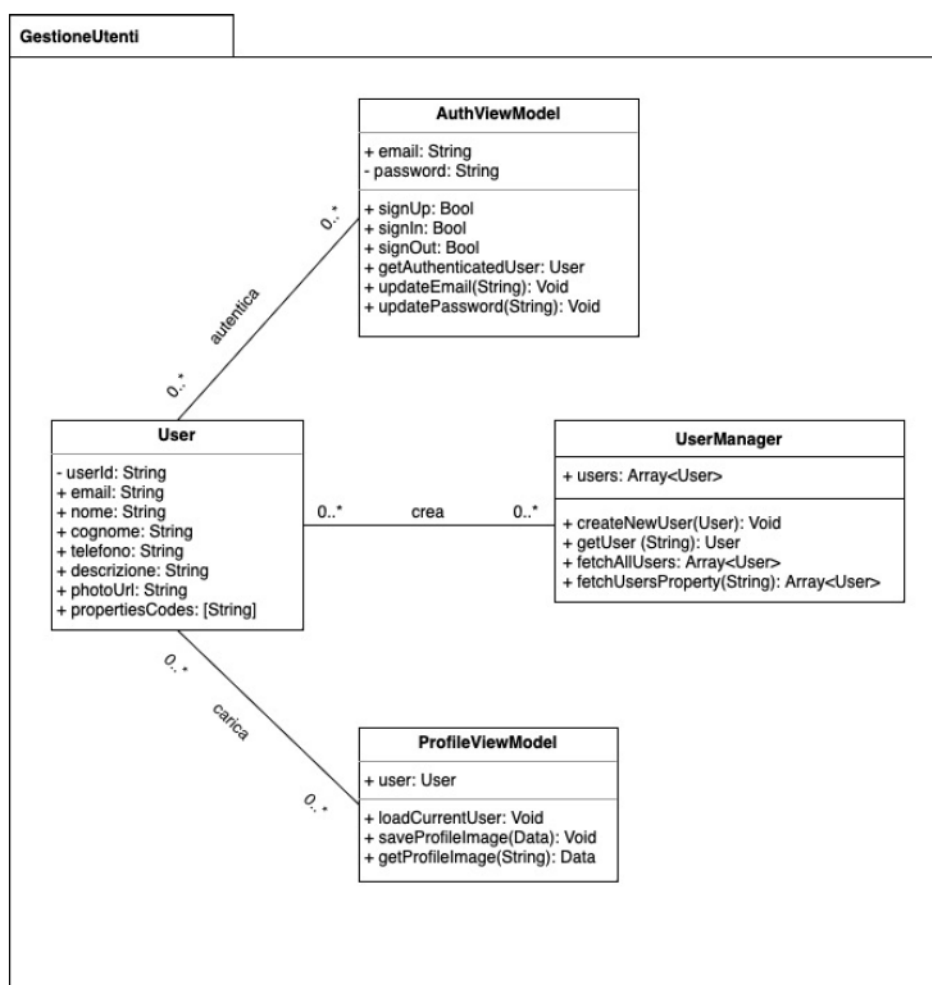


Figura 3.5: Diagramma delle classi del package GestioneUtenti

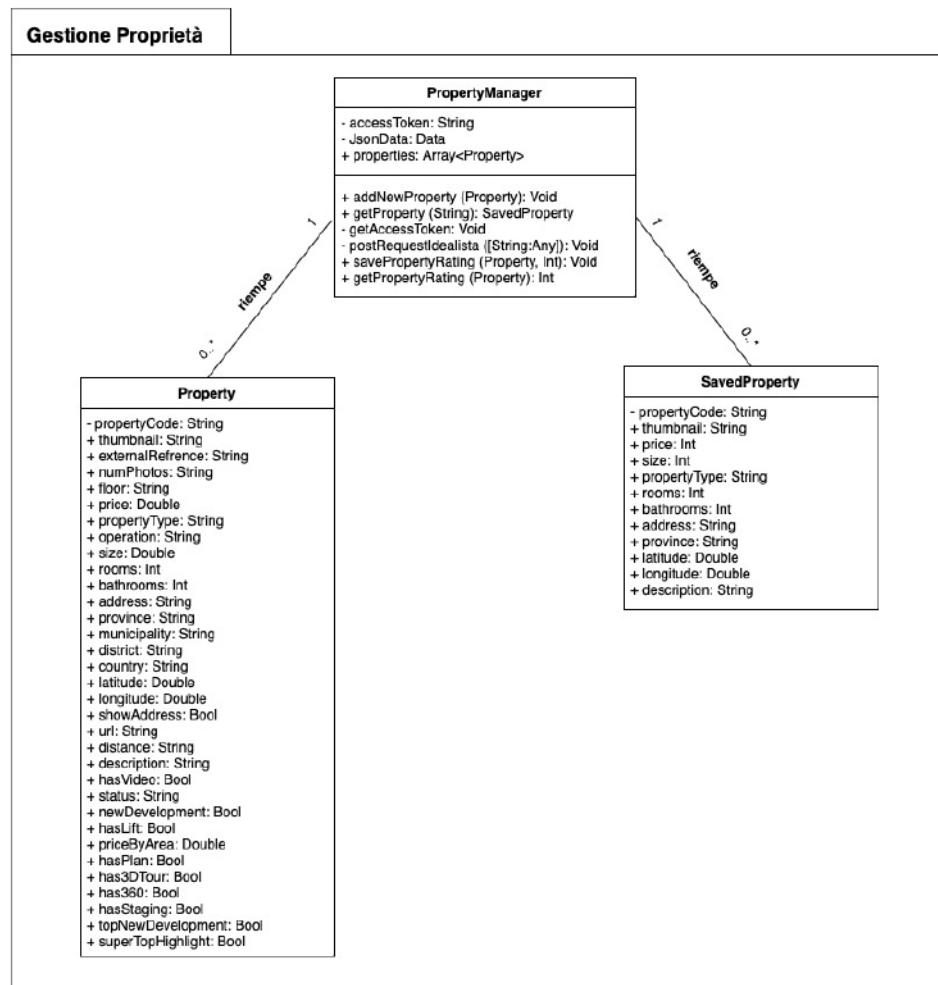


Figura 3.6: Diagramma delle classi del package GestioneProprietà

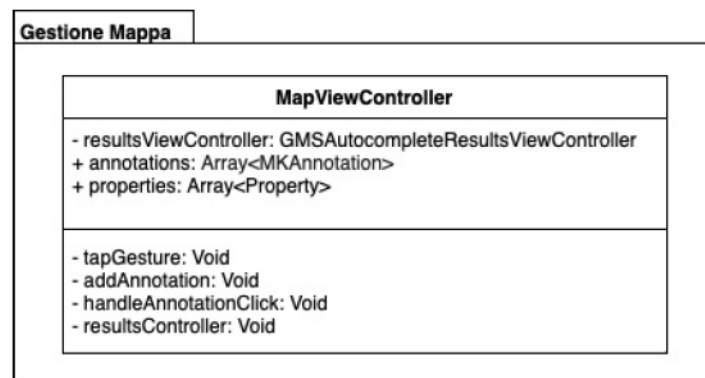


Figura 3.7: Diagramma delle classi del package GestioneMappa

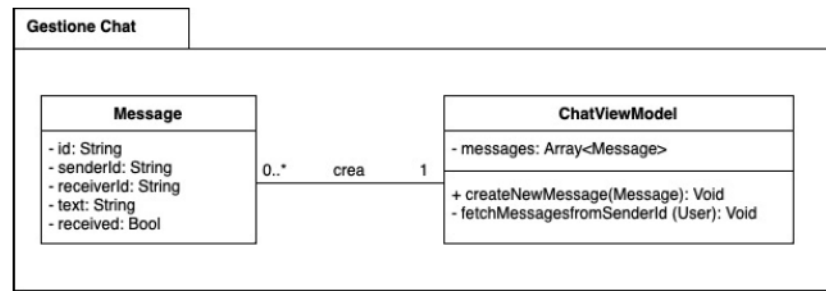


Figura 3.8: Diagramma delle classi del package GestioneChat

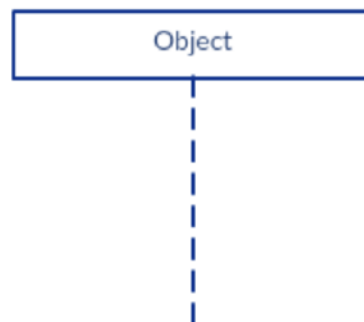


Figura 3.9: Rappresentazione della linea di vita

ha una colonna e i messaggi scambiati tra di loro sono rappresentati da frecce. Iniziamo a dare alcune definizioni:

- *Linea di vita*: rappresenta il tempo che passa estendendosi verso il basso (Figura 3.9).
- *Barra di attivazione*: il riquadro posto sulla linea di vita che serve per indicare che un oggetto è attivo durante un'interazione tra due oggetti. La lunghezza del rettangolo indica la durata degli oggetti che rimangono attivi (Figura 3.10).
- *Messaggio*, ovvero un tipo specifico di comunicazione tra due linee di vita in un'interazione. I messaggi vengono rappresentati da delle frecce che vanno da una linea di vita all'altra. Esistono sette tipi di messaggio (Figura 3.11).

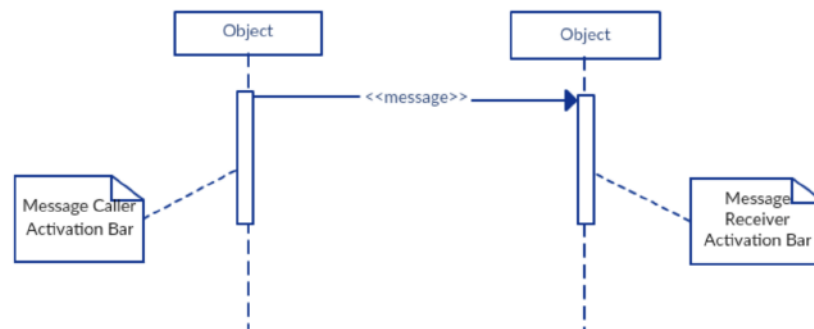


Figura 3.10: Rappresentazione della barra di attivazione

Sintassi	Nome	Descrizione
unMessaggio (unParametro) →	Messaggio Sincrono	Il mittente aspetta che il destinatario risponda all'esecuzione del messaggio
unMessaggio (unParametro) →	Messaggio Asincrono	Il mittente invia il messaggio e continua l'esecuzione: non aspetta la risposta del destinatario
←-----	Messaggio di ritorno	Il destinatario di un messaggio precedente restituisce il focus di controllo al mittente di quel messaggio
<<crea>> unMessaggio() 	Creazione dell'oggetto	Il mittente crea un'istanza del classificatore specificato dal destinatario
<<distruggi>> 	Distruzione dell'oggetto	Il mittente distrugge il destinatario. Se la sua linea di vita ha una coda, questa termina con una X
● →	Messaggio trovato	Il mittente del messaggio è al di fuori dell'ambito dell'interazione. Lo si utilizza quando si vuole mostrare la ricezione di un messaggio, ma non si vuole mostrare da dove proviene
→ ●	Messaggio perso	Il messaggio non raggiunge mai la sua destinazione. Può essere utilizzato per indicare condizioni di errore in cui i messaggi vanno persi.

Figura 3.11: Tipologie di messaggio

Dopo aver dato queste definizioni, nelle Figure 3.12-3.16, sono rappresentati i diagrammi di sequenza dei casi d'uso della nostra applicazione.

3.3 Diagramma di attività

I diagrammi di attività consentono di modellare un processo come un'attività costituita da un insieme di nodi connessi da archi. È possibile associare un'attività a qualunque elemento di modellazione al solo fine di modellare il comportamento di tale elemento. L'essenza di un diagramma di attività è quella di comunicare uno specifico aspetto del comportamento dinamico di un sistema e, quindi, è necessario che il suo livello di astrazione sia quello più adatto a comunicare questo aspetto agli utenti, contenendo la minore quantità di informazioni. Normalmente i diagrammi di attività vengono usati per modellare un caso d'uso come una serie di azioni.

Ogni attività comincia con un singolo nodo di controllo, che indica il punto in cui l'esecuzione inizierà quando essa viene invocata; uno o più nodi finali indicano il punto in cui l'attività termina. Ogni attività è rappresentata con un rettangolo dai bordi arrotondati, al cui interno viene spiegata a parole l'azione; le diramazioni condizionali, caratterizzate da ingressi e uscite multiple, e la fusione di più flussi di attività vengono rappresentati da un rombo.

Nelle Figure 3.17-3.21 vengono riportati i diagrammi di attività relativi ai casi d'uso della nostra applicazione.

3.4 Architettura dell'app

L'architettura di un'applicazione è l'infrastruttura che sta alla base del suo funzionamento e ne definisce chiaramente la struttura della logica applicativa. Oltre alla chiarezza, i vantaggi che una buona architettura porta sono molteplici, come ad esempio:

