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

TESI DI LAUREA

Progettazione e sviluppo di un'applicazione per dispositivi iOS per la pianificazione intelligente di viaggi

Design and development of an application for iOS devices for smart travel planning

Relatore

Dott. Enrico Corradini

Candidato

Pietro Salvatore

ANNO ACCADEMICO 2023-2024

Si faccia una vita
interiore - di studio, di
affetti, d'interessi umani
che non siano soltanto
di «arrivare», ma di
«essere» - e vedrà che la
vita avrà un significato

Cesare Pavese

Sommario

La semplicità con cui è possibile accedere alle informazioni nell'era digitale costituisce una grande evoluzione rispetto ai decenni passati, soprattutto grazie alla diffusione di Internet. Tuttavia, l'abbondanza dei dati, in particolare nel campo della pianificazione dei viaggi, causa una frammentazione delle informazioni, arrecando confusione e disorientamento negli utenti. Questa tesi propone una soluzione a tale problematica, centralizzando tutte le informazioni necessarie in un'unica applicazione, e mostrando il suo sviluppo dalle fasi di analisi e progettazione a quella di implementazione. Il risultato del lavoro di tesi è un'app in Swift per dispositivi iOS che consente di pianificare e prenotare viaggi, preservando intuitività ed eleganza.

Keyword: Pianificazione di viaggi, Applicazione Mobile, SwiftUI, iOS, Prenotazione di viaggi

Indice

Introduzione **1**

1 Descrizione e analisi **3**
1.1 Descrizione dell'applicazione . . . 3
1.1.1 Struttura del software . . . 3
1.2 Requisiti dell'applicazione . . . 4
1.2.1 Requisiti non funzionali . . . 5
1.2.2 Requisiti funzionali . . . 5
1.3 Casi d'uso . . . 6
1.3.1 Gestione Sistema . . . 7
1.3.2 Gestione Utente . . . 11
1.3.3 Gestione Galleria . . . 15
1.3.4 Gestione Preferiti . . . 17
1.3.5 Gestione Recensioni . . . 20
1.4 Classi di analisi . . . 23

2 Progettazione **24**
2.1 Classi di progettazione . . . 24
2.2 Mockup . . . 27

3 Implementazione **33**
3.1 Tecnologie utilizzate . . . 33
3.2 Avvio dell'applicazione . . . 34
3.3 Gestione dei dati dell'utente . . . 35
3.4 Ricerca di una destinazione . . . 37
3.5 Destinazioni popolari . . . 41
3.6 Città recenti . . . 44
3.7 Ricerca dei luoghi di interesse . . . 46
3.8 Ricerca dei piatti tipici . . . 48
3.9 Gestione delle recensioni . . . 49
3.10 Gestione dei preferiti . . . 51
3.11 Gestione della galleria . . . 57
3.12 Gestione delle prenotazioni . . . 62
3.12.1 Prenotazione degli aerei . . . 62
3.12.2 Prenotazione dei treni e dei bus . . . 65

3.12.3 Prenotazione degli hotel 67

4 Test, funzionamento e discussione **69**
4.1 Autenticazione e accesso 69
4.2 Home e navigazione 70
4.3 Ricerca delle informazioni di una città 70
4.4 Gestione dei preferiti 71
4.5 Prenotazione di viaggi 72
4.6 Gestione dei ricordi di un viaggio 73
4.7 Gestione dell'area riservata 74

Conclusioni **75**

Bibliografia **77**

Ringraziamenti **78**

Elenco delle figure

1.1 Casi d'uso di Gestione Sistema 7
1.2 Casi d'uso di Gestione Utente 11
1.3 Casi d'uso di Gestione Galleria 15
1.4 Casi d'uso di Gestione Preferiti 18
1.5 Casi d'uso di Gestione Recensioni 20
1.6 Diagramma delle classi di analisi 23

2.1 Package delle classi di progettazione 24
2.2 Models 25
2.3 Controllers 26
2.4 Views 27
2.5 Insieme delle viste posizionate in ordine logico 28

4.1 Sezione autenticazione 70
4.2 Home 70
4.3 Informazioni di una città 71
4.4 Gestione dei preferiti 72
4.5 Prenotazione di un viaggio 73
4.6 Gestione dei ricordi di un viaggio 74
4.7 Area riservata 74

Elenco delle tabelle

1.1 Caso d'Uso CreaNuovaDestinazione 8
1.2 Sequenza degli eventi alternativa di CreaNuovaDestinazione: annulla 8
1.3 Caso d'Uso VisualizzaLuoghiDiInteresse 9
1.4 Caso d'Uso VisualizzaPiattiTipici 9
1.5 Caso d'Uso VisualizzaAlloggi 10
1.6 Caso d'Uso VisualizzaLuoghiDiInteresse 10
1.7 Caso d'Uso CRUD Utente 12
1.8 Sequenza degli eventi alternativa di CRUD Utente: annulla 13
1.9 Caso d'Uso AutenticaUtente 13
1.10 Caso d'Uso VerificaCredenziali 14
1.11 Caso d'Uso VisualizzaMetePopolari 14
1.12 Caso d'Uso VisualizzaMeteRecenti 15
1.13 Caso d'Uso VisualizzaGalleria 16
1.14 Caso d'Uso CRD FileMultimediale 17
1.15 Sequenza degli eventi alternativa di CRD FileMultimediale: annulla 17
1.16 Caso d'Uso CRD CittàPreferite 18
1.17 Caso d'Uso CRD MonumentiPreferiti 19
1.18 Caso d'Uso CRD PietanzePreferite 20
1.19 Caso d'Uso CR Recensioni 21
1.20 Sequenza degli eventi alternativa di CR Recensione: annulla 21
1.21 Caso d'Uso FiltraRecensioni 22

Introduzione

Negli ultimi decenni, la diffusione di Internet ha rivoluzionato la vita delle persone, rendendo disponibili le informazioni ovunque e in qualsiasi momento. Ogni giorno sono effettuate milioni di ricerche di qualunque tipologia sul web; un settore che di anno in anno sta riscontrando una crescita esponenziale è quello dell'organizzazione di viaggi. Tuttavia, nonostante l'accesso a informazioni e servizi sia oggi più diffuso che mai, la frammentazione di queste risorse rende spesso la ricerca complessa e inefficiente. Di fatti, esistendo una quantità innumerevole di piattaforme e siti web, l'attività di pianificazione diventa disorientante per gli utenti.

Nasce, in questo contesto, l'esigenza di strumenti integrati ed intuitivi, che rendano il processo di organizzazione di viaggi centralizzato e semplice. L'obiettivo di questa tesi, quindi, è la progettazione e lo sviluppo di un'applicazione per dispositivi iOS in SwiftUI per la completa gestione dei viaggi, dalle fasi di ricerca di destinazioni a quelle di prenotazione di mezzi di trasporto e alloggi. Le funzionalità proposte personalizzabile l'esperienza degli utenti, come la possibilità di aggiungere destinazioni, attrazioni e piatti tipici tra i preferiti, e di conservare ricordi attraverso una galleria multimediale.

La tesi si articola in diverse sezioni, che descrivono in dettaglio le fasi di progettazione e sviluppo dell'applicazione. Inizialmente sarà mostrata la fase di analisi, ponendo enfasi su cosa l'applicazione è in grado di fare. Si proseguirà, dunque, con la fase di progettazione, in cui sarà illustrato in che modo i vari componenti collaborano per la realizzazione dei servizi offerti. Successivamente saranno mostrati i mockup dell'applicazione, ovvero la progettazione della sua interfaccia grafica. A seguire saranno presentate le scelte implementative utilizzate, avvalorandole con dei blocchi di codice. Sarà, poi, illustrato il funzionamento dell'applicazione, sottolineandone la semplicità di utilizzo. La sezione finale della tesi sarà dedicata alla presentazione dei risultati ottenuti durante lo sviluppo, e saranno illustrate le possibili direzioni per futuri aggiornamenti e miglioramenti, contribuendo a far sì che l'app possa continuare a rispondere efficacemente alle esigenze dei viaggiatori in un panorama tecnologico in continua evoluzione.

La presente tesi è articolata in quattro capitoli strutturati come specificato di seguito:

- Il Capitolo 1 sarà interamente dedicato alla descrizione dell'applicazione e alla fase di analisi. Qui, inizialmente sarà illustrato il problema che si mira ad affrontare, illustrando la struttura del software prodotto. Successivamente saranno evidenziati i requisiti che l'applicazione deve rispettare, i casi d'uso e il diagramma delle classi di analisi, per illustrare cosa è offerto dal software.

- Il Capitolo 2 si occuperà della fase di progettazione, partendo dal diagramma dei Models, proseguendo con quello dei Controllers, e concludendo con quello delle Views. Il capitolo si concluderà con la presentazione dei mockup.
- Il Capitolo 3 sarà incentrato sulla fase di implementazione. In primis saranno esposte le tecnologie utilizzate per lo sviluppo del software, e in secondo luogo sarà mostrato come ogni funzionalità è stata realizzata.
- Il Capitolo 4, infine, mirerà ad illustrare il funzionamento dell'applicazione, offrendo una visione chiara delle potenzialità del software.

CAPITOLO 1

Descrizione e analisi

In questo capitolo si vuole mostrare la descrizione in linguaggio naturale dell'applicazione proposta, evidenziandone i requisiti. Successivamente, verranno mostrati i diagrammi di analisi, ovvero: casi d'uso e diagramma delle classi di analisi.

1.1 Descrizione dell'applicazione

Oggigiorno, grazie all'ingente diffusione di internet, è molto semplice reperire informazioni di qualsiasi tipo. Tuttavia, non sempre ciò può essere considerato un vantaggio, in quanto la loro alta reperibilità può comportare problematiche quali difficoltà nello scegliere la fonte corretta o la loro dispersione. Un esempio pratico possono essere le ricerche da effettuare, prima di prenotare un viaggio. Infatti, in questo caso, bisogna informarsi su diversi punti, come la nazione e la città da visitare, i principali luoghi di interesse, il miglior modo per raggiungere la destinazione e i piatti tipici della destinazione. La maggior parte delle volte, però, tutte queste informazioni non sono uniformate, rendendo difficile trovare tutto il necessario.

Da qui, nasce l'esigenza di sviluppare un'applicazione che racchiuda tutto il necessario per organizzare un viaggio in qualsiasi parte del mondo. Questo è l'obiettivo di TravelMate che, oltre a fare quanto detto in precedenza, consiglia a chi la utilizza destinazioni future, in base ai suoi gusti personali, mostrando le recensioni e le foto di coloro che hanno già visitato quei luoghi. Essa, infine, sarà disponibile per dispositivi iOS.

Si scenderà, ora, più nel dettaglio, esaminando le caratteristiche dell'applicazione.

1.1.1 Struttura del software

In questa sezione, verrà mostrato ciò che il software sarà in grado di espletare. Verranno, quindi, analizzate le funzionalità principali, aggiuntive e la gestione delle operazioni eseguibili da un utente.

Funzionalità principali

Sono qui mostrate tutte le funzionalità cardine dell'applicazione:

- *Luoghi di interesse*, ovvero tutte le attrazioni turistiche, punti di interesse storico, musei e parchi. L'applicazione li mostrerà, semplicemente digitando il nome della città, regione

o nazione da visitare. Dopo aver effettuato la ricerca, saranno mostrate informazioni dettagliate sulla destinazione, come il rating da parte degli utenti, le recensioni e le foto del luogo.

- *Piatti tipici*, ovvero la cucina del posto e i suoi piatti tradizionali. Una volta selezionato uno di questi, TravelMate mostrerà le seguenti informazioni: ingredienti, preparazione e potenziali allergeni. Inoltre, sarà possibile visualizzare sulle mappe ristoranti e bar nelle vicinanze, che offrono l'alimento selezionato.
- *Alloggi*, ossia hotel, ostelli, B&B e stanze. Questi saranno comparabili attraverso le piattaforme più popolari per la loro prenotazione. Al fine di facilitare l'esperienza, sarà possibile registrarsi con il proprio ID Apple, e pagare con Apple Pay.
- *Trasporti*, ossia tutti i mezzi con cui l'utente può raggiungere la propria destinazione. L'applicazione permetterà, quindi, di prenotare voli, treni, bus e trasferimenti da e per l'aeroporto, appoggiandosi ai servizi principali di viaggio. Inoltre, sarà possibile visualizzare aggiornamenti su orari, ritardi e cancellazioni.
- *Collezione di foto geo-localizzate*, ovvero un'organizzazione delle foto scattate durante i viaggi in album, permettendo agli utenti di condividere le proprie avventure.

Gestione dell'utente

Ogni utente possiede una propria area riservata, e ha la possibilità di scrivere recensioni, caricare ricordi e cercare destinazioni all'interno dell'applicazione.

Account

Quando un utente scaricherà l'applicazione, dovrà creare il proprio account, inserendo i seguenti dati: nome, cognome, username, password, data di nascita, nazionalità, e-mail e numero di telefono. L'utente potrà sempre aggiornare o eliminare il proprio account.

L'autenticazione è necessaria per l'utilizzo dei servizi offerti.

Recensioni

Una volta effettuato l'accesso, l'utente potrà scrivere, leggere o cancellare una recensione di un luogo visitato, o che vorrebbe visitare. Le recensioni saranno caratterizzate da una data e un ID.

File multimediali

Una volta effettuato l'accesso, l'utente potrà caricare, visualizzare o cancellare file multimediali, come foto e video dei luoghi visitati. Tutti i file multimediali saranno associati ad una geolocalizzazione.

Nuova destinazione

Quando l'utente selezionerà la barra di ricerca, potrà digitare il nome di una città, una regione o una nazione. Successivamente, l'applicazione visualizzerà luoghi di interesse, piatti tipici, foto e video degli altri utenti, alloggi e trasporti relativi alla destinazione cercata.

1.2 Requisiti dell'applicazione

Si mostrano, adesso, i requisiti di TravelMate. Essi sono divisi in due gruppi:

- *Requisiti non funzionali*, ovvero eventuali limitazioni imposte per lo sviluppo dell'applicazione;
- *Requisiti funzionali*, che specificano ciò che l'applicazione è in grado di fare, ovvero le funzionalità.

1.2.1 Requisiti non funzionali

- RNF1 - **Implementazione in Swift**:
  Il sistema dovrà essere implementata in linguaggio Swift.
- RNF2 - **Username**:
  Il sistema dovrà gestire lo username per accedere alla propria area riservata.
- RNF3 - **Password**:
  Il sistema dovrà gestire la password per accedere all'area riservata.
- RNF4 - **Interfaccia grafica**:
  Il sistema avrà un'interfaccia grafica.

1.2.2 Requisiti funzionali

Gestione Sistema

- RF1 - **VisualizzaLuoghiDiInteresse**:
  Il sistema mostrerà i luoghi di interesse della destinazione cercata dall'utente.
- RF2 - **VisualizzaPiattiTipici**:
  Il sistema mostrerà i piatti tipici della destinazione cercata dall'utente.
- RF3 - **VisualizzaMezziDiTrasporto**:
  Il sistema mostrerà i mezzi di trasporto disponibili verso la destinazione cercata dall'utente.
- RF4 - **VisualizzaAlloggi**:
  Il sistema mostrerà gli alloggi disponibili per la destinazione cercata dall'utente.
- RF5 - **AutenticaUtente**:
  Il sistema dovrà permettere all'utente di accedere alla propria area riservata.

Gestione Utente

- RF6 - **CRUD Utente**:
  Il sistema dovrà gestire le attività di CRUD sul profilo dell'utente.
- RF7 - **VisualizzaMetePopolari**:
  Il sistema dovrà gestire la visualizzazione delle mete popolari tra gli utenti.
- RF8 - **VisualizzaMeteRecenti**:
  Il sistema dovrà gestire la visualizzazione delle mete cercate recentemente dall'utente.
- RF9 - **CreaNuovoViaggio**:
  Il sistema dovrà gestire la creazione di un nuovo viaggio.

Gestione Galleria

- RF10 - **VisualizzaGalleria**:
  Il sistema dovrà gestire la visualizzazione dei file multimediali caricati dall'utente raggruppati per destinazione.
- RF11 - **CRD FileMultimediale**:
  Il sistema dovrà gestire le attività di CRD sui file multimediali caricati dall'utente.

Gestione Preferiti

- RF12 - **CRD CittàPreferite**:
  Il sistema dovrà gestire le attività di CRD sulle città preferite dall'utente.
- RF13 - **CRD MonumentiPreferiti**:
  Il sistema dovrà gestire le attività di CRD sui monumenti preferiti dall'utente.
- RF14 - **CRD PietanzePreferite**:
  Il sistema dovrà gestire le attività CRD sui piatti tipici preferiti dall'utente.

Gestione Recensioni

- RF15 - **CR Recensione**:
  Il sistema dovrà gestire le attività di CR sulle recensioni.
- RF16 - **FiltraRecensioni**:
  Il sistema dovrà gestire la visualizzazione delle recensioni in base al numero di stelle attribuite dagli utenti.

1.3 Casi d'uso

Si mostrano qui i casi d'uso dell'applicazione. Essi specificano una sequenza di azioni, che un sistema è in grado di fare, interagendo con gli attori. Ogni caso d'uso è attivato da un attore, ed è descritto dal suo punto di vista. Ognuno di essi, infine, è descritto attraverso i seguenti valori: nome, ID, breve descrizione, attore primario (colui che avvia il caso d'uso), attori secondari (coloro che sono affetti dal caso d'uso), precondizioni (cioè lo stato del sistema prima che venga avviato il caso d'uso), sequenza degli eventi principale (ciò che il caso d'uso è in grado di fare passo dopo passo), postcondizioni (lo stato del sistema, una volta terminata l'esecuzione del caso d'uso), e sequenza degli eventi alternativa, qualora avvenga un errore.

1.3.1 Gestione Sistema

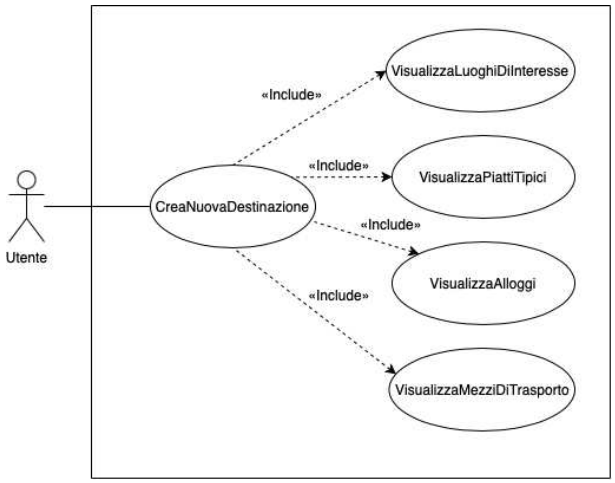


Figura 1.1: Casi d'uso di Gestione Sistema

Caso d'Uso: CreaNuovaDestinazione
ID: 1
Breve Descrizione: Il sistema crea una nuova destinazione, mostrando all'utente le informazioni necessarie
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente deve aver effettuato l'accesso
Sequenza degli eventi principale:

1. Il caso d'uso inizia, quando l'utente seleziona "Nuova destinazione"
2. *While* l'utente scrive la destinazione
   (a) Il sistema mostra le destinazioni
   (b) *If* la destinazione esiste
      i. Il sistema mostra la destinazione
   (c) *Else*
      i. Il sistema informa l'utente che la destinazione non esiste
3. *Include*(VisualizzaLuoghiDiInteresse)
4. *Include*(VisualizzaPiattiTipici)
5. *Include*(VisualizzaAlloggi)
6. *Include*(VisualizzaMezziDiTrasporto)
7. Il sistema mostra la destinazione cercata, permettendo all'utente di visualizzarne i luoghi di interesse, i piatti tipici, i mezzi di trasporto e gli alloggi

Postcondizioni:
Una nuova destinazione è stata creata

Sequenza degli eventi alternativa:
Annulla

Tabella 1.1: Caso d'Uso CreaNuovaDestinazione

Sequenza degli eventi alternativa: CreaNuovaDestinazione: Annulla
ID: 1.1
Breve descrizione: L'utente interrompe il processo di creazione di una nuova destinazione
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: Nessuna
Sequenza degli eventi principale: 1. La sequenza alternativa inizia in un qualsiasi momento 2. L'utente ritorna alla Home Page
Postcondizioni: Il sistema non crea una nuova destinazione

Tabella 1.2: Sequenza degli eventi alternativa di CreaNuovaDestinazione: annulla

Caso d'Uso: VisualizzaLuoghiDiInteresse
ID: 2
Breve Descrizione: Il sistema mostra una lista di luoghi di interesse della destinazione selezionata
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente seleziona la destinazione
Sequenza degli eventi principale: 1. Il caso d'uso inizia, quando l'utente seleziona "Luoghi di interesse" 2. Il sistema cerca i luoghi di interesse più popolari della destinazione selezionata 3. Il sistema mostra la lista di tutte le attrazioni
Postcondizioni: Nessuna
Sequenza degli eventi alternativa: Nessuna

Tabella 1.3: Caso d'Uso VisualizzaLuoghiDiInteresse

Caso d'Uso: VisualizzaPiattiTipici
ID: 3
Breve Descrizione: Il sistema mostra una lista di piatti tipici della destinazione selezionata
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente seleziona la destinazione
Sequenza degli eventi principale: 1. Il caso d'uso inizia, quando l'utente seleziona "Piatti tipici" 2. Il sistema cerca i piatti tipici più popolari della destinazione selezionata 3. Il sistema mostra la lista di tutte i piatti tipici
Postcondizioni: Nessuna
Sequenza degli eventi alternativa: Nessuna

Tabella 1.4: Caso d'Uso VisualizzaPiattiTipici

Caso d'Uso: VisualizzaAlloggi
ID: 4
Breve Descrizione: Il sistema mostra una lista di luoghi di alloggi per la destinazione selezionata
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente seleziona "Prenota"
Sequenza degli eventi principale: 1. Il caso d'uso inizia, quando l'utente seleziona "Booking" o "airbnb" 2. Il sistema cerca gli alloggi per la destinazione selezionata 3. Il sistema mostra la lista di tutti gli alloggi disponibili
Postcondizioni: Nessuna
Sequenza degli eventi alternativa: Nessuna

Tabella 1.5: Caso d'Uso VisualizzaAlloggi

Caso d'Uso: VisualizzaMezziDiTrasporto
ID: 5
Breve Descrizione: Il sistema mostra una lista di mezzi di trasporto disponibili per la destinazione selezionata
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente seleziona "Prenota"
Sequenza degli eventi principale: 1. Il caso d'uso inizia, quando l'utente seleziona "Skyscanner", "Trainline" o "Omio" 2. Il sistema cerca i mezzi di trasporto disponibili per la destinazione selezionata 3. Il sistema mostra la lista di tutti i mezzi di trasporto
Postcondizioni: Nessuna
Sequenza degli eventi alternativa: Nessuna

Tabella 1.6: Caso d'Uso VisualizzaLuoghiDiInteresse

1.3.2 Gestione Utente

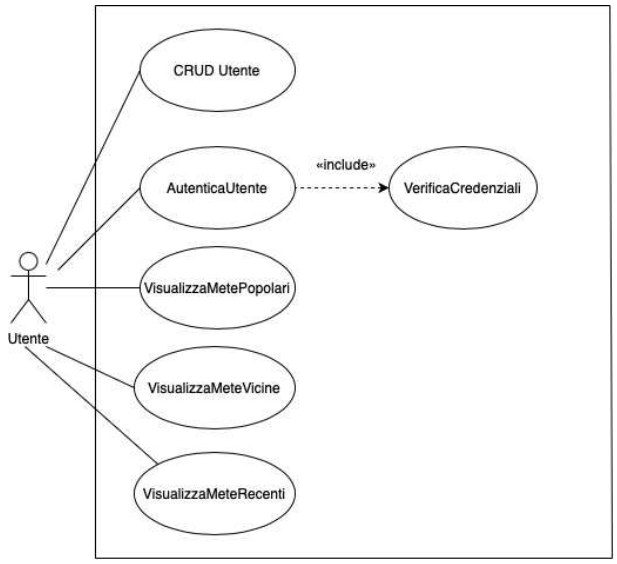


Figura 1.2: Casi d'uso di Gestione Utente

Caso d'Uso: CRUD Utente
ID: 6
Breve Descrizione: Il sistema permette all'utente di creare, visualizzare, modificare o eliminare il proprio account. Durante la creazione dell'account, la password deve soddisfare i seguenti requisiti: almeno otto caratteri di lunghezza, almeno una lettera maiscola, almeno un numero e almeno un carattere speciale
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente deve essere autenticato, per visualizzare, modificare o eliminare il suo account
Sequenza degli eventi principale:

1. Il caso d'uso inizia, quando l'utente vuole eseguire un'operazione di CRUD per il suo account
2. *If* l'utente seleziona "Registrati"
   - L'utente inserisce i propri dati personali
   - L'utente inserisce una password, che soddisfi i requisiti
   - *While* la password non soddisfa i requisiti
     - Il sistema mostra un messaggio in base al requisito non soddisfatto
     - L'utente scrive la password nuovamente
   - Il sistema crea l'account per l'utente
3. *Else if* l'utente seleziona "Account"
   - Il sistema mostra all'utente il suo account
4. *Else if* l'utente seleziona "Modifica Account"
   - L'utente modifica i suoi dati personali
   - L'utente seleziona "Salva modifiche"
5. *Else*
   - L'utente seleziona "Elimina Account"
   - Il sistema visualizza un messaggio di conferma per l'eliminazione dell'account
   - L'utente seleziona "Sì"

Postcondizioni: L'account dell'utente è creato, visualizzato, modificato o eliminato
Sequenza degli eventi alternativa: Annulla

Tabella 1.7: Caso d'Uso CRUD Utente

Sequenza degli eventi alternativa: CRUD Utente: Annulla
ID: 6.1
Breve descrizione: L'utente interrompe il processo di creazione, modifica o eliminazione dell'account
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: Nessuna
Sequenza degli eventi principale:

1. La sequenza alternativa inizia in un qualsiasi momento
2. L'utente interrompe il processo di creazione, modifica o eliminazione dell'account

Postcondizioni:
Il sistema non crea, modifica o elimina l'account dell'utente

Tabella 1.8: Sequenza degli eventi alternativa di CRUD Utente: annulla

Caso d'Uso: AutenticaUtente

ID: 7

Breve Descrizione:
Il sistema permette all'utente di accedere al proprio account

Attore Primario:
Utente

Attori Secondari:
Nessuno

Precondizioni:
Nessuna

Sequenza degli eventi principale:

1. Il caso d'uso inizia, quando l'utente seleziona "Login"
2. *Include*(VerificaCredenziali)
3. *If* l'account esiste
    - Il sistema permette all'utente di accedere al proprio account
4. *Else*
    - Il sistema informa l'utente che l'account non esiste

Postcondizioni:
L'utente accede al proprio account

Sequenza degli eventi alternativa:
Nessuna

Tabella 1.9: Caso d'Uso AutenticaUtente

Caso d'Uso: VerificaCredenziali

ID: 8

Breve Descrizione:
Il sistema verifica se le credenziali inserite dall'utente sono corrette

Attore Primario:
Utente

Attori Secondari:
Nessuno

Precondizioni:
L'utente seleziona "Login"

Sequenza degli eventi principale: 1. Il caso d'uso inizia, quando l'utente seleziona "Login" 2. *While* le credenziali scritte dall'utente non sono valide • Il sistema chiede all'utente di inserire nuovamente le sue credenziali • Il sistema veridica se le credenziali inserite sono corrette
Postcondizioni: L'utente accede al proprio account
Sequenza degli eventi alternativa: Nessuna

Tabella 1.10: Caso d'Uso VerificaCredenziali

Caso d'Uso: VisualizzaMetePopolari
ID: 9
Breve Descrizione: Il sistema visualizza le mete più popolari
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente deve essere autenticato
Sequenza degli eventi principale: 1. Il caso d'uso inizia, quando l'utente seleziona "Popolari" 2. Il sistema cerca le mete più popolari 3. Il sistema mostra le mete trovate
Postcondizioni: Nessuna
Sequenza degli eventi alternativa: Nessuna

Tabella 1.11: Caso d'Uso VisualizzaMetePopolari

Caso d'Uso: VisualizzaMeteRecenti
ID: 10
Breve Descrizione: Il sistema visualizza le mete cercate recentemente dall'utente
Attore Primario: Utente
Attori Secondari: Nessuno

Precondizioni: L'utente deve essere autenticato
Sequenza degli eventi principale: 1. Il caso d'uso inizia, quando l'utente seleziona "Recenti" 2. Il sistema cerca le mete cercate recentemente dall'utente 3. *If* l'utente ha effettuato ricerche • Il sistema mostra le mete trovate 4. *Else* • Il sistema informa l'utente che non vi sono ricerche recenti
Postcondizioni: Nessuna
Sequenza degli eventi alternativa: Nessuna

Tabella 1.12: Caso d'Uso VisualizzaMeteRecenti

1.3.3 Gestione Galleria

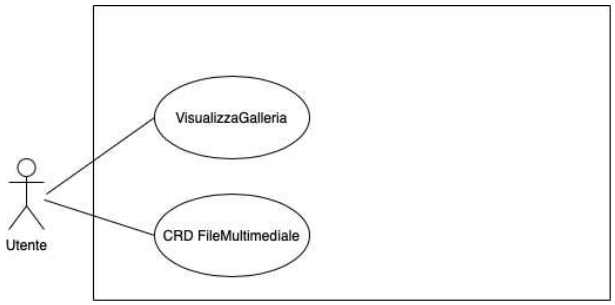


Figura 1.3: Casi d'uso di Gestione Galleria

Caso d'Uso: VisualizzaGalleria
ID: 11
Breve Descrizione: Il sistema visualizza i file multimediali caricati dall'utente raggruppati per destinazione
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente deve essere autenticato
Sequenza degli eventi principale:

1. Il caso d'uso inizia, quando l'utente seleziona "Galleria"
2. Il sistema cerca i file multimediali caricati dall'utente
3. *If* l'utente ha caricato dei file multimediali
    - Il sistema mostra i file multimediali trovati e raggruppati per luogo
4. *Else*
    - Il sistema informa l'utente che non ha ancora caricato alcun file multimediale

Postcondizioni: Nessuna
Sequenza degli eventi alternativa: Nessuna

Tabella 1.13: Caso d'Uso VisualizzaGalleria

Caso d'Uso: CRD FileMultimediale
ID: 12
Breve Descrizione: Il sistema permette all'utente di creare, visualizzare o eliminare un file multimediale caricato
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente deve essere autenticato
Sequenza degli eventi principale:

1. Il caso d'uso inizia, quando l'utente vuole eseguire un'operazione di CRD per un file multimediale 2. *If* l'utente seleziona "Nuovo ricordo" • L'utente inserisce luogo, data e file multimediale • L'utente seleziona "Pubblica" • Il sistema pubblica il file multimediale caricato 3. *Else if* l'utente seleziona un file multimediale dalla galleria • Il sistema visualizza il file selezionato 4. *Else* • L'utente seleziona un file multimediale dalla galleria • L'utente seleziona "Elimina ricordo" • Il sistema visualizza un messaggio di conferma per l'eliminazione del file • L'utente seleziona "Sì"
Postcondizioni: Un file multimediale è caricato, visualizzato o eliminato
Sequenza degli eventi alternativa: Annulla

Tabella 1.14: Caso d'Uso CRD FileMultimediale

Sequenza degli eventi alternativa: CRD FileMultimediale: Annulla
ID: 12.1
Breve descrizione: L'utente interrompe il processo di creazione, o eliminazione del file multimediale
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: Nessuna
Sequenza degli eventi principale: 1. La sequenza alternativa inizia in un qualsiasi momento 2. L'utente interrompe il processo di creazione, o eliminazione del file multimediale
Postcondizioni: Il sistema non crea, o elimina il file multimediale

Tabella 1.15: Sequenza degli eventi alternativa di CRD FileMultimediale: annulla

1.3.4 Gestione Preferiti

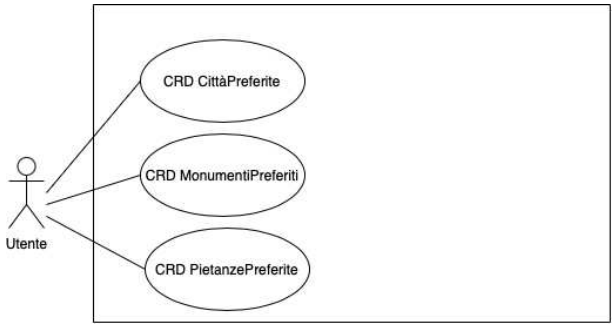


Figura 1.4: Casi d'uso di Gestione Preferiti

Caso d'Uso: CRD CittàPreferite
ID: 13
Breve Descrizione: Il sistema permette all'utente di creare, visualizzare o eliminare una città dai preferiti
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente deve essere autenticato
Sequenza degli eventi principale: 1. Il caso d'uso inizia, quando l'utente vuole eseguire un'operazione di CRD per la lista delle città preferite 2. *If* l'utente seleziona l'icona a forma di cuore • Il sistema inserisce la città tra i preferiti 3. *Else if* l'utente seleziona "Preferiti" o "Città" • *If* sono presenti città preferite – Il sistema visualizza le città preferite dell'utente • *Else* – Il sistema informa l'utente che non vi sono città preferite 4. *Else* • L'utente seleziona l'icona a forma di cuore su una città preferita • Il sistema elimina la città dalla lista dei preferiti
Postcondizioni: Una città è caricata, visualizzata o eliminata dalla lista dei preferiti
Sequenza degli eventi alternativa: Nessuna

Tabella 1.16: Caso d'Uso CRD CittàPreferite

Caso d'Uso: CRD MonumentiPreferiti
ID: 14
Breve Descrizione: Il sistema permette all'utente di creare, visualizzare o eliminare un monumento dai preferiti
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente deve essere autenticato
Sequenza degli eventi principale: 1. Il caso d'uso inizia, quando l'utente vuole eseguire un'operazione di CRD per la lista dei monumenti preferiti 2. *If* l'utente seleziona l'icona a forma di cuore • Il sistema inserisce il monumento tra i preferiti 3. *Else if* l'utente seleziona "Monumenti" • *If* sono presenti monumenti preferiti – Il sistema visualizza i monumenti preferiti dell'utente • *Else* – Il sistema informa l'utente che non vi sono monumenti preferiti 4. *Else* • L'utente seleziona l'icona a forma di cuore su un monumento preferito • Il sistema elimina il monumento dalla lista dei preferiti
Postcondizioni: Un monumento è caricato, visualizzato o eliminato dalla lista dei preferiti
Sequenza degli eventi alternativa: Nessuna

Tabella 1.17: Caso d'Uso CRD MonumentiPreferiti

Caso d'Uso: CRD PietanzePreferite
ID: 15
Breve Descrizione: Il sistema permette all'utente di creare, visualizzare o eliminare una pietanza dai preferiti
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente deve essere autenticato
Sequenza degli eventi principale:

1. Il caso d'uso inizia, quando l'utente vuole eseguire un'operazione di CRD per la lista delle pietanze preferite
2. *If* l'utente seleziona l'icona a forma di cuore
   - Il sistema inserisce la pietanza tra i preferiti
3. *Else if* l'utente seleziona "Pietanze"
   - *If* sono presenti pietanze preferite
     - Il sistema visualizza le pietanze preferite dell'utente
   - *Else*
     - Il sistema informa l'utente che non vi sono pietanze preferite
4. *Else*
   - L'utente seleziona l'icona a forma di cuore su una pietanza preferita
   - Il sistema elimina la pietanza dalla lista dei preferiti

Postcondizioni: Una pietanza è caricata, visualizzata o eliminata dalla lista dei preferiti
Sequenza degli eventi alternativa: Nessuna

Tabella 1.18: Caso d'Uso CRD PietanzePreferite

1.3.5 Gestione Recensioni

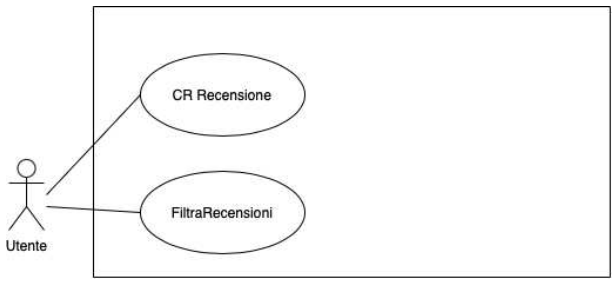


Figura 1.5: Casi d'uso di Gestione Recensioni

Caso d'Uso: CR Recensione
ID: 16
Breve Descrizione: Il sistema permette all'utente di creare o visualizzare una recensione
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni:

L'utente deve essere autenticato
Sequenza degli eventi principale: 1. Il caso d'uso inizia, quando l'utente seleziona "Recensioni" 2. *If* sono presenti recensioni • Il sistema visualizza le recensioni 3. *Else* • Il sistema informa l'utente che non sono presenti recensioni 4. *If* l'utente seleziona "Scrivi una recensione" • L'utente inserisce un voto da 1 a 5 • L'utente scrive la recensione • L'utente seleziona "Pubblica" • Il sistema pubblica la recensione
Postcondizioni: Una recensione è caricata o visualizzata
Sequenza degli eventi alternativa: Annulla

Tabella 1.19: Caso d'Uso CR Recensioni

Sequenza degli eventi alternativa: CR Recensioni: Annulla
ID: 16.1
Breve Descrizione: L'utente interrompe il processo di creazione della recensione
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: Nessuna
Sequenza degli eventi principale: 1. La sequenza alternativa inizia in un qualsiasi momento 2. L'utente interrompe il processo di creazione della recensione
Postcondizioni: Il sistema non crea la recensione

Tabella 1.20: Sequenza degli eventi alternativa di CR Recensione: annulla

Caso d'Uso: FiltraRecensioni
ID: 17

Breve Descrizione: Il sistema filtra le recensioni in base alle stelle inserite dagli utenti
Attore Primario: Utente
Attori Secondari: Nessuno
Precondizioni: L'utente deve essere autenticato
Sequenza degli eventi principale: 1. Il caso d'uso inizia, quando l'utente seleziona un numero di stelle associate ad una recensione 2. Il sistema cerca le recensioni con il numero di stelle selezionate 3. *If* sono presenti recensioni • Il sistema mostra le recensioni cercate 4. *Else* • Il sistema informa l'utente che non vi sono recensioni con quella valutazione
Postcondizioni: Nessuna
Sequenza degli eventi alternativa: Nessuna

Tabella 1.21: Caso d'Uso FiltraRecensioni

1.4 Classi di analisi

Si mostrano, infine, le classi di analisi, con lo scopo di evidenziare in che modo le classi interagiscano, per la realizzazione dei casi d'uso.

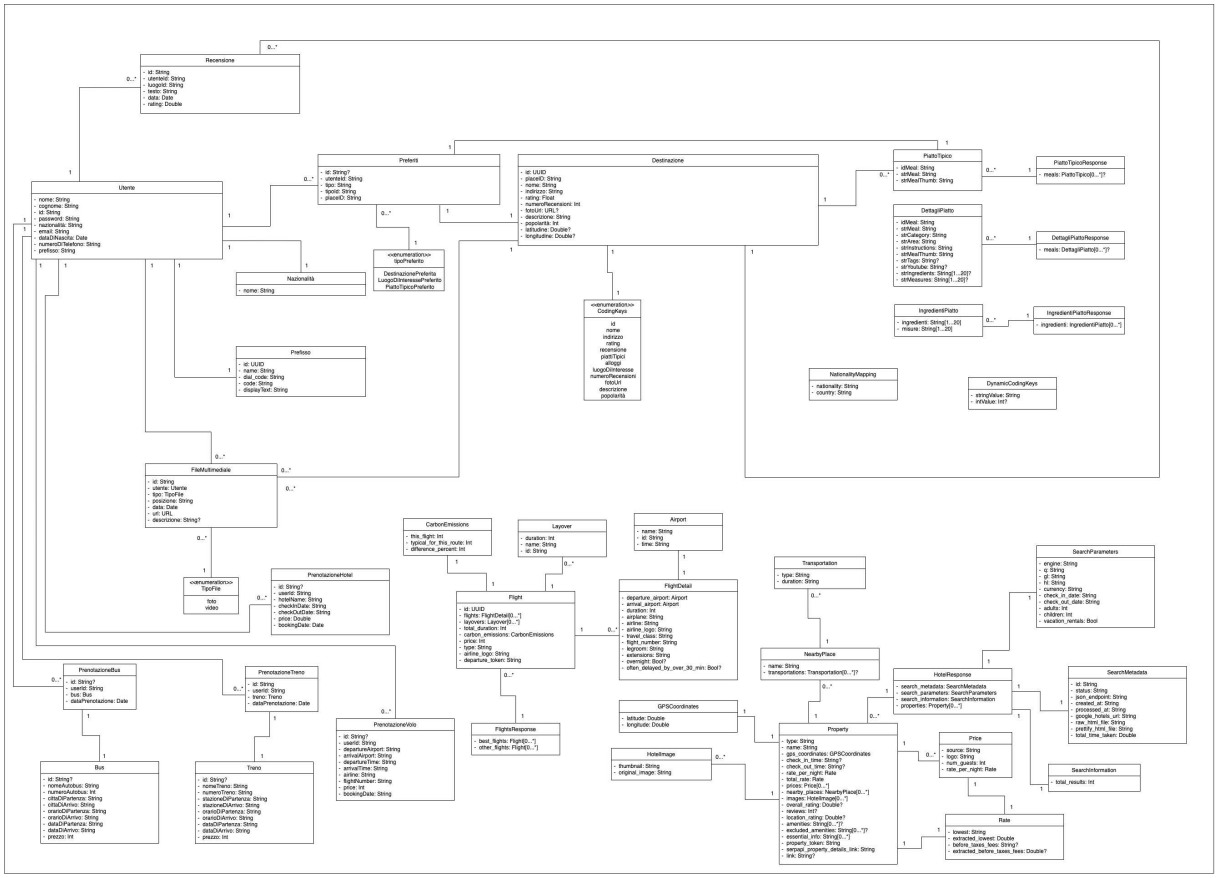


Figura 1.6: Diagramma delle classi di analisi

CAPITOLO 2

Progettazione

In questo capitolo si vuole mostrare la progettazione dell'applicazione iOS. Verranno qui mostrati i diagrammi delle classi di progettazione e, successivamente, i mockup.

2.1 Classi di progettazione

Si mostrano le classi di progettazione, con lo scopo di risaltare ciò che esse possono fare e come.

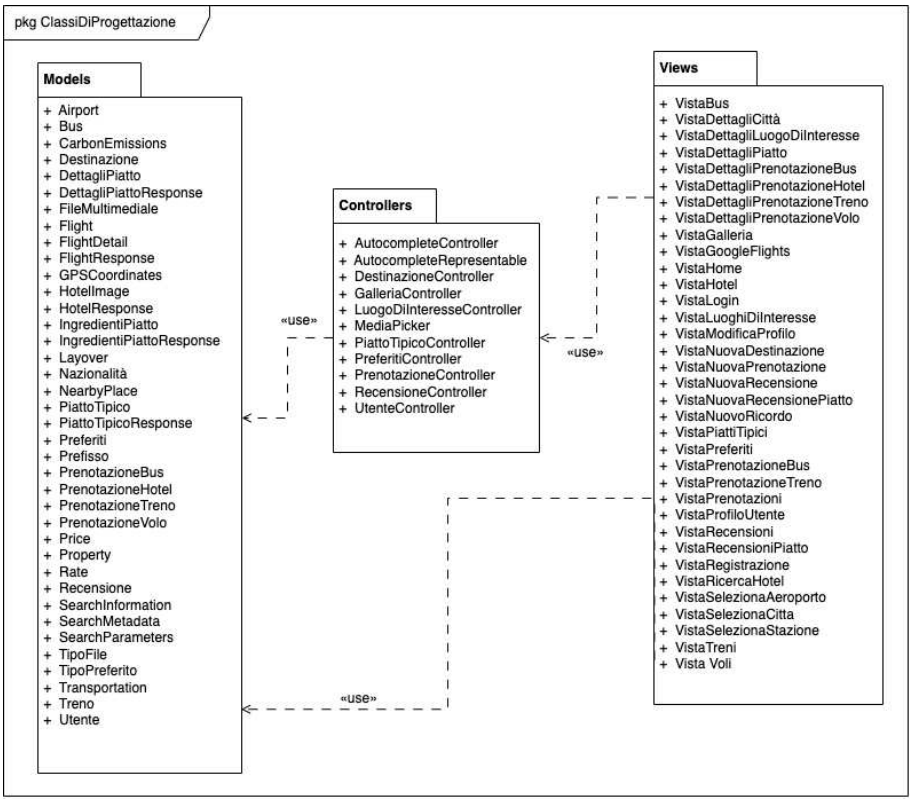


Figura 2.1: Package delle classi di progettazione

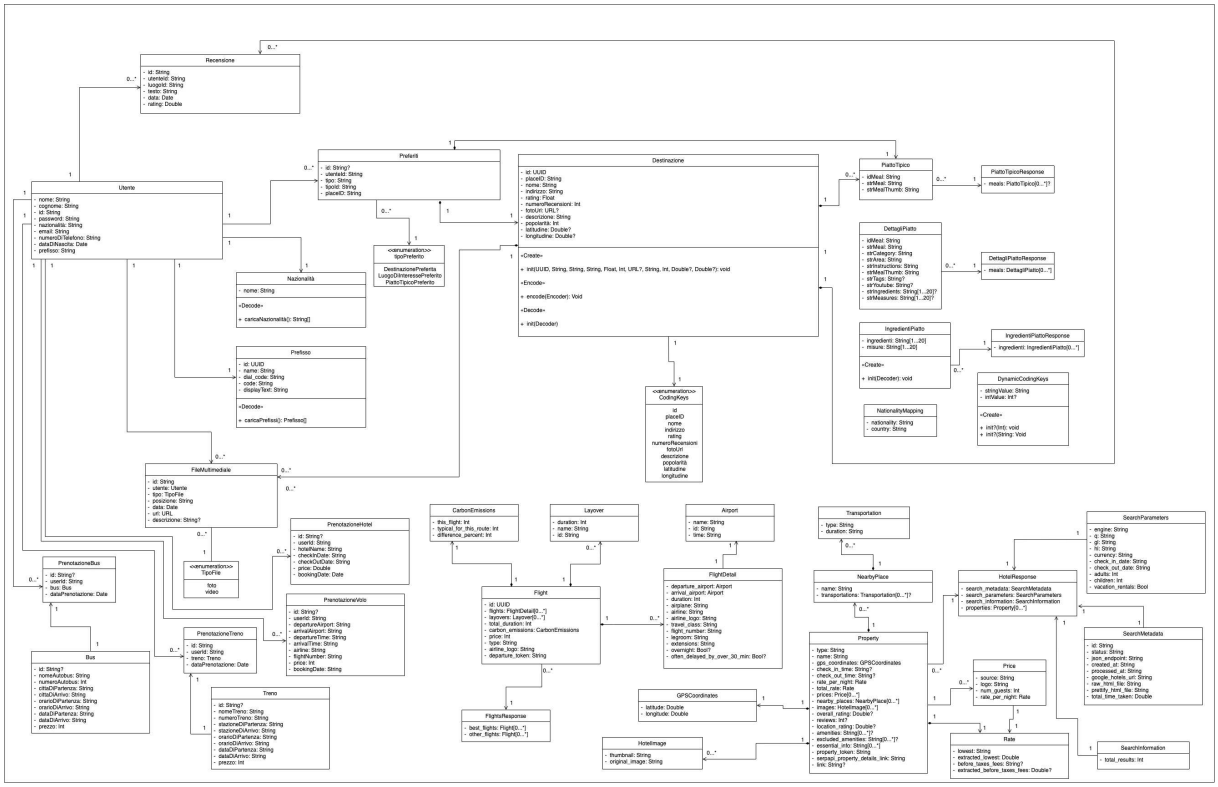


Figura 2.2: Models

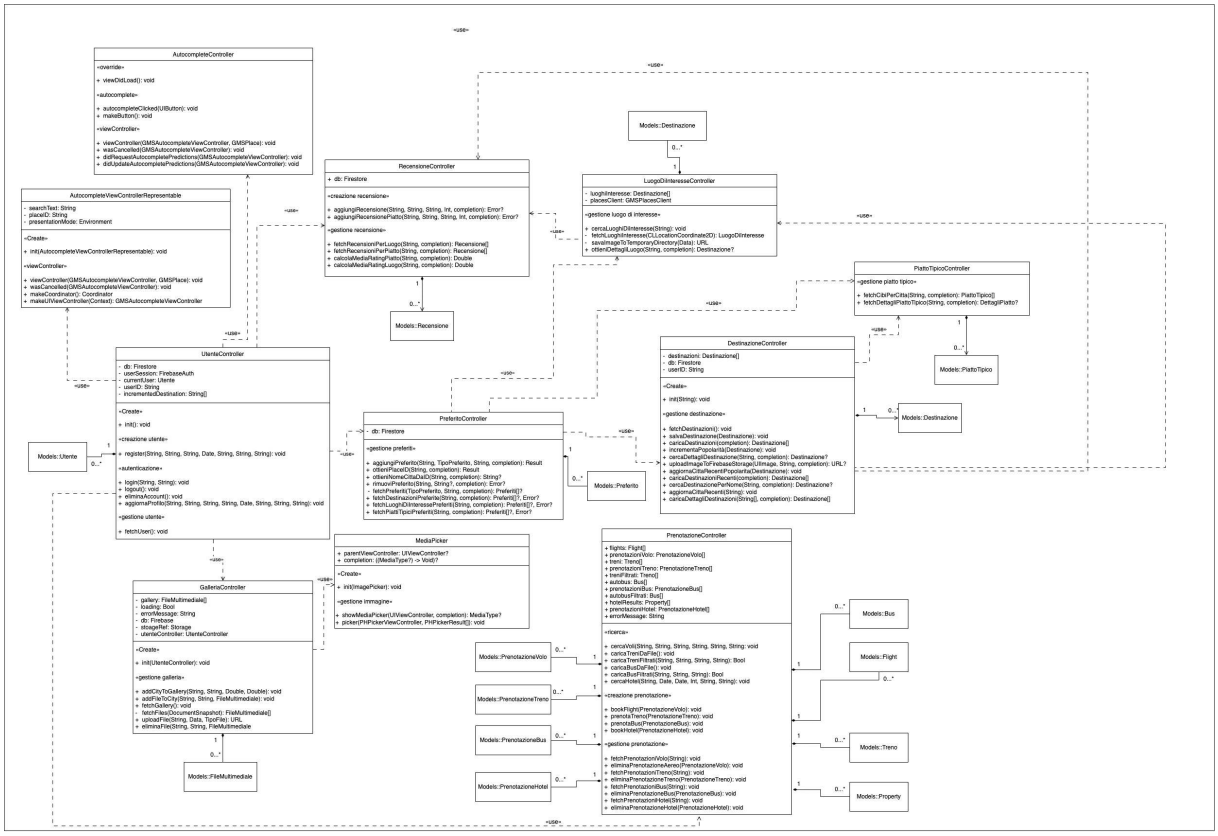


Figura 2.3: Controllers

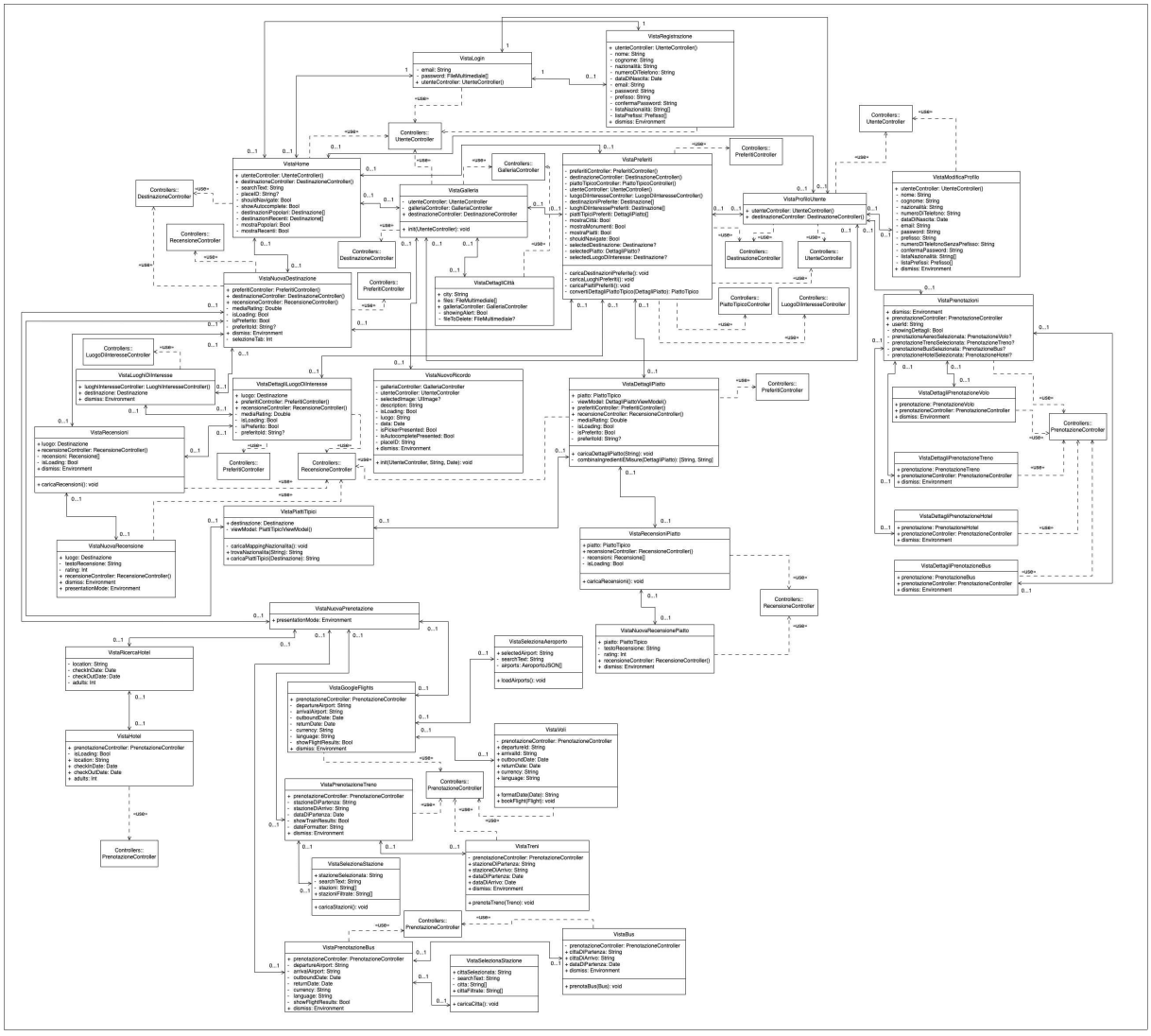


Figura 2.4: Views

2.2 Mockup

Si mostrano, ora, i mockup, ovvero le rappresentazioni visive delle viste che l'applicazione avrà.

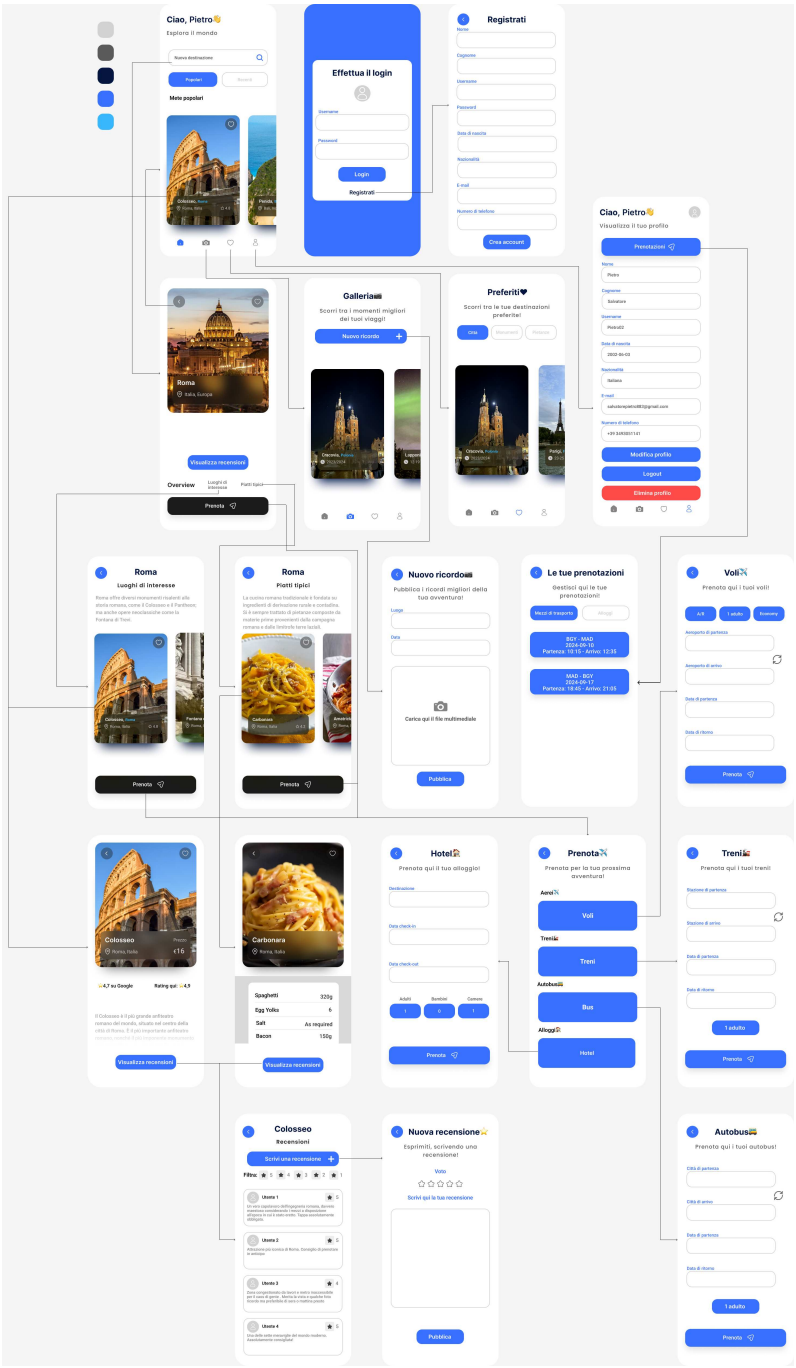
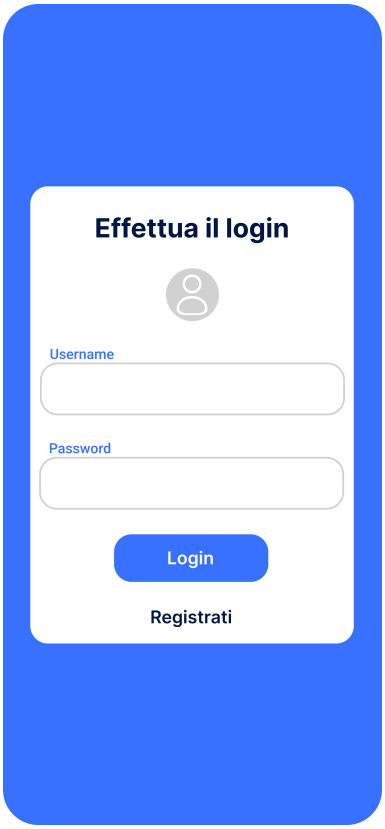
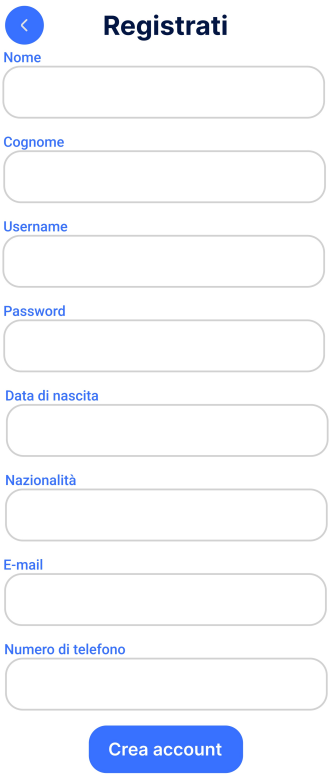


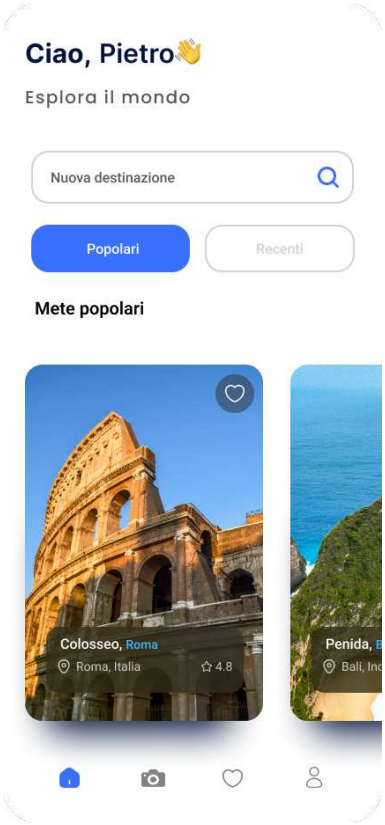
Figura 2.5: Insieme delle viste posizionate in ordine logico



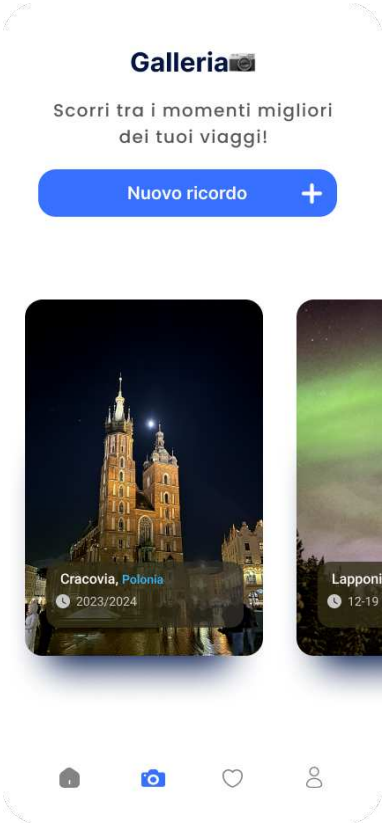
(a) Vista login



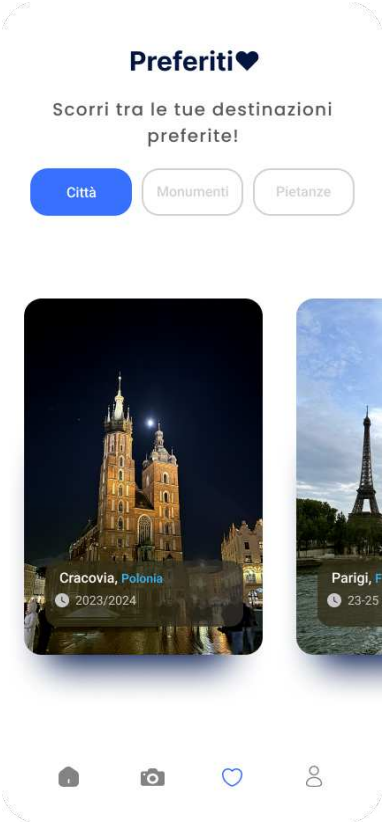
(b) Vista registrazione account



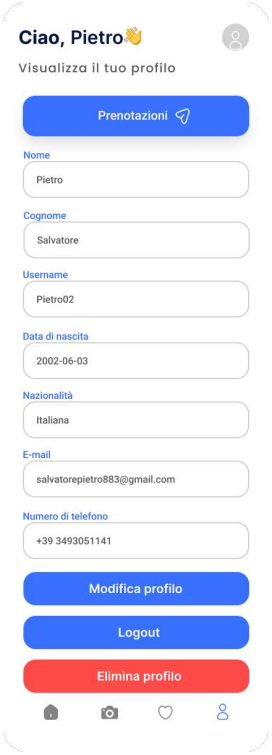
(a) Vista della home



(b) Vista della galleria

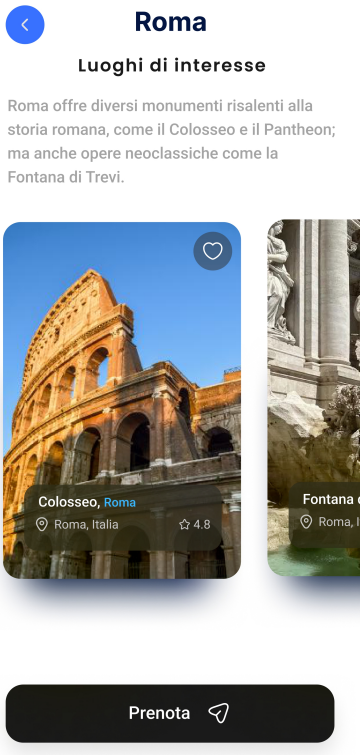


(a) Vista dei preferiti

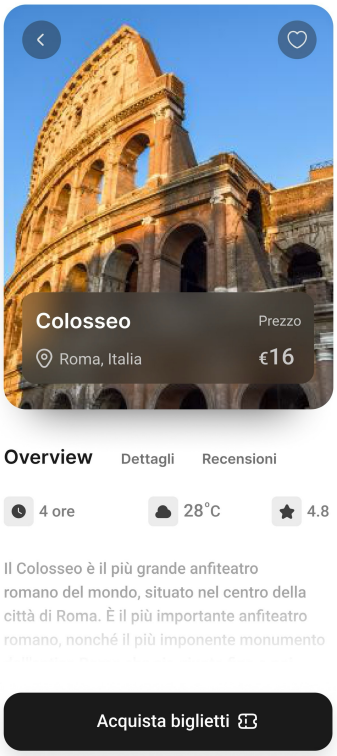


(b) Vista del profilo utente

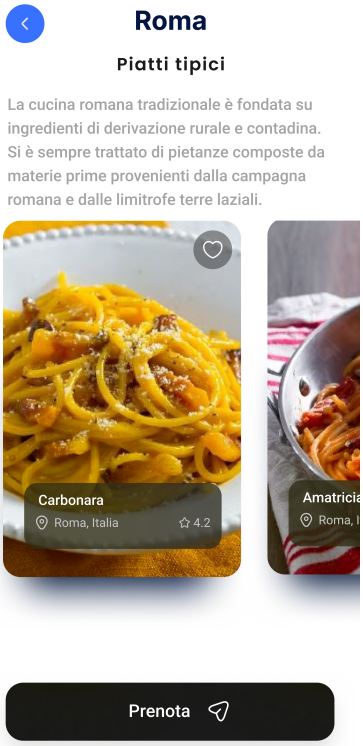
(a) Vista di nuova destinazione



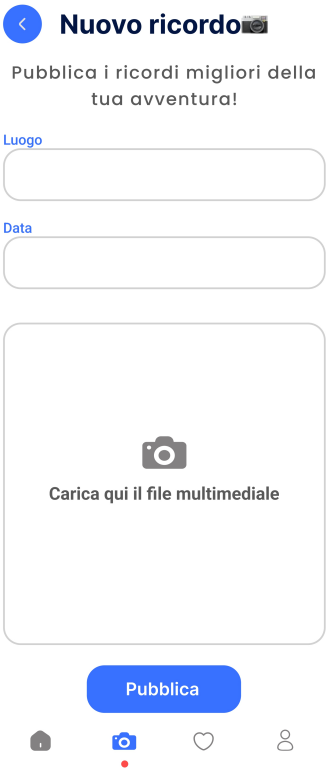
(b) Vista dei luoghi di interesse di una destinazione



(a) Vista dei dettagli di un luogo di interesse



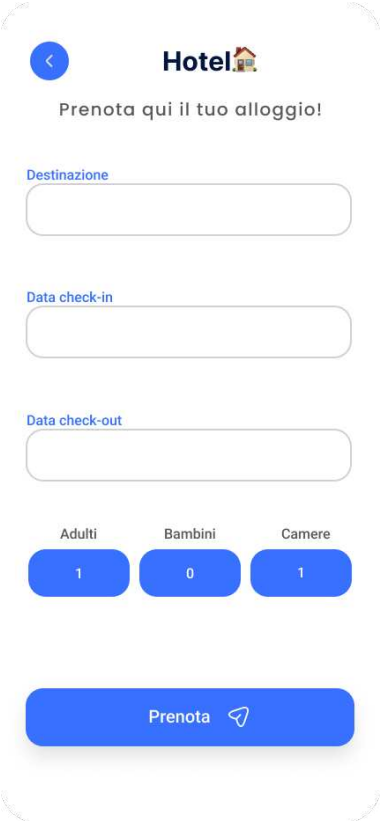
(b) Vista dei piatti tipici di una destinazione



(a) Vista nuovo ricordo

(b) Vista nuova prenotazione

(a) Vista per la prenotazione dei voli



(b) Vista per la prenotazione di un alloggio tramite Booking

(a) Vista delle recensioni

(b) Vista nuova recensione

CAPITOLO 3

Implementazione

Questo capitolo mira a mostrare l'implementazione dell'applicazione. In particolare, si vogliono evidenziare le soluzioni adottate per la realizzazione di quanto progettato. Inizialmente, verranno elencate tutte le tecnologie utilizzate per lo sviluppo dell'applicazione, e poi verrà mostrato in che modo esse sono state utilizzate per la sua implementazione.

3.1 Tecnologie utilizzate

Swift

Il linguaggio utilizzato per lo sviluppo dell'applicazione proposta è Swift5. Esso è un linguaggio compilato, conciso e tipizzato. Swift eredita una serie di caratteristiche da altri linguaggi moderni, come gli Optionals e le collezioni. Inoltre, offre la possibilità di non specificare esplicitamente il tipo di dato per variabili e costanti grazie al type inference. Infine, supporta efficientemente architetture come l'MVC e l'MVVM grazie a framework per la creazione di interfacce grafiche, come UIKit e SwiftUI.

SwiftUI

Per la realizzazione delle interfacce grafiche è stato utilizzato il framework SwiftUI, in quanto semplice e intuitivo. Esso, inoltre, offre la possibilità di visualizzare in tempo reale, attraverso una preview, la vista che lo sviluppatore sta implementando. Infine, è importante menzionare la possibilità di creare applicazioni con compatibilità cross-platform.

Firebase

Per quanto riguarda la gestione dei dati, è stato utilizzato Firebase. Esso, infatti, è un database NoSQL, che per l'applicazione proposta risulta la soluzione più ottimale. Tuttavia, per TravelMate non viene sfruttato solo l'utilizzo come database di Firebase, ma viene utilizzato anche per gestire l'autenticazione degli utenti attraverso indirizzo email e password, e per lo storage di file multimediali.

API

Poiché l'applicazione deve permettere all'utente di reperire informazioni attendibili su destinazioni, voli, hotel e cibi tipici, sono state utilizzate alcune API per lo sviluppo di essa:

- **Google Places API**: Per la ricerca di destinazioni, è stata utilizzata l'API Google Places. Inizialmente, la ricerca viene facilitata grazie all'autocomplete. Successivamente alla ricerca, sono utilizzate le API Place Details e Place Photos, per ottenere tutte le informazioni necessarie sulla città e una fotografia di essa. Per la ricerca dei luoghi di interesse viene utilizzata l'API Nearby Search, che, date le coordinate di una città, restituisce musei e attrazioni turistiche all'utente.
- **Google Flights API**: Per la ricerca dei voli, è stata utilizzata l'API di Google Flights. Essa simula una ricerca su google dei voli disponibili per una destinazione. Per fare ciò, all'utente basta selezionare gli aeroporti di partenza e arrivo, le date di partenza e ritorno, la valuta e la lingua. I risultati della ricerca offrono tutti i dettagli sul volo cercato, come la compagnia che effettua la tratta selezionata, orari di partenza e di arrivo, durata dell'eventuale scalo, prezzo ed emissioni di anidride carbonica.
- **Google Hotels API**: Analoga alla ricerca dei voli è quella degli hotel. All'utente, infatti, basta inserire la città dove vorrebbe alloggiare, le date di check-in e check-out, il numero di adulti, la valuta, la nazione e la lingua della ricerca. Una volta ottenuti i risultati, all'utente saranno accessibili tutte le informazioni necessarie, come il nome dell'hotel, il prezzo e i servizi offerti.
- **TheMealDB**: Per la ricerca dei piatti tipici è stata utilizzata l'API TheMealDB. Essa, una volta inserita una nazione, mostra all'utente delle pietanze tipiche con rispettiva fotografia e ingredienti.

3.2 Avvio dell'applicazione

Quando l'applicazione viene avviata per la prima volta, o dopo aver effettuato un logout, all'utente viene mostrata un'interfaccia grafica con la form per effettuare il login.

Registrazione

Qualora si tratti di un nuovo utente, quest'ultimo può creare un nuovo account. I dati necessari per la registrazione sono i seguenti: nome, cognome, password, nazionalità, email, prefisso, numero di telefono e data di nascita. Il metodo seguente gestisce la registrazione di un nuovo utente:

```
func register(withEmail email: String, password: String, nome:
String, cognome: String, dataDiNascita: Date, nazionalità: String,
numeroDiTelefono: String, prefisso: String) async throws {
    do {
        let result = try await Auth.auth().createUser(
        withEmail: email,password: password)
        self.userSession = result.user
        let user = Utente(
            nome: nome,
            cognome: cognome,
            id: result.user.uid,
            password: password,
            nazionalità: nazionalità,
            email: email,
            numeroDiTelefono: "\(prefisso)\(numeroDiTelefono)",
            dataDiNascita: dataDiNascita,
            prefisso: prefisso
        )
        let encodedUser = try Firestore.Encoder().encode(user)
        try await Firestore.firestore().collection("users").
        document(user.id).setData(encodedUser)
        await fetchUser()
```

```
    } catch {
        print("DEBUG: Fallimento nella creazione dell'utente con
        errore \(error.localizedDescription)")
    }
}
```

Il metodo crea l'oggetto Utente e il documento con tutti i dati nel database. Una volta effettuata la registrazione, l'utente può subito iniziare ad utilizzare l'applicazione.

Login

Nel caso in cui un utente disponga già di un account, quest'ultimo può effettuare il login. Il metodo che gestisce l'autenticazione è il seguente:

```
func login(withEmail email: String, password: String) async throws {
    do {
        let result = try await Auth.auth().signIn(withEmail: email,
        password: password)
        self.userSession = result.user
        await fetchUser()
    } catch {
        print("DEBUG: fallimento nel login con errore \
        (error.localizedDescription)")
    }
}
```

Il metodo controlla se le credenziali sono corrette, e, se lo sono, crea una sessione per l'utente, permettendogli di utilizzare l'applicazione.

3.3 Gestione dei dati dell'utente

All'interno di questa sezione sono mostrate le soluzioni relative alla gestione dei dati dell'utente.

Persistenza della sessione

Qualora l'utente chiuda l'applicazione senza effettuare il logout, la sessione verrebbe preservata. Il metodo che gestisce ciò è il seguente:

```
func fetchUser() async {
    guard let uid = Auth.auth().currentUser?.uid else { return }
    guard let snapshot = try? await Firestore.firestore()
    .collection("users").document(uid)
    .getDocument() else { return }
    self.currentUser = try? snapshot.data(as: Utente.self)
    self.incrementedDestinations = snapshot.data()?
    ["incrementedDestinations"] as? [String] ?? []
    print(uid)
}
```

Questo metodo ricerca all'interno del database la presenza di un utente in base al proprio id.

Modifica del profilo

Qualora l'utente avesse sbagliato nell'inserimento di alcune informazioni durante la registrazione, o, con il passare del tempo, alcuni dati dovessero essere cambiati, potrebbe sempre aggiornare il proprio profilo, modificando qualsiasi dato:

```
func aggiornaProfilo(nome: String, cognome: String, nazionalità:
String, numeroDiTelefono: String, dataDiNascita: Date, email: String,
password: String, prefisso: String) async {
    guard let uid = userSession?.uid else { return }

    do {
        let user = Auth.auth().currentUser
        // Aggiorna l'email
        if user?.email != email {
            try await user?.updateEmail(to: email)
        }
        // Aggiorna la password
        if !password.isEmpty {
            try await user?.updatePassword(to: password)
        }

        // Aggiorna i dati nel Firestore
        let updatedUser = Utente(
            nome: nome,
            cognome: cognome,
            id: uid,
            password: password,
            nazionalità: nazionalità,
            email: email,
            numeroDiTelefono: "\(numeroDiTelefono)",
            dataDiNascita: dataDiNascita,
            prefisso: prefisso
        )
        let encodedUser = try Firestore.Encoder()
        .encode(updatedUser)
        try await Firestore.firestore().collection("users")
        .document(uid).setData(encodedUser)

        // Ricarica l'utente aggiornato
        await fetchUser()
    } catch {
        print("Errore durante l'aggiornamento del profilo:
        \(error.localizedDescription)")
    }
}
```

Il metodo aggiorna tutti i dati dell'utente e il relativo documento presente all'interno del database.

Logout ed eliminazione dell'account

Qualora l'utente decidesse di effettuare il logout o di voler eliminare il proprio profilo, potrebbe farlo senza alcun problema attraverso i seguenti metodi:

```
func logout() {
    do {
        try Auth.auth().signOut()
        self.userSession = nil
        self.currentUser = nil
        print("Utente disconnesso e sessione:
        \(String(describing: self.userSession))")
    } catch {
        print("DEBUG: Fallimento nel logout con errore
        \(error.localizedDescription)")
    }
}
```

Il metodo di logout imposta la sessione dell'utente collegato a nil.

```
func eliminaAccount() async {
    guard let uid = Auth.auth().currentUser?.uid else { return }

    do {
```

```
            try await Firestore.firestore().collection("users")
            .document(uid).delete()
            try await Auth.auth().currentUser?.delete()

            self.userSession = nil
            self.currentUser = nil
            self.incrementedDestinations = []

            NotificationCenter.default.post(
            name: .userDidLogout, object: nil)
        } catch {
            print("DEBUG: Errore nella rimozione dell'account con errore
            \(error.localizedDescription)")
        }
    }
```

Tale metodo, oltre ad impostare a nil la sessione dell'utente collegato, elimina anche il documento all'interno del database.

3.4 Ricerca di una destinazione

La ricerca di una destinazione inizia nella home dell'applicazione, e, grazie all'API di Google, è possibile ottenere tutte le informazioni a riguardo.

Autocomplete

Quando l'utente interagisce con la barra di ricerca, viene chiamato il seguente metodo dell'AutocompleteController:

```
func makeButton() {
    let btnLaunchAc = UIButton(frame: CGRect(x: 5, y: 150,
    width: 300, height: 35))
    btnLaunchAc.backgroundColor = .blue
    btnLaunchAc.setTitle("Launch autocomplete", for: .normal)
    btnLaunchAc.addTarget(self, action:
    #selector(autocompleteClicked),for: .touchUpInside)
    self.view.addSubview(btnLaunchAc)
}
```

Una volta premuto il bottone, viene chiamato il seguente metodo:

```
    @objc func autocompleteClicked(_ sender: UIButton) {
        let autocompleteController = GMSAutocompleteViewController()
        autocompleteController.delegate = self

        // Specify the place data types to return.
        let fields: GMSPlaceField = GMSPlaceField(rawValue:
        UInt64(UInt(GMSPlaceField.name.rawValue) |
        UInt(GMSPlaceField.placeID.rawValue)))
        autocompleteController.placeFields = fields

        // Specify a filter.
        let filter = GMSAutocompleteFilter()
        filter.type = .city
        autocompleteController.autocompleteFilter = filter

        // Display the autocomplete view controller.
        present(autocompleteController, animated: true,
        completion: nil)
    }
```

Inizialmente viene impostata l'istanza corrente come delegato per la gestione di errori. Successivamente sono selezionati i campi da restituire della destinazione, e la ricerca è filtrata per città.

Con i metodi ora mostrati sono gestiti gli errori, l'annullamento della ricerca da parte dell'utente e le previsioni di autocompletamento:

```
func viewController(_ viewController: GMSAutocompleteViewController,
didFailAutocompleteWithError error: Error) {
    // TODO: handle the error.
    print("Error: ", error.localizedDescription)
}

// User canceled the operation.
func wasCancelled(_ viewController: GMSAutocompleteViewController) {
    dismiss(animated: true, completion: nil)
}

// Turn the network activity indicator on and off again.
func didRequestAutocompletePredictions(_ viewController:
GMSAutocompleteViewController) {
    UIApplication.shared.isNetworkActivityIndicatorVisible = true
}

func didUpdateAutocompletePredictions(_ viewController:
GMSAutocompleteViewController) {
    UIApplication.shared.isNetworkActivityIndicatorVisible = false
}
```

Dettagli di una destinazione

Una volta effettuata la ricerca, attraverso le API Place Details e Place Photos sono estratte tutte le informazioni necessarie riguardo alla città:

```
func cercaDettagliDestinazione(placeID: String, completion: @escaping
(Destinazione?) -> Void) {
    let placesClient = GMSPlacesClient.shared()

    let fields: GMSPlaceField = GMSPlaceField(rawValue:
    UInt64(UInt(GMSPlaceField.name.rawValue) |
           UInt(GMSPlaceField.placeID.rawValue) |
           UInt(GMSPlaceField.formattedAddress.rawValue) |
           UInt(GMSPlaceField.rating.rawValue) |
           UInt(GMSPlaceField.userRatingsTotal.rawValue) |
           UInt(GMSPlaceField.photos.rawValue) |
           UInt(GMSPlaceField.coordinate.rawValue)))

    let sessionToken = GMSAutocompleteSessionToken.init()

    placesClient.fetchPlace(fromPlaceID: placeID, placeFields: fields,
    sessionToken: sessionToken) { (place, error) in
        if let error = error as NSError? {
            print("Error code: \(error.code)")
            print("Error description: \(error.localizedDescription)")
            completion(nil)
            return
        }

        guard let place = place else {
            print("No place details found.")
            completion(nil)
            return
        }

        // Estrai i dettagli necessari
        let nome = place.name ?? "Unknown"
        let indirizzo = place.formattedAddress ?? "Unknown"
        let rating = place.rating
        let numeroRecensioni = Int(place.userRatingsTotal)
        let latitudine = place.coordinate.latitude
        let longitudine = place.coordinate.longitude
        let placeID = place.placeID ?? ""
```

```
        // Carica la foto se disponibile
        if let photoMetadata = place.photos?.first {
            placesClient.loadPlacePhoto(photoMetadata)
            { (image, error) in
                if let error = error {
                    print("Error loading photo:
                    \(error.localizedDescription)")
                } else if let image = image {
                    // 1. Carica l'immagine su Firebase Storage
                    self.uploadImageToFirebaseStorage(image: image,
                    placeID: placeID) { url in
                        guard let imageUrl = url else {
                            completion(nil)
                            return
                        }

                        // 2. Crea l'oggetto Destinazione con l'URL
                        della foto
                        let destinazione = Destinazione(
                            nome: nome,
                            indirizzo: indirizzo,
                            rating: rating,
                            numeroRecensioni: numeroRecensioni,
                            fotoUrl: imageUrl,
                            descrizione: "",
                            popolarità: 1,
                            placeID: placeID,
                            id: UUID(uuidString: placeID) ?? UUID(),
                            latitudine: latitudine,
                            longitudine: longitudine
                        )

                        // 3. Salva la destinazione in Firestore
                        self.salvaDestinazione(destinazione)
                        completion(destinazione)
                    }
                }
            }
        } else {
            // Gestione caso senza foto
            let destinazione = Destinazione(
                nome: nome,
                indirizzo: indirizzo,
                rating: rating,
                numeroRecensioni: numeroRecensioni,
                fotoUrl: nil,
                descrizione: "",
                popolarità: 1,
                placeID: placeID,
                id: UUID(uuidString: placeID) ?? UUID(),
                latitudine: latitudine,
                longitudine: longitudine
            )
            self.salvaDestinazione(destinazione)
            completion(destinazione)
        }
    }
}
```

Il metodo appena mostrato è fondamentale per ottenere tutti i dettagli di una città. Inizialmente viene creata un'istanza di GMSPlacesClient, per interagire con il servizio di Google Places. Sono, ora, selezionati nella costante fields tutti i dati da restituire.

Il passaggio successivo è la richiesta effettiva all'API. Essa avviene attraverso l'ID della città cercata dall'utente. Qualora avvenga un errore o non vengano trovati dettagli riguardanti la città, viene gestito l'errore con una completion vuota. Se i dettagli sono recuperati con successo, sono estratti.

Ora, viene effettuata la chiamata all'API di Google Photos, in cui viene richiesta la prima

foto della città cercata. Se è presente, viene caricata su Firebase Storage attraverso il seguente metodo:

```
func uploadImageToFirebaseStorage(image: UIImage, placeID: String,
completion: @escaping (URL?) -> Void) {
    let storageRef = Storage.storage().reference()
    .child("destinazioni/\(placeID).jpg")

    if let imageData = image.jpegData(compressionQuality: 0.8) {
        storageRef.putData(imageData, metadata: nil)
        { metadata, error in
            if let error = error {
                print("Errore durante il caricamento dell'immagine:
                \(error.localizedDescription)")
                completion(nil)
                return
            }

            // Recupera l'URL dell'immagine caricata
            storageRef.downloadURL { (url, error) in
                if let error = error {
                    print("Errore nel recuperare l'URL dell'immagine:
                    \(error.localizedDescription)")
                    completion(nil)
                    return
                }
                completion(url)
            }
        }
    } else {
        completion(nil)
    }
}
```

Qui viene creato un percorso: destinazioni/placeID.jpg, l'immagine viene compressa e viene recuperato l'URL dell'immagine. Una volta caricata l'immagine sul database, viene creato l'oggetto destinazione con tutti i dati recuperati. Qualora non fosse presente alcuna immagine, verrebbe creato un oggetto destinazione senza di essa.

Il passaggio finale del metodo cercaDettagliDestinazione è il salvataggio della destinazione:

```
func salvaDestinazione(_ destinazione: Destinazione) {
    // Verifica se la destinazione con lo stesso placeID esiste già
    let query = db.collection("destinazioni").whereField("placeID",
    isEqualTo: destinazione.placeID)

    query.getDocuments { (snapshot, error) in
        if let error = error {
            print("Errore durante la verifica del placeID esistente:
            \(error)")
            return
        }

        if let snapshot = snapshot, !snapshot.isEmpty {
            // La destinazione esiste già, aggiorna i dettagli
            if let document = snapshot.documents.first {
                self.db.collection("destinazioni")
                .document(document.documentID).updateData([
                    "nome": destinazione.nome,
                    "indirizzo": destinazione.indirizzo,
                    "rating": destinazione.rating,
                    "numeroRecensioni": destinazione.numeroRecensioni,
                    "fotoUrl": destinazione.fotoUrl?
                    .absoluteString ?? "",
                    "descrizione": destinazione.descrizione,
                    "popolarità": destinazione.popolarità,
                    "latitudine": destinazione.latitudine ?? 0.0,
                    "longitudine": destinazione.longitudine ?? 0.0
```

```
                ]) { error in
                    if let error = error {
                        print("Errore nell'aggiornamento della
                        destinazione: \(error)")
                    } else {
                        print("Destinazione aggiornata con successo!")
                    }
                }
            }
        } else {
            // La destinazione non esiste, crea un nuovo documento
            self.db.collection("destinazioni")
            .document(destinazione.id.uuidString).setData([
                "nome": destinazione.nome,
                "indirizzo": destinazione.indirizzo,
                "rating": destinazione.rating,
                "numeroRecensioni": destinazione.numeroRecensioni,
                "fotoUrl": destinazione.fotoUrl?
                .absoluteString ?? "",
                "descrizione": destinazione.descrizione,
                "popolarità": destinazione.popolarità,
                "userID": Auth.auth().currentUser?.uid ?? "",
                "latitudine": destinazione.latitudine ?? 0.0,
                "longitudine": destinazione.longitudine ?? 0.0,
                "placeID": destinazione.placeID
            ]) { error in
                if let error = error {
                    print("Errore nel salvataggio della
                    destinazione: \(error)")
                } else {
                    print("Destinazione salvata con successo!")
                }
            }
        }
    }
}
```

Questo metodo esegue una query, per verificare se la destinazione è già presente nel database. Qualora lo fosse, aggiorna semplicemente i dati; se, invece, non lo fosse, creerebbe un nuovo documento con tutti i dati recuperati dalla ricerca.

3.5 Destinazioni popolari

Per permettere agli utenti dell'applicazione di scoprire quali sono le mete più cercate, è stato implementato un meccanismo di popolarità.

```
func incrementaPopolarità(per destinazione: Destinazione) {
    guard let userID = Auth.auth().currentUser?.uid else {
        print("ID dell'utente non valido ")
        return
    }

    let visitDocument = db.collection("users")
    .document(userID).collection("visite")
    .document(destinazione.id.uuidString)

    visitDocument.getDocument { document, error in
        if let error = error {
            print("Errore durante il recupero del documento
            \(error.localizedDescription)")
        }
        if let document = document, document.exists {
            print("L'utente ha già visitato questa destinazione")
        } else {
            let query =
            self.db.collection("destinazioni")
            .whereField("placeID", isEqualTo: destinazione.placeID)
```

```
            query.getDocuments { snapshot, error in
                if let error = error {
                    print("Errore nel recupero della destinazione:
                    \(error)")
                    return
                }

                if let snapshot = snapshot, !snapshot.isEmpty {
                    if let document = snapshot.documents.first {
                        let destinazioneRef = document.reference
                        destinazioneRef.updateData(["popolarità":
                        FieldValue.increment(Int64(1))]) { error in
                            if let error = error {
                                print("Errore nell'aggiornamento
                                dellapopolarità: \(error)")
                            } else {
                                print("Popolarità aggiornata con
                                successo!")
                            }
                        }

                        // Salva la visita dell'utente
                        visitDocument.setData(["visitato": true])
                        { error in
                            if let error = error {
                                print("Errore nel salvare la visita:
                                \(error)")
                            } else {
                                print("Visita salvata con successo")
                            self.aggiornaCittaRecentiPopolarita
                            (destinazione)
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Il metodo appena presentato viene chiamato ogni volta che un utente cerca una destinazione. Il suo scopo è verificare se chi è collegato abbia già cercato quella destinazione oppure no. Se è la prima volta che effettua quella ricerca, l'indice di popolarità della città viene incrementato; altrimenti rimane invariato.

Viene, infine, chiamato il seguente metodo:

```
func aggiornaCittaRecentiPopolarita(_ destinazione: Destinazione) {
    guard let userID = Auth.auth().currentUser?.uid else {
        print("ID dell'utente non valido")
        return
    }

    let userRef = db.collection("users").document(userID)

    print("Aggiornamento città recenti per: \(destinazione.nome)")

    userRef.updateData([
        "ultimeCittaCercate": FieldValue
        .arrayUnion([destinazione.nome])
    ]) { error in
        if let error = error {
            print("Errore nell'aggiornamento delle città recenti:
            \(error)")
        } else {
            print("Città recenti aggiornate correttamente")
        }
    }
}
```

Il suo scopo è quello di aggiornare le città recentemente cercate dall'utente.

Per mostrare nella vista le città popolari, è chiamato il seguente metodo:

```
func caricaDestinazioni(completion: @escaping
([Destinazione]) -> Void) {
    db.collection("destinazioni").getDocuments
    { (QuerySnapshot, error) in
        if let error = error {
            print("Errore nel caricamento delle destinazioni:
            \(error)")
            completion([])
        } else {
            var destinazioni = [Destinazione]()

            for document in QuerySnapshot!.documents {
                let data = document.data()
                let id = UUID(uuidString: document.documentID)
                ?? UUID()

                if destinazioni.first
                (where: { $0.id == id }) == nil {
                    let nome = data["nome"] as?
                    String ?? ""
                    let indirizzo = data["indirizzo"] as?
                    String ?? ""
                    let rating = data["rating"] as? Float ?? 0.0
                    let numeroRecensioni = data["numeroRecensioni"]
                    as? Int ?? 0
                    let descrizione = data["descrizione"] as?
                    String ?? ""
                    let popolarità = data["popolarità"] as? Int ?? 0
                    let fotoUrlString = data["fotoUrl"] as? String
                    let fotoUrl = fotoUrlString != nil ? URL(string:
                    fotoUrlString!) : nil
                    let placeID = data["placeID"] as? String ?? ""

                    let destinazione = Destinazione(
                        nome: nome,
                        indirizzo: indirizzo,
                        rating: rating,
                        numeroRecensioni: numeroRecensioni,
                        fotoUrl: fotoUrl,
                        descrizione: descrizione,
                        popolarità: popolarità,
                        placeID: placeID,
                        id: id
                    )

                    destinazioni.append(destinazione)
                }
            }
            completion(destinazioni)
        }
    }
}
```

Tale metodo ha l'obiettivo di estrarre tutte le destinazioni presenti nel database, eliminando i duplicati se l'id della destinazione è presente più volte. Il risultato dell'estrazione viene inserito in un array, che viene filtrato nella vista come segue:

```
    self.destinazioniPopolari = Array(destinazioni.sorted
    { $0.popolarità > $1.popolarità }.prefix(10))
```

Così facendo, sono mostrate le dieci città con popolarità maggiore.

3.6 Città recenti

Ogni volta che un utente cerca una città, il suo nome viene memorizzato in un array nel database. Così facendo, è possibile creare una cronologia delle ultime ricerche dell'utente. La dimensione massima dell'array è di dieci elementi. Il primo passo per l'implementazione avviene con il seguente metodo:

```
func aggiornaCittaRecenti(citta: String) {
    guard let userID = Auth.auth().currentUser?.uid else {
        print("ID dell'utente non valido")
        return
    }

    let userRef = db.collection("users").document(userID)

    userRef.getDocument { document, error in
        if let document = document, document.exists {
            var recentiNomi = document.data()?["ultimeCittaCercate"]
            as?[String] ?? []

            // Rimuovi la città se è già presente
            if let index = recentiNomi.firstIndex(of: citta) {
                recentiNomi.remove(at: index)
            }

            // Aggiungi la città in fondo
            recentiNomi.append(citta)

            // Mantieni solo le ultime 10 città
            recentiNomi = Array(recentiNomi.suffix(10))

            // Aggiorna Firestore con la nuova lista
            userRef.updateData(["ultimeCittaCercate": recentiNomi]) {
            error in
                if let error = error {
                    print("Errore nell'aggiornamento delle città
                    recenti: \(error)")
                } else {
                    print("Lista città recenti
                    aggiornata correttamente")
                }
            }
        } else if let error = error {
            print("Errore nel recupero del documento utente:
            \(error)")
        }
    }
}
```

Lo scopo di questo metodo è analizzare l'array ultimeCittaCercate a seguito di una nuova ricerca da parte dell'utente. Se la città cercata è già presente nella lista, essa viene rimossa e inserita in fondo all'array; altrimenti viene rimosso il primo elemento e viene inserita la nuova città nell'ultima posizione. Per mostrare la cronologia nella vista viene utilizzato il seguente metodo:

```
func caricaDestinazioniRecenti(completion: @escaping
([Destinazione]) -> Void) {
    guard let userID = Auth.auth().currentUser?.uid else {
        print("ID dell'utente non valido")
        return
    }

    let userRef = db.collection("users").document(userID)

    userRef.getDocument { document, error in
        if let document = document, document.exists {
            if var recentiNomi = document.data()?
```

```
            ["ultimeCittaCercate"] as? [String] {
                recentiNomi = Array(recentiNomi.suffix(10))
                var destinazioniRecenti = Array<Destinazione?>
                (repeating: nil, count: recentiNomi.count)

                let group = DispatchGroup()

                for (index, nome) in recentiNomi.enumerated() {
                    group.enter()
                    self.cercaDestinazionePerNome(nome)
                    { destinazione in
                        if let destinazione = destinazione {
                            destinazioniRecenti[index] =
                            destinazione
                        }
                        group.leave()
                    }
                }

                group.notify(queue: .main) {
                    let destinazioniValide =
                    destinazioniRecenti.compactMap { $0 }
                    completion(destinazioniValide.reversed())
                }
            } else {
                print("Non ci sono città recenti salvate")
                completion([])
            }
        } else if let error = error {
            print("Errore nel recupero del documento utente:
            \(error)")
            completion([])
        }
    }
}
```

Il metodo caricaDestinazioniRecenti cerca all'interno del documento dell'utente nel database la presenza dell'array ultimeDestinazioniCercate. Se è presente, va ad estrarre in base al nome della destinazione le ultime dieci città cercate, restituendo l'array in ordine invertito, in modo tale da avere le ricerche ordinate dalla più recente alla meno recente; qualora l'array non fosse presente, l'errore viene gestito.

```
func cercaDestinazionePerNome(_ nome: String, completion: @escaping
(Destinazione?) -> Void) {
    let destinazioniRef = db.collection("destinazioni")
    let query = destinazioniRef.whereField("nome", isEqualTo: nome)

    query.getDocuments { snapshot, error in
        if let error = error {
            print("Errore nella ricerca della destinazione:
            \(error)")
            completion(nil)
            return
        }

        if let document = snapshot?.documents.first {
            let data = document.data()
            let destinazione = Destinazione(
                nome: data["nome"] as? String ?? "",
                indirizzo: data["indirizzo"] as? String ?? "",
                rating: data["rating"] as? Float ?? 0.0,
                numeroRecensioni: data["numeroRecensioni"]
                as? Int ?? 0,
                fotoUrl: URL(string: data["fotoUrl"]
                as? String ?? ""),
                descrizione: data["descrizione"] as? String ?? "",
                popolarità: data["popolarità"] as? Int ?? 0,
                placeID: data["placeID"] as? String ?? "",
                id: UUID(),
```

```
                latitudine: data["latitudine"] as? CLLocationDegrees,
                longitudine: data["longitudine"]
                as? CLLocationDegrees
            )
            completion(destinazione)
        } else {
            print("Nessuna destinazione trovata per il nome:
            \(nome)")
            completion(nil)
        }
    }
}
```

Il metodo cercaDestinazionePerNome, infine, ha lo scopo di estrarre i dati necessari in base al nome della città presente nell'array ultimeDestinazioniCercate.

3.7 Ricerca dei luoghi di interesse

Per effettuare una ricerca dei luoghi di interesse, l'utente deve innanzitutto cercare la destinazione in cui essi si trovano.

```
func cercaLuoghiDiInteresse(intornoACitta citta: String) {
    // Geocodifica per latitudine e longitudine
    let geocoder = CLGeocoder()
    geocoder.geocodeAddressString(citta)
    { [weak self] (placemarks, error) in
        guard let coordinate = placemarks?.first?.location?
        .coordinate else {
            print("Errore nella geocodifica della città
            \(error?.localizedDescription ?? "Errore sconosciuto")")
            return
        }

        Task {
            await self?.fetchLuoghiInteresse(coordinate: coordinate)
        }
    }
}
```

La ricerca dei luoghi di interesse inizia attraverso una geocodifica di latitudine e longitudine della destinazione. Se essa non va a buon fine, l'errore viene gestito; altrimenti viene chiamato il seguente metodo:

```
private func fetchLuoghiInteresse(coordinate:
CLLocationCoordinate2D) {
    // Raggio di ricerca
    let circularLocationRestriction =
    GMSPlaceCircularLocationOption(coordinate, 5000)

    // Proprietà aggiuntive richieste
    let placeProperties = [
        GMSPlaceProperty.name.rawValue,
        GMSPlaceProperty.coordinate.rawValue,
        GMSPlaceProperty.placeID.rawValue,
        GMSPlaceProperty.photos.rawValue,
        GMSPlaceProperty.formattedAddress.rawValue,
        GMSPlaceProperty.rating.rawValue,
        GMSPlaceProperty.userRatingsTotal.rawValue
    ]

    // Richiesta
    var request = GMSPlaceSearchNearbyRequest(locationRestriction:
    circularLocationRestriction, placeProperties: placeProperties)

    // Tipi di luoghi da includere
    let includedTypes = ["tourist_attraction", "museum"]
```

```
    request.includedTypes = includedTypes

    let callback: GMSPlaceSearchNearbyResultCallback = { [weak self]
    results, error in
        guard let self = self, error == nil else {
            if let error = error {
                print("Errore nella ricerca dei luoghi di interesse:
                \(error.localizedDescription)")
            }
            return
        }

        guard let results = results else {
            print("Nessun risultato trovato")
            return
        }

        var destinazioni: [Destinazione] = []
        let dispatchGroup = DispatchGroup()

        for place in results {
            let nome = place.name ?? "Sconosciuto"
            let indirizzo = place.formattedAddress ?? "Indirizzo non
            disponibile"
            let rating = place.rating
            let numeroRecensioni = place.userRatingsTotal
            let descrizione = place.attributions?.string ??
            "Descrizione non disponibile"
            let placeID = place.placeID ?? ""

            var fotoUrl: URL? = nil
            if let photoMetadata = place.photos?.first {
                dispatchGroup.enter()
                self.placesClient.loadPlacePhoto(photoMetadata)
                { (photo, error) in
                    if let error = error {
                        //print("Errore nel caricamento della foto:
                        \(error.localizedDescription)")
                        fotoUrl = URL(string:
                        "https://via.placeholder.com/200")
                    } else if let photo = photo, let imageData =
                    photo.pngData() {
                        fotoUrl =
                        self.saveImageToTemporaryDirectory
                        (imageData: imageData)
                    }
                    dispatchGroup.leave()
                }
            }

            let dettaglio = Destinazione(
                nome: nome,
                indirizzo: indirizzo,
                rating: rating,
                numeroRecensioni: Int(numeroRecensioni),
                fotoUrl: fotoUrl,
                descrizione: descrizione,
                popolarità: 0,
                placeID: placeID
            )
            destinazioni.append(dettaglio)
        }

        dispatchGroup.notify(queue: .main) {
            self.luoghiInteresse = destinazioni
        }
    }

    placesClient.searchNearby(with: request, callback: callback)
}
```

fetchLuoghiInteresse crea intorno alle coordinate della città un'area circolare di cinque chilometri. Successivamente sono selezionati i dati da estrarre, e viene creata una richiesta. Quest'ultima, poi, viene filtrata per musei e attrazioni turistiche. I risultati della ricerca, se presenti, sono inseriti all'interno di un array. Esattamente come per le destinazioni, viene gestita la foto del luogo di interesse.

3.8 Ricerca dei piatti tipici

Anche in questo caso, per l'utente è necessario cercare inizialmente la destinazione.

Lista dei piatti tipici

Per ottenere una lista dei piatti tipici, viene chiamato il seguente metodo:

```
func fetchCibiPerCitta(area: String, completion: @escaping
([PiattoTipico]) -> Void) {
    let urlString =
    "https://www.themealdb.com/api/json/v1/1/filter.php?a=\(area)"

    guard let url = URL(string: urlString) else {
        print("URL non valido")
        return
    }

    let task = URLSession.shared.dataTask(with: url) {
    data, response, error in
        if let error = error {
            print("Errore durante il caricamento dei piatti:
            \(error.localizedDescription)")
            completion([])
            return
        }

        guard let data = data else {
            print("No data")
            completion([])
            return
        }

        do {
            let decoder = JSONDecoder()
            let response =
            try decoder.decode(PiattoTipicoResponse.self,
            from: data)
            let meals = response.meals ?? []
            completion(meals)
        } catch {
            print("Errore nella decodifica JSON:
            \(error.localizedDescription)")
            completion([])
        }
    }

    task.resume()
}
```

fetchCibiPerCitta ha lo scopo di eseguire una chiamata all'API TheMealDB. L'unica informazione necessaria per la richiesta è la nazione in cui la città si trova. A questo punto, avviene una decodifica del file JSON restituito dall'API.

Dettagli di un piatto

Qualora l'utente volesse ottenere informazioni su un piatto tipico, verrebbe chiamato il seguente metodo:

```
func fetchDettagliPiatto(id: String, completion: @escaping
(DettagliPiatto?) -> Void) {
    let urlString =
    "https://www.themealdb.com/api/json/v1/1/lookup.php?i=\(id)"

    guard let url = URL(string: urlString) else {
        print("URL non valido")
        completion(nil)
        return
    }

    let task = URLSession.shared.dataTask(with: url) {
    data, response, error in
        if let error = error {
            print("Errore: \(error.localizedDescription)")
            completion(nil)
            return
        }

        guard let data = data else {
            print("Nessun dato ricevuto")
            completion(nil)
            return
        }

        do {
            let decoder = JSONDecoder()
            let response =
            try decoder.decode(DettagliPiattoResponse.self,
            from: data)
            let dettagliPiatto = response.meals.first
            completion(dettagliPiatto)
        } catch {
            print("Errore nel parsing: \(error.localizedDescription)")
            completion(nil)
        }
    }
    task.resume()
}
```

Qui viene effettuata una chiamata all'API in base all'id del piatto. Il risultato della richiesta è un file JSON contenenente gli ingredienti della pietanza, che viene decodificato.

3.9 Gestione delle recensioni

Ogni utente può inserire una recensione ad una destinazione, ad un luogo di interesse e ad un piatto tipico. Essa si compone di un voto da una a cinque stelle e di un breve testo, in cui può esprimere il proprio parere a riguardo. È, ora, mostrato il meccanismo per un luogo di interesse e una destinazione. Il funzionamento per i piatti tipici è analogo.

Creazione di una recensione

Per aggiungere una nuova recensione è utilizzato il seguente metodo:

```
func aggiungiRecensione(destinazioneId: String, utenteId: String,
testo: String, rating: Int, completion: @escaping (Error?) -> Void) {
    let recensioneId = UUID().uuidString
    let recensioneData: [String: Any] = [
        "id": recensioneId,
```

```
        "utenteId": utenteId,
        "luogoId": destinazioneId,
        "testo": testo,
        "rating": rating,
        "data": Timestamp()
    ]

    db.collection("recensioni").document(recensioneId)
    .setData(recensioneData) { error in
        completion(error)
    }
}
```

Qui viene aggiunta una nuova recensione al database.

Recupero delle recensioni

Per recuperare tutte le recensioni presenti di un dato luogo, è chiamato il seguente metodo:

```
func fetchRecensioniPerLuogo(luogoId: String, completion: @escaping
([Recensione]) -> Void) {
    db.collection("recensioni")
        .whereField("luogoId", isEqualTo: luogoId)
        .getDocuments { (querySnapshot, error) in
            if let error = error {
                print("Errore nel fetch delle recensioni:
                \(error.localizedDescription)")
                completion([])
                return
            }

            guard let documents = querySnapshot?.documents else {
                print("Nessun documento trovato")
                completion([])
                return
            }

            var recensioni: [Recensione] = []
            for document in documents {
                let data = document.data()

                // Stampa i dati recuperati per verificare
                print("Dati recensione recuperati: \(data)")

                let recensione = Recensione(
                    id: document.documentID,
                    utenteId: data["utenteId"] as? String ?? "",
                    luogoId: data["luogoId"] as? String ?? "",
                    testo: data["testo"] as? String ?? "",
                    data: (data["data"] as? Timestamp)?.dateValue()
                    ?? Date(),
                    rating: data["rating"] as? Double ?? 0.0
                )
                recensioni.append(recensione)
            }

            completion(recensioni)
        }
}
```

Il suo scopo è quello di eseguire una query nel database ed estrarre eventuali documenti contenenti le recensioni lasciate dagli utenti. Se l'estrazione va a buon fine, viene creato un array contenente i risultati.

Calcolo del rating di un luogo

Ogni volta che un utente visualizza un luogo, sarà presente il rating dato dagli altri utenti ad esso.

```
func calcolaMediaRatingLuogo(luogoId: String, completion: @escaping
(Double) -> Void) {
    db.collection("recensioni")
        .whereField("luogoId", isEqualTo: luogoId)
        .getDocuments { (querySnapshot, error) in
            if let error = error {
                print("Erorre nel recuperare le recensioni:
                \(error.localizedDescription)")
                completion(0.0)
                return
            }

            guard let documents = querySnapshot?.documents,
            !documents.isEmpty else {
                completion(0.0)
                return
            }

            var totalRating: Double = 0.0
            var count: Double = 0.0

            for document in documents {
                let data = document.data()
                let rating = data["rating"] as? Double ?? 0.0
                totalRating += rating
                count += 1
            }

            let mediaRating = totalRating / count
            completion(mediaRating)
        }
}
```

Il metodo qui mostrato va ad estrarre tutte le recensioni presenti nel database in base all'id di un luogo. Successivamente, itera attraverso tutti i documenti trovati, ed esegue una media aritmetica dei voti degli utenti.

3.10 Gestione dei preferiti

In questa sezione viene analizzata l'implementazione dei preferiti. Ogni utente può aggiungere, eliminare e visualizzare città, luoghi di interesse e piatti tipici preferiti.

Aggiunta di un preferito

```
func aggiungiPreferito(utenteId: String, tipoPreferito:
TipoPreferito, tipoId: String, completion: @escaping
(Result<String, Error>) -> Void) {
    guard let userID = Auth.auth().currentUser?.uid else {
        print("ID dell'utente non valido")
        return
    }

    switch tipoPreferito {
    case.piattoTipicoPreferito:
        let preferito = Preferiti(
            id: nil,
            utenteId: userID,
            tipo: tipoPreferito.rawValue,
            tipoId: tipoId,
```

```
            placeID: ""
        )
        salvaPreferito(preferito: preferito, completion: completion)

    case .destinazionePreferita:
        ottieniNomeCittaDaID(id: tipoId) { nomeCitta in
            guard let nomeCitta = nomeCitta else {
                completion(.failure(NSError(domain: "Nome città non
                trovato", code: 0, userInfo: nil)))
                return
            }

            self.ottieniPlaceID(nomeCitta: nomeCitta) { result in
                switch result {
                case .success(let placeID):
                    let preferito = Preferiti(
                        id: nil,
                        utenteId: userID,
                        tipo: tipoPreferito.rawValue,
                        tipoId: tipoId,
                        placeID: placeID
                    )
                    self.salvaPreferito(preferito: preferito,
                    completion: completion)

                case .failure(let error):
                    print("Errore nel recupero del place: ID:
                    \(error)")
                    completion(.failure(error))
                }
            }
        }

    case .luogoDiInteressePreferito:
        luogoDiInteresseController.ottieniDettagliLuogo
        (placeID: tipoId)
        { result in
            switch result {
            case .success(let destinazione):
                // Se la destinazione è stata trovata, crea l'oggetto
                Preferiti
                let preferito = Preferiti(
                    id: nil,
                    utenteId: userID,
                    tipo: tipoPreferito.rawValue,
                    tipoId: tipoId,
                    placeID: destinazione.placeID
                )
                // Salva il preferito
                self.salvaPreferito(preferito: preferito, completion:
                completion)

            case .failure(let error):
                // Se c'è stato un errore, restituisci l'errore nel
                completion
                completion(.failure(NSError(domain:
                "Errore nel recupero del luogo di interesse:
                \(error.localizedDescription)",
                code: 0, userInfo: nil)))
            }
        }
    }
}
```

Il metodo appena presentato gestisce l'aggiunta di un nuovo preferito. Innanzitutto verifica che l'id dell'utente sia valido, e poi, in base alla tipologia di preferito, viene gestita l'aggiunta. Si hanno tre casi:

1. Il primo caso riguarda i piatti tipici, per cui viene semplicemente creato l'oggetto Preferito e viene salvato.

2. Il secondo caso riguarda le destinazioni, e inizialmente si ricava il nome della città dall'id del documento nel database, attraverso una query:

```
    func ottieniNomeCittaDaID(id: String, completion: @escaping
    (String?) -> Void) {
    let query = db.collection("destinazioni").document(id)
    .getDocument {
    (document, error) in
        if let error = error {
            print("Errore nel recupero del nome della città:
            \(error)")
            completion(nil)
            return
        }

        guard let document = document, document.exists,
        let data = document.data() else {
            print("Documento non trovato")
            completion(nil)
            return
        }

        let nomeCitta = data["nome"] as? String
        completion(nomeCitta)
    }
}
```

Successivamente esso viene utilizzato per ricavare il placeID attraverso l'API di Google:

```
    func ottieniPlaceID(nomeCitta: String, completion: @escaping
    (Result<String, Error>) -> Void) {
    let apiKey = "AIzaSyDAwm9PsexK1IRuPbOSvg4TjqivqqWwjOI"

    guard let encodedNomeCitta =
    nomeCitta.addingPercentEncoding(withAllowedCharacters:
    .urlQueryAllowed) else {
            completion(.failure(NSError(domain:
            "Encoding Error", code: 0, userInfo:
            [NSLocalizedDescriptionKey: "Error
            encoding city name"])))
            return
        }

    let urlString =
    "https://maps.googleapis.com/maps/api/place
    /findplacefromtext/json ?input=
    \(encodedNomeCitta)&inputtype=textquery&key=\(apiKey)"

    guard let url = URL(string: urlString) else {
        completion(.failure(NSError(domain: "Invalid URL",
        code: 0, userInfo: nil)))
        return
    }

    URLSession.shared.dataTask(with: url)
    { (data, response, error) in DispatchQueue.main.async {
            if let error = error {
                completion(.failure(error))
                return
            }

            guard let data = data else {
                completion(.failure(NSError(domain: "No data
                received", code: 0, userInfo: nil)))
                return
            }

            do {
                if let json = try JSONSerialization.jsonObject
```

```
                (with: data, options: []) as? [String: Any] {
                    // Controlla lo stato della risposta
                    if let status = json["status"] as? String,
                    status == "OK" {
                        if let candidates = json["candidates"]
                        as? [[String: Any]],
                        let firstCandidate = candidates.first,
                        let placeID = firstCandidate["place_id"]
                        as? String {
                            completion(.success(placeID))
                        } else {
                            completion(.failure(NSError(
                            domain: "No place ID found", code: 0,
                            userInfo: nil)))
                        }
                    } else {
                        let errorMessage = json["error_message"]
                        as? String ?? "No results found"
                        completion(.failure(NSError(domain:
                        errorMessage, code: 0, userInfo: nil)))
                    }
                } else {
                    completion(.failure(NSError(domain: "Invalid
                    JSON format", code: 0, userInfo: nil)))
                }
            } catch let error {
                completion(.failure(error))
            }
        }
    }.resume()
}
```

L'API restituisce il placeID della città in un file JSON, che viene decodificato. Se non ci sono errori, viene creato l'oggetto Preferito e salvato.

3. L'ultimo caso riguarda i luoghi di interesse. Inizialmente viene chiamato il seguente metodo:

```
func ottieniDettagliLuogo(placeID: String, completion:
@escaping (Result<Destinazione, Error>) -> Void) {
// Specifica il client di Google Places
let client = GMSPlacesClient.shared()

// Specifica le proprietà del luogo da recuperare
let myProperties = [
    GMSPlaceProperty.name.rawValue,
    GMSPlaceProperty.website.rawValue,
    GMSPlaceProperty.formattedAddress.rawValue,
    GMSPlaceProperty.rating.rawValue,
    GMSPlaceProperty.userRatingsTotal.rawValue,
    GMSPlaceProperty.photos.rawValue,
    GMSPlaceProperty.placeID.rawValue
]

// Crea l'oggetto GMSFetchPlaceRequest
let fetchPlaceRequest = GMSFetchPlaceRequest(
placeID: placeID, placeProperties: myProperties,
sessionToken: nil)

// Effettua la richiesta per ottenere i dettagli del luogo
client.fetchPlace(with: fetchPlaceRequest, callback: {
(place: GMSPlace?, error: Error?) in
    guard let place = place, error == nil else {
        if let error = error {
            completion(.failure(error))
        } else {
            completion(.failure(NSError(domain:
            "Luogo non trovato", code: 0, userInfo: nil)))
        }
```

```
                return
            }

            // Estrai i dettagli del luogo
            let nome = place.name ?? "Nome non disponibile"
            let indirizzo = place.formattedAddress ??
            "Indirizzo non disponibile"
            let rating = place.rating
            let numeroRecensioni = place.userRatingsTotal
            let placeID = place.placeID

            // Recupera la foto se disponibile
            var fotoUrl: URL? = nil
            if let photoMetadata = place.photos?.first {
                client.loadPlacePhoto(photoMetadata)
                { (photo, error) in
                    if let photo = photo, let imageData = photo
                    .pngData() {
                        fotoUrl = self.saveImageToTemporaryDirectory
                        (imageData: imageData)
                    }
                }
            }

            // Crea l'oggetto Destinazione con i dati recuperati
            let destinazione = Destinazione(
                nome: nome,
                indirizzo: indirizzo,
                rating: rating,
                numeroRecensioni: Int(numeroRecensioni),
                fotoUrl: fotoUrl,
                descrizione: "",
                popolarità: 0,
                placeID: placeID ?? ""
            )

            // Restituisci il risultato con l'oggetto Destinazione
            completion(.success(destinazione))
        })
    }
```

Esso estrae i dettagli di un luogo di interesse in base al suo placeID. Viene, ora, creato l'oggetto Preferito, e salvato.

Il salvataggio di un preferito avviene come segue:

```
private func salvaPreferito(preferito: Preferiti, completion:
@escaping (Result<String, Error>) -> Void) {
    do {
        var documentRef: DocumentReference? = nil
        documentRef = try self.db
        .collection("users").document(preferito.utenteId)
        .collection("preferiti").addDocument(from: preferito)
        { error in
            if let error = error {
                completion(.failure(error))
            } else {
                if let documentId = documentRef?.documentID {
                    self.db.collection("users")
                    .document(preferito.utenteId)
                    .collection("preferiti").document(documentId)
                    .updateData(["id": documentId])
                    completion(.success(documentId))
                }
            }
        }
        print("Preferito aggiunto con ID:
        \(documentRef?.documentID ?? "")")
    } catch let error {
        completion(.failure(error))
```

```
    }
}
```

Il metodo aggiunge un oggetto Preferito al database, o gestisce un eventuale errore. A livello visivo viene riempita l'icona di un cuore, per mostrare all'utente che l'aggiunta ha avuto successo.

Rimozione di un preferito

Ogni utente può rimuovere un preferito come segue:

```
func rimuoviPreferito(utenteId: String, preferitoId: String?,
completion: @escaping (Error?) -> Void) {
    guard let userID = Auth.auth().currentUser?.uid else {
        print("ID dell'utente non valido")
        return
    }

    guard let preferitoId = preferitoId else {
        print("Errore: preferitoId non valido")
        return
    }

    db.collection("users").document(userID).collection("preferiti")
    .document(preferitoId).delete { error in
        if let error = error {
            print("Errore nella rimozione del preferito:
            \(error.localizedDescription)")
        } else {
            print("Preferito rimosso correttamente")
        }
        completion(error)
    }
}
```

Il metodo va ad eliminare il documento del preferito dal database. A livello visivo l'icona del cuore viene svuotata, per mostrare all'utente che la rimozione ha avuto successo.

Estrazione dei preferiti

Per mostrare all'utente i propri preferiti, viene eseguita una query sul database:

```
private func fetchPreferiti(tipoPreferito: TipoPreferito, utenteId:
String, completion: @escaping ([Preferiti]?, Error?) -> Void) {
    guard let userID = Auth.auth().currentUser?.uid else {
        print("ID dell'utente non valido")
        return
    }

    db.collection("users").document(userID).collection("preferiti")
        .whereField("tipo", isEqualTo: tipoPreferito.rawValue)
        .getDocuments { (snapshot, error) in
            guard let documents = snapshot?.documents,
            error == nil else {
                completion(nil, error)
                return
            }

            let preferiti = documents.compactMap
            { doc -> Preferiti? in
                var preferito = try? doc.data(as: Preferiti.self)
                preferito?.id = doc.documentID
                return preferito
            }
            completion(preferiti, nil)
```

```
        }
}
```

Per facilitare la visualizzazione dei preferiti all'utente, essi sono divisi in base al tipo:

```
func fetchDestinazioniPreferite(utenteId: String, completion:
@escaping ([Preferiti]?, Error?) -> Void) {
    fetchPreferiti(tipoPreferito: .destinazionePreferita,
    utenteId: utenteId, completion: completion)
}

func fetchLuoghiDiInteressePreferiti(utenteId: String,
completion: @escaping ([Preferiti]?, Error?) -> Void) {
    fetchPreferiti(tipoPreferito: .luogoDiInteressePreferito,
    utenteId: utenteId, completion: completion)
}

func fetchPiattiTipiciPreferiti(utenteId: String,
completion: @escaping ([Preferiti]?, Error?) -> Void) {
    fetchPreferiti(tipoPreferito: .piattoTipicoPreferito,
    utenteId: utenteId) { preferiti, error in
        if let preferiti = preferiti {
            print("Trovati \(preferiti.count) piatti preferiti")
        } else if let error = error {
            print("Errore nel fetch dei piatti preferiti:
            \(error.localizedDescription)")
        }
        completion(preferiti, error)
    }
}
```

Ognuno di questi metodi chiama il metodo fetchPreferiti, passando come parametro il tipo di preferito.

3.11 Gestione della galleria

All'interno dell'applicazione è possibile avere una propria galleria privata in cui inserire i migliori ricordi di un viaggio. Ogni ricordo si compone di una data, il nome della città e un file multimediale (foto o video).

Inserimento di un ricordo

L'inserimento di un nuovo ricordo inizia con la selezione della città attraverso l'autocomplete di Google, e poi si seleziona la data del ricordo. Infine, attraverso un media picker si seleziona il tipo di file multimediale:

```
func showMediaPicker(from viewController: UIViewController,
completion: @escaping (MediaType?) -> Void) {
    self.parentViewController = viewController
    self.completion = completion

    var configuration = PHPickerConfiguration()
    configuration.filter = .any(of: [.images, .videos])
    // Limita la selezione a un solo elemento
    configuration.selectionLimit = 1

    let picker = PHPickerViewController(configuration: configuration)
    picker.delegate = self
    viewController.present(picker, animated: true)
}
```

Qui si accede alla galleria dell'utente, dove sono presenti immagini e video, e viene stabilito un limite di upload pari ad un singolo file.

```
func picker(_ picker: PHPickerViewController,
didFinishPicking results: [PHPickerResult]) {
    picker.dismiss(animated: true)

    guard let provider = results.first?.itemProvider else {
        completion?(nil)
        return
    }

    if provider.hasItemConformingToTypeIdentifier(
    UTType.image.identifier) {
        // Se è stata selezionata un'immagine
        provider.loadObject(ofClass: UIImage.self)
        { [weak self] image, error in
            guard let image = image as? UIImage, error == nil else {
                self?.completion?(nil)
                return
            }
            self?.completion?(.photo(image))
        }
    } else if provider.hasItemConformingToTypeIdentifier
    (UTType.movie.identifier) {
        // Se è stato selezionato un video
        provider.loadFileRepresentation
        (forTypeIdentifier: UTType.movie.identifier)
        { [weak self] fileURL, error in
            guard let fileURL = fileURL, error == nil else {
                self?.completion?(nil)
                return
            }

            let tempURL = FileManager.default.temporaryDirectory
            .appendingPathComponent(fileURL.lastPathComponent)

            do {
                try FileManager.default.copyItem(at: fileURL,
                to: tempURL)
                self?.completion?(.video(tempURL))
            } catch {
                print("Errore nella copia del file video:
                \(error.localizedDescription)")
                self?.completion?(nil)
            }
        }
    }
}
```

Qui viene gestita la selezione dell'utente. Se si tratta di una foto, viene restituito un file di tipo UIImage; se si tratta di un video, esso viene caricato in uno storage temporaneo, e viene restituito l'url.

La gestione dell'upload di un ricordo inizia dalla vista. Quando l'utente preme il bottone "pubblica", viene chiamato il seguente metodo:

```
private func uploadMedia() async {
    guard let user = utenteController.currentUser else { return }
    isLoading = true

    if let selectedImage = selectedImage,
    let imageData = selectedImage
    .jpegData(compressionQuality: 0.8) {
        // Upload dell'immagine
        if let url = await galleriaController.uploadFile(
            userId: user.id,
            fileData: imageData,
            fileType: .foto
        ) {
            let file = FileMultimediale(
                id: UUID().uuidString,
                utente: user,
```

```
                tipo: .foto,
                posizione: luogo.isEmpty ? cityName : luogo,
                data: data,
                url: url,
                descrizione: description
            )

            if let latitude = cityLatitude,
            let longitude = cityLongitude {
                await galleriaController.addCityToGallery(
                    userId: user.id,
                    city: luogo.isEmpty ? cityName : luogo,
                    lat: latitude,
                    long: longitude
                )
            }

            await galleriaController.addFileToCity(
                userId: user.id,
                city: luogo.isEmpty ? cityName : luogo,
                file: file
            )
        }
    } else if let selectedVideoURL = selectedVideoURL {
        do {
            // Converti il video in data binario
            let videoData = try Data(contentsOf: selectedVideoURL)
            // Upload del video
            if let url = await galleriaController.uploadFile(
                userId: user.id,
                fileData: videoData,
                fileType: .video
            ) {
                let file = FileMultimediale(
                    id: UUID().uuidString,
                    utente: user,
                    tipo: .video,
                    posizione: luogo.isEmpty ? cityName : luogo,
                    data: data,
                    url: url,
                    descrizione: description
                )

                if let latitude = cityLatitude,
                let longitude = cityLongitude {
                    await galleriaController.addCityToGallery(
                        userId: user.id,
                        city: luogo.isEmpty ? cityName : luogo,
                        lat: latitude,
                        long: longitude
                    )
                }

                await galleriaController.addFileToCity(
                    userId: user.id,
                    city: luogo.isEmpty ? cityName : luogo,
                    file: file
                )
            }
        } catch {
            print("Errore durante la conversione del video in dati:
            \(error.localizedDescription)")
        }
    }

    isLoading = false
    dismiss()
}
```

Se il file selezionato è un'immagine, essa viene compressa e avviene l'upload:

```
func uploadFile(userId: String, fileData: Data, fileType: TipoFile)
async -> URL? {
    let fileName = UUID().uuidString +
    (fileType == .foto ? ".jpg" : ".mov")
    let fileRef = storageRef.child("files/\(userId)/\(fileName)")

    do {
        let metadata = StorageMetadata()
        metadata.contentType = fileType ==
        .foto ? "image/jpeg" : "video/quicktime"

        // Carica i dati
        _ = try await fileRef.putDataAsync(
        fileData, metadata: metadata)

        // Ottieni l'URL di download
        let downloadURL = try await fileRef.downloadURL()
        return downloadURL
    } catch {
        print("Errore durante il caricamento del file:
        \(error.localizedDescription)")
        return nil
    }
}
```

Viene quindi creato e caricato un file in Firebase Storage, e viene restituito l'url di download del file. In caso di errori, essi vengono gestiti.

Dopo l'upload, viene creato l'oggetto FileMultimediale, e, in base alle coordinate della città, questa viene aggiunta alla galleria:

```
func addCityToGallery(userId: String, city: String,
lat: Double, long: Double) async {
    let cityData: [String: Any] = [
        "nomeCitta": city,
        "latitudine": lat,
        "longitudine": long
    ]

    do {
        try await db.collection("users").document(userId)
            .collection("galleria").document(city)
            .setData(cityData, merge: true)

        await fetchGallery()
    } catch {
        self.errorMessage = "Errore nell'aggiunta della città:
        \(error.localizedDescription)"
    }
}
```

Lo scopo di questo metodo è raggruppare i file multimediali per città. Qui viene invocato fetchGallery, che recupera la galleria dell'utente collegato:

```
func fetchGallery() async {
    guard let userId = utenteController.currentUser?.id else {
        self.errorMessage = "Utente non autenticato"
        return
    }

    self.loading = true
    self.gallery = [:]

    do {
        let snapshot = try await db.collection("users")
        .document(userId).collection("galleria").getDocuments()

        var newGallery: [String: [FileMultimediale]] = [:]
```

```
            for document in snapshot.documents {
                let cityName = document.documentID
                let files = try await fetchFiles(for: document)
                newGallery[cityName] = files
            }

            self.gallery = newGallery
        } catch {
            self.errorMessage = "Errore nel caricamento della galleria:
            \(error.localizedDescription)"
        }

        self.loading = false
    }
```

Il metodo fetchFiles ha l'obiettivo di estrarre tutti i file multimediali di un utente, che vengono raggruppati nell'array files:

```
private func fetchFiles(for document: DocumentSnapshot)
async throws -> [FileMultimediale] {
    var files: [FileMultimediale] = []

    let fileSnapshot = try await document.
    reference.collection("fileMultimediali").getDocuments()

    for fileDoc in fileSnapshot.documents {
        let data = fileDoc.data()
        let tipoString = data["tipo"] as? String ?? "foto"
        let tipo: TipoFile = tipoString == "foto" ? .foto : .video
        let timestamp = data["data"] as? Timestamp ??
        Timestamp(date: Date())
        let date = timestamp.dateValue()
        let urlString = data["url"] as? String ?? ""

        guard let url = URL(string: urlString) else {
            print("URL non valido: \(urlString)")
            continue
        }
        let file = FileMultimediale(
            id: fileDoc.documentID,
            utente: self.utenteController.currentUser!,
            tipo: tipo,
            posizione: document.documentID,
            data: date,
            url: url,
            descrizione: data["descrizione"] as? String ?? ""
        )
        files.append(file)
    }

    return files
}
```

Una volta caricata la città nella galleria, viene aggiunto il file alla città:

```
func addFileToCity(userId: String, city: String,
file: FileMultimediale) async {
    do {
        let fileData: [String : Any] = [
            "tipo": file.tipo == .foto ? "foto" : "video",
            "data": Timestamp(date: file.data),
            "url": file.url.absoluteString,
            "descrizione": file.descrizione as Any
        ]

        try await db.collection("users").document(userId)
            .collection("galleria").document(city)
            .collection("fileMultimediali").document(file.id)
            .setData(fileData)
```

```
            await fetchGallery()

            DispatchQueue.main.async {
                self.gallery[city, default: []].append(file)
            }
        } catch {
            print("Errore nell'aggiunta del file: \(error)")
        }
    }
```

Qui il file viene associato alla città, ed è aggiornata la galleria locale. Il ragionamento in caso di video è analogo.

Eliminazione di un file multimediale

Ogni utente può eliminare i file multimediali caricati.

```
func eliminaFile(userId: String, city: String,
file: FileMultimediale) async {
    do {
        let filePath = file.url.pathComponents.
        suffix(3).joined(separator: "/")

        let fileRef = storageRef.child(filePath)
        try await fileRef.delete()

        try await db.collection("users").document(userId)
            .collection("galleria").document(city)
            .collection("fileMultimediali").document(file.id)
            .delete()

        await fetchGallery()

        print("File eliminato con successo")
    } catch {
        print("Errore durante l'eliminazione del file:
        \(error.localizedDescription)")
    }
}
```

L'eliminazione comprende un recupero del percorso del file e la successiva eliminazione nel cloud storage e nel database. Infine la galleria viene aggiornata.

3.12 Gestione delle prenotazioni

In questa sezione viene analizzato il funzionamento delle prenotazioni all'interno dell'applicazione. Ogni utente può prenotare aerei, treni, bus e hotel.

3.12.1 Prenotazione degli aerei

Ricerca di un volo

Il primo passo per prenotare un volo è la ricerca di esso:

```
func cercaVoli(departureId: String, arrivalId: String,
outboundDate: String, returnDate: String,
currency: String, language: String) {
    let apiKey = "9aee355d056ba2680284906025487-
    7fb8ab08b2856dba38c5ebf791487ba7a94"

    // Costruisci la stringa dell'URL
    let urlString = "https://serpapi.com/search.json?engine=
    google_flights&departure_id=\(departureId)&arrival_id=
```

```
    \(arrivalId)&outbound_date=\(outboundDate)&return_date=
    \(returnDate)&currency=\(currency)&hl=
    \(language)&api_key=\(apiKey)"

    guard let url = URL(string: urlString) else {
        print("Invalid URL")
        return
    }

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

    let task = URLSession.shared.dataTask(with: request)
    { data, response, error in
        if let error = error {
            self.errorMessage = "Error:
            \(error.localizedDescription)"
            return
        }

        guard let data = data else {
            self.errorMessage = "No data"
            return
        }

        do {
            let decoder = JSONDecoder()
            let flightsResponse = try decoder.decode(
            FlightsResponse.self, from: data)

            DispatchQueue.main.async {
                self.flights = flightsResponse.best_flights ??
                flightsResponse.other_flights ?? []
                if self.flights.isEmpty {
                    self.errorMessage = "No flights found"
                }
            }
        } catch {
            DispatchQueue.main.async {
                self.errorMessage = "Decoding error:
                \(error.localizedDescription)"
            }
        }
    }
    task.resume()
}
```

Il metodo appena presentato ha lo scopo di cercare i voli grazie all'API di Google Flights. I parametri passati alla richiesta sono: i codici IATA degli aeroporti di partenza e arrivo, le date di partenza e arrivo, la valuta e la lingua della ricerca. Per rendere migliore l'esperienza di utilizzo dell'applicazione, è presente un file JSON contenente i principali aeroporti del mondo con nome e codice da inserire nella ricerca. Tale file viene decodificato come segue:

```
func loadAirports() {
    if let url = Bundle.main.url(forResource: "airports",
    withExtension: "json") {
        do {
            let data = try Data(contentsOf: url)
            let decoder = JSONDecoder()
            let allAirports = try decoder
            .decode([AeroportoJSON].self, from: data)
            airports = allAirports.filter { $0.name != nil }
        } catch {
            print("Error decoding JSON: \(error)")
        }
    } else {
        print("airports.json file not found in the bundle.")
    }
}
```

Una volta eseguita la richiesta, il risultato viene mostrato a schermo.

Gestione della prenotazione di un volo

Dopo aver trovato il volo perfetto per le esigenze dell'utente, quest'ultimo può procedere con la prenotazione. La procedura inizia non appena viene premuto il bottone "Prenota":

```
func bookFlight(flight: Flight) {
    let firstFlight = flight.flights.first!
    let prenotazione = PrenotazioneVolo(
        id: UUID().uuidString,
        userId: Auth.auth().currentUser!.uid,
        departureAirport: firstFlight.departure_airport.name,
        arrivalAirport: firstFlight.arrival_airport.name,
        departureTime: firstFlight.departure_airport.time,
        arrivalTime: firstFlight.arrival_airport.time,
        airline: firstFlight.airline,
        flightNumber: firstFlight.flight_number,
        price: flight.price,
        bookingDate: Date()
    )

    prenotazioneController.bookFlight(flight: prenotazione)
    print("Prenotazione effettuata per il volo
    \(prenotazione.flightNumber)")
}
```

Qui viene creato l'oggetto PrenotazioneVolo, che viene passato al controller:

```
func bookFlight(flight: PrenotazioneVolo) {
    let reservationData: [String: Any] = [
        "id": flight.id as Any,
        "userId": flight.userId,
        "departureAirport": flight.departureAirport,
        "arrivalAirport": flight.arrivalAirport,
        "departureTime": flight.departureTime,
        "arrivalTime": flight.arrivalTime,
        "airline": flight.airline,
        "flightNumber": flight.flightNumber,
        "price": flight.price,
        "bookingDate": Timestamp(date: flight.bookingDate)
    ]

    db.collection("prenotazioni").document(flight.id!)
    .setData(reservationData) { error in
        if let error = error {
            print("Errore nel salvataggio della prenotazione:
            \(error)")
        } else {
            print("Prenotazione salvata con successo!")
        }
    }
}
```

Viene, ora, creata la prenotazione effettiva e caricata nel database.

Accedendo alla propria area riservata, l'utente ha la possibilità di visualizzare tutte le prenotazioni da lui effettuate:

```
func fetchPrenotazioniVolo(forUserId userId: String) {
    db.collection("prenotazioni").whereField("userId",
    isEqualTo: userId).getDocuments { (querySnapshot, error) in
        if let error = error {
            print("Errore nella fetch delle prenotazioni: \(error)")
            self.errorMessage = "Errore nel caricamento delle
            prenotazioni"
            return
        }
```

```
        guard let documents = querySnapshot?.documents else {
            self.errorMessage = "Nessuna prenotazione trovata"
            print("Nessuna prenotazione trovata")
            return
        }

        print("Documenti trovati: \(documents.count)")
        self.prenotazioniVolo = documents.compactMap
        { queryDocumentSnapshot -> PrenotazioneVolo? in
            do {
                let prenotazione = try queryDocumentSnapshot
                .data(as: PrenotazioneVolo.self)
                return prenotazione
            } catch {
                print("Errore nella conversione del documento:
                \(error)")
                return nil
            }
        }

        if self.prenotazioniVolo.isEmpty {
            self.errorMessage = "Nessuna prenotazione trovata"
        }
    }
}
```

Il metodo esegue una query all'interno del database, e va a recuperare tutti i documenti contenenti le prenotazioni effettuate dall'utente.

È sempre possibile, inoltre, eliminare una prenotazione:

```
func eliminaPrenotazioneAereo(_ prenotazione: PrenotazioneVolo) {
    guard let prenotazioneId = prenotazione.id else { return }

    db.collection("prenotazioni").document(prenotazioneId)
    .delete() { error in
        if let error = error {
            print("Errore durante l'eliminazione della
            prenotazione: \(error.localizedDescription)")
            self.errorMessage = "Errore durante l'eliminazione"
        } else {
            print("Prenotazione eliminata con successo")
            self.prenotazioniVolo.removeAll
            { $0.id == prenotazioneId }
        }
    }
}
```

Il metodo ha lo scopo di eliminare il file della prenotazione dal database. Il funzionamento per gli altri mezzi di trasporto e degli hotel è analogo.

3.12.2 Prenotazione dei treni e dei bus

La ricerca di treni e bus disponibili per una tratta è gestita internamente all'applicazione, quindi il meccanismo è molto simile. Per praticità verrà mostrata l'implementazione per i treni.

Ricerca di un treno

Nell'applicazione è presente un file JSON contenente i dati di alcune tratte di treni con i seguenti dati: stazione di partenza e arrivo, date e orari di partenza e arrivo, nome e numero del treno e prezzo:

```
func caricaTreniDaFile() {
    guard let fileURL = Bundle.main.url(forResource: "treni",
    withExtension: "json") else {
        print("File JSON non trovato")
        return
    }

    do {
        let data = try Data(contentsOf: fileURL)
        let decoder = JSONDecoder()
        var treni = try decoder.decode([Treno].self, from: data)

        // Assegna un id se è nullo
        treni = treni.map { treno in
            var trenoConID = treno
            trenoConID.id = treno.id ?? UUID().uuidString
            return trenoConID
        }

        DispatchQueue.main.async {
            self.treni = treni
        }
    } catch {
        print("Errore nella lettura del file JSON:
        \(error.localizedDescription)")
        self.errorMessage = "Errore nel caricamento dei treni"
    }
}
```

La ricerca inizia dall'estrazione dei dati relativi ai treni all'interno del file JSON. Successivamente, in base alle esigenze dell'utente, essi vengono filtrati:

```
func caricaTreniFiltrati(stazioneDiPartenza: String,
stazioneDiArrivo: String, dataDiPartenza: String,
dataDiRitorno: String) {
    // Carica tutti i treni
    caricaTreniDaFile()

    DispatchQueue.main.async {
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "yyyy-MM-dd"
        dateFormatter.timeZone = TimeZone(secondsFromGMT: 0)

        let userDateFormatter = DateFormatter()
        userDateFormatter.dateFormat = "yyyy-MM-dd"
        userDateFormatter.timeZone = TimeZone(secondsFromGMT: 0)

        // Parse selected dates
        guard let selectedDataDiPartenza = userDateFormatter
        .date(from: dataDiPartenza),
              let selectedDataDiRitorno = userDateFormatter
              .date(from: dataDiRitorno) else {
            print("Errore nella conversione delle date selezionate")
            return
        }

        // For debugging
        let debugDateFormatter = DateFormatter()
        debugDateFormatter.dateFormat = "yyyy-MM-dd HH:mm:ss Z"
        debugDateFormatter.timeZone = TimeZone.current

        let calendar = Calendar.current

        self.treniFiltrati = self.treni.filter { treno in
            let stazionePartenzaMatch = treno.stazioneDiPartenza
            .trimmingCharacters(in: .whitespaces)
            .lowercased() == stazioneDiPartenza
            .trimmingCharacters(in: .whitespaces).lowercased()
            let stazioneArrivoMatch = treno.stazioneDiArrivo
            .trimmingCharacters(in: .whitespaces)
```

```
            .lowercased() == stazioneDiArrivo
            .trimmingCharacters(in: .whitespaces).lowercased()

            if let dataDiPartenzaTreno = dateFormatter
            .date(from: treno.dataDiPartenza),
               let dataDiArrivoTreno = dateFormatter
               .date(from: treno.dataDiArrivo) {

                // Normalize dates to start of the day
                let startOfDayDataDiPartenzaTreno =
                calendar.startOfDay(for: dataDiPartenzaTreno)
                let startOfDayDataDiArrivoTreno =
                calendar.startOfDay(for: dataDiArrivoTreno)
                let startOfDaySelectedDataDiPartenza =
                calendar.startOfDay(for: selectedDataDiPartenza)
                let startOfDaySelectedDataDiRitorno =
                calendar.startOfDay(for: selectedDataDiRitorno)

                // Compare dates
                let dataPartenzaMatch = calendar.
                isDate(startOfDayDataDiPartenzaTreno,
                inSameDayAs: startOfDaySelectedDataDiPartenza)
                let dataArrivoMatch = calendar.
                isDate(startOfDayDataDiArrivoTreno,
                inSameDayAs: startOfDaySelectedDataDiRitorno)

                return stazionePartenzaMatch &&
                stazioneArrivoMatch &&
                dataPartenzaMatch &&
                dataArrivoMatch
            }

            return false
        }

        if self.treniFiltrati.isEmpty {
            self.errorMessage = "Nessun treno trovato per
            i criteri selezionati."
        } else {
            self.errorMessage = nil
        }
    }
}
```

Il metodo appena mostrato ha l'obiettivo di filtrare i treni presenti nel file in base alla ricerca dell'utente. Le date passate come parametro sono convertite secondo il formato "yyyy-MM-dd" e in base al fuso orario UTC. Viene, ora, eseguito un parsing delle date da stringa a tipo Date. Infine avviene il filtro effettivo, in cui si verifica la corrispondenza delle stazioni e delle date inserite dall'utente.

3.12.3 Prenotazione degli hotel

Viene, ora, mostrata la fase di ricerca degli hotel:

```
func cercaHotel(location: String, checkInDate: Date,
checkOutDate: Date, adults: Int, currency: String,
language: String) {
    let apiKey = "9aee355d056ba268028490602548-
    77fb8ab08b2856dba38c5ebf791487ba7a94"

    let formattedCheckInDate = formatDate(checkInDate)
    let formattedCheckOutDate = formatDate(checkOutDate)

    let urlString = "https://serpapi.com/search.json?
    engine=google_hotels&q=
    \(location)&vacation_rentals=true&check_in_date=
    \(checkInDate)&check_out_date=\(checkOutDate)&adults=
    \(adults)&currency=\(currency)&hl=
```

```
    \(language)&api_key=\(apiKey)"

    guard let url = URL(string: urlString) else {
        print("URL non valido")
        return
    }

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

    let task = URLSession.shared.dataTask(with: request)
    { data, response, error in
        if let error = error {
            print("Errore: \(error.localizedDescription)")
            return
        }

        guard let data = data else {
            print("No data")
            return
        }
        if let jsonString = String(data: data, encoding: .utf8) {
               print("JSON Response: \(jsonString)")
           }

        do {
            let decoder = JSONDecoder()
            let hotelResponse = try decoder
            .decode(HotelResponse.self, from: data)

            DispatchQueue.main.async {
                self.hotelResults = hotelResponse.properties
            }
        } catch {
            DispatchQueue.main.async {
                print("Decoding error: \(error)")
            }
        }
    }
    task.resume()
}
```

Il metodo fa una chiamata all'API Google Hotels, passando come parametri: la città, le date di check-in e check-out, il numero di adulti, la valuta e la lingua della ricerca. Il file JSON restituito viene decodificato e mostrato all'utente, gestendo gli eventuali errori.

CAPITOLO 4

Test, funzionamento e discussione

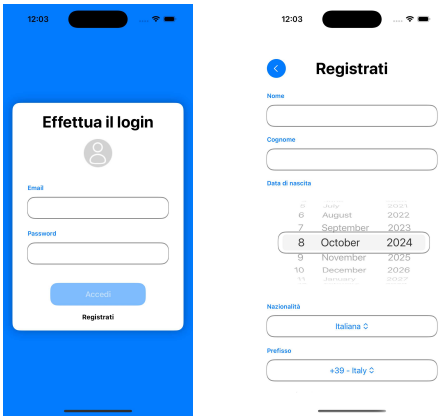
In questo capitolo verrà descritta l'applicazione ultimata nel suo sviluppo, risaltandone le caratteristiche. Saranno esaminate nel dettaglio le funzionalità di ricerca e prenotazione di servizi di alloggio e di trasporto e la gestione dei contenuti multimediali. Lo scopo, quindi, è quello di offrire una visione chiara e approfondita delle potenzialità offerte dall'applicazione.

4.1 Autenticazione e accesso

L'avvio dell'applicazione porta gli utenti alla schermata di autenticazione. Qui si richiede di accedere tramite e-mail e password, o di creare un nuovo account.

Registrazione e login
La fase di registrazione è il primo passo per accedere alle funzionalità dell'applicazione. Il meccanismo è stato progettato per essere semplice e sicuro, assicurando che i dati personali degli utenti siano protetti. Questi sono guidati attraverso una schermata in cui inserire informazioni, come nome, indirizzo email e password. Se i dati inseriti non sono corretti, il sistema informa in tempo reale gli utenti attraverso chiari messaggi di avviso.

Se si possiede già un account, è possibile accedere tramite indirizzo e-mail e password. Una volta inserite le credenziali, si è immediatamente reindirizzati alla home dell'app.



(a) Schermata di login

(b) Schermata registrazione

Figura 4.1: Sezione autenticazione

4.2 Home e navigazione

Nella home risiede il cuore dell'applicazione, infatti da qui è agevole accedere alle principali funzionalità offerte.

Nella parte alta della schermata è posizionata una barra di ricerca dove è possibile digitare il nome della città da visitare. Al centro, poi, attraverso dei bottoni, è possibile visualizzare le mete popolari e le mete recenti. Le prime prime permettono agli utenti di conoscere quali sono le città più cercate; le seconde, invece, rappresentano una vera e propria cronologia di ricerca. In fondo, infine, si trova un menù, con cui si può accedere facilmente alle schermate della galleria, dei preferiti e del profilo dell'utente.

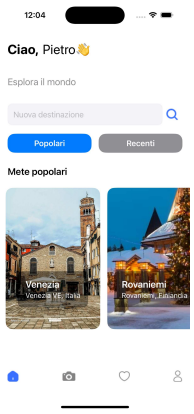


Figura 4.2: Home

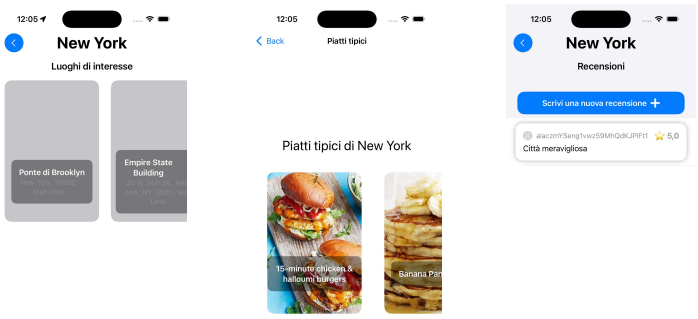
4.3 Ricerca delle informazioni di una città

La funzionalità di ricerca non solo offre la possibilità di scoprire nuove destinazioni per le prossime avventure, ma anche di arricchire l'esperienza culturale del viaggiatore grazie alla possibilità di cercare le attrazioni turistiche e i piatti tipici del luogo. L'utente, infatti, ha la possibilità di ottenere tutte le informazioni desiderate attraverso un menù semplice ed intuitivo.

Luoghi di interesse
Una volta selezionata la città, l'utente può esplorare quali sono le principali attrazioni turistiche, come musei e monumenti. Ogni risultato è accompagnato da una foto, una breve descrizione e la valutazione presente su Google e sull'applicazione. Questa funzionalità consente di pianificare itinerari personalizzati, in base alle proprie esigenze.

Piatti Tipici
Per vivere appieno l'avventura, l'applicazione propone all'utente una serie di piatti tipici della città selezionata. Ognuno di essi è accompagnato da una foto. Inoltre, possono essere visualizzati in dettaglio gli ingredienti e le valutazioni degli altri utenti.

Recensioni
Per avere ulteriori informazioni, l'utente può consultare le recensioni degli altri viaggiatori. Esse sono disponibili per città, luoghi di interesse e piatti tipici, e si compongono di un breve testo e una valutazione da una a cinque stelle. Ogni utente, dopo un'avventura, può pubblicarne una, non solo per aiutare gli altri viaggiatori, ma anche per migliorare l'accuratezza delle informazioni presenti nell'applicazione.



(a) Luoghi di interesse **(b)** Piatti tipici **(c)** Recensioni

Figura 4.3: Informazioni di una città

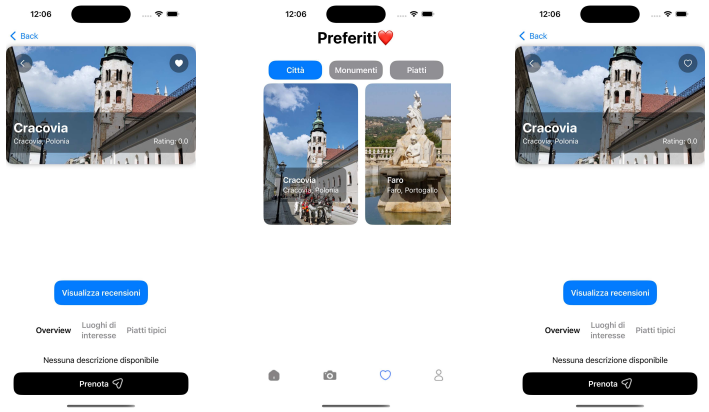
4.4 Gestione dei preferiti

La funzione di gestione dei preferiti è stata progettata per rendere più semplice e veloce la pianificazione di un viaggio. Essa è disponibile per città, attrazioni turistiche e piatti tipici.

Aggiunta di un preferito
Durante la navigazione, l'utente può aggiungere facilmente una destinazione o un servizio alla lista dei preferiti, cliccando sull'apposita icona a forma di cuore. Grazie a ciò, è possibile creare una vera e propria lista dei desideri o un itinerario dei luoghi da visitare.

Visualizzazione dei preferiti
La visualizzazione dei preferiti risulta agevole, poiché la sezione ad essi dedicata è raggiungibile direttamente dalla home. Qui quest'ultimi sono suddivisi in base alla loro categoria.

Rimozione di un preferito
Se, infine, una destinazione o un servizio non è più utile, gli utenti possono facilmente rimuoverlo dalla lista dei preferiti, cliccando sull'apposita icona. Questa funzionalità offre flessibilità nel mantenere aggiornata la lista dei luoghi da visitare, consentendo agli utenti di adattarla alle proprie esigenze di viaggio.



(a) Aggiunta di un preferito

(b) Preferiti

(c) Rimozione di un preferito

Figura 4.4: Gestione dei preferiti

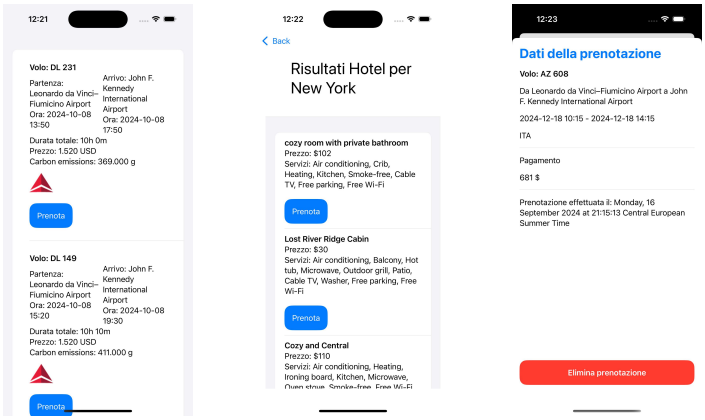
4.5 Prenotazione di viaggi

La funzionalità di prenotazione di viaggi è stata progettata, per offrire agli utenti un'esperienza fluida e intuitiva nella pianificazione e prenotazione di voli, treni, autobus e hotel. Grazie ad essa, infatti, si possono confrontare le varie soluzioni e scegliere quella più adatta.

Ricerca di un mezzo di trasporto
Il processo di prenotazione inizia con una ricerca personalizzata. Gli utenti, infatti, possono specificare la città o stazione di partenza e arrivo, le date del viaggio e il numero di passeggeri. L'applicazione è integrata con i servizi delle compagnie aeree, ferroviarie e di autobus, e offre al viaggiatore la lista delle migliori opzioni disponibili. Questa comprende la compagnia di viaggio, gli orari e date di partenza e arrivo e il prezzo.

Ricerca di un hotel
La ricerca di un hotel è molto simile a quella di un mezzo di trasporto. Infatti, basta inserire la città, le date in cui effettuare check-in e check-out e il numero di adulti, e, grazie all'integrazione con il servizio di Google, saranno mostrati i risultati migliori. Tra le informazioni disponibili si trovano: il nome dell'hotel, il prezzo e i servizi offerti.

Prenotazione di un mezzo di trasporto o hotel
Una volta trovata la soluzione che più si addice all'utente, esso può prenotarlo, e visualizzare la sua prenotazione direttamente nella sua area riservata. Essa comprende tutti i dati necessari per il viaggio, come il mezzo di trasporto scelto, date e orari, il prezzo pagato e la data in cui è stata effettuata la prenotazione. Nel caso di un alloggio i dati sono analoghi.



(a) Ricerca di un volo

(b) Ricerca di un hotel

(c) Prenotazione

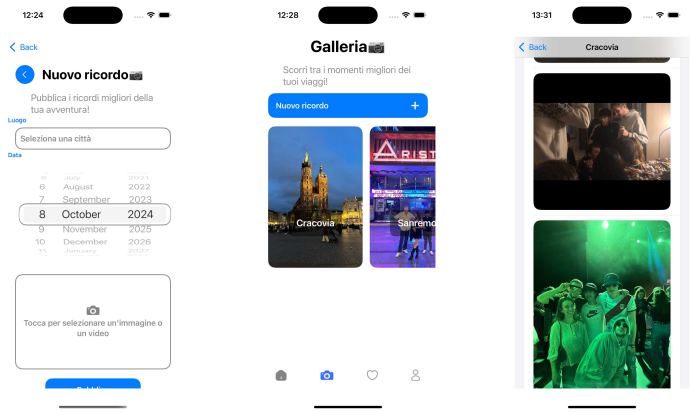
Figura 4.5: Prenotazione di un viaggio

4.6 Gestione dei ricordi di un viaggio

Una delle emozioni più grandi è rivivere le esperienze passate attraverso i ricordi. Per questo motivo è stata progettata una funzionalità che permette agli utenti di caricare foto e video delle proprie avventure.

Upload di un ricordo
Il meccanismo che permette agli utenti di inserire un ricordo è molto agevole. Esso prevede la selezione di una città, la data in cui è stata immortalata la memoria e il file da caricare. Una volta concessi i permessi per accedere alla galleria, l'utente può selezionare la foto o il video che desidera caricare.

Visualizzazione della galleria
La galleria interna all'applicazione si mostra ordinata ed elegante. Essa, infatti, raggruppa tutti i ricordi degli utenti in base alla città. Una volta scelto l'album, è possibile scorrere tra tutte le memorie contenute in esso, e rivivere anche a distanza di anni le emozioni che avevano reso unica quell'avventura.

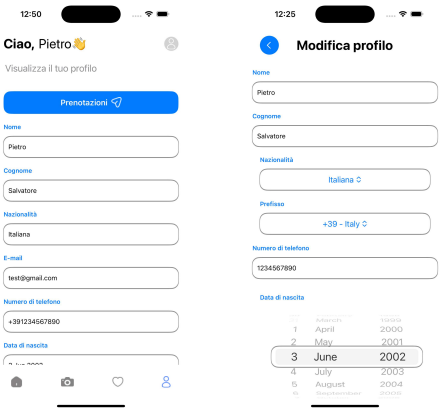


(a) Nuovo ricordo **(b)** Galleria **(c)** Ricordi di una città

Figura 4.6: Gestione dei ricordi di un viaggio

4.7 Gestione dell'area riservata

L'area riservata dell'utente rappresenta uno spazio sicuro all'interno dell'applicazione raggiungibile semplicemente dalla home del sito. Qui ogni utente può gestire i propri dati, aggiornando le informazioni personali e i dati sensibili, e le prenotazioni, visualizzandone i dettagli o annullandole.



(a) Profilo di un utente **(b)** Modifica profilo

Figura 4.7: Area riservata

Conclusioni

Nel corso di questa tesi è stata mostrata l'idea e il successivo sviluppo di un'applicazione iOS. Inizialmente, è stato introdotto il problema che l'applicazione proposta mira a risolvere e la descrizione in linguaggio naturale delle sue funzionalità. Segue la fase di analisi, in cui si evidenziano: i requisiti, suddivisi tra funzionali e non funzionali; i casi d'uso, con lo scopo di evidenziare in che modo tali funzionalità verranno implementate, e il diagramma delle classi di analisi, per evidenziare come le sezioni dell'applicazione interagiscano tra di loro. Successivamente, è stata presentata la fase di progettazione. Qui è stato specificato che l'architettura utilizzata per lo sviluppo dell'applicazione è la Model-View-Controller. Inizialmente è stato mostrato il diagramma dei Models, dove si evidenziano le relazioni tra le classi che costituiscono l'applicazione; segue il diagramma dei Controllers, in cui si precisano le azioni compiute per la realizzazione dei casi d'uso; e infine è stato presentato il diagramma delle Views, con lo scopo di mostrare come esse sono collegate tra di loro. A seguire i mockup, per esporre come l'applicazione appare visivamente.

Il capitolo successivo è stato interamente dedicato all'implementazione dell'applicazione. In primis, sono state esposte le tecnologie utilizzate nel progetto, come il linguaggio di programmazione e i servizi esterni; in secondo luogo la spiegazione di come ogni sezione è stata realizzata, accompagnata da blocchi di codice.

L'ultimo capitolo della tesi si occupa di accompagnare un futuro utente in un tour completo tra le funzionalità offerte, evidenziandone la praticità e semplicità di utilizzo.

Lo sviluppo dell'applicazione ha rappresentato un percorso complesso, ma che ha portato a risultati altamente soddisfacenti. Durante le varie fasi di progettazione ed implementazione, sono stati conseguiti numerosi obiettivi, che hanno permesso alla piattaforma di diventare matura e funzionale. Particolare attenzione è stata posta nella realizzazione di interfacce grafiche intuitive e nella gestione sicura dei dati sensibili degli utenti.

Sebbene l'applicazione abbia raggiunto un livello di funzionalità avanzato, sono presenti diverse aree che saranno migliorate nei futuri aggiornamenti. Sono, infatti, elencate ora alcune idee per futuri aggiornamenti:

- *Planner dell'itinerario*, ovvero l'organizzazione di attività giornaliere e dettagli di trasporto. Esso sarà sincronizzato al calendario.
- *Integrazione social*, ovvero la condivisione di itinerario, consigli, foto e video, qualora un utente pianificasse un viaggio con qualcuno.
- *Assistente linguistico*, ossia un traduttore in tempo reale, qualora l'utente viaggi in una nazione dove è parlata una lingua a lui sconosciuta.

- *Accesso offline*, ovvero la possibilità di scaricare le informazioni essenziali sul viaggio, come mappe e itinerari. Infatti un utente, viaggiando all'estero, potrebbe avere roaming dati limitato o, addirittura, non averlo.

Con grande riguardo saranno ascoltati i feedback degli utenti per l'implementazione di ulteriori funzionalità.

In conclusione, lo sviluppo dell'applicazione ha portato alla creazione di un prodotto versatile e competitivo, che pone le basi per miglioramenti futuri e ampliamenti delle funzionalità.

Bibliografia

APPLE (2022), «The Swift Programming Language (Swift 5.7)», *Apple Inc.*

Siti Web consultati

- Draw.io – `https://www.drawio.com`
- Figma – `https://www.figma.com`
- Google Flights API – `https://serpapi.com/google-flights-api`
- Google Hotels API – `https://serpapi.com/google-hotels-api`
- HTML – `https://www.html.it/pag/54084/introduzione-al-linguaggio-swift/`
- Medium – `https://medium.com/@veeranjain04/understanding-the-basics-of-swiftui-a-beginners-guide-69f263786d8f`
- Place Autocomplete – `https://developers.google.com/maps/documentation/places/ios-sdk/place-autocomplete?hl=it`
- Place Details – `https://developers.google.com/maps/documentation/places/ios-sdk/details-place?hl=it`
- Place Photos – `https://developers.google.com/maps/documentation/places/ios-sdk/place-photos?hl=it`
- TheMealDB – `https://www.themealdb.com`

Ringraziamenti

I miei primi ringraziamenti sono rivolti al professor Enrico Corradini, che mi ha accompagnato durante la stesura di questa tesi, mostrandosi sempre disponibile e affidabile.

Ringrazio la mia bisononna e mia zia, che con il loro supporto hanno reso possibile questo traguardo.

Ringrazio mia madre e la mia famiglia, che mi sono stati sempre vicini, nei bei momenti e in quelli più difficili. Ringrazio in particolare mio zio Vittorio, che mi ha supportato ed aiutato per la realizzazione di questo lavoro.

Un pensiero speciale va a tutti i miei amici, che hanno reso indimenticabili questi anni, diventando una vera e propria famiglia.

Infine, non per importanza, ma seguendo un ordine cronologico, ringrazio i miei compagni dell'Erasmus. La loro presenza e i momenti di condivisione sono stati fondamentali per la mia crescita.

Vi porterò sempre nel cuore.