



UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

DIPARTIMENTO DI INGEGNERIA ELETTRICA
E DELLE TECNOLOGIE DELL'INFORMAZIONE

Specifiche, progettazione,
implementazione e validazione del
Sistema Informativo
NaTour21

Studenti

Bianca Giada CHEHADE N86003209
Mario LIGUORI N86003258
Mattia ROSSI N86003211

Docenti

Sergio DI MARTINO
Francesco CUTUGNO



Indice

1 Documento dei Requisiti Software	3
1.1 Descrizione del progetto	3
1.2 Presentazione dell'idea progettuale	3
1.3 Individuazione del target degli utenti	4
1.4 Requisiti funzionali	5
1.4.1 Autenticazione	5
1.4.2 Interazione con un itinerario	6
1.4.3 Interazione con un post	7
1.4.4 Gestione dati personali e conversazioni private	8
1.4.5 Interazione con una compilation	8
1.4.6 Requisiti amministratore	9
1.5 Requisiti non funzionali	10
1.6 Requisiti di dominio	12
1.7 Modellazione dei Casi d'Uso	13
1.7.1 Autenticazione	14
1.7.2 Interazione con un itinerario	15
1.7.3 Interazione con un post	16
1.7.4 Gestione profilo	17
1.7.5 Gestione messaggi	18
1.7.6 Funzionalità amministratore	19
1.8 Tabelle di Cockburn dei casi d'uso	20
1.8.1 Inserisce un itinerario	20
1.8.2 Segnala un itinerario	23
1.9 UX Design	24
1.9.1 Definizione delle Personas	24
1.9.2 Prototipazione dell'esperienza utente	27
1.9.3 Definizione di uno stile di design	29
1.10 Mock-Up dell'applicazione	30
1.10.1 Inserimento itinerario	30
1.10.2 Segnalazione itinerario	33
1.11 Valutazione dell'usabilità a priori	34
1.11.1 Definizione dei task utente	34
1.11.2 Risultati ottenuti dallo svolgimento dei task	35
1.12 Prototipazione funzionale via statechart dell'interfaccia grafica	36
1.12.1 Segnalazione itinerario	36
1.12.2 Inserimento itinerario	37
1.13 Classi, oggetti e relazioni di analisi	41
1.13.1 Autenticazione	41
1.13.2 Interazione con un itinerario	45
1.13.3 Interazione con un post	51
1.13.4 Interazione con una compilation	55
1.13.5 Gestione profilo e interazione con gli utenti	59
1.13.6 Funzionalità riservate agli amministratori	62
1.14 Diagrammi di sequenza di analisi	64
1.14.1 Segnalazione itinerario	64

1.14.2 Ricerca itinerario	65
1.15 Diagrammi di attività	66
1.15.1 Autenticazione	66
1.15.2 Interazione con un itinerario	70
1.15.3 Interazione con un post	76
1.15.4 Interazione con una compilation	79
1.15.5 Gestione profilo e interazione con utenti	82
1.15.6 Funzionalità riservate agli amministratori	86
2 Documento di Design del sistema	88
2.1 Analisi dell'architettura e criteri di design	88
2.1.1 Diagramma di design del sistema	88
2.1.2 Software deployment	89
2.1.3 Google Maps Platform	90
2.1.4 Architettura del REST Service	90
2.2 Architettura dell'applicativo	93
2.3 Le scelte implementative	96
2.3.1 Applicazioni mobile native e ibride	96
2.3.2 Perchè Kotlin?	96
2.3.3 Diagramma di design dell'applicativo	98
2.3.4 Pattern architetturale utilizzato	99
2.4 Diagramma delle classi di design	100
2.4.1 Autenticazione	100
2.4.2 Interazione con un itinerario	106
2.4.3 Interazione con un post	113
2.4.4 Interazione con una compilation	118
2.4.5 Interazione con altri utenti	123
2.4.6 Gestione profilo	126
2.4.7 Amministratore	129
2.5 Diagrammi di sequenza di design	132
2.6 Definizione delle gerarchie funzionali	134
3 Definizione di un piano di testing e valutazione sul campo dell'usabilità.	135
3.1 Codice xUnit per testing di 3 metodi	135
3.1.1 Metodo <i>formValidator</i>	135
3.1.2 Metodo <i>getChatMyMembersUseCase</i>	139
3.1.3 Metodo <i>retrofitSafeCall</i>	144
3.2 Valutazione dell'usabilità sul campo	149

1 Documento dei Requisiti Software

1.1 Descrizione del progetto

NaTour21 è un sistema complesso e distribuito finalizzato ad offrire un moderno social network multipiattaforma per appassionati di escursioni.

Il sistema consiste in:

- un Back-End sicuro, performante e scalabile;
- un client mobile attraverso cui gli utenti possono fruire delle funzionalità del sistema in modo intuitivo, rapido e piacevole;
- un client mobile attraverso cui gli amministratori possono gestire i contenuti inseriti in piattaforma.

NaTour21 ha lo scopo di creare una community sicura e affidabile dove condividere la propria passione per l'escursionismo.

In questo scenario, l'utente si configura come protagonista: oltre alla possibilità di inserire itinerari (dettagliati da informazioni), compilation e post personali sulla piattaforma, è lasciato ampio spazio all'individualità personale.

Tutto ciò si concretizza con la possibilità di interagire con gli altri utenti, in modo da poter avere un contatto più diretto con la realtà dell'escursionismo, e di lasciare valutazioni personali su qualunque itinerario si desideri.

Il sistema valuta essenziale la sicurezza degli utenti: questi potranno segnalare informazioni inesatte o contenuti inappropriati al fine di rendere la permanenza nella community piacevole per tutti.

1.2 Presentazione dell'idea progettuale

L'applicazione **NaTour21** nasce in seguito all'esigenza di uno spazio virtuale dove poter condividere la passione per l'escursionismo.

Creare una community di escursionisti, esperti o meno, ha lo scopo di rendere le esperienze individuali degli utenti più sicure e informate, oltre a promuovere la condivisione di contenuti personali.

NaTour21 mette a disposizione degli utenti registrati diverse funzionalità.

Il lato prettamente informativo consente di ricercare - nonchè inserire in piattaforma - diversi itinerari, e tutte le informazioni relative ad essi. Ciò comprende la possibilità di visualizzarli su mappa, e, per garantire un'accuratezza maggiore delle informazioni, di lasciare un feedback personale per ciascuno di essi.

Ci sono, però, ulteriori funzionalità che coinvolgono direttamente l'utente: tra queste la condivisione di post, la creazione di compilation personalizzate e la visualizzazione di

una homepage con post riguardanti diversi itinerari.

Particolare attenzione è riservata alla sicurezza degli utenti, i quali possono segnalare contenuti inappropriati o informazioni inesatte.

1.3 Individuazione del target degli utenti

NaTour21 si configura come community in cui chiunque può intraprendere la passione per l'escursionismo e condividerla.

Ciononostante, è stato possibile individuare un target di utenti ben definito grazie ai dati statistici¹ raccolti nell'anno 2020 dall'ISTAT, Istituto Nazionale di Statistica.

In seguito all'analisi delle informazioni statistiche sull'escursionismo italiano è stato individuato come target utenti la popolazione, caratterizzata da un numero di circa pari entità di uomini e di donne, nella fascia di età 35-64. La maggior parte della platea considerata risulta essere appartenente alla condizione lavorativa di *dirigente, quadro o impiegato*.



L'Istat ha inoltre conteggiato nell'anno 2020 circa 41194 escursioni giornaliere, di cui il 42.7% situate temporalmente nel terzo trimestre dell'anno. Da questa informazione si deduce che il periodo preferito per attività escursionistiche sia quello dei mesi estivi.

Il motivo prevalente per queste escursioni risulta essere stato *piacere personale, svago e tempo libero*, con un numero di escursioni così orientate pari a 23223: questo dato risulta particolarmente significativo, pari a un ordine di grandezza in più rispetto agli altri motivi registrati. È importante anche notare quanto il numero di escursioni in paesi esteri sia di gran lunga minore rispetto a quelle nelle regioni italiane.

¹Fonte: [sito ufficiale Istat](#).

Le regioni del Nord Italia sembrano essere state le preferite per visite giornaliere (la più visitata è stata il Veneto, con una media di 6200), seguite immediatamente da quelle del centro-Sud. Il Molise, invece, non risulta aver raggiunto la minima cifra considerata nella raccolta dati.

1.4 Requisiti funzionali

In questa sezione saranno esposti i requisiti *funzionali* dell'applicazione NaTour21, cioè le funzionalità offerte. Le funzionalità richieste sono presenti nella loro totalità; inoltre, è stato ritenuto opportuno inserire alcune funzionalità non esplicitamente richieste, ma ritenute dal team di sviluppo come desiderabili per qualsiasi applicativo di questo tipo.

1.4.1 Autenticazione

Nome	Descrizione
Registrazione	Il sistema deve consentire a un utente di potersi registrare indicando e-mail, username e password, oppure utilizzando account di terze parti (Google o Facebook).

Tabella 1: RF.1

Nome	Descrizione
Accesso	Il sistema deve consentire a un utente registrato di poter effettuare l'accesso alla piattaforma avendo eseguito precedentemente una registrazione.

Tabella 2: RF.2

Nome	Descrizione
Reimpostazione password	Il sistema deve consentire a un utente registrato di poter reimpostare la propria password se dimenticata o persa.

Tabella 3: RF.3

Nome	Descrizione
Uscita dall'account	Il sistema deve consentire a un utente autenticato di poter uscire dall'account con il quale ha eseguito l'accesso.

Tabella 4: RF.4

1.4.2 Interazione con un itinerario

Nome	Descrizione
Visualizzazione itinerari	Il sistema deve consentire a un utente autenticato di visualizzare i dettagli degli itinerari pubblicati (tra cui la sua posizione su mappa) e i post ad essi associati.

Tabella 5: RF.5

Nome	Descrizione
Inserimento itinerario	Il sistema deve consentire a un utente autenticato di inserire nuovi itinerari (sentieri) in piattaforma. Un sentiero è caratterizzato da un nome, una durata (max 16 ore), un livello di difficoltà, un punto di inizio e di fine, delle foto (max 5), una descrizione (opzionale), e un tracciato geografico (opzionale) che lo rappresenta su una mappa. Il tracciato geografico può essere inseribile manualmente (interagendo con una mappa interattiva) oppure tramite file in formato standard GPX.

Tabella 6: RF.6

Nome	Descrizione
Ricerca itinerari	Il sistema deve consentire a un utente autenticato di effettuare ricerche di itinerari tra quelli presenti in piattaforma, con possibilità di filtrare i risultati per area geografica (compresa in un raggio espresso in km scelto dall'utente), per livello di difficoltà, per durata e per accessibilità ai disabili.

Tabella 7: RF.7

Nome	Descrizione
Valutazione itinerario	Il sistema deve consentire a un utente autenticato di indicare un punteggio di difficoltà e/o un tempo di percorrenza diverso da quello indicato dall'utente che ha inserito il sentiero. In questo caso, il punteggio di difficoltà e il tempo di percorrenza per il sentiero saranno ri-calcolati come la media delle difficoltà e dei tempi indicati.

Tabella 8: RF.8

Nome	Descrizione
Salvataggio itinerario	Il sistema deve consentire a un utente autenticato di salvare gli itinerari nelle proprie compilation personalizzate.

Tabella 9: RF.9

Nome	Descrizione
Segnalazione itinerario	Il sistema deve consentire a un utente autenticato di segnalare degli itinerari le quali informazioni potrebbero non essere corrette e/o aggiornate.

Tabella 10: RF.10

Nome	Descrizione
Rimozione itinerario	Il sistema deve consentire a un utente autenticato di eliminare itinerari di cui è l'autore.

Tabella 11: RF.11

1.4.3 Interazione con un post

Nome	Descrizione
Visualizzazione post	Il sistema deve consentire a un utente autenticato di visualizzare i dettagli di un post.

Tabella 12: RF.12

Nome	Descrizione
Inserimento post	Il sistema deve consentire a un utente autenticato di inserire un post associato ad un itinerario, caratterizzato da foto (max 5) e una descrizione (opzionale).

Tabella 13: RF.13

Nome	Descrizione
Segnalazione post	Il sistema deve consentire a un utente autenticato di segnalare post con fotografie inappropriate.

Tabella 14: RF.14

Nome	Descrizione
Rimozione post	Il sistema deve consentire a un utente autenticato di eliminare post di cui è l'autore.

Tabella 15: RF.15

1.4.4 Gestione dati personali e conversazioni private

Nome	Descrizione
Gestione profilo	Il sistema deve consentire a un utente autenticato di gestire il suo profilo, quindi di visualizzare i contenuti inseriti ed eliminarli. Inoltre è possibile modificare la foto profilo.

Tabella 16: RF.16

Nome	Descrizione
Gestione conversazioni private	Il sistema deve consentire a un utente autenticato lo scambio di informazioni con altri utenti. È possibile avviare la conversazione a partire dai contenuti pubblicati. Inoltre è possibile ricercare un destinatario.

Tabella 17: RF.17

1.4.5 Interazione con una compilation

Nome	Descrizione
Creazione compilation	Il sistema deve consentire a un utente autenticato di creare delle <i>compilation personalizzate</i> , caratterizzate da un titolo, una foto e da una descrizione.

Tabella 18: RF.18

Nome	Descrizione
Rimozione compilation	Il sistema deve consentire a un utente autenticato di eliminare le proprie compilation personalizzate.

Tabella 19: RF.19

Nome	Descrizione
Visualizzazione compilation	Il sistema deve consentire a un utente autenticato di visualizzare i dettagli delle proprie compilation.

Tabella 20: RF.20

Nome	Descrizione
Rimozione itinerario da compilation	Il sistema deve consentire a un utente autenticato di rimuovere gli itinerari dalle proprie compilation personalizzate.

Tabella 21: RF.21

1.4.6 Requisiti amministratore

Il seguente insieme di funzionalità è ridotto ad un gruppo ristretto di utenti: gli amministratori.²

Nome	Descrizione
Visualizzazione segnalazioni	Il sistema deve consentire a un utente autenticato con i privilegi di amministratore di visualizzare la lista di segnalazioni inviate.

Tabella 22: RF.22

Nome	Descrizione
Amministrazione itinerario	Il sistema deve consentire a un utente autenticato con i privilegi di amministratore di modificare o rimuovere un itinerario segnalato.

Tabella 23: RF.23

²È importante sottolineare che un amministratore è un utente semplice con particolari privilegi: un amministratore potrà, dunque, usufruire di tutte le funzionalità riservate agli utenti. Non vale, però, il viceversa: gli utenti "semplici" non avranno alcun potere sui contenuti pubblicati da altri utenti, se non la possibilità di segnalarli agli amministratori.

1.5 Requisiti non funzionali

Questa sezione conterrà invece i requisiti *non funzionali* dell'applicazione, ovvero tutti i vincoli sui servizi offerti dal sistema.

Nome	Descrizione
Back-End scalabile	Il sistema deve essere scalabile e performante rispetto ai cambiamenti del Back-End, in modo da garantire un'agile manutenibilità nel tempo.

Tabella 24: RNF1

Nome	Descrizione
Prestazioni	Il sistema deve essere operativo entro 3 secondi dall'avvio e, inoltre, non deve ostacolare in alcun modo gli input dell'utente (e.g. presentando dei rallentamenti).

Tabella 25: RNF2

Nome	Descrizione
Reattività	Il sistema deve essere responsivo, sia nei confronti degli input da parte dell'utente, sia in caso di interruzioni esterne con alta priorità (e.g. chiamate in arrivo). Considerando il caso delle interruzioni: il sistema deve essere in grado di salvare il proprio stato e ripresentarlo all'utente.

Tabella 26: RNF3

Nome	Descrizione
Usabilità	L'applicazione deve presentare una UX semplice, che renda il flusso delle operazioni comprensibile anche ad utenti meno esperti.

Tabella 27: RNF4

Nome	Descrizione
Sicurezza	Il sistema deve essere sicuro, con dati criptati e protetti da attacchi esterni. Nel caso dell'autenticazione per i client, verrà salvato un token (nello specifico JWT) sul dispositivo.

Tabella 28: RNF5

Nome	Descrizione
Sicurezza password	Il sistema non deve consentire di inserire password che non contengano almeno 8 caratteri.

Tabella 29: RNF6

Nome	Descrizione
Numero massimo di foto	Il sistema non deve consentire di inserire più di 5 foto per itinerario o post.

Tabella 30: RNF7

1.6 Requisiti di dominio

Infine, questa sezione conterrà i requisiti *di dominio* dell'applicazione, ovvero tutti i vincoli generali a cui l'applicativo deve attenersi.

Nome	Descrizione
ISO/IEC 27018:2019	Il sistema deve essere conforme allo standard ISO/IEC 27018:2019, incentrato sulla protezione dei dati personali nel cloud.

Tabella 31: RD1

Nome	Descrizione
GDPR	Il sistema deve essere conforme al GDPR (Regolamento Generale sulla Protezione dei Dati), incentrato sul trattamento dei dati personali e sulla privacy dell'utente.

Tabella 32: RD2

1.7 Modellazione dei Casi d'Uso

In questa sezione è riportato il diagramma Use Case per l'applicazione.

Per garantire una maggiore leggibilità e data la molteplicità di funzioni garantite dal sistema, si è fatta la scelta di raggruppare le funzioni logicamente legate tra loro in packages.

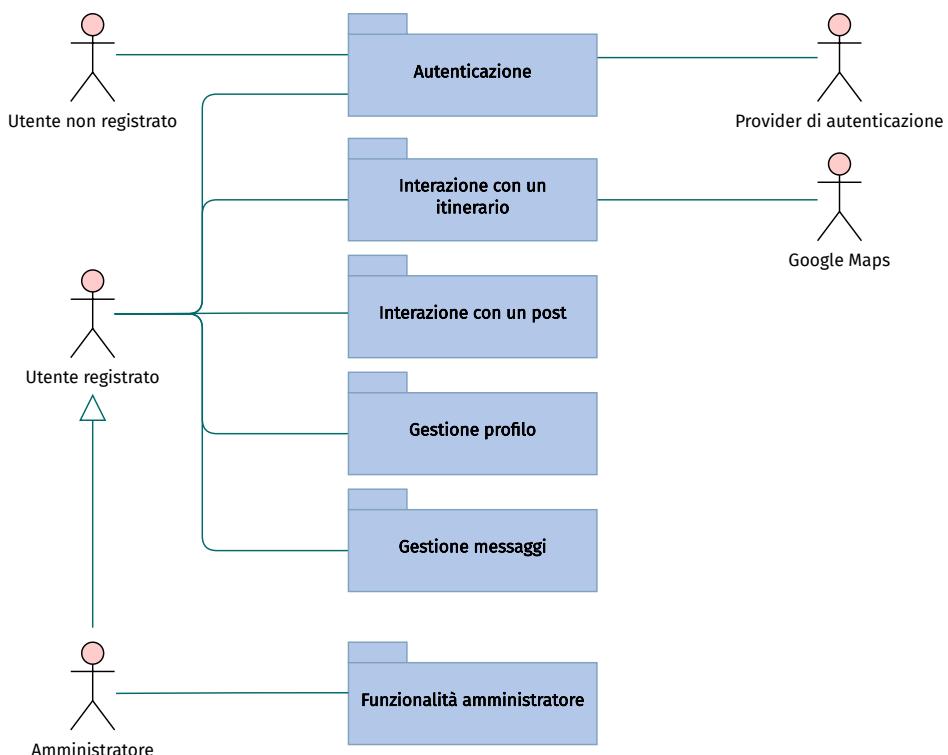


Figura 1: Use Case Diagram

1.7.1 Autenticazione

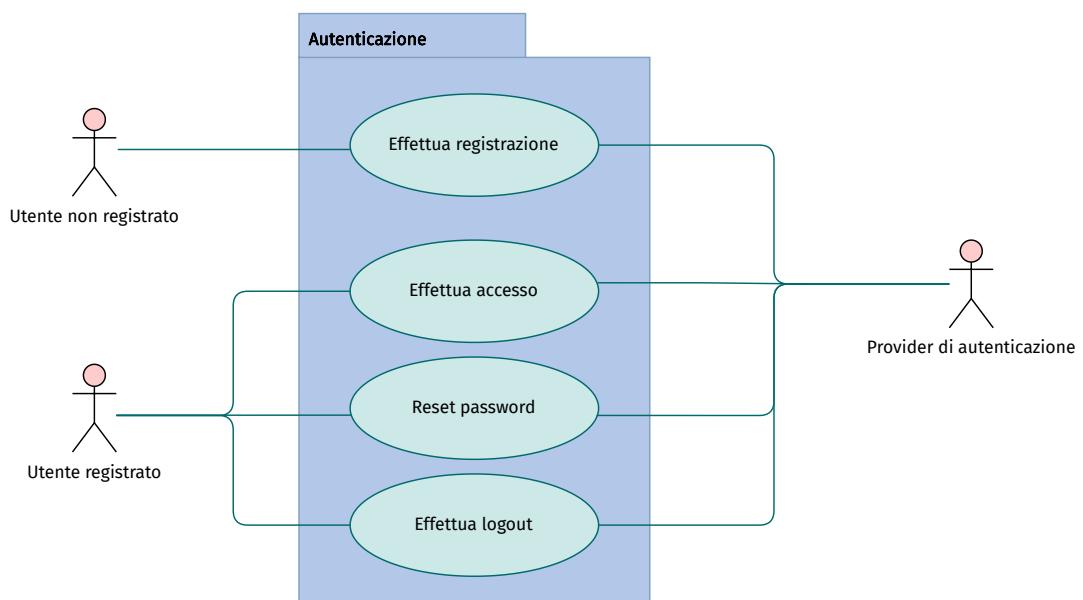


Figura 2: Package 1 - Autenticazione

1.7.2 Interazione con un itinerario

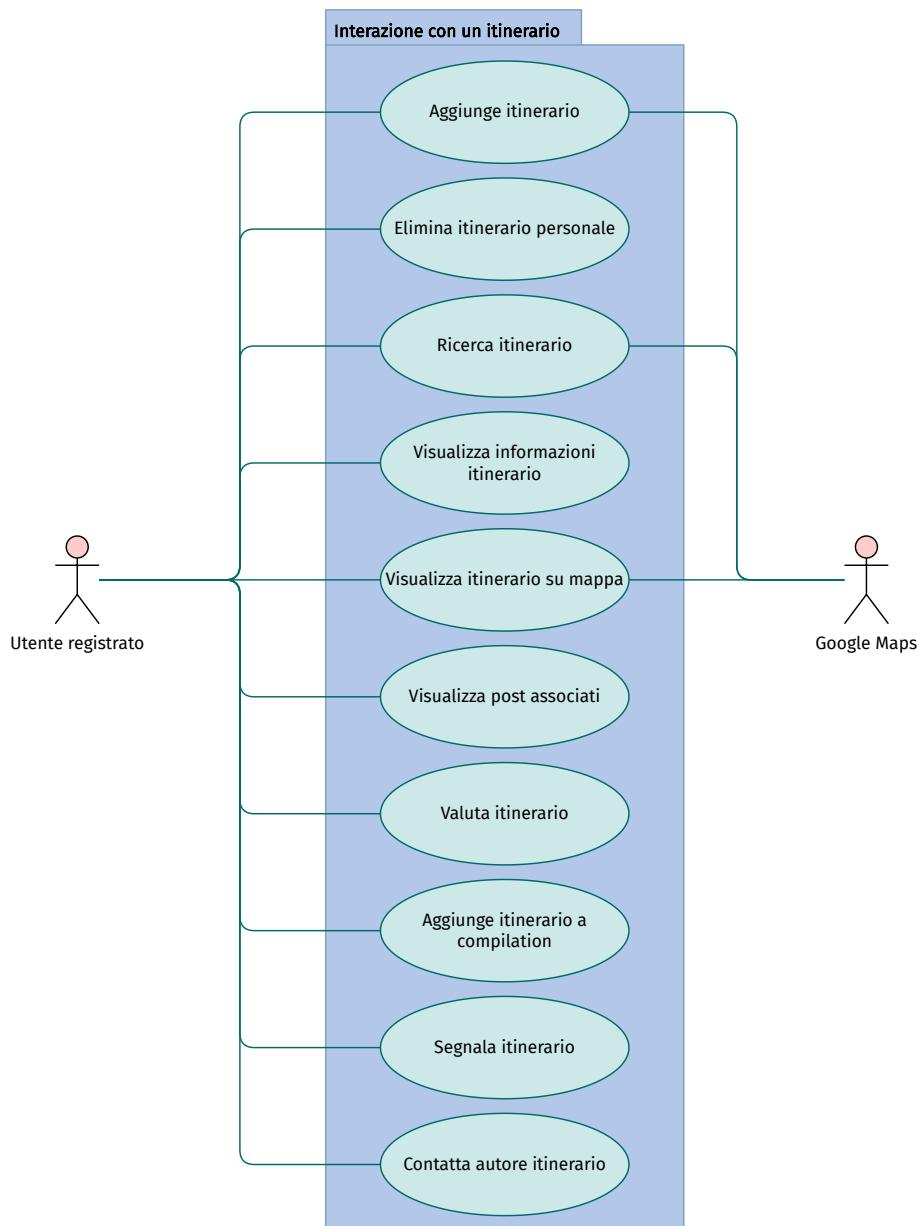


Figura 3: Package 2 - Interazione con un itinerario

1.7.3 Interazione con un post

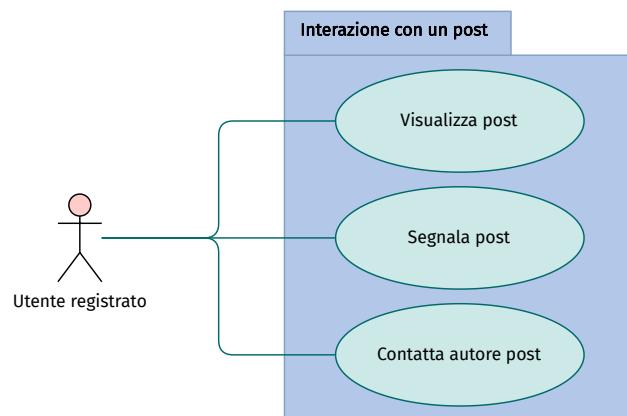


Figura 4: Package 3 - Interazione con un post

1.7.4 Gestione profilo

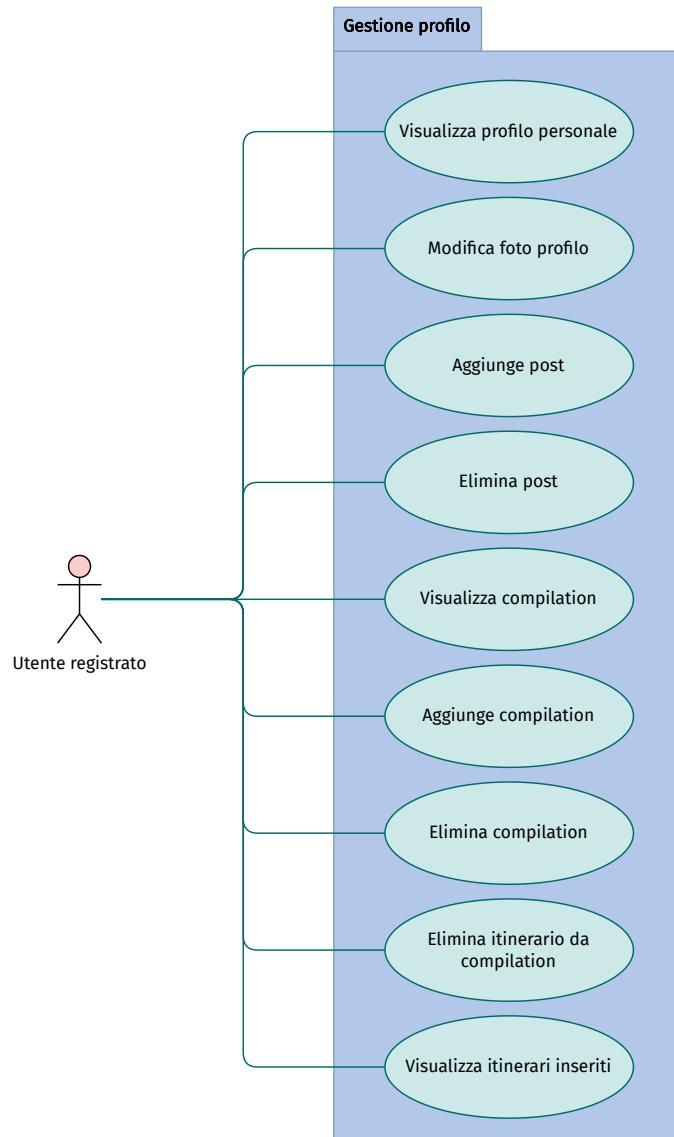


Figura 5: Package 4 - Gestione profilo

1.7.5 Gestione messaggi

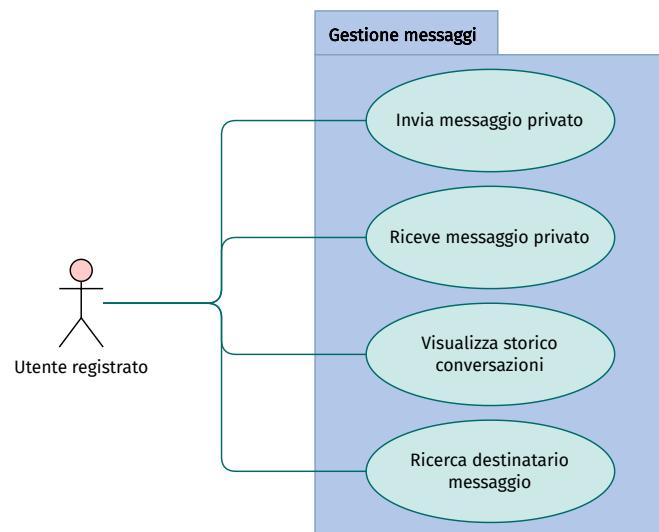


Figura 6: Package 5 - Gestione messaggi

1.7.6 Funzionalità amministratore

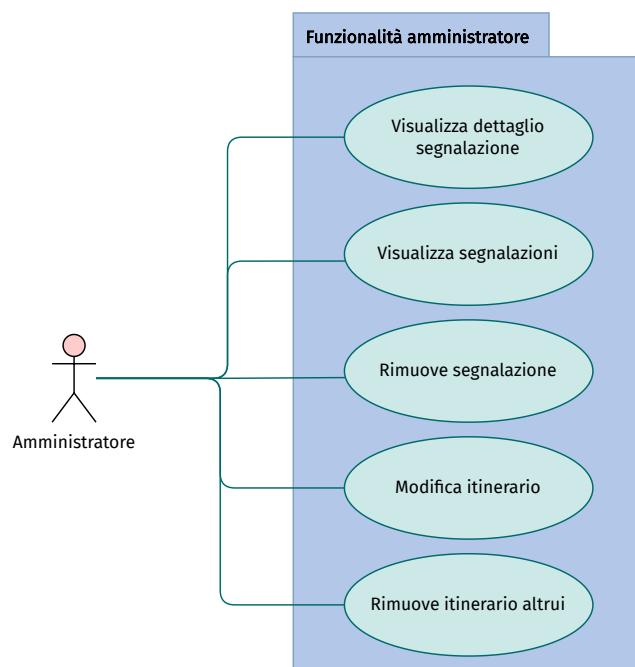


Figura 7: Package 6 - Funzionalità amministratore

1.8 Tabelle di Cockburn dei casi d'uso

Sono presentate in questa sezione le tabelle di Cockburn relative a due casi d'uso significativi dello Use Case Diagram.

1.8.1 Inserisce un itinerario

USE CASE #1	Inserisce un itinerario			
Goal in Context	L'utente vuole inserire un itinerario in piattaforma			
Preconditions	L'utente è autenticato			
Success End Conditions	L'utente riesce ad inserire il nuovo itinerario			
Failed End Conditions	L'utente non riesce ad inserire il nuovo itinerario			
Primary Actor	Utente registrato			
Secondary Actor	Google Maps			
Trigger	L'utente preme il Floating Action Button “Nuovo itinerario” nella schermata “RoutesUI”			
	Step	Utente registrato	Sistema	Google Maps
	1	Preme il Floating Action Button “Nuovo itinerario” nella schermata “RoutesUI”		
	2		Mostra la schermata “CreateRouteInfoUI”	
	3	Inserisce il nome dell'itinerario, (optionalmente) la descrizione, la durata approssimativa, il livello di difficoltà e la possibilità di accessibilità per disabili		
	4	Preme il Floating Action Button “nextFab”		
	5		Mostra la schermata “CreateRouteMapUI”	

	6	Preme sull'icona di ricerca		
	7	Inserisce la posizione della prima tappa del percorso per visualizzarla sulla mappa		
	8			Mostra la posizione desiderata sulla mappa
	9	Seleziona le tappe del percorso tramite la mappa interattiva		Recupera la posizione dei punti selezionati sulla mappa e traccia i percorsi tra essi
	10	Preme il Floating Action Button "nextFab"		
	11		Mostra la schermata "CreateRoutePhotosUI"	
	12	Preme il bottone "Seleziona foto"		
	13		Mostra il photo picker per la selezione delle foto	
	14	Seleziona un numero di foto compreso tra 1 e 5		
	15	Preme il bottone "Fatto"		
	16		Mostra la schermata "CreateRoutePhotosUI"	

	17	Preme il Floating Action Button “Inserisci itinerario”		
EXTENSION #1 Inserimento mappa con file GPX	Step	Utente registrato	Sistema	Google Maps
	9.1	Apre il menu a tendina in alto a destra della schermata e seleziona “Importa GPX”		
	10.1		Mostra la schermata per la selezione di un file GPX	
	11.1	Seleziona il file GPX		
	12.1		Mostra la schermata "Create-Route-MapUI" aggiornata	

1.8.2 Segnala un itinerario

USE CASE #2	Segnala un itinerario		
Goal in Context	L'utente vuole segnalare un itinerario per informazioni inesatte o non accurate		
Preconditions	L'utente è autenticato		
Success End Conditions	L'utente riesce a segnalare l'itinerario correttamente		
Failed End Conditions	L'utente non riesce a segnalare l'itinerario		
Primary Actor	Utente registrato		
Trigger	L'utente preme sull'icona di segnalazione nella schermata di dettaglio di un itinerario		
	Step	Utente registrato	Sistema
	1	Preme l'icona di segnalazione nella schermata "RouteDetailsUI"	
	2		Mostra la full dialog "ReportRoute-FullDialog"
	3	Inserisce un titolo e una descrizione per la segnalazione	
	4	Preme il bottone "Invia"	
	5		Mostra la schermata "RouteDetailsUI" con un toast contenente un messaggio di segnalazione andata a buon fine, aggiungendo all'itinerario un warning (se non è già presente) per possibili informazioni inesatte

1.9 UX Design

1.9.1 Definizione delle Personas

Uno dei principali obiettivi dello sviluppo dell'applicativo è stato quello di *conoscere i potenziali utenti* e di delineare dei loro probabili comportamenti, al fine di garantire all'intero target di utenza la migliore esperienza possibile.

Si è ritenuto dunque opportuno elaborare dei profili ideali e immaginari: conoscendo caratteristiche e comportamenti del destinatario di un prodotto, la progettazione di quest'ultimo risulta notevolmente più efficiente.

Oltre a creare semplicemente un'ottima user experience, dunque, si è prestata una particolare attenzione ad utilizzare un approccio *customer-centered*.

I profili elaborati prendono il nome di **personas**. Le *personas* sono veri e propri identikit, creati sulla base di ricerche, che rappresentano un particolare gruppo di clienti che potrebbero utilizzare un prodotto. Delineare le caratteristiche di tali clienti permette di conoscere e comprendere i loro bisogni, obiettivi, paure e motivazioni.

Proprio questi identikit, in conclusione, hanno contribuito allo sviluppo di una **UX** piacevole, funzionale ed elegante. Di seguito sono mostrati tre di questi profili elaborati.

Liam Kumar



"Il successo non è finale come il fallimento non è fatale: è il coraggio di continuare che conta."

Età: 23
Lavoro: Consulente finanziario
Status: Single
Residenza: Dubai

Bio

Al momento lavoro come consulente finanziario presso un'azienda a Dubai dove sono anche residente. Per me tenersi in forma è una priorità ed adoro fare escursioni, soprattutto con persone con più esperienza.

Personalità

Introversione	Estroversione
Pensiero	Sentimento
Sensazione	Intuizione
Giudizio	Percezione

Obiettivi

- Essere sempre aggiornato sulle novità del mondo finanziario
- Scrivere una biografia
- Essere sempre al massimo della forma

Frustrazioni

- Saltare gli allenamenti quotidiani
- Rimanere indietro rispetto la massa

Motivazione

Paura	
Crescita	
Potere	
Socialità	

Competenze

Tecnologiche	
Sociali	
Apprendimento	
Varie	

Marchi & Influenze



ethereum





24

È stato scelto di differenziare le caratteristiche delle personalità prese in analisi, in modo da coprire una maggiore fetta di utenza.

Juanna Cheng



"Quando hai un sogno, lasciarlo andare è il più grande errore tu possa fare."

Età: 25
Lavoro: Capo reparto
Status: Single
Residenza: Milano

Bio

Correntemente vivo a Milano. Sono attualmente vice-leader di un reparto presso l'azienda nella quale lavoro. Amo molto fare lunghe passeggiate nei weekend sia da sola che in compagnia.

Personalità

Introversione	Estroversione
Pensiero	Sentimento
Sensazione	Intuizione
Giudizio	Percezione

Obiettivi

- Bisogno di trovare persone con simili abilità per migliorarsi costantemente
- Diventare un modello da seguire
- Aumentare il suo bagaglio culturale

Frustrazioni

- Non essere in grado di rispettare delle scadenze
- Mancanza di confronto

Motivazione

Paura	
Crescita	Potere
Socialità	

Competenze

Tecnologiche	
Sociali	Apprendimento
Varie	

Marchi & Influenze





Modellando dei potenziali utenti è possibile comprendere quelli che potrebbero essere i punti critici del prodotto.

Alfredo Bianchi



Bio

Vivo a New York con la mia compagna. Da 3 anni lavoro presso un'agenzia come consulente di viaggi. Dovendo consigliare dove viaggiare, spesso mi porto in tali destinazioni.

Personalità

Introversione	Estroversione
Pensiero	Sentimento
Sensazione	Intuizione
Giudizio	Percezione

"La cattiva notizia è che il tempo vola, la buona è che tu sei il pilota."

Età: 29
Lavoro: Consulente di viaggi
Status: Fidanzato
Residenza: New York

Obiettivi

- Aprire la sua agenzia di viaggi
- Toccare ogni continente
- Creare una famiglia

Frustrazioni

- Non esser preso in considerazione
- Doversi impostare dei limiti

Motivazione

Paura	
Crescita	
Potere	
Socialità	

Competenze

Tecnologiche	
Sociali	
Apprendimento	
Varie	

Marchi & Influenze





Come si può notare dai precedenti modelli, non ci si è concentrati soltanto su probabili *skills* degli utenti, ma anche sul loro temperamento: si ritiene fondamentale, infatti, adattare l'esperienza utente in modo che possa risultare piacevole a tutte le diverse personalità.

1.9.2 Prototipazione dell'esperienza utente

Attraverso una fase di **brainstorming** sono state generate idee per sfruttare le opportunità e risolvere i problemi emersi durante la creazione delle *personas*. La raccolta di opinioni e suggerimenti dei componenti del team di progettazione ha analizzato anche le proposte più semplici ed intuitive: è stato ritenuto importante, infatti, valutare ogni possibilità nata dal processo creativo.

Una volta approfondite le proposte, è stato deciso di realizzare dei prototipi dinamici con un tool di **CASE**; nello specifico, la scelta di quest'ultimo è ricaduta su *Figma*.

Periodicamente ogni prototipo è stato discusso e rielaborato, in un processo coinvolgente tutti i membri del team e avente come oggetto, in particolare, le seguenti analisi:

- Facilità di navigazione;
- Tempo di reperimento dei contenuti;
- Coerenza nella creazione di nuovi contenuti.

Progettare l'interazione tra utenti e prodotto è un processo complicato e spesso anche dispendioso.

Una possibile soluzione a questo problema, però, esiste: si tratta di un metodo molto utilizzato in **UX Design**, avente lo scopo di ridurre al minimo i test utente. Tale processo è noto come *valutazione euristica*.

Le **Euristiche di Nielsen** sono il cardine dell'esperienza utente ideata.

Il focus dell'analisi è stato rivolto verso lo *stato del sistema*, con particolare attenzione agli stati di *caricamento* e di *errore*; si è ritenuto opportuno, in queste condizioni, fornire all'utente dei feedback.

Per quanto riguarda le situazioni di errore è stata adottata una politica di *problem avoidance* piuttosto che di *problem solving*: ciò non implica, ovviamente, che le circostanze di errore non siano state gestite o degnate della giusta attenzione. Si è preferito, però, concentrarsi sul *prevenire* gli errori. Ove possibile, infatti, sono stati posti dei "limiti": grazie alla loro presenza, non sarebbe possibile procedere con l'utilizzo di una particolare funzionalità se i requisiti necessari per un corretto funzionamento di quest'ultima non fossero rispettati.

Le funzionalità più "comuni", in quanto di prassi nel mondo del social networking, si trovano sotto nomi ben noti: una delle motivazioni di questa scelta è stata sicuramente la volontà di creare una corrispondenza tra il sistema e il mondo reale. La motivazione principale, però, è stata la volontà di ottenere una sorta di *continuità* nell'utilizzo del prodotto: sfruttando convenzioni ormai consolidate nell'esperienza utente, è stato ritenuto naturale e improntato a una migliore comprensione del funzionamento dell'applicativo sfruttare i meccanismi e comportamenti - che per un utente abituale risultano involontari - che entrano in gioco nell'utilizzo di un'applicativo informatico.

L'interfaccia non presenta limitazioni; l'utente non viene mai posto in condizione di dover visualizzare funzionalità o componenti indesiderate. È stato ritenuto però necessario stabilire delle eccezioni, relative esclusivamente ad operazioni *distruttive*, che potrebbero causare modifiche permanenti ed irreversibili ai contenuti inseriti. Tali operazioni presentano dunque la necessità di essere confermate prima di essere portate a termine: risulta

chiara, anche qui, la volontà di evitare situazioni di disagio causate da meri errori dell’utente.

L’interfaccia realizzata per il software risulta essere estremamente minimalista; i contenuti più importanti sono posti in risalto attraverso sfumature, colori, elevazione ed un posizionamento strategico delle informazioni.

Le scelte di design sono dunque improndate a creare un ambiente visivo semplice e di immediata comprensione. Ci si trova in presenza di un ambiente totalmente isolato, in cui le diverse schermate sono caratterizzate da una forte **consistenza**. Per *consistenza* si intende il mantenimento di una linea generale di presentazione di contenuti attraverso l’intero applicativo: le diverse interfacce non presentano mai cambi di layout estremi, garantendo una maggiore velocità d’apprendimento agli utenti.

Prima di raffinare l’esperienza utente è stato ritenuto necessario coinvolgere un ristretto bacino d’utenza per ottenere riscontri esterni rispetto al contesto di prototipazione e sviluppo.

1.9.3 Definizione di uno stile di design

Per raffinare l'esperienza utente è stato ritenuto un buon approccio quello di affidarsi ad uno specifico **UI Design System**. La scelta è stata effettuata tra diversi sistemi concorrenti, ricadendo infine su **Material**³.



Material Design si ispira al mondo fisico, includendo riflessi e ombre. Tipografie, griglie, spazi e colori creano gerarchie finalizzate a rendere l'utilizzatore totalmente immerso. Gli effetti di movimento mirano a focalizzare l'attenzione generando continuità spaziale attraverso feedback e transizioni. Le componenti sono altamente dinamiche, si trasformano e organizzano l'ambiente in base alle interazioni, generando altre trasformazioni. Le componenti Material sono blocchi interattivi che formano l'interfaccia utente. Queste includono stati built-in system per comunicare focus, selezione, attivazione, errore, hovering, pressione, trascinamento, e disabilitazione.

Ogni componente è realizzata avendo specifiche caratteristiche e finalità, tra le quali:

- **Esposizione** - Piazzare ed organizzare contenuti correlati usando componenti come *card* e *liste*;
- **Navigazione** - Consentire agli utenti di muoversi nel prodotto finito utilizzando *drawer* o *tabs*;
- **Azioni** - Consentire agli utenti di eseguire task utilizzando componenti di alto rilievo, come i *floating action button*;
- **Input** - Consentire agli utenti di inserire informazioni o selezionare tra un certo numero di opzioni, attraverso *campi di testo dinamici* e *chips*;
- **Comunicazione** - Dare informazioni agli utenti attraverso *snackbar*, *toast* e *dialog*.

³Material è un design system creato da Google per facilitare lo sviluppo di UX di altissima qualità per Android, iOS, Flutter, e Web-App.

1.10 Mock-Up dell'applicazione

Vengono ora mostrati i Mock-Up dei casi d'uso significativi dettagliati nella sottosezione precedente.

1.10.1 Inserimento itinerario

Il processo di inserimento di un itinerario consiste in tre fasi:

- Inserimento informazioni sull'itinerario;
- Inserimento mappa itinerario;
- Inserimento foto itinerario.

Sono presentati i Mock-Up che rappresentano un inserimento fittizio, con scopo illustrativo.



The mock-up shows a mobile application screen titled "Nuovo itinerario". It contains four input fields: "Nome itinerario", "Descrizione (opzionale)", "Durata", and "Livello di difficoltà". Below the "Durata" field, there is a note: "Durata da 1 a 16 ore". Under "Livello di difficoltà", there are three buttons: "Facile" (selected), "Intermedio", and "Difficile". Below "Livello di difficoltà", there is a section for accessibility: "Accessibilità ai disabili" with a note "Non accessibile a persone con disabilità" and a toggle switch that is off. At the bottom right is a large grey button with a right-pointing arrow.

Figura 8: Inserimento informazioni - form vuoto



The mock-up shows the same mobile application screen as Figure 8, but with filled-in data. The "Nome itinerario" field contains "Percorso Maradona". The "Durata" field shows "4 ore". The "Livello di difficoltà" section has the "Facile" button selected. The accessibility section now says "Accessibile a persone con disabilità" and the toggle switch is on. The large grey button at the bottom right now has a teal background and a white right-pointing arrow.

Figura 9: Inserimento informazioni - compilato

Il processo di inserimento della mappa può avvenire sia interagendo con la mappa interattiva sia importando un file GPX dall'apposito menu. In entrambi i casi è possibile ricercare una posizione per visualizzarla sulla mappa ed eventualmente apporre i marker; inoltre è possibile eliminare i marker inseriti.

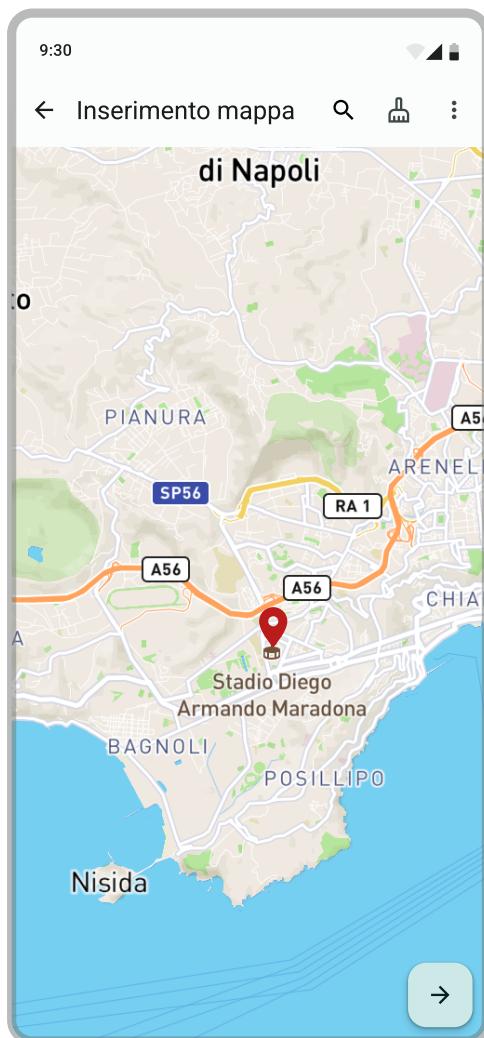


Figura 10: Inserimento mappa

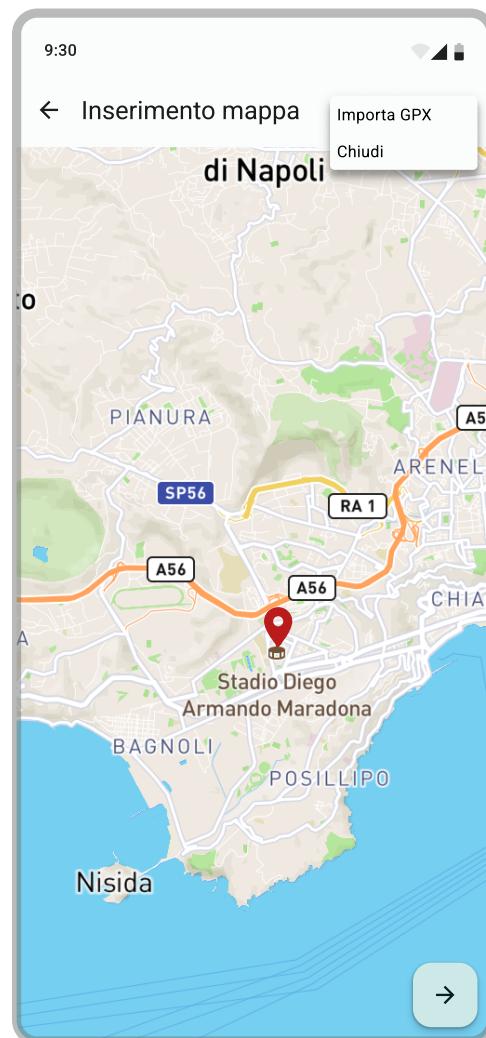


Figura 11: Inserimento mappa - menu

L'inserimento delle foto per l'itinerario usufruisce di un photo picker esterno, il quale rimanda ad una schermata di sistema che consente la selezione di più foto, fino a un massimo di 5. Una volta effettuata la selezione desiderata, le foto appariranno nella schermata precedente con la possibilità di eliminarle.

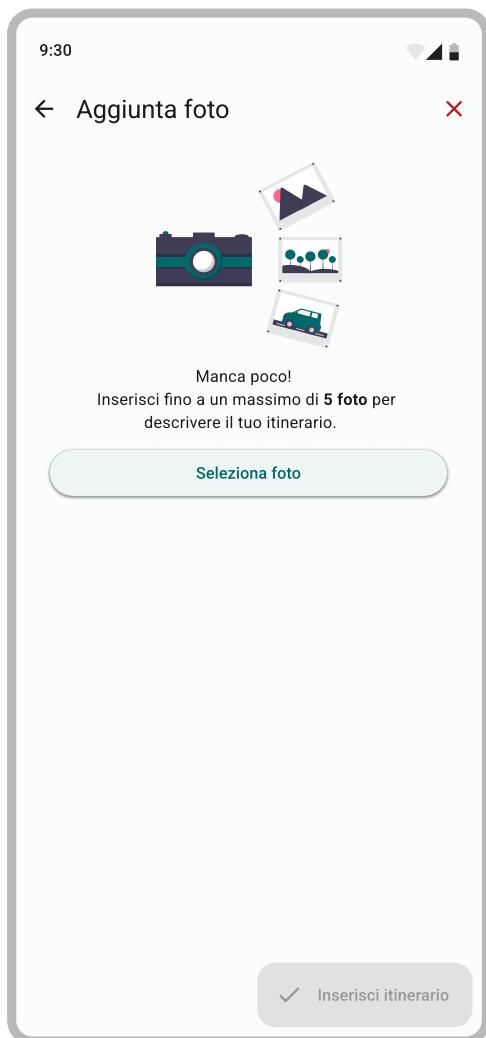


Figura 12: Inserimento foto

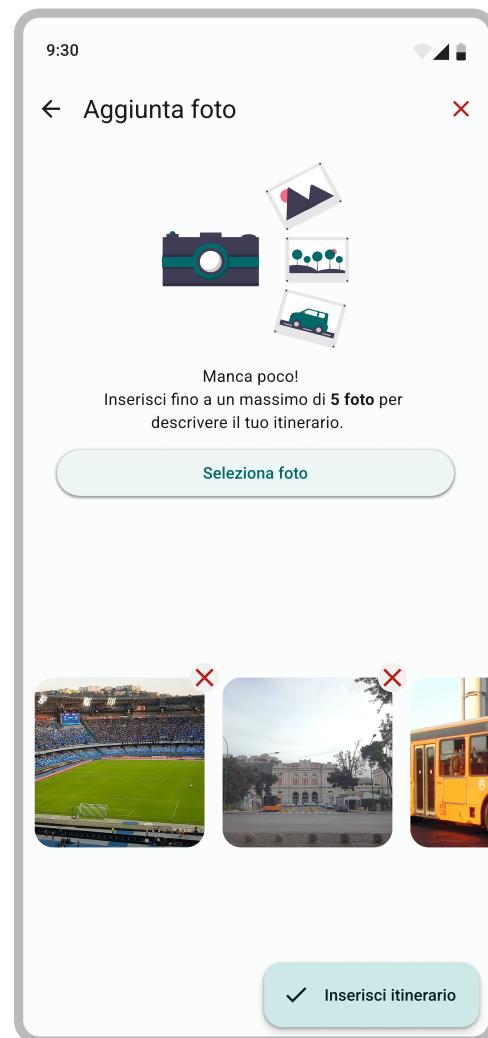


Figura 13: Inserimento foto - foto inserite

È importante sottolineare che - per ridurre al minimo i possibili errori nell'inserimento - le *affordance* che consentono la navigazione tra le diverse fasi dell'inserimento restano disattivate fino all'inserimento di input adeguati.

Una caratteristica di particolare rilevanza è il *salvataggio dello stato* dell'inserimento tra le schermate: sarà quindi possibile navigare tra di esse senza perdere i dati inseriti.

1.10.2 Segnalazione itinerario

La segnalazione di un itinerario avviene tramite una full dialog, a seguito del riempimento dell'apposito form.



9:30

✗ Invia una segnalazione Fatto

Titolo segnalazione

Descrizione



9:30

✗ Invia una segnalazione Fatto

Titolo segnalazione

Informazioni inesatte

Descrizione

Il percorso inserito risulta essere pericoloso; si richiede la rimozione immediata da parte di un amministratore.

Figura 14: Segnalazione - form vuoto

Figura 15: Segnalazione - compilato

1.11 Valutazione dell’usabilità a priori

Una parte fondamentale del processo creativo alla base dello sviluppo di un’applicazione è capire la direzione da seguire per migliorare l’esperienza dei clienti. Al fine di rendere la user experience quanto più piacevole possibile si è ricorso alla messa in pratica di alcune tecniche, con il fine di testare l’usabilità del prodotto. L’approccio adottato richiede l’esecuzione di alcun task ad un gruppo di utenti con differenti retroscena.

Tali task possono avere un riscontro differente a seconda dell’utente:

- L’utente è stato in grado di compiere il task tramite gli strumenti offerti dall’app;
- L’utente non è stato in grado di compiere il task tramite gli strumenti offerti dall’app;
- L’utente è stato in grado di compiere il task, ma ha trovato difficoltà oppure trova l’interfaccia poco intuitiva.

Verranno fornite solo indicazioni di base: il sistema dovrebbe essere intuitivo. In caso di una percentuale di riscontri altamente negativa, il sistema sarà rielaborato alla luce dei feedback utente.

1.11.1 Definizione dei task utente

1. Creazione itinerario

Hai deciso di aggiungere un nuovo itinerario all’interno della piattaforma: a questo scopo dovrà andare nella sezione degli itinerari e cliccare sul bottone di creazione. Nelle schermate successive dovrà inserire le informazioni, la mappa e le foto del nuovo itinerario.

2. Aggiunta itinerario ad una compilation

Hai trovato un itinerario interessante che hai scelto di inserire all’interno di una compilation. Seleziona l’icona di salvataggio per poi decidere in quale compilation inserirlo.

3. Segnalazione itinerario

Hai trovato un itinerario con informazioni errate, così hai deciso di segnalarlo agli amministratori. Dalla schermata dell’itinerario in questione clicca sull’icona di segnalazione e scegli un titolo e una descrizione per la segnalazione da inviare.

4. Contatto con autore itinerario

Hai trovato un itinerario ma hai dei dubbi riguardo ad esso, così hai deciso di contattare l’utente che l’ha creato. Dalla schermata dell’itinerario clicca l’icona di messaggio privato per visualizzare la chat.

5. Modifica immagine profilo

Ti sei appena registrato all’applicazione, ma il tuo profilo è completamente spoglio: così hai deciso di modificare l’immagine del tuo profilo. Spostati nella sezione profilo e modifica la tua foto.

1.11.2 Risultati ottenuti dallo svolgimento dei task

È stato preferibile selezionare i possibili clienti secondo serie di parametri quali:

- **Competenza tecnologica:** per favorire un'analisi migliore dell'intuitività fornita dalla UX;
- **Posizione lavorativa:** per spaziare tra background di carriera diversi, che potrebbero influire sull'esperienza con UX simili a quella progettata;

Di seguito è presente una tabella di utenti