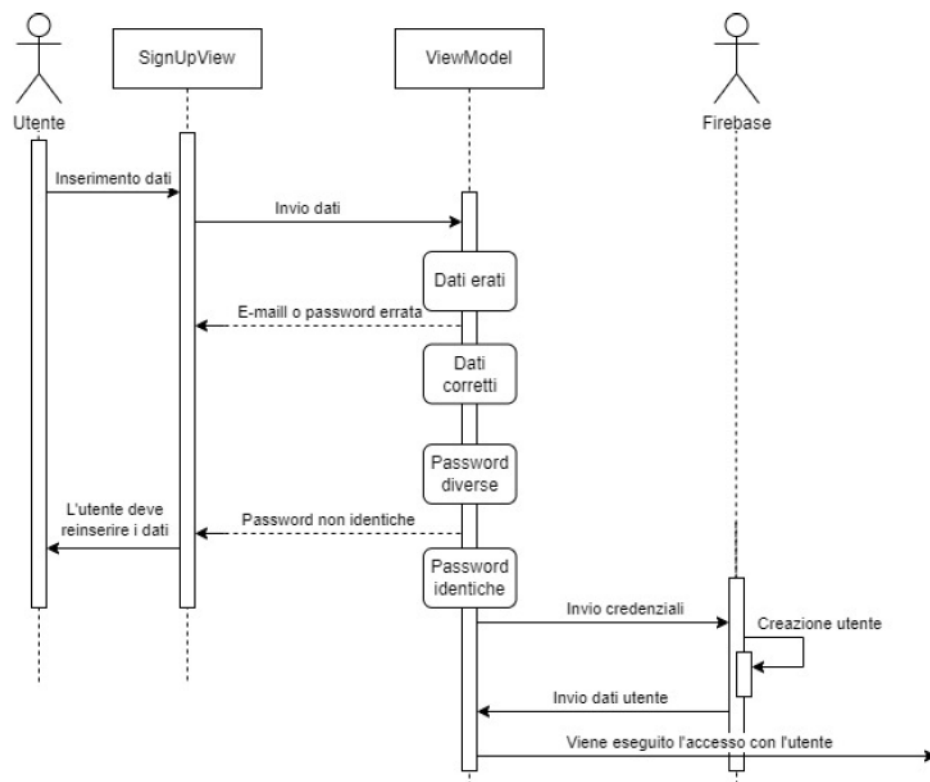


Figura 3.12: Diagramma di sequenza di Registrazione

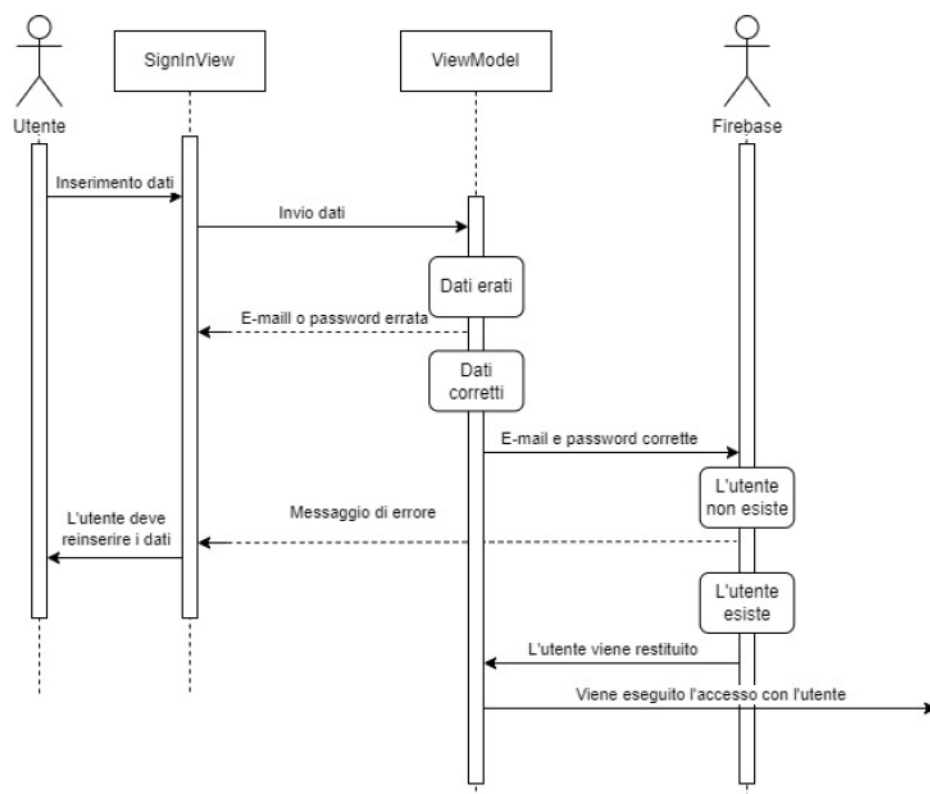


Figura 3.13: Diagramma di sequenza di Login

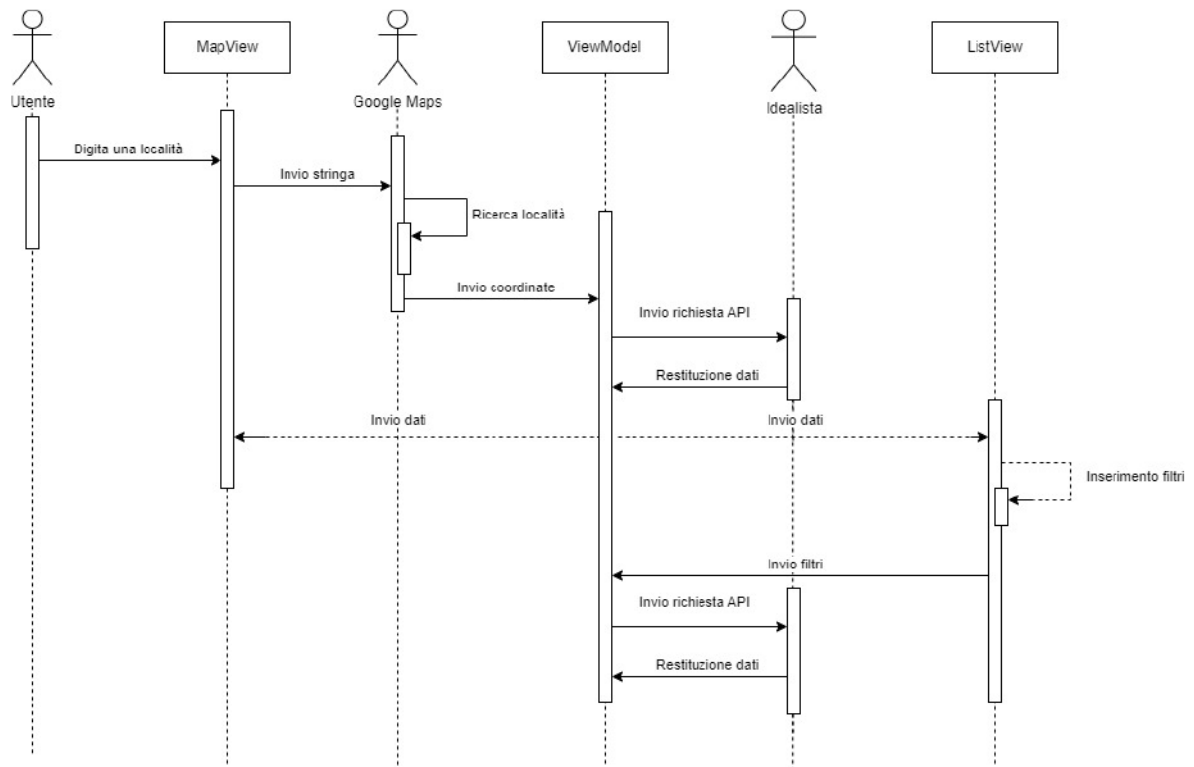


Figura 3.14: Diagramma di sequenza di RicercaFiltrata

- rendere l'app modulare, ovvero suddividerla in più componenti, l'una indipendente dall'altra, e quindi renderla più manutenibile;
- rendere più facile lo sviluppo e la collaborazione tra sviluppatori;
- rendere il codice più facile da testare e l'app più scalabile¹.

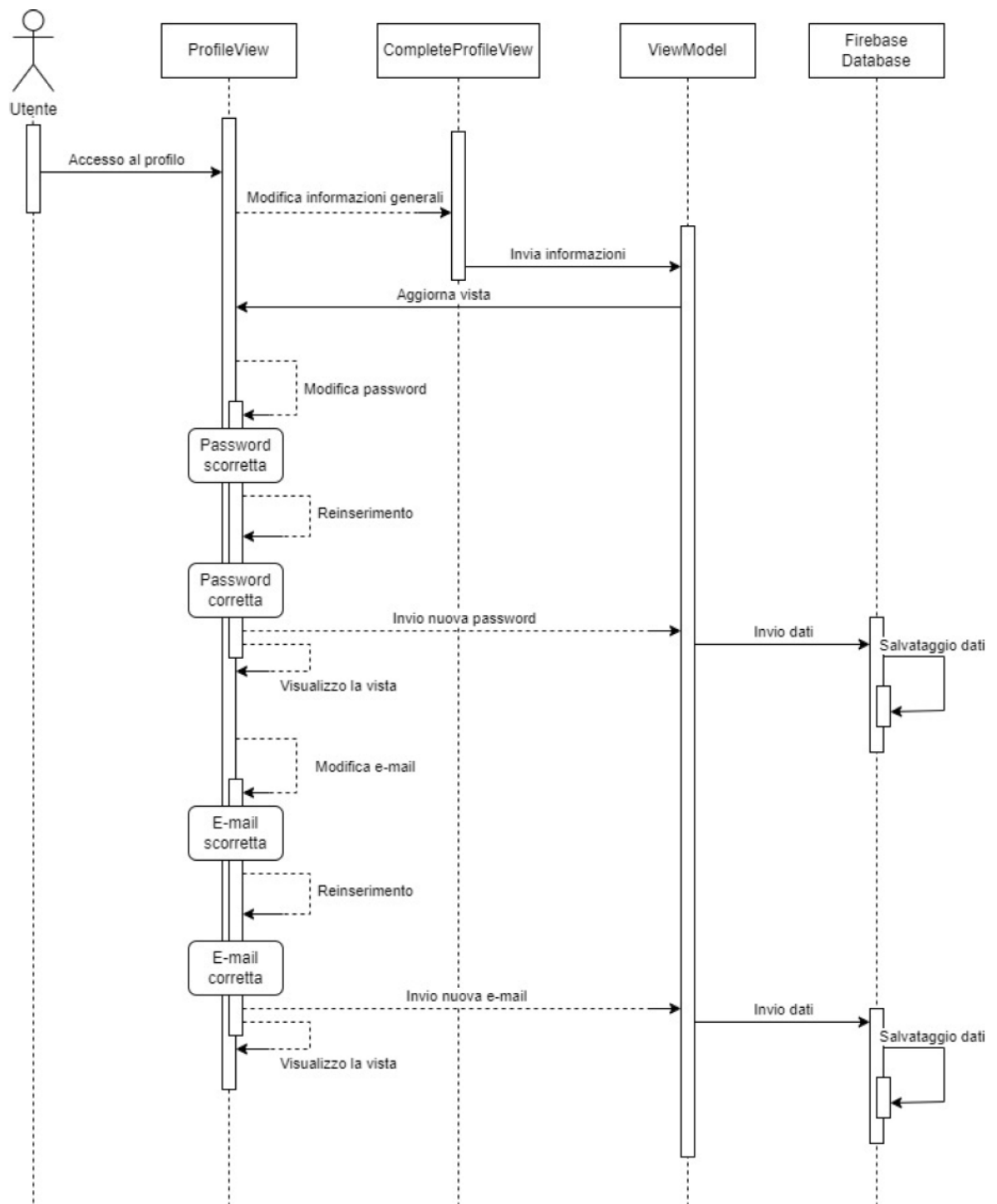
Esistono numerosi tipi di architetture, ma quella che abbiamo adottato nella nostra applicazione è la *Model-View-ViewModel* (MVVM). Analizziamo le tre componenti principali dell'architettura, che sono:

- *Model*, che rappresenta le strutture dati (le cosiddette *data class*);
- *View*, che rappresenta le interfacce utente, quindi ciò che l'utente vede;
- *ViewModel*, ovvero la componente che gestisce la logica. Il compito del ViewModel è quello di preparare (in seguito ad elaborazioni) i dati per la UI;² la sua particolarità è che i dati possono essere aggiornati all'interno di un *LiveData*, cioè dei dati osservabili (Sezione 1.2.3); ogni cambiamento di stato verrà notificato all'osservatore, ovvero la View, mostrando gli aggiornamenti all'utente.

Nelle Figure 3.22 e 3.23 vengono riportati un esempio di Model ed un esempio di View-Model; per quanto riguarda la View, viene fornito un esempio della schermata di Login in Figura 3.24. Per comodità, nell'applicazione, oltre alle tre componenti principali, abbiamo utilizzato anche dei file contenenti della logica, che però non fanno parte del ViewModel. Nella Figura 3.25, è riportata una suddivisione in file dell'app.

¹Capacità di un'applicazione di aggiungere nuove funzionalità senza cambiare la struttura preesistente.

²User Interface (Interfaccia Utente).

**Figura 3.15:** Diagramma di sequenza di ModificaProfilo

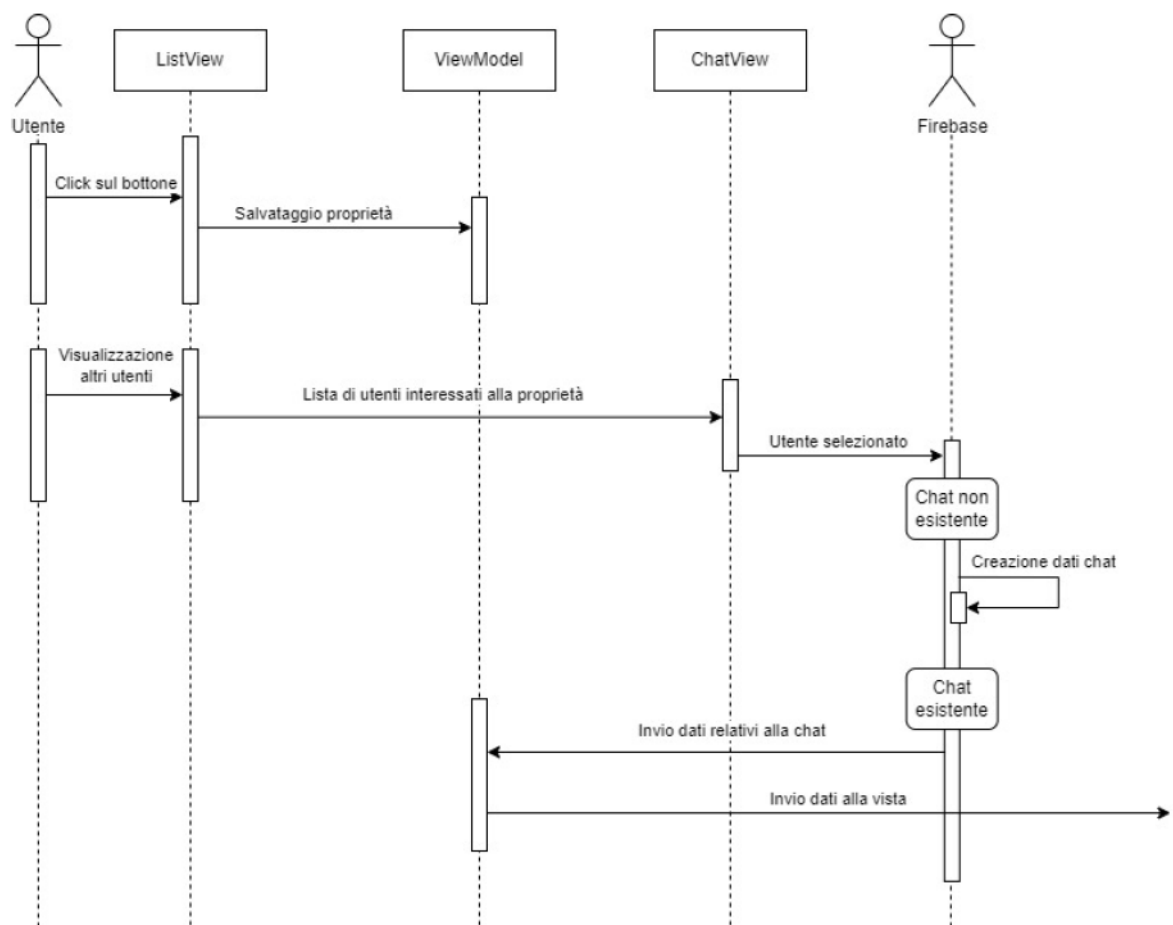
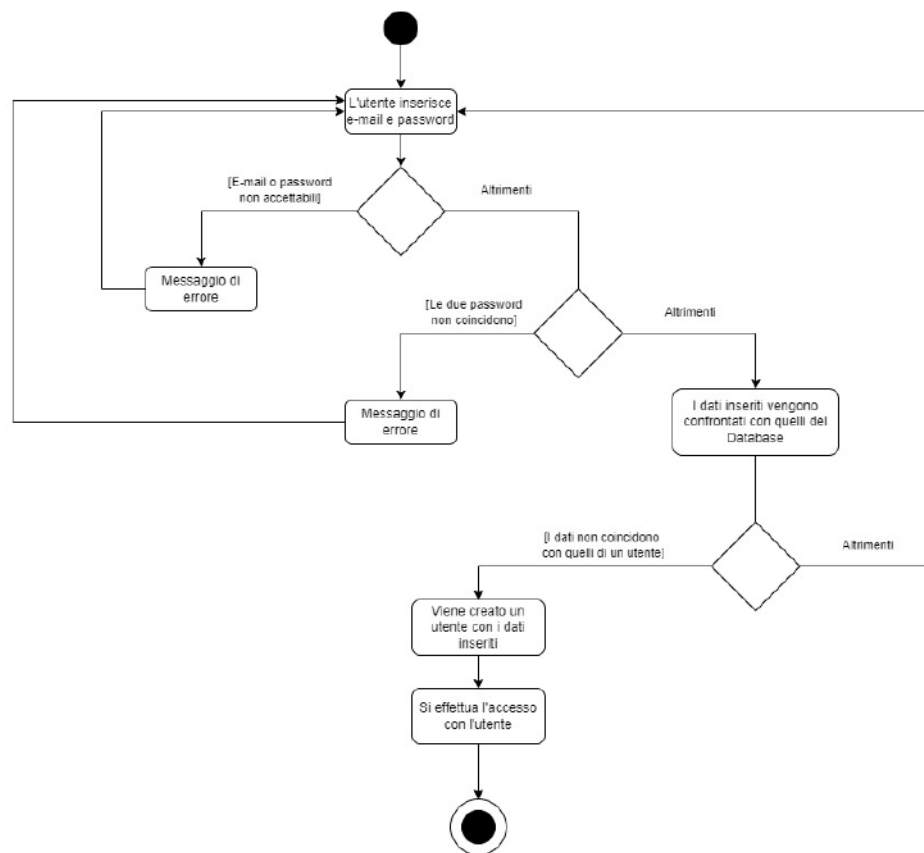
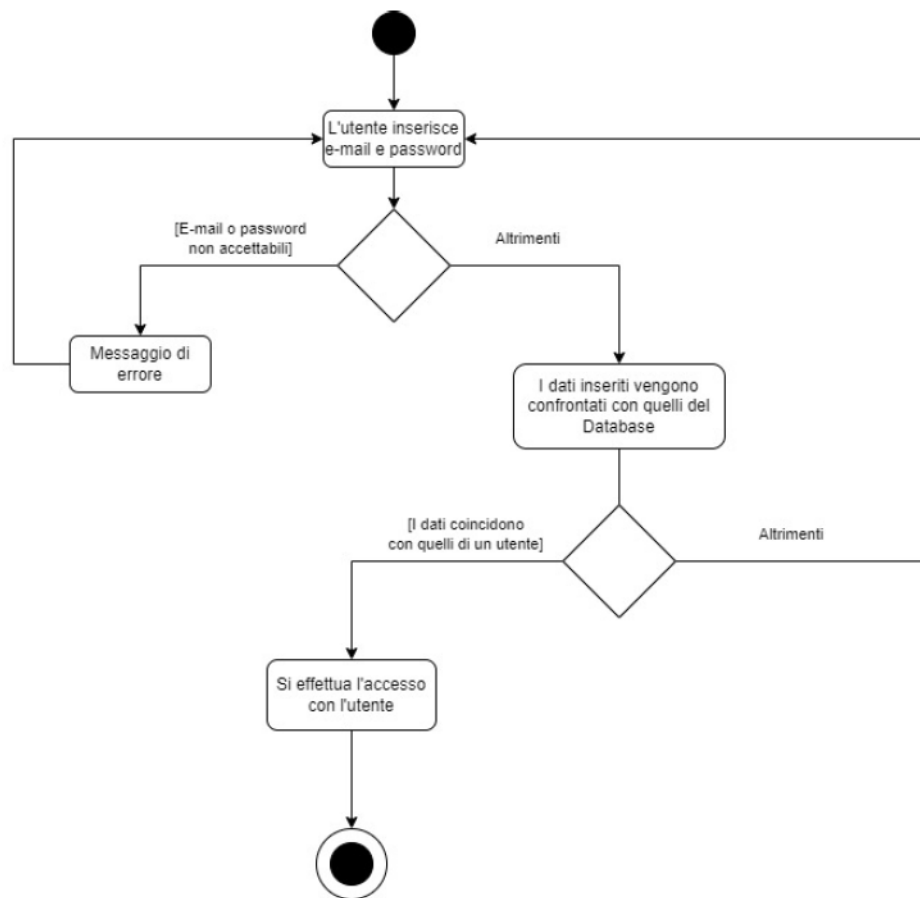
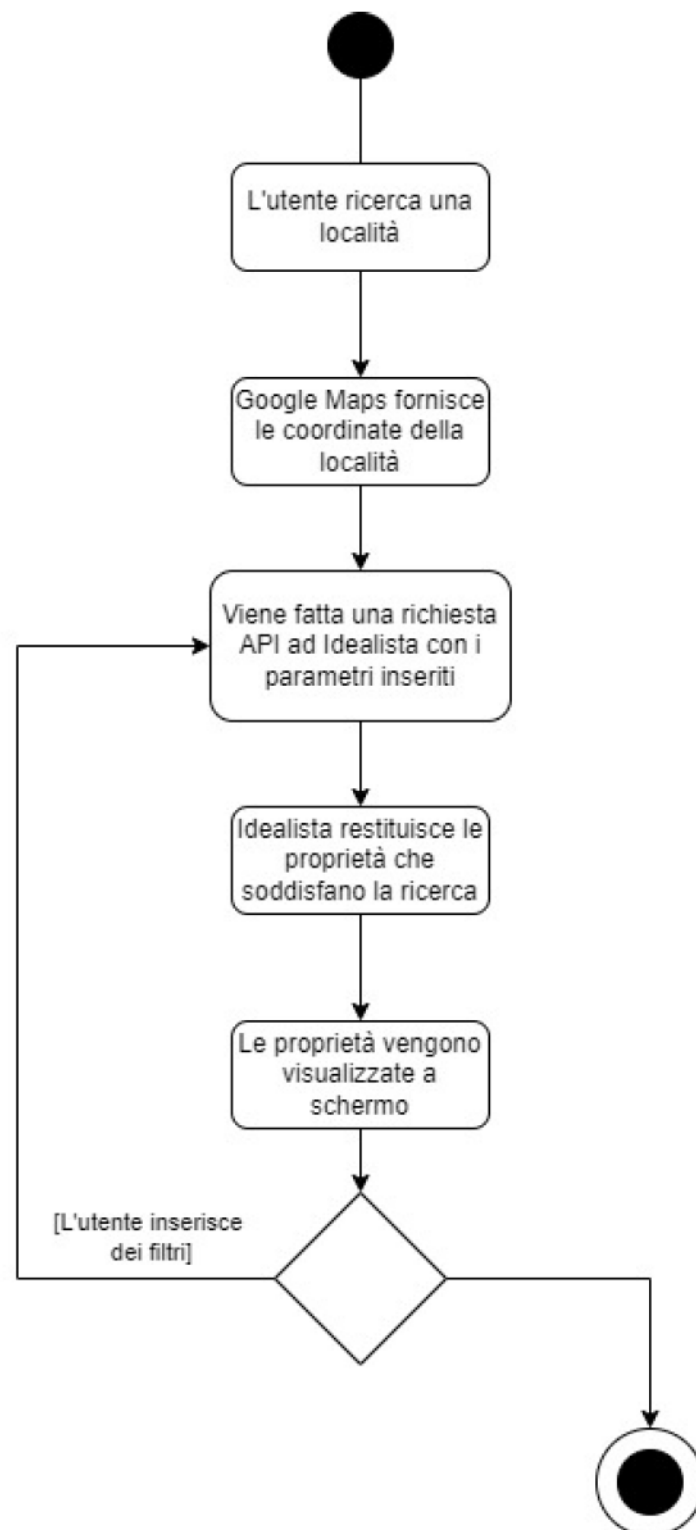
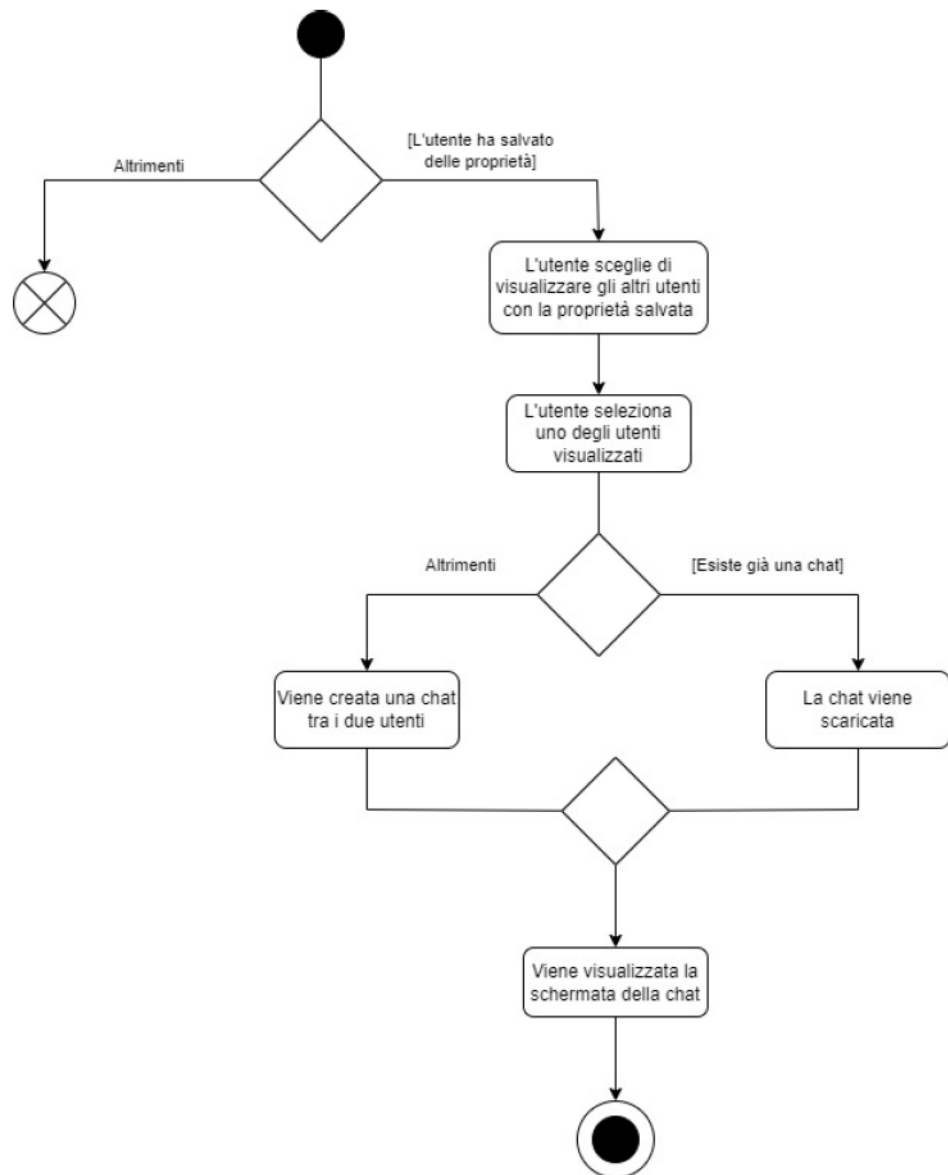


Figura 3.16: Diagramma di sequenza di Chat

**Figura 3.17:** Diagramma di attività di Registrazione

**Figura 3.18:** Diagramma di attività di Login

**Figura 3.19:** Diagramma di attività di RicercaFiltrata

**Figura 3.20:** Diagramma di attività di Chat

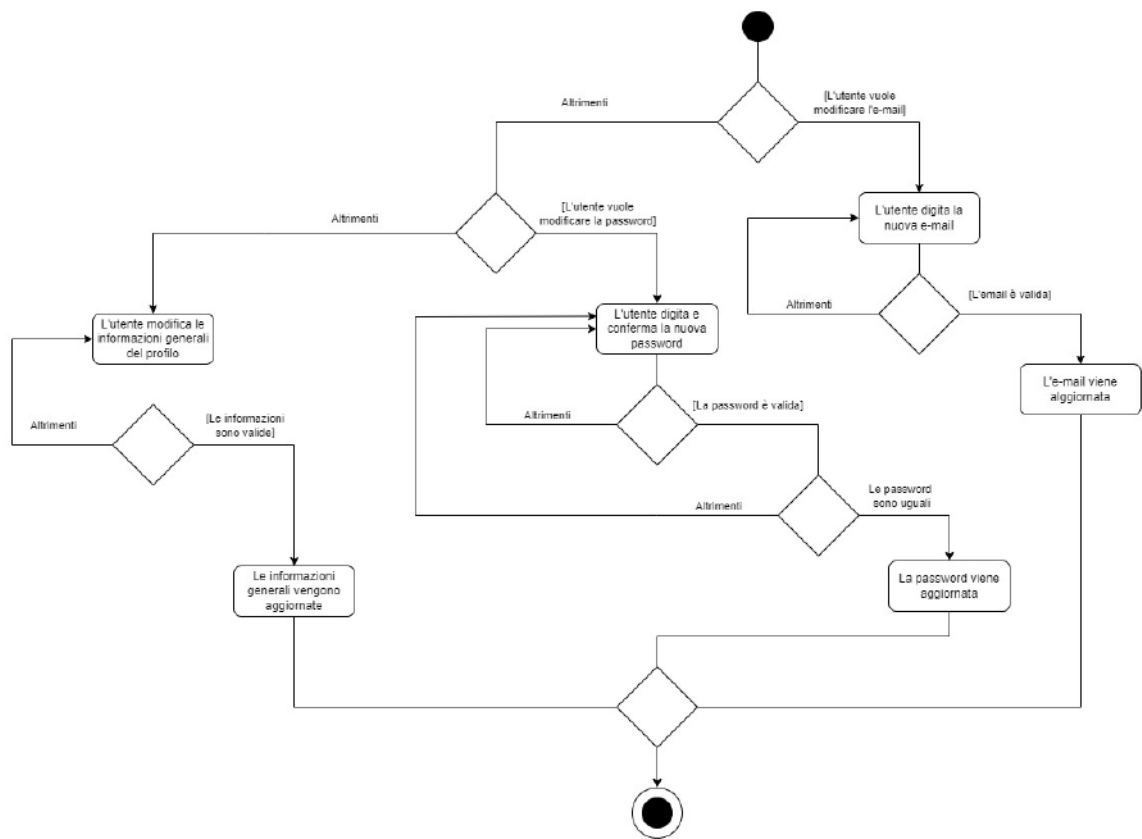


Figura 3.21: Diagramma di attività di ModificaProfilo

```

struct DBUser: Identifiable {
    var id: String { userId }

    var userId: String
    var email: String?
    var nome: String?
    var cognome: String?
    var telefono: String?
    var descrizione: String?
    var photoUrl: String?
    var propertiesCodes: [String]?
}

```

Figura 3.22: Model di User

```

final class AuthViewModel : ObservableObject {

    @Published var email = ""
    @Published var password = ""
    var properties: Array<Property> = []
    var loggedIn: Bool = false
    var signedUp: Bool = false

    //Logica di Registrazione
    @discardableResult
    func createUser(email: String, password: String) async throws -> AuthDataResultModel {

        let authDataResult = try await Auth.auth().createUser(withEmail: email, password: password)

        return AuthDataResultModel(user: authDataResult.user)
    }

    func signUp () async throws -> Bool{

        guard !email.isEmpty, !password.isEmpty else {
            print ("No email or password found")
            return false
        }

        do{
            let authDataResult = try await createUser(email: email, password: password)
            try await UserManager.shared.createNewUser(auth: authDataResult)
            return true
        }
        catch {
            print ("Error: \(error)")
            return false
        }
    }

    //Logica di Login
    @discardableResult
    func signInUser (email: String, password: String) async throws -> AuthDataResultModel{
        let authDataResult = try await Auth.auth().signIn(withEmail: email, password: password)
        return AuthDataResultModel(user: authDataResult.user)
    }

    func signIn () async throws -> Bool {

        guard !email.isEmpty, !password.isEmpty else {
            return false
        }

        do{
            try await signInUser(email: email, password: password)
            self.loggedIn = true
            return true
        }
        catch {
            self.loggedIn = false
            return false
        }
    }
}

```

Figura 3.23: Esempio di ViewModel



UNI FIND

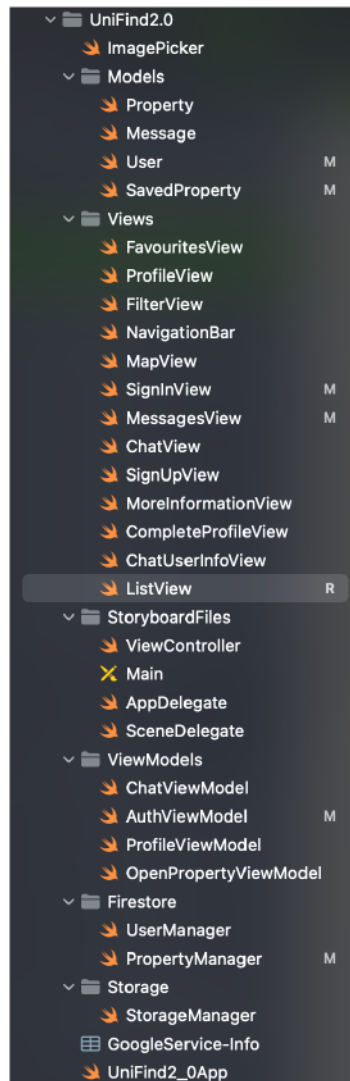
Email:

Password:

Login

Non sei ancora registrato? [Registrati](#)

Figura 3.24: Esempio di view

**Figura 3.25:** Suddivisione in file dell'applicazione

La fase di implementazione è l'ultima parte della fase di elaborazione del ciclo di vita di un progetto. L'implementazione riguarda la trasformazione del modello di progettazione in codice eseguibile. Nella prima parte del capitolo ci concentreremo di più sull'interfaccia utente, ovvero su come l'applicazione si mostra e si rende utilizzabile all'utente, mentre nella seconda parte tratteremo della logica che sta dietro all'applicazione, rendendola funzionante, e di alcuni strumenti software utilizzati nello sviluppo dell'app. In entrambe le sezioni verranno riportati alcuni esempi di codice.

4.1 Frontend

Con il termine "frontend" si denota la parte di un programma visibile all'utente e con cui egli può interagire. Esso è responsabile dell'acquisizione dei dati di ingresso e della loro elaborazione, tali da renderli utilizzabili dal backend.

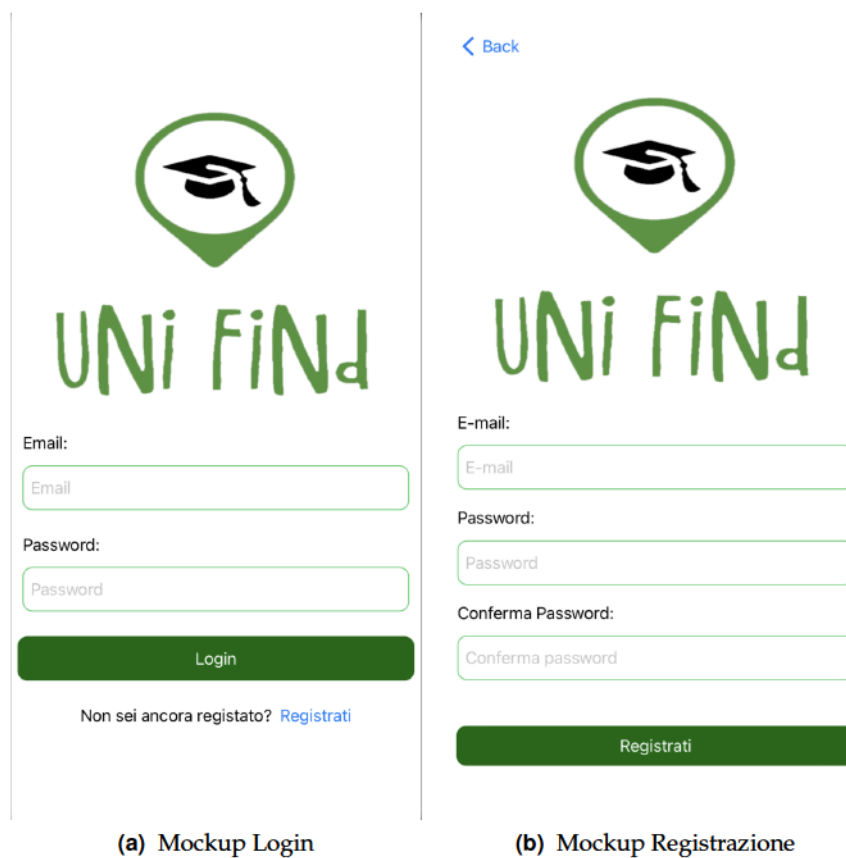
4.1.1 Mockup

Un mockup è una realizzazione a scopo illustrativo di un'interfaccia, anche senza tutte le funzioni di quando sarà completa. Un mockup può rappresentare la totalità o solo una parte dell'interfaccia finale (già esistente o in fase di progetto), essere in scala reale oppure variata. Nelle Figure 4.1-4.7 sono mostrati i mockup della nostra applicazione.

4.1.2 Utilizzo di SwiftUI nel codice

Come detto precedentemente, SwiftUI è il framework, realizzato interamente in Swift, che facilita la costruzione dell'interfaccia utente. Esso fornisce, infatti, numerosi costrutti che permettono di posizionare automaticamente elementi all'interno di una vista. Alcuni esempi di questi elementi sono:

- `VStack`, ovvero una vista che dispone gli elementi al suo interno in modo verticale;
- `HStack`, ovvero una vista che dispone gli elementi al suo interno in modo orizzontale;
- `ScrollView`, ovvero una vista scorribile. Essa viene solitamente utilizzata con una lista di elementi che non rientrano all'interno di un'unica schermata ed è, quindi, necessario scorrerla.



The figure displays two mobile app mockups for 'UNI FIND'. Both screens feature a logo at the top consisting of a green location pin with a black graduation cap inside. Below the logo, the text 'UNI FIND' is written in a green, hand-drawn font.

(a) Mockup Login: This screen has two input fields. The first is labeled 'Email:' and contains the placeholder text 'Email'. The second is labeled 'Password:' and contains the placeholder text 'Password'. Below these fields is a solid green button with the white text 'Login'. At the bottom, there is a link that says 'Non sei ancora registrato? [Registrati](#)'.

(b) Mockup Registrazione: This screen includes a blue '< Back' link at the top left. It has three input fields: 'E-mail' (placeholder 'E-mail'), 'Password' (placeholder 'Password'), and 'Conferma Password' (placeholder 'Conferma password'). A solid green button with the white text 'Registrati' is positioned at the bottom.

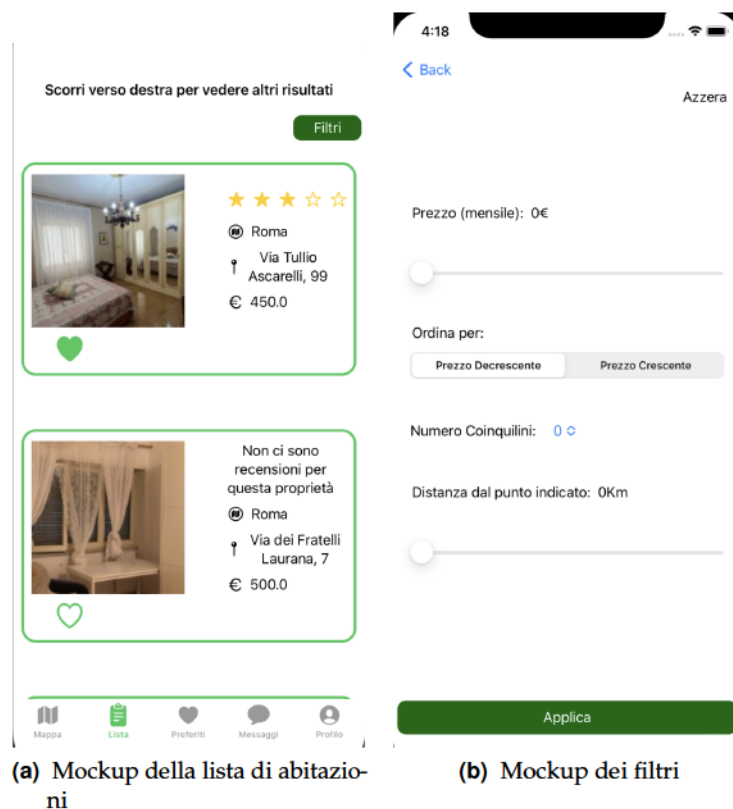
(a) Mockup Login

(b) Mockup Registrazione

Figura 4.1: Mockup per l'autenticazione



Figura 4.2: Mockup della mappa



(a) Mockup della lista di abitazioni

(b) Mockup dei filtri

Figura 4.3: Mockup della lista di abitazioni come risultato della ricerca e dei filtri

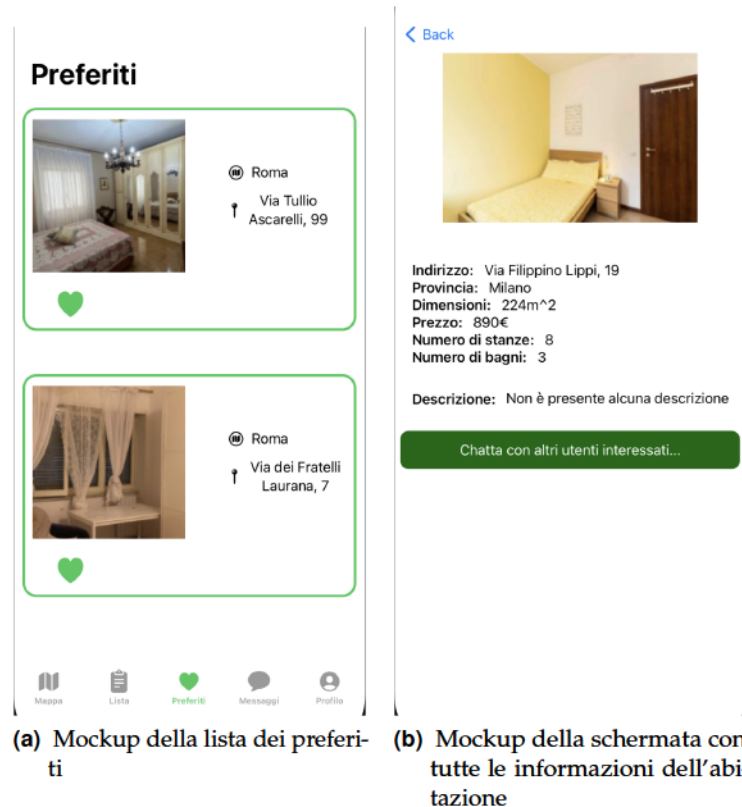


Figura 4.4: Mockup dei preferiti e della schermata con tutte le informazioni

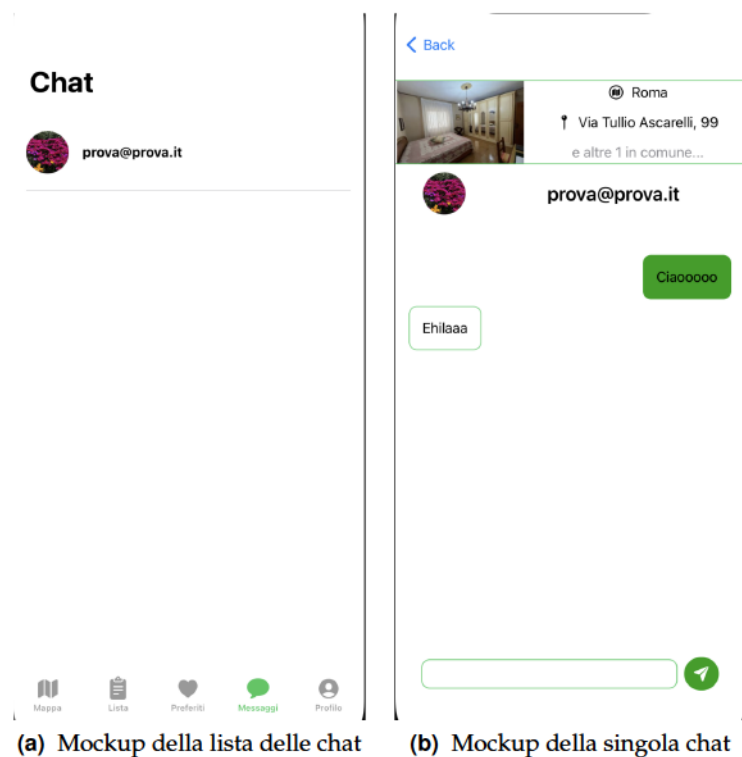
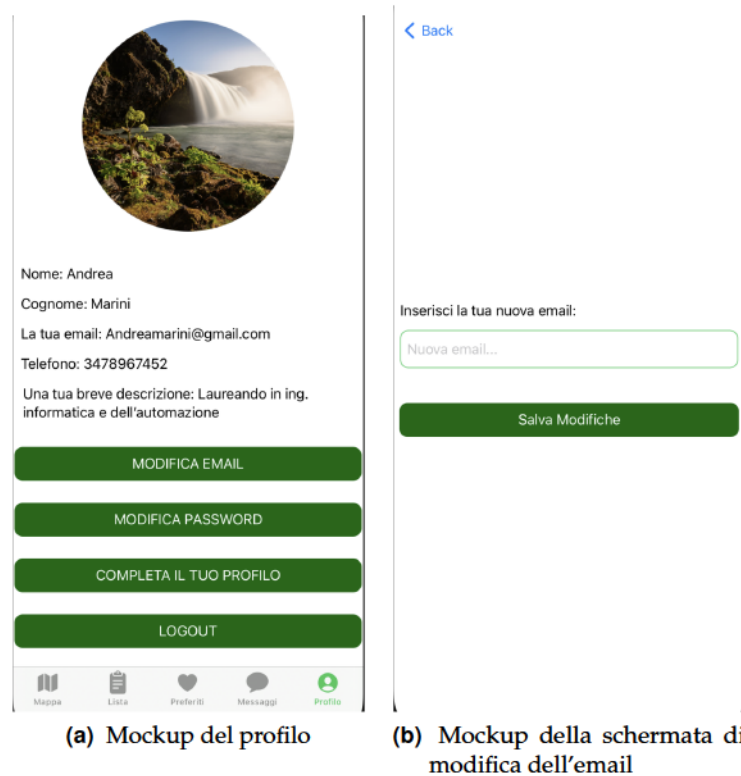
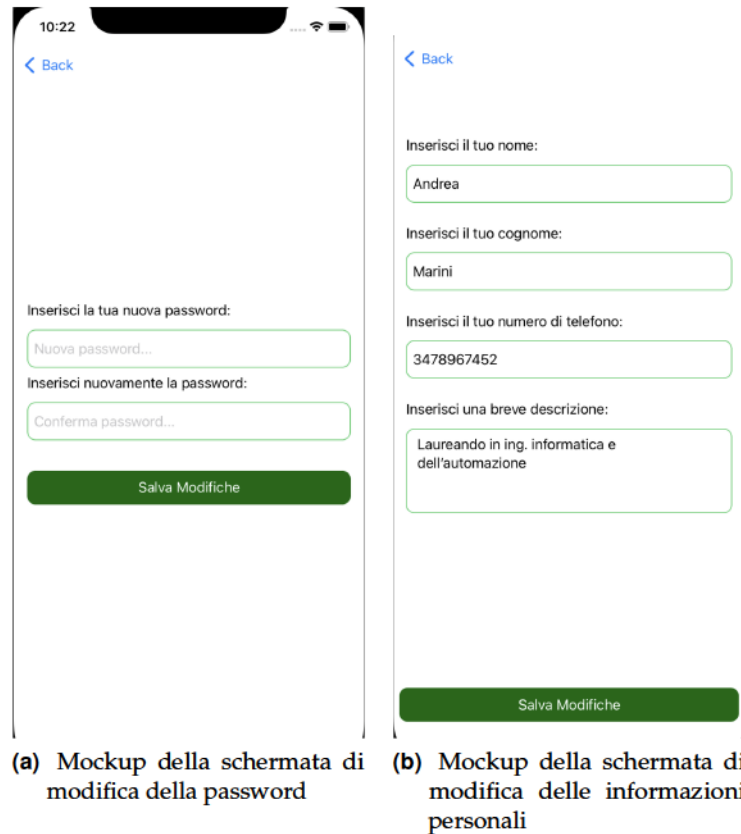


Figura 4.5: Mockup della lista delle chat e della schermata con la chat singola

**Figura 4.6:** Mockup del profilo e della schermata di modifica dell'email**Figura 4.7:** Mockup della schermata di modifica della password e di modifica delle informazioni personali

Oltre a questi elementi, che vengono spesso chiamati *layout*, SwiftUI ne mette a disposizione anche altri, come:

- `Text`, ovvero un elemento che mostra una o più righe di testo.
- `TextField`, ovvero una casella di testo riempibile. Essa rappresenta, quindi, una classica interazione tra utente e applicazione e i dati inseriti costituiscono un input che potrà essere elaborato.
- `Button`, ovvero il classico pulsante, che è costituito da una `label`, che determina il suo aspetto, e da un `action`, cioè l'azione che si verifica al click su di esso.

Inoltre, la disposizione di elementi, come quelli sopra indicati, può essere regolata all'interno del layout mediante alcuni metodi, come:

- `.frame`, ovvero un metodo che permette di definire larghezza e altezza del layout e l'allineamento del contenuto di quest'ultimo;
- `.padding`, ovvero un metodo che permette di definire uno spazio dai bordi di un elemento.

Con questi pochi elementi, è già possibile realizzare la maggior parte della schermata di login, rappresentata in Figura 4.1. Il codice, utilizzato per la sua realizzazione, è, invece, visibile in Figura 4.8.

Approfondiamo, ora, due concetti estremamente importanti, ovvero la navigazione e la programmazione asincrona.

La navigazione è quel meccanismo che permette all'applicazione di passare da una schermata all'altra. Ad esempio, sempre facendo riferimento alla schermata di login in Figura 4.1, cliccando su "Registrati", si viene indirizzati ad una schermata diversa da quella iniziale, ovvero la schermata di registrazione (Figura 4.1). I costrutti che rendono possibile questo meccanismo sono:

- `NavigationView`, che indica che la vista è il punto di inizio di un *percorso di navigazione*;
- `NavigationLink`, che permette di definire un link, ovvero una zona cliccabile, e una destinazione. Infatti, al click sul link, l'utente verrà reindirizzato alla schermata indicata come destinazione.

Il loro utilizzo è osservabile in Figura 4.8.

La programmazione asincrona è una forma di programmazione parallela che permette ad un'unità di lavoro di funzionare separatamente dal thread¹ principale. Essa viene usata, ad esempio, per il download di informazioni in background, per operazioni di calcolo pesanti oppure, come nel caso della nostra applicazione, per leggere o scrivere da/su un database.

SwiftUI mette a disposizione un metodo particolare, `.task`, che viene definito al termine dell'interfaccia e permette di gestire la programmazione asincrona. Il metodo `.task` consente di svolgere una serie di istruzioni in modo asincrono prima che la vista per cui è definito sia visualizzabile all'utente.

Un esempio di come abbiamo usato `.task` nella nostra applicazione è riportato nella Figura 4.9. In questo caso, il metodo viene definito per la schermata che mostra la chat tra due utenti che hanno la stessa abitazione salvata nei preferiti (Figura 4.5). Abbiamo utilizzato

¹Un thread di esecuzione è una suddivisione di un processo in due o più sottoprocessi che vengono eseguiti concorrentemente.

```
import SwiftUI

struct Login: View {

    @State var email: String
    @State var password: String

    var body: some View {
        NavigationView{
            VStack(spacing: 25){
                Image("WhiteMode")
                    .resizable()
                    .frame(width: 340, height: 300)

                VStack(alignment:.leading){
                    Text("Email:")
                    TextField("Email", text:$email)
                        .frame(height: 45)
                        .padding([.leading, .trailing],8)
                        .background(Color.white)
                        .overlay(RoundedRectangle(cornerRadius:10.0).strokeBorder(Color.green, style: StrokeStyle(lineWidth: 1.0)))
                }
                .padding([.leading, .trailing], 12)

                VStack(alignment:.leading){
                    Text("Password:")
                    SecureField("Password", text: $password)
                        .frame(height: 45)
                        .padding([.leading, .trailing],8)
                        .background(Color.white)
                        .overlay(RoundedRectangle(cornerRadius:10.0).strokeBorder(Color.green, style: StrokeStyle(lineWidth: 1.0)))
                }
                .padding([.leading, .trailing], 12)

                VStack(){
                    Button(action: {}, label: {
                        Text("Login")
                            .frame(width: 400, height: 45)
                            .background(Color(hue: 0.341, saturation: 1.0, brightness: 0.405))
                            .foregroundColor(Color.white)
                            .cornerRadius(10)
                    })
                }

                HStack(){
                    Text("Non sei ancora registrato?")
                    NavigationLink("Registrati", destination: SignUpView())
                }
            }
            .padding(.top)
            .padding(.bottom, 30)
        }
    }
}
```

Figura 4.8: Codice per la costruzione della schermata di Login

```

.task {
  do{
    self.propertyChatUser = []
    let data = try? await StorageManager.shared.getImage(path: chatUser.userId)
    self.imageData = data
    if (imageData == nil){}
    else{
      let image = UIImage(data: imageData!)
      imageProva = image
    }

    let snapshot = try? await Firestore.firestore().collection("messages").document(currentUser!).getDocument()
    let propertyCode = snapshot?.data()!["properties"] as! String
    let propertyManager = PropertyManager()
    let userManager = UserManager()
    property = try! await propertyManager.getProperty(propertyId: propertyCode)
    for code in chatUser.propertiesCodes!{
      if try await (userManager.getUser(userId: currentUser!).propertiesCodes!.contains(code)) {
        let property=try await propertyManager.getProperty(propertyId: code)
        self.propertyChatUser.append(property)
      }
    }
  }
} catch {
  print ("Errore")
}
}

```

Figura 4.9: Esempio di utilizzo del metodo .task

il metodo per leggere dati dal database, in particolare per ricavare l'immagine del profilo dell'utente con cui si sta chattando e per caricare i messaggi tra i due utenti. Le istruzioni asincrone che rendono possibile la lettura dal database sono:

- `try? await StorageManager.shared.getImage(path: chatUser.userId)`
- `try? await Firestore.firestore().collection("messages").document(currentUser!).getDocument()`

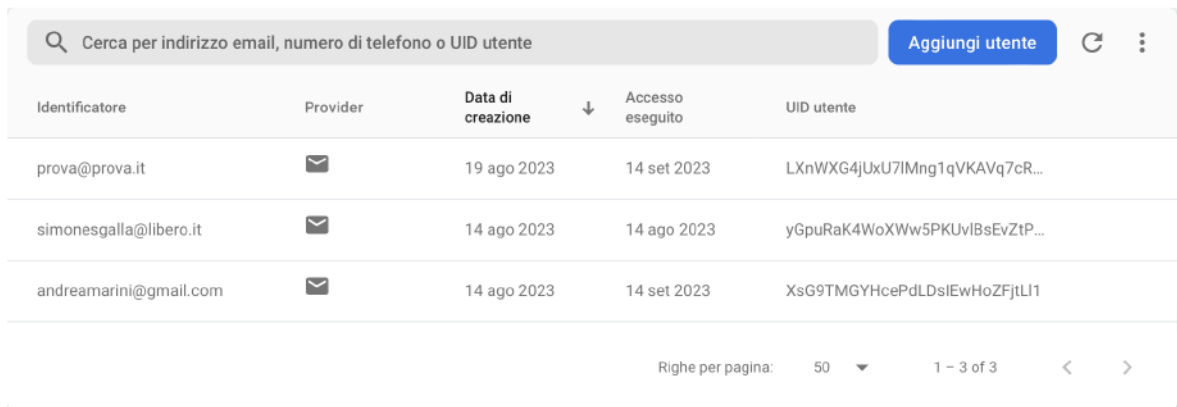
4.2 Backend

Il backend è la parte dell'applicazione nascosta agli utenti. Con questo termine si intende tutto ciò che è necessario ad un'applicazione per funzionare, ma che gli utenti non vedono e con cui non possono interagire. La parte di backend si contrappone alla parte di frontend, che include, invece, tutti gli elementi che sono a diretto contatto con gli utenti. Il backend, dunque, svolge funzioni come memorizzare e organizzare i dati, gestire i collegamenti con il database, interagire con le API e richiamare librerie.

Prima di proseguire, chiariamo, quindi, cosa sono le API. L'acronimo API (*Application Programming Interfaces*) indica delle interfacce, intese come insieme di definizioni e protocolli, che permettono ad un'applicazione di comunicare e scambiare informazioni con altre applicazioni. In questo modo è stato possibile ottenere, ad esempio, le informazioni sulle abitazioni (dal sito Idealista), la funzionalità di ricerca su mappa e l'interazione con il database Firebase.

4.2.1 Google Maps

Google Maps è un servizio Internet geografico sviluppato da Google che consente la ricerca di località, rendendone fruibile la visualizzazione tramite carte geografiche di buona parte della Terra. Da un po' di anni a questa parte, Google ha messo a disposizione degli utenti le API key, che permettono di accedere ad alcuni servizi, come, in questo caso, la gestione e l'utilizzo di mappe. Le API di Google sono gratuite, ma sono richieste delle chiavi di accesso



The screenshot shows the Firebase Authentication database interface. At the top, there is a search bar with the placeholder text "Cerca per indirizzo email, numero di telefono o UID utente" and a blue button labeled "Aggiungi utente". Below the search bar is a table with the following columns: "Identificatore", "Provider", "Data di creazione", "Accesso eseguito", and "UID utente". The table contains three rows of user data. At the bottom right, there is a pagination control showing "Righe per pagina: 50" and "1 - 3 of 3".

Identificatore	Provider	Data di creazione	Accesso eseguito	UID utente
prova@prova.it	✉	19 ago 2023	14 set 2023	LXnWXG4jUxU7IMng1qVKAVq7cR...
simonesgalla@libero.it	✉	14 ago 2023	14 ago 2023	yGpuRaK4WoXWw5PKUvIBsEvZtP...
andreamarini@gmail.com	✉	14 ago 2023	14 set 2023	XsG9TMGYHcePdLDsiEwHoZFjtLI1

Figura 4.10: Database di autenticazione

per utilizzare il prodotto. La procedura è piuttosto semplice: basta autenticarsi con il proprio account google a GoogleCloudPlatform, creare un nuovo progetto, selezionare il tasto "Console", andare nella sezione "API e servizi" e premere, infine, il tasto "Crea Credenziali". In questo modo si ottiene la chiave di accesso personale, che può essere gestita all'interno di GoogleCloudPlatform. A questo punto, basta inserire l'API key nell'apposita sezione di codice. Una spiegazione più chiara e completa può essere trovata nella documentazione ufficiale di GooglePlatform, cliccando all'indirizzo: <https://developers.google.com/maps/documentation/ios-sdk?hl=it>.

4.2.2 Firebase

Firebase è una piattaforma online che permette di usufruire di vari servizi per la gestione di dati. Nel mese di ottobre 2014, Firebase è entrato a fare parte dei servizi Google e, quindi, anche in questo caso, l'utente può usufruire delle API gratuite, che gli permettono di integrare Firebase all'interno di un progetto, mediante la gestione delle dipendenze dello stesso. Firebase offre diverse tipologie di funzionalità, alcune delle quali sono:

- Autenticazione, grazie alla quale, personalizzando il meccanismo (tramite email/password, con il numero di telefono, tramite fornitori di terze parti, come Google o Facebook...), lo sviluppatore può gestire le procedure di autenticazione di un utente;
- Firestore, ovvero un database per salvare informazioni generiche;
- Storage, che consente agli sviluppatori di archiviare file di grandi dimensioni come immagini e video.

Tutti i dati su Firebase vengono salvati in formato JSON ².

Nelle Figure 4.10 e 4.11 vengono riportati, rispettivamente, il database di Autenticazione ed il database di Firestore utilizzati nell'applicazione. Il meccanismo di autenticazione usato è quello classico di email/password.

Una delle caratteristiche principali di Firebase è che offre un database in tempo reale, ovvero sincronizza i dati tra tutti i dispositivi connessi in tempo reale. Questo consente di avere un aggiornamento *istantaneo* dei dati, ad esempio all'interno di una vista, e di mostrarli all'utente, senza la necessità di distruggere e ricreare da zero l'interfaccia (si pensi semplicemente ad una chat: ad ogni messaggio inviato/ricevuto l'interfaccia dovrà aggiornarsi con nuovi dati).

²Formato per lo scambio di dati.

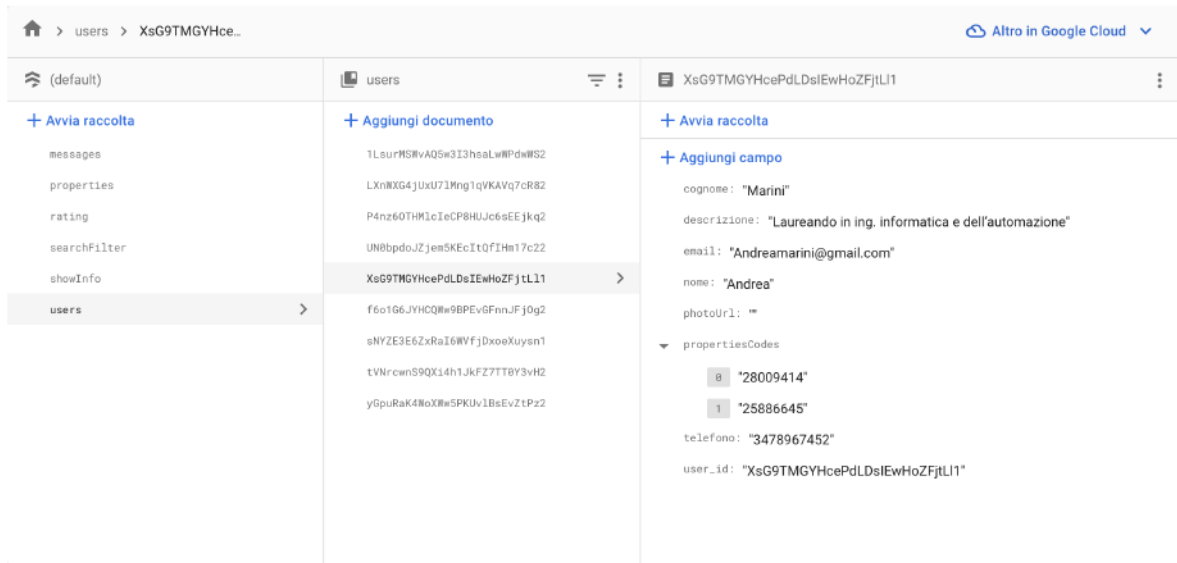


Figura 4.11: Database Firestore con le informazioni relative ad un utente

Una spiegazione più chiara e completa può essere trovata nella documentazione ufficiale di Firebase, cliccando all'indirizzo: <https://firebase.google.com/docs?hl=it>.

4.2.3 Postman

Postman è un ambiente di sviluppo API completo. Esso consente agli sviluppatori di effettuare richieste API, ispezionare i dati di richiesta e risposta ed eseguire il debug degli endpoint API. Le chiamate API possono essere di diverso tipo, ovvero:

- *Get*, per ottenere dati da un'API;
- *Post*, per inviare dati ad un'API;
- *Put/Patch*, per aggiornare dei dati esistenti;
- *Delete*, per eliminare dei dati esistenti.

Postman gestisce diversi tipi di formati di risposta come JSON, XML, HTML e testo semplice.

Infine, il software fornisce una funzione chiamata "Server simulato" che consente di simulare le risposte dell'API e testare le richieste senza la necessità di un server reale.

In Figura 4.12 è riportato un esempio di chiamata, che abbiamo effettivamente usato nell'applicazione, per ottenere l'`accessToken`³ necessario per fare la richiesta API ad Idealista.

4.2.4 Esempi significativi nel codice

In Figura 4.9 abbiamo mostrato come SwiftUI metta a disposizione il metodo `.task` per gestire la programmazione asincrona. Al di fuori dell'interfaccia, invece, questa può essere gestita mediante l'uso di due parole chiave, che sono:

- `async`, definita nell'intestazione di una funzione, che indica che quella funzione può eseguire una o più operazioni asincrone;

³Un token (codice) di accesso è utilizzato nell'autenticazione basata su token per consentire ad un'applicazione di accedere a un'API.

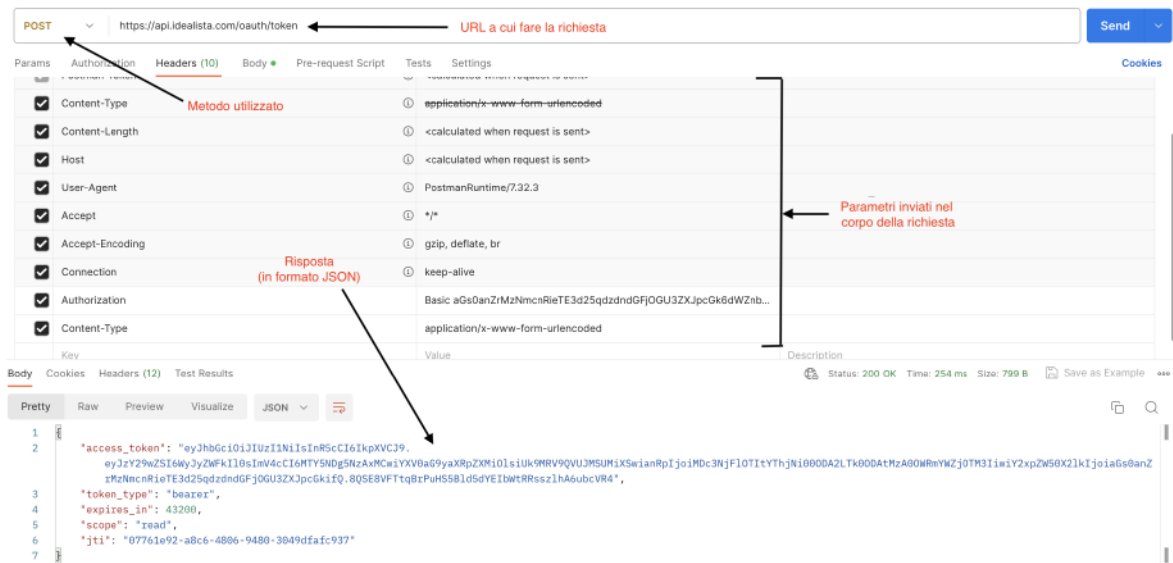


Figura 4.12: Chiamata API per l'ottenimento dell'accessToken

```
func loadData() async {
    let currentUser = try? authViewModel.getAuthenticatedUser()
    let snapshot = try? await Firestore.firestore().collection("users").document(currentUser!.uid).getDocument()
    savedProperties = snapshot!.data()?["propertiesCodes"] as? Array<String>
}
```

Figura 4.13: Uso di async e await

- `await`, utilizzata all'interno del corpo della funzione, che serve per sospendere quella funzione, fintanto che l'operazione asincrona marcata con `await` viene eseguita completamente, per poi ripartire dal punto in cui si è sospesa.

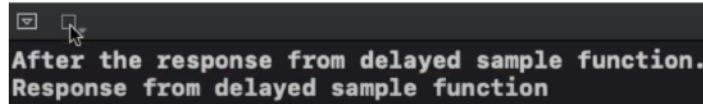
In Figura 4.13 vediamo un esempio per capire meglio questo funzionamento. In questo caso abbiamo definito una funzione `loadData`, nella cui intestazione è presente la parola chiave `async` per indicare che la funzione contiene istruzioni asincrone. Nel corpo della funzione, la seconda istruzione è marcata con `await`, e ciò indica che la funzione `loadData` si sospende finché l'istruzione non si è risolta (e quindi la variabile `snapshot` si è istanziata) per poi riprendere con la terza istruzione che utilizza la variabile `snapshot`. Tutto ciò è necessario in quanto la lettura da database può richiedere più o meno tempo e, senza l'uso di `await`, si rischierebbe di svolgere la terza istruzione senza che la seconda si sia completata, risultando, quindi, la variabile `snapshot` vuota.

Un'altra soluzione che abbiamo adottato nel nostro codice è la parola chiave `completion`. Il suo utilizzo è conveniente quando un'applicazione sta eseguendo qualcosa che potrebbe richiedere un po' di tempo, come effettuare una chiamata API, e bisogna fare qualcosa al termine dell'attività, come aggiornare l'interfaccia utente per mostrare i dati ottenuti. La parola chiave `completion` consente alla funzione di completare l'attività in modo asincrono, mettendo in pausa una componente, senza ritardare il resto dell'esecuzione del programma. In pratica, essa permette di definire delle istruzioni che verranno eseguite una volta che le istruzioni sul thread principale sono state completate. Vediamo il suo funzionamento con un'esempio nelle Figure 4.14 e 4.15. Nella prima figura è possibile notare come venga eseguita prima l'istruzione `print` ("After the response from delayed sample function") nonostante sia definita dopo l'istruzione `print` ("Response from delayed sample function"). Questo accade grazie all'uso del-

```
func sampleFunction2() {
    DispatchQueue.main.asyncAfter (deadline: DispatchTime.now() + 2) {
        print ("Response from delayed sample function" )
    }
    print ("After the response from delayed sample function.")
}

sampleFunction2 ()
```

(a) Funzione senza l'uso di completion



```
After the response from delayed sample function.
Response from delayed sample function
```

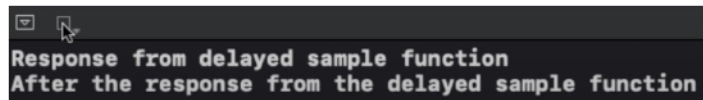
(b) Risultato in console senza l'uso di completion

Figura 4.14: Esempio senza completion

```
func sampleFunction3 (completion: @escaping () -> Void) {
    DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + 2) {
        print ("Response from delayed sample function")
    }
    completion ()
}

sampleFunction3 ( completion: {
    print ("After the response from delayed sample function.")
})
```

(a) Funzione con l'uso di completion



```
Response from delayed sample function
After the response from the delayed sample function
```

(b) Risultato in console con l'uso di completion

Figura 4.15: Esempio con completion

la funzione `DispatchQueue`, che ritarda l'esecuzione del thread principale di due secondi (viene usato per simulare la quantità di tempo maggiore che può richiedere, ad esempio, una chiamata API). Nella seconda figura, invece, l'istruzione `print ("After the response from delayed sample function")` viene eseguita dopo l'istruzione `print ("Response from delayed sample function")`, in quanto quest'ultima si trova nel thread principale che deve completarsi per primo (grazie all'uso di `completion`). Questo esempio fa capire come `completion` permetta di attendere il completamento di un'istruzione per mettere a disposizione, ad esempio, dati che richiedono più tempo per essere ottenuti. Senza l'uso di `completion` avremmo, magari, una vista che non tiene conto di questi dati in quanto viene costruita prima del loro ottenimento.

In ogni caso, le parole chiave `async` e `await` sono più utilizzate di `completion`, poichè permettono di fare le stesse cose in modo più efficiente e con un codice più "pulito".

Nelle Figure 4.16-4.18 è riportato l'utilizzo di `completion` nella nostra applicazione per l'ottenimento dell'`accessToken`, mediante chiamata API, che sarà poi utilizzato per fare un'altra chiamata API ad Idealista per l'ottenimento delle informazioni sulle abitazioni.

```
func getAccessToken(completion: @escaping () -> Void) {  
  
    if let url = URL(string: "https://api.idealista.com/oauth/token"){  
        var request = URLRequest(url: url)  
        request.httpMethod = "POST"  
        let parameters: [String : Any] = [  
            "grant_type": "client_credentials",  
            "scope": "read"  
        ]  
  
        request.setValue("Basic aGs0anZrMzNmcmRieTE3d25qdzdndGFjOGU3ZXJpcGk6dWZnbWVhVWZaRHV0", forHTTPHeaderField: "Authorization")  
        //request.setValue("Basic bm15M2ptMG11ODBqenYwcDV1b3dhcHNjeXVhbGl6OTc6dk5YM0t2QUNPNzZz", forHTTPHeaderField: "Authorization")  
        request.httpBody = parameters.percentEscaped().data(using: .utf8)  
  
        URLSession.shared.dataTask(with: request) { (data, response, error) in  
            guard let data = data else {  
                if error == nil{  
                    print(error?.localizedDescription ?? "Unknown Error")  
                }  
                return  
            }  
  
            if let response = response as? HTTPURLResponse{  
                guard (200 ... 299) ~= response.statusCode else {  
                    print("Status code :- \(response.statusCode)")  
                    print(response)  
                    return  
                }  
            }  
  
            do{  
                let json = try JSONSerialization.jsonObject(with: data, options: []) as? [String: Any]  
                self.accessToken = (json!["access_token"] as? String)!  
            }catch let error{  
                print(error.localizedDescription)  
            }  
            completion()  
        }.resume()  
    }  
}
```

Figura 4.16: Codice della funzione per ottenere l'accessToken

```

func postRequestIdealista(completion: @escaping () -> Void, data: [String: Any]) {

    if let url = URL(string: "https://api.idealista.com/3.5/it/search"){

        let latitude = data["latitude"] as! String
        let longitude = data["longitude"] as! String

        var request = URLRequest(url: url)
        request.httpMethod = "POST"

        var parameters: [String : Any] = [
            "propertyType": "bedrooms",
            "locale": "it",
            "operation": "rent",
            "center": latitude+","+longitude,
            "distance": data["distance"] as! Double,
            "maxPrice": data["price"] as! Double
        ]

        if(data["distance"] as! Double == 0) {
            parameters["distance"] = 15000
        }
        if(data["price"] as! Double == 0) {
            parameters["maxPrice"] = 1000
        }

        request.setValue("Bearer \(self.accessToken)", forHTTPHeaderField: "Authorization")
        request.httpBody = parameters.percentEscaped().data(using: .utf8)

        URLSession.shared.dataTask(with: request) { (data, response, error) in
            guard let data = data else {
                if error == nil{
                    print(error?.localizedDescription ?? "Unknown Error")
                }
                return
            }

            if let response = response as? HTTPURLResponse{
                guard (200 ... 299) ~= response.statusCode else {
                    print("Status code :- \(response.statusCode)")
                    print(response)
                    return
                }
            }

            do{
                let jsonDecoder = JSONDecoder()
                let jsonString = String(data: data, encoding: .utf8)
                self.JsonData = jsonString!.data(using: .utf8)
                let decodedData = try jsonDecoder.decode(JSONObject.self, from: self.JsonData!)

                let properties = decodedData.elementList
                self.properties = properties
            }catch let error{
                print(String(describing: error))
            }

            completion()
        }.resume()
    }
}

```

Figura 4.17: Codice della funzione per fare una chiamata API a Idealista

```
func getIdealistaProperties(completion: @escaping () -> Void) async{
    var data: [String: Any]
    let currentUser = try? AuthViewModel().getAuthenticatedUser().uid
    let snapshot = try? await Firestore.firestore().collection("searchFilter").document(currentUser!).getDocument()
    if (snapshot?.data() != nil) {
        data = snapshot!.data()!
        self.getAccessToken() {
            self.postRequestIdealista(completion: {completion()}, data: data)
        }
    }
}
```

Figura 4.18: Utilizzo di completion con getAccessToken e postRequestIdealista

In questo capitolo verranno riportate le immagini ed una spiegazione dettagliata del funzionamento dell'applicazione per un utente comune, in modo che la stessa possa essere utilizzata completamente e senza ambiguità.

5.1 Autenticazione

La prima cosa che un utente deve fare non appena avvia l'applicazione è l'autenticazione, in modo da recuperare tutte le informazioni relative al proprio profilo. Ci sono due modi per autenticarsi: tramite login, nel caso in cui l'utente abbia già un profilo, oppure tramite registrazione, nel caso in cui l'utente utilizza l'applicazione per la prima volta. In Figura 5.1 sono riportate le schermate di login e di registrazione.

Poichè la prima schermata che si presenta quando si avvia l'applicazione è quella di login, nel caso in cui non abbia mai eseguito l'accesso, l'utente deve cliccare sul link "Registrati", evidenziato in celeste. In questo modo l'utente verrà portato alla schermata di registrazione. Sia nel caso del login che della registrazione, se l'utente inserisce i dati in modo corretto, verrà autenticato e potrà finalmente usare l'applicazione.

5.2 Utilizzo della mappa

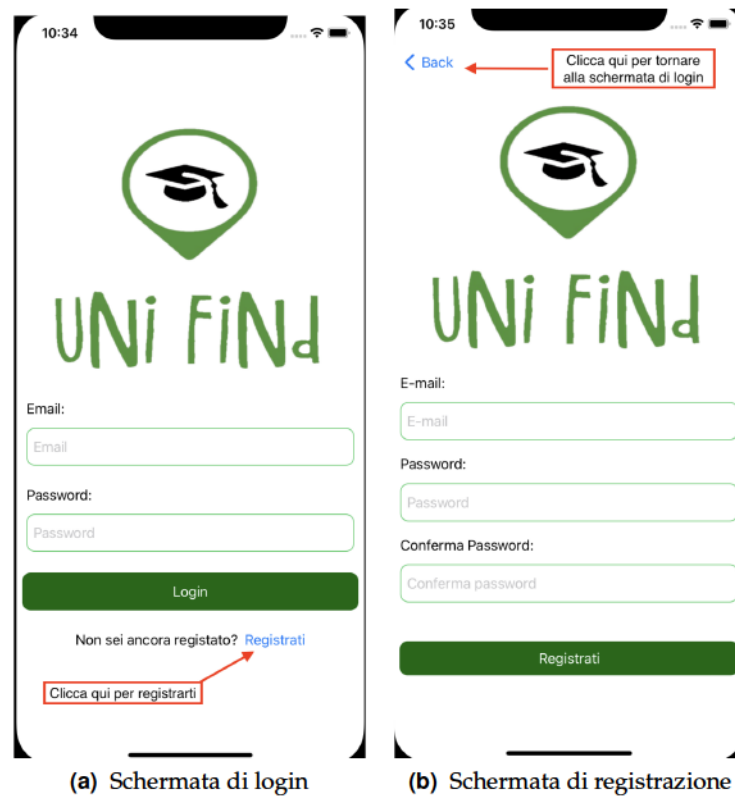
Una volta all'interno dell'applicazione, la schermata che si mostra all'utente è quella relativa alla mappa (Figura 5.2).

L'utente può muoversi tra le varie schermate utilizzando la barra di navigazione situata nella parte inferiore dello schermo.

Come già spiegato precedentemente, l'utente può ricercare una località di interesse in due modi, tramite barra di ricerca, digitando il nome della città di interesse, oppure tramite click sulla mappa.

Supponiamo di voler ricercare delle abitazioni a Roma. In Figura 5.3 viene mostrato il risultato di una ricerca con uno dei metodi sopra indicati.

Le icone nere con le case rappresentano le abitazioni trovate come risultato della ricerca. Cliccando su una delle icone è possibile ottenere più informazioni sulla relativa abitazione (Figura 5.4).

**Figura 5.1:** Schermata di login e di registrazione**Figura 5.2:** Schermata con la mappa

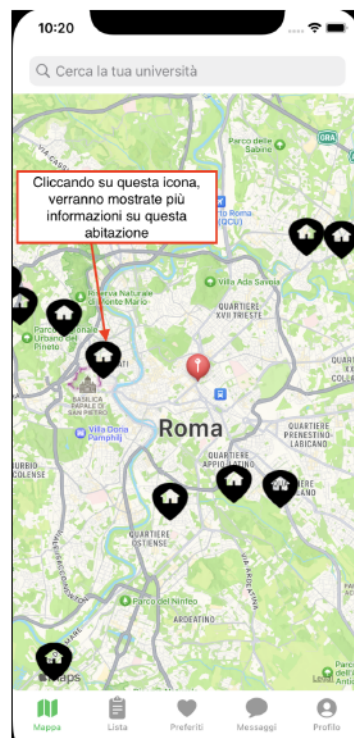


Figura 5.3: Risultato della ricerca tramite barra di ricerca o click sulla mappa



Figura 5.4: Maggiori informazioni su un'abitazione

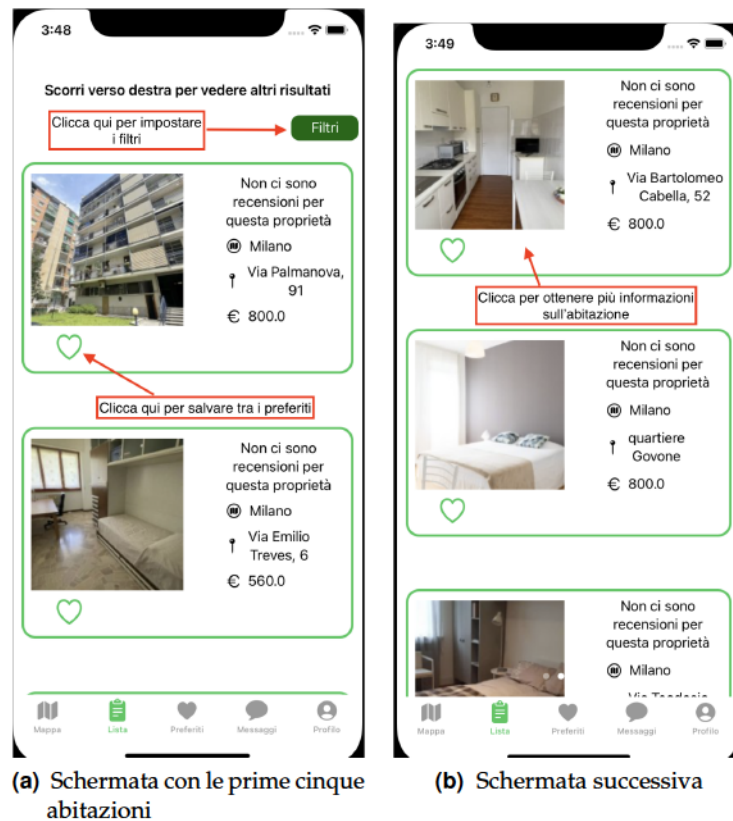


Figura 5.5: Schermate della lista di abitazioni

5.3 Utilizzo della lista delle abitazioni

Se si avvia questa sezione prima di effettuare una ricerca su mappa, possono verificarsi due situazioni: se l'utente non ha mai effettuato una ricerca, allora, al centro della schermata, sarà presente una scritta del tipo "La ricerca non ha prodotto risultati"; altrimenti, se l'utente ha effettuato una nuova ricerca o ne ha effettuate precedentemente, sarà presente una lista di abitazioni risultanti dalla ricerca. Potendo tale lista essere eccessivamente lunga, per avere una visione più ordinata, abbiamo suddiviso la stessa in più schermate (massimo 4, con 5 elementi per schermata). Per passare da una schermata all'altra, è necessario scorrere con il dito verso destra. In Figura 5.5 vengono riportate la schermata con le prime cinque abitazioni e una delle schermate successive.

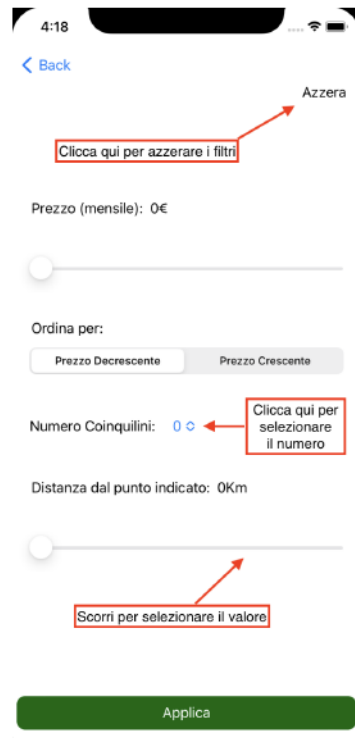
Cliccando su una qualsiasi abitazione, è possibile ricondursi alla schermata che mostra maggiori informazioni su di essa (Figura 5.4).

Inoltre, dalla prima schermata si può accedere alla sezione relativa ai filtri (Figura 5.6), tramite il pulsante omonimo, per impostare la ricerca in base alle esigenze dell'utente. Cliccando sul pulsante "Applica", viene effettuata una nuova ricerca, sempre nella città indicata nella ricerca precedente, in base ai parametri di filtraggio.

Infine, in tutte le schermate della lista delle abitazioni, è possibile aggiungere/togliere una proprietà ai/dai preferiti. Se l'abitazione si trova nella lista dei preferiti, questa sarà denotata da un cuore pieno, altrimenti da un cuore vuoto.

Per accedere alla schermata dei preferiti (Figura 5.7), è necessario cliccare sul pulsante relativo (per intenderci, quello a forma di cuore) nella barra di navigazione.

Anche in questa sezione è possibile accedere alla schermata relativa alle maggiori informazioni (Figura 5.4), cliccando, semplicemente, sul rettangolo dell'abitazione.

**Figura 5.6:** Schermata dei filtri

Infine, se si vuole togliere un'abitazione dalla lista dei preferiti direttamente da questa schermata, è necessario cliccare sull'icona a forma di cuore.

5.4 Utilizzo della chat

La chat tra due utenti è possibile se entrambi hanno almeno un'abitazione in comune tra i preferiti.

Si può accedere alla schermata relativa alla chat in due modi: tramite la barra di navigazione nella parte inferiore dello schermo oppure tramite il pulsante "Chatta con altri utenti interessati" nella schermata che mostra le informazioni di un'abitazione (Figura 5.4). In entrambi i casi, l'utente viene prima indirizzato ad una schermata contenente una lista di utenti con le stesse abitazioni tra i preferiti (Figura 5.8) e poi, selezionando l'utente di interesse, si apre la schermata con la relativa chat (Figura 5.9).

Per inviare un nuovo messaggio, è necessario cliccare sulla casella di testo nella parte inferiore dello schermo, scrivere il messaggio e poi inviarlo cliccando sull'icona posta alla destra della casella di testo.

Inoltre, cliccando sull'email o sulla foto profilo dell'utente con cui si sta chattando, si viene indirizzati alla schermata che mostra le informazioni relative a quell'utente (Figura 5.10).

Infine, sempre dalla schermata della chat tra due utenti (Figura 5.9), è possibile visualizzare tutte le proprietà che i due utenti hanno in comune. Per fare ciò, è necessario cliccare sul rettangolo con l'abitazione indicata nella parte superiore dello schermo. A questo punto, l'utente verrà indirizzato alla schermata relativa alle maggiori informazioni di un'abitazione (Figura 5.4), e, se presenti più abitazioni in comune, scorrendo con il dito verso destra, sarà possibile visualizzare tutte le informazioni relative alle altre abitazioni in comune.



Figura 5.7: Schermata dei preferiti

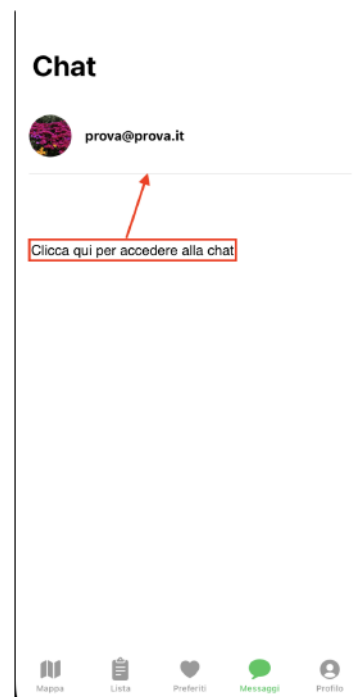


Figura 5.8: Schermata con la lista degli utenti con cui chattare

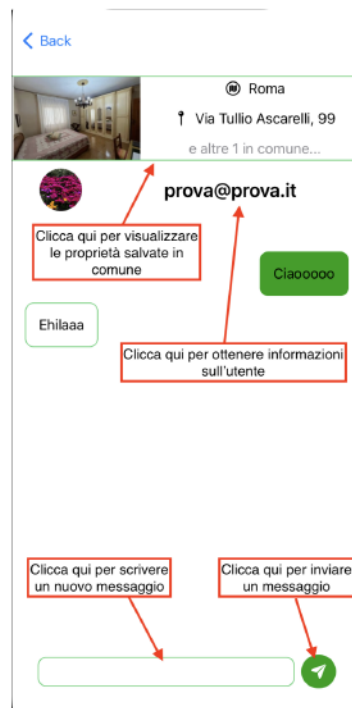


Figura 5.9: Schermata della chat tra due utenti

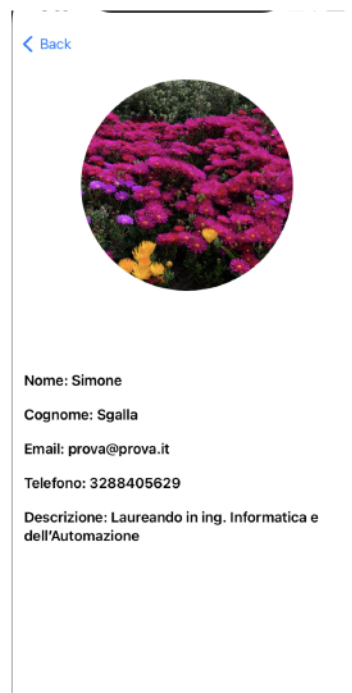


Figura 5.10: Informazioni sull'utente con cui si sta chattando



Figura 5.11: Schermata relativa al proprio profilo

5.5 Utilizzo della sezione del profilo

Cliccando sull'ultima icona della barra di navigazione, è possibile accedere alla schermata relativa al proprio profilo (Figura 5.11).

Premendo sull'immagine nella parte superiore dello schermo, è possibile cambiare la foto del profilo. Infatti, si aprirà la galleria del proprio telefono e, selezionando un'immagine di interesse, la foto del profilo si aggiornerà con la nuova immagine selezionata.

Inoltre, cliccando gli opportuni pulsanti, è possibile modificare le proprie credenziali di accesso e le informazioni del profilo.

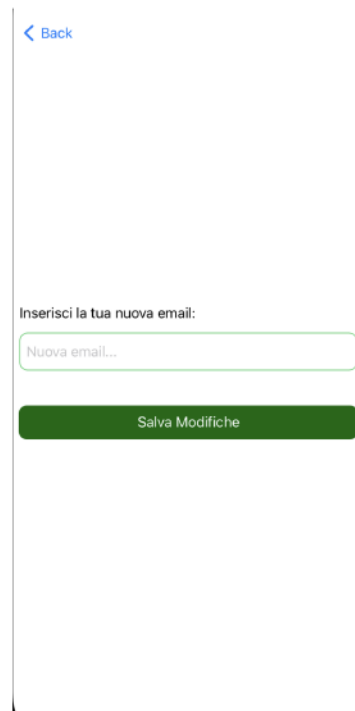
Partiamo dal pulsante "Modifica Email". Quando viene cliccato, l'utente viene indirizzato alla schermata per la modifica della propria email di accesso (Figura 5.12). Se egli inserisce un'email valida nella casella di testo, premendo sul pulsante "Salva Modifiche", l'email di accesso verrà aggiornata.

Cliccando sul pulsante "Modifica Password", l'utente viene indirizzato alla schermata per la modifica della propria password di accesso (Figura 5.13). Se l'utente inserisce una password valida e le password inserite nelle due caselle di testo coincidono, premendo sul pulsante "Salva Modifiche", la password di accesso verrà aggiornata.

Cliccando sul pulsante "Completa il tuo profilo", l'utente viene indirizzato alla schermata per la modifica delle proprie informazioni personali (Figura 5.14). Riempendo le caselle di testo con le opportune informazioni, che vengono indicate sopra ogni casella di testo, e premendo sul pulsante "Salva Modifiche", le informazioni del profilo di un utente verranno aggiornate e visualizzate in Figura 5.11.

Infine, cliccando sul pulsante di logout, verrà effettuata la disconnessione dal proprio profilo e l'utente verrà indirizzato alla schermata di login (Figura 5.1).

In molte schermate, nella parte in alto a sinistra, è presente un pulsante con scritto "Back", che serve per tornare alla schermata precedente.



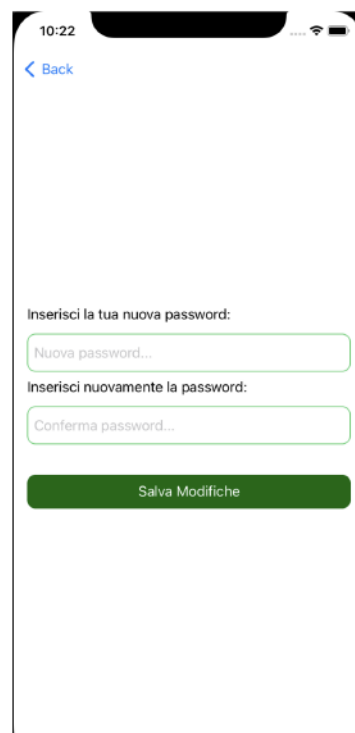
< Back

Inserisci la tua nuova email:

Salva Modifiche

This is a mobile app screen for modifying an email. It features a blue back arrow and the text '< Back' at the top left. Below this, the instruction 'Inserisci la tua nuova email:' is displayed. A text input field with a light green border contains the placeholder text 'Nuova email...'. At the bottom of the screen is a dark green button with the white text 'Salva Modifiche'.

Figura 5.12: Schermata per la modifica dell'email



10:22

< Back

Inserisci la tua nuova password:

Inserisci nuovamente la password:

Salva Modifiche

This is a mobile app screen for modifying a password. The status bar at the top shows the time '10:22' and signal icons. A blue back arrow and the text '< Back' are at the top left. The instruction 'Inserisci la tua nuova password:' is followed by a text input field with a light green border containing the placeholder 'Nuova password...'. Below this, the instruction 'Inserisci nuovamente la password:' is followed by another text input field with a light green border containing the placeholder 'Conferma password...'. At the bottom is a dark green button with the white text 'Salva Modifiche'.

Figura 5.13: Schermata per la modifica della password

< Back

Inserisci il tuo nome:

Andrea

Inserisci il tuo cognome:

Marini

Inserisci il tuo numero di telefono:

3478967452

Inserisci una breve descrizione:

Laureando in ing. informatica e dell'automazione

Salva Modifiche

Figura 5.14: Schermata per aggiornare le informazioni del proprio profilo

In questo capitolo verrà trattata, in un primo momento, l'analisi SWOT, ovvero uno strumento di pianificazione strategica fondamentale per qualsiasi impresa, che permette, attraverso l'attenta valutazione delle caratteristiche che definiscono l'ambiente interno ed esterno all'azienda, di individuare obiettivi specifici e tracciare un percorso chiaro verso il successo del business. Utilizzando, poi, i risultati ottenuti dall'analisi SWOT, nella seconda parte del capitolo verranno descritti alcuni siti web di spicco nel mercato immobiliare, che serviranno per fare un confronto con l'applicazione da noi realizzata.

6.1 Analisi SWOT del sistema realizzato

L'analisi SWOT è una tecnica utilizzata per identificare i punti di forza, di debolezza, le opportunità e le minacce di un progetto specifico. Essa costituisce un potente strumento che aiuta a identificare opportunità di miglioramento e, a livelli più alti, a pianificare adeguatamente la crescita di un'organizzazione, intesa come azienda e team di sviluppatori. L'acronimo SWOT sta per *Strengths* (punti di forza), *Weaknesses* (punti di debolezza), *Opportunities* (opportunità) e *Threats* (minacce).

6.1.1 Punti di forza

I punti di forza nell'analisi SWOT sono quelle iniziative interne che danno buoni risultati. Questa valutazione aiuta a capire cosa funziona; una volta individuati i punti di forza, si potranno applicare in altri ambiti, ad esempio per migliorare l'efficienza del team.

Quando si esaminano i punti di forza della propria organizzazione, ci si pongono domande del tipo:

- Cosa facciamo bene?
- Cosa ha di unico il nostro team?

Fatta questa piccola premessa, discutiamo alcuni punti di forza del nostro sistema.

Spiegando ad Idealista in cosa consistesse la nostra applicazione, ci hanno fornito l'accesso e la documentazione delle loro API, in modo da poterle utilizzare al meglio. Le API di Idealista non sono, infatti, aperte a tutti, ma, per ottenerle, è necessario convincere della validità della propria idea gli amministratori del sito.

Per quanto riguarda, invece, le API di Google Maps e Firebase, che, come già detto precedentemente, sono aperte a tutti e gratuite, abbiamo imparato ad utilizzarle in modo discreto, coprendo tutte le principali funzionalità.

6.1.2 Punti di debolezza

I punti di debolezza nell'analisi SWOT si riferiscono a iniziative interne che non rendono quanto dovrebbero. Identificare i punti deboli interni fornisce un punto di partenza per migliorare l'organizzazione e il progetto.

Per la ricerca dei punti di debolezza, alcune domande che ci si possono porre sono:

- Quali iniziative sono poco performanti e perché?
- Cosa può essere migliorato?
- Quali risorse potrebbero migliorare le nostre prestazioni?

Elenchiamo, ora, alcuni punti di debolezza del nostro sistema.

Uno dei principali fattori di debolezza del nostro team è il numero ristretto di componenti dello stesso e questo causa un minore flusso e scambio di idee. Inoltre, le risorse e le capacità del gruppo saranno, inevitabilmente, limitate ed ognuno dei componenti del team dovrà avere una conoscenza approssimativa su tutto, senza essere specializzato in un ambito piuttosto che in un altro. Però, è intuibile che questo porta alla realizzazione di un prodotto utilizzabile, ma non di valore.

Un altro grande limite della nostra applicazione è il numero limitato di richieste API. Idealista, infatti, ci ha messo a disposizione 100 richieste al mese, che sono appena sufficienti per lo sviluppo dell'applicazione. Con queste risorse non è, quindi, possibile mettere a disposizione l'applicazione di un pubblico effettivo.

6.1.3 Opportunità

Le opportunità nell'analisi SWOT sono il risultato degli attuali punti di forza e debolezza, insieme a qualsiasi iniziativa esterna che metterà l'azienda in una posizione competitiva più forte. Le opportunità potrebbero trattarsi di punti deboli che si vorrebbero migliorare o aree che non sono state esplorate nelle prime due fasi dell'analisi (punti di forza e debolezza).

Poiché ci sono vari modi per trovare delle opportunità, è utile considerare queste domande prima di iniziare:

- Quali risorse possiamo utilizzare per migliorare i punti deboli?
- Ci sono aspetti del mercato non coperti dai nostri servizi?

Essendo studenti universitari, nel nostro contesto, non abbiamo molte opportunità. Un'idea valida potrebbe essere aumentare il nostro team con altri studenti e, partendo da questo, il panorama delle opportunità potrebbe espandersi. Una delle piccole opportunità che mi viene in mente, una volta che il team si è ben composto ed affermato, potrebbe essere richiedere API illimitate ad Idealista, oppure ad altri siti, in modo da poter realizzare un prodotto software più complesso.

6.1.4 Minacce

Le minacce nell'analisi SWOT si riferiscono ad aree che potrebbero creare problemi. Esse sono diverse dai punti deboli, in quanto sono esterne e generalmente fuori dal controllo dell'organizzazione. Esse possono essere di vario tipo, a partire da un imprevisto fino ad arrivare ad un cambiamento nel panorama competitivo.

Queste sono alcune domande che ci si dovrebbe porre per identificare le minacce esterne:

- Dove la concorrenza fa meglio di noi?

- Quali aspetti nel settore sono motivo di preoccupazione?

Nel caso della nostra applicazione ci sono già moltissimi siti Internet che permettono di ricercare abitazioni in affitto, come <https://www.immobiliare.it> oppure <https://www.booking.com/index.it.html>. La grande differenza sta nel fatto che questi siti permettono all'utente di mettersi direttamente in contatto con i proprietari delle abitazioni, oppure, ai proprietari stessi, di inserire annunci delle abitazioni che intendono affittare. Nella nostra applicazione, invece, l'utente può soltanto visualizzare le case (messe a disposizione da Idealista) e chattare con altri utenti interessati alla stessa abitazione; ad esempio, l'applicazione non prevede la funzione, per i proprietari delle abitazioni, di inserire un annuncio.

6.2 Confronto con sistemi correlati

In questa sezione confronteremo il nostro software con altri siti web relativi al settore immobiliare. Prima di partire con una breve descrizione di alcuni portali di spicco per l'acquisto/la vendita di immobili, è opportuno ricordare che il nostro prodotto è stato realizzato per scopi accademici e quindi il confronto risulterà, fin dall'inizio, impari. Cercheremo, in ogni caso, di evidenziare alcune caratteristiche che potrebbero migliorare la nostra applicazione.

6.2.1 Immobiliare

Immobiliare è stato lanciato nel 2007 e, da quel momento, è diventato uno dei principali portali italiani per l'affitto e la vendita delle case. Esso offre anche servizi per valutare la propria abitazione e per seguire l'andamento del mercato immobiliare. Il portale risulta facilmente utilizzabile da tutti, sia su desktop (tramite sito web) sia su smartphone (tramite applicazione), supportando gli agenti immobiliari nell'attività di intermediazione.

Per usare Immobiliare, è necessario collegarsi alla relativa home page (cliccando sul link <https://www.immobiliare.it> oppure scaricando l'applicazione Figura 6.1), selezionare l'opzione "Affitta", digitare il nome della città in cui si intende effettuare la ricerca e selezionare il suggerimento pertinente dal menù che compare. In seguito, bisogna assicurarsi che, nel menù a tendina posto sulla destra, risulti selezionata l'opzione "Case - Appartamenti". Se si è interessati ad altro, cliccando sul simbolo "v", si può ricercare un immobile specifico (ad esempio, Appartamento, Loft, Rustico - Casale, etc.) Si possono applicare ulteriori filtri alla ricerca, come prezzo (minimo e massimo) e superficie dell'abitazione. Dopo aver compiuto questi passaggi, si è pronti per avviare la ricerca della casa, cliccando sul pulsante "Cerca" collocato a destra. Nella nuova pagina con i risultati degli annunci, si può individuare quello di interesse dall'elenco e cliccarci, così da poterlo visionare in maniera dettagliata (Figura 6.2). Le abitazioni sono visualizzabili anche su mappa, facendo clic sul simbolo del segnaposto che si trova in corrispondenza del titolo dell'inserzione (dopo averla aperta). Se si è realmente interessati, ci si può mettere in contatto con l'inserzionista telefonicamente, oppure tramite email, usando i pulsanti e i campi per i recapiti collocati a destra, nel box "Contatta l'inserzionista".

Se si è, invece, un inserzionista, nella home page del sito, è presente il pulsante "Pubblica", al cui click, si aprirà una pagina che guida l'inserzionista nella pubblicazione di un nuovo annuncio (Figura 6.3). Sarà, infine, compito di Immobiliare verificare l'autenticità e l'affidabilità dell'inserzionista e del suo annuncio.



Figura 6.1: Home Page di Immobiliare.it

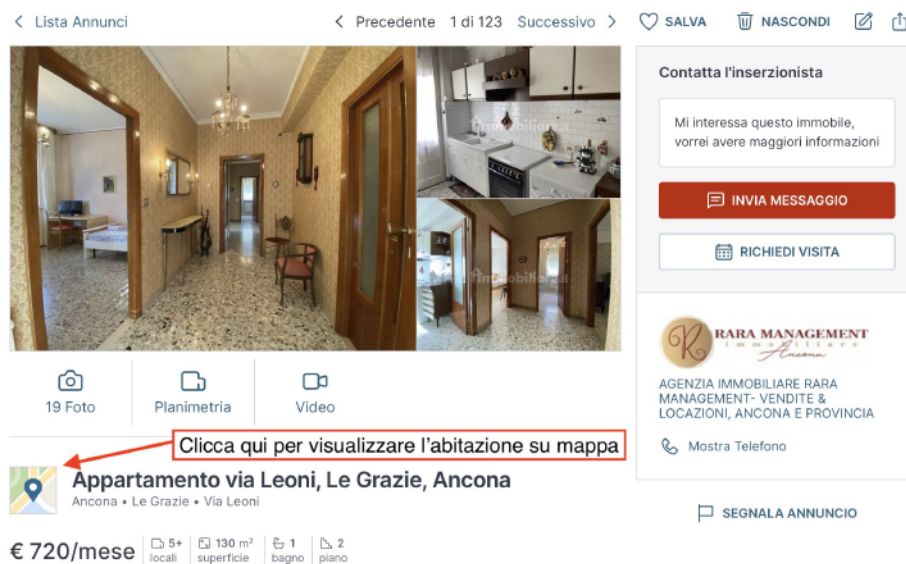


Figura 6.2: Schermata con informazioni dettagliate di un'abitazione (Immobiliare)



Figura 6.3: Schermata per la pubblicazione di un annuncio (Immobiliare)

6.2.2 Idealista

Le stesse funzionalità elencate per Immobiliare sono presenti anche su Idealista, ovvero un altro dei principali siti web per la ricerca e la pubblicazione di immobili. Come nel caso di Immobiliare, anche Idealista è supportato da un'applicazione per dispositivi, sia con sistema operativo Android sia con sistema operativo iOS.

Per completezza, nelle Figure 6.4, 6.5 e 6.6 riportiamo la home page, la schermata con tutte le informazioni per un'abitazione e la schermata per la pubblicazione di un annuncio da parte di un inserzionista.

6.2.3 Confronto con la nostra applicazione

Esistono anche tanti altri siti web (con le relative applicazioni) che presentano funzionalità di questo tipo (Booking, Casa, etc.) e, quindi, per evitare di ripeterci, abbiamo preferito ripor-



Figura 6.4: Home Page di Idealista.it

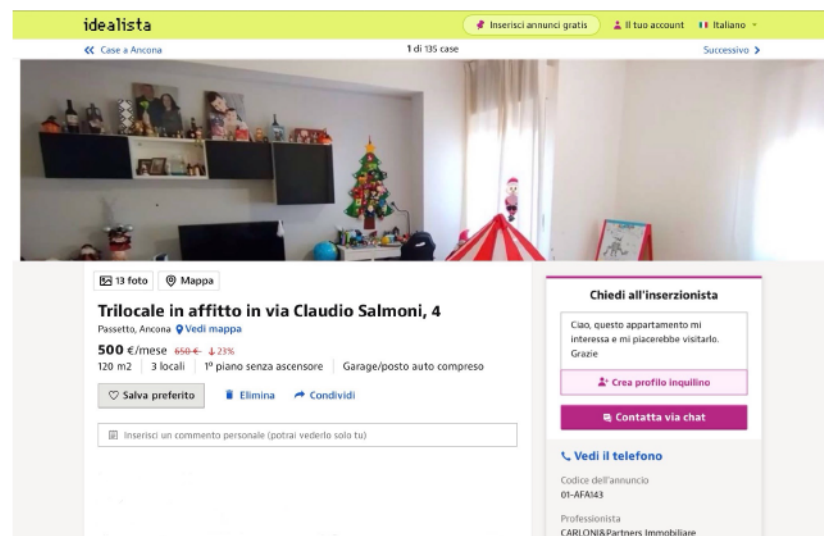


Figura 6.5: Schermata con informazioni dettagliate di un'abitazione (Idealista)

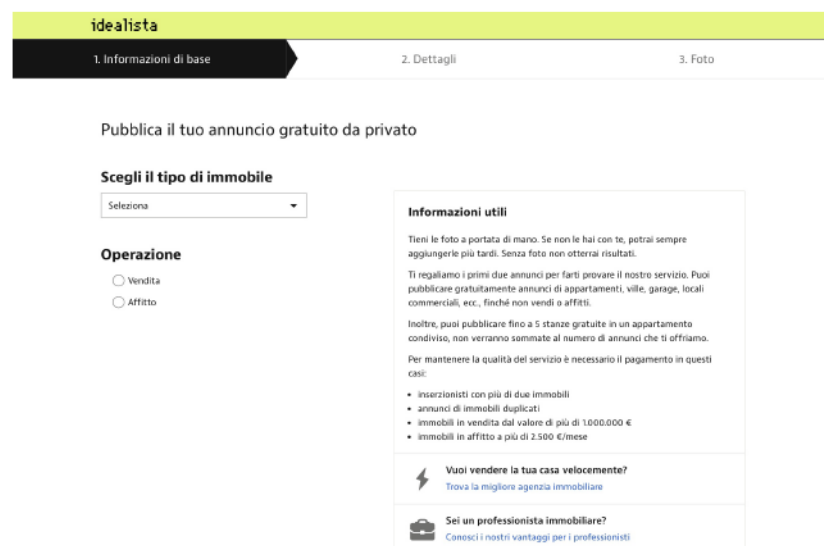


Figura 6.6: Schermata per la pubblicazione di un annuncio (Idealista)

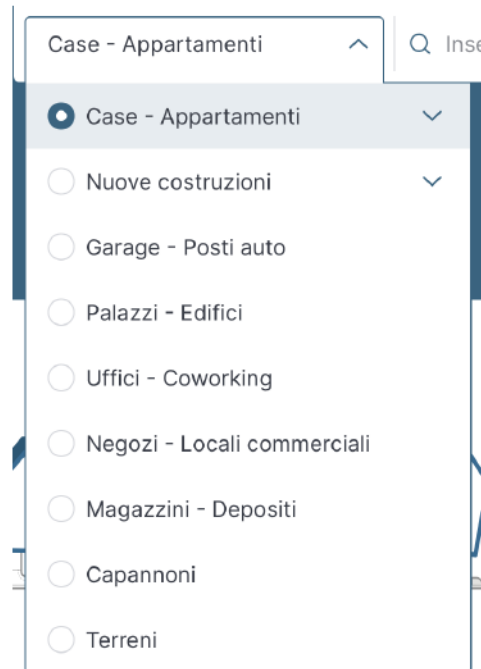


Figura 6.7: Tipologie di immobile in Immobiliare

tare <https://www.immobiliare.it> e <https://www.idealista.it>, di cui abbiamo usato le API.

A parte alcune peculiarità, come la possibilità di disegnare direttamente su mappa la propria area di interesse (anche con poligoni irregolari), e di essere notificati nel caso in cui venga aggiunto un annuncio con parametri simili alle ricerche precedenti, la differenza sostanziale tra tutti questi siti e la nostra applicazione è la mancanza di tutta la parte relativa alla gestione degli annunci. La nostra applicazione è limitata all'utente che intende comprare/affittare un'abitazione, escludendo, quindi, tutti coloro che, invece, vorrebbero inserire nel mercato immobiliare la propria dimora. Sempre da questo punto di vista, UniFind non prevede alcun tipo di comunicazione tra utenti e inserzionisti, nè, tantomeno, l'intermediazione da agenzie.

Inoltre, sempre per la scarsità di API messa a nostra disposizione, in Immobiliare ed Idealista, è presente una gamma più vasta di tipologie di immobili. È, infatti, possibile selezionare immobili di ogni tipo, a partire da ville ed appartamenti fino ad arrivare a garage e terreni (Figura 6.7).

Conclusioni

In questa tesi, siamo partiti con un'introduzione a Swift, trattando sia aspetti storici che tecnici, a SwiftUI, ovvero il framework per la realizzazione di interfacce utente, e a XCode, cioè l'IDE utilizzato per lo sviluppo dell'applicazione. Ciò che è emerso è sicuramente l'innovazione e la facilità d'uso portate da questo linguaggio di programmazione. Successivamente siamo entrati nel dettaglio dell'applicazione realizzata. Siamo partiti con la stesura e con la rappresentazione dei requisiti funzionali e non funzionali e dei modelli dei casi d'uso, il tutto realizzato utilizzando il linguaggio UML. Dopodichè, entrando nel pieno della fase di progettazione, abbiamo trattato il diagramma delle classi, contenente classi, metodi ed attributi che rientrano, effettivamente, nella logica implementativa. Sempre nello stesso capitolo, abbiamo rappresentato anche diagrammi di attività e di sequenza, in modo da esplicitare meglio le interazioni dell'utente con l'applicazione e il comportamento di quest'ultima. Come ultima fase, abbiamo parlato dell'implementazione, facendo la distinzione tra frontend, ovvero la parte di applicazione visibile all'utente, e backend, ovvero tutta la parte che gestisce la logica. Oltre ai mockup delle schermate, in questo capitolo abbiamo riportato degli esempi salienti di codice, contenenti la costruzione di interfacce utente e soluzioni interessanti adottate. Finita la parte tecnica di descrizione dell'applicazione, siamo passati al manuale utente, che ha il compito di illustrare il funzionamento ad un utente comune ed eliminare ogni tipo di ambiguità d'uso. Abbiamo riportato delle immagini, che rappresentano le schermate dell'applicazione, con delle frecce che indicano all'utente cosa fare. Infine, nel penultimo capitolo, abbiamo trattato l'analisi SWOT, ovvero lo strumento, per eccellenza, per l'analisi dell'ambiente interno ed esterno dal nostro team e per l'individuazione delle opportunità di miglioramento. Nello stesso capitolo, abbiamo anche parlato di alcuni siti immobiliari affermati, analizzandone le caratteristiche e confrontandole con quelle della nostra applicazione.

Per quanto riguarda i possibili sviluppi futuri, pensiamo, innanzitutto, di perfezionare ciò che è già presente nella nostra applicazione, ottimizzandola e rendendola più user-friendly. Inoltre, pensiamo di aggiungere tutta la parte relativa alla gestione degli annunci, in modo che l'applicazione possa essere utilizzata anche da inserzionisti e agenzie immobiliari, che fungono da intermediarie. In questo modo potremmo, finalmente, farci notare dai siti Internet interessati al mercato immobiliare e far sì che ci facciano usufruire, in modo illimitato, delle loro API. Da questo momento in poi, la nostra applicazione potrebbe risultare pressoché completa e, grazie ad API illimitate e, magari, anche a sponsorizzazioni, essere inserita nel mercato, rendendola disponibile ad un pubblico più vasto. Nostro compito sarà, poi, aggiornarla continuamente per renderla competitiva e ben adattata al mercato.

- BERLIN, J. e CACHEAUX, R. (2022), *Advanced iOS App Architecture (Fourth Edition): Real-World App Architecture in Swift*, Razeware LLC.
- BRAGAGNOLO, L. (2015), *Swift. Il linguaggio per creare applicazioni iOS e OS X*, Apogeo.
- DIAS, R., HUSSAIN, A. e STEELE, J. (2022), *Real-World iOS by Tutorials (First Edition): Professional App Development With Swift*, Razeware LLC.
- GAUCHAT, J. (2022), *SwiftUI for Masterminds 3rd Edition 2022: How to take advantage of Swift and SwiftUI to create insanely great apps for iPhones, iPads, and Macs*.
- GIORDANO SCALZO, E. N. (2021), *SwiftUI Cookbook: A guide to solving the most common problems and learning best practices while building SwiftUI apps, 2nd Edition*, Packt Publishing.
- KULSHRESTHA, A. (2021), *iOS 15 Application Development for Beginners: Learn Swift Programming and Build iPhone Apps with SwiftUI and Xcode 13*, bpb.
- LAU, K. e NGO, V. (2021), *Data Structures & Algorithms in Swift (Fourth Edition): Implementing Practical Data Structures with Swift*, Razeware LLC.
- MAZZONE, C. A. (2022), *Sviluppare per il web e il mobile*, Edizione LSWR.
- NEUBURG, N. (2021), *Ios 15 Programming Fundamentals With Swift: Swift, Xcode, and Cocoa Basics*, Oreilly & Associates Inc.
- SAHAR, A. e CLAYTON, C. (2022), *iOS 16 Programming for Beginners: Kickstart your iOS app development journey with a hands-on guide to Swift 5.7 and Xcode 14, 7th Edition*, Packt Publishing.

Siti Web consultati

- Macitynet – <https://www.macitynet.it>
- Apple, il sito ufficiale – <https://www.apple.com/it>
- HTML, che offre numerose guide – <https://www.html.it>

- XCoding – <https://www.xcoding.it>
- Creately – <https://creately.com>
- Appmaster – <https://appmaster.io>
- Google Cloud Platform – <https://cloud.google.com>
- Firebase – <https://firebase.google.com>
- Idealista, un portale immobiliare – <https://www.idealista.it>
- Immobiliare, un portale immobiliare – <https://www.immobiliare.it>
- Wikipedia – www.wikipedia.org

Ringraziamenti

Vorrei ringraziare, innanzitutto, il professor Ursino, per la sua infinita pazienza e disponibilità, per nulla scontate, durante lo sviluppo di tutta la tesi. Lo stesso vale per il dottor Enrico Corradini, che, con i suoi consigli, ci ha aiutato a perfezionare l'applicazione.

Un altro importante ringraziamento va alla mia famiglia, per essermi stata da sempre accanto, nonostante i numerosi momenti di difficoltà.

Ovviamente devo anche ringraziare la mia ragazza e i miei amici, che mi sopportano e supportano ormai da diversi anni, e che mi hanno fatto svagare anche nei momenti più impegnativi.

Infine, un ringraziamento speciale va a Simone, che ha condiviso con me questo percorso triennale e tutti i progetti realizzati durante tale percorso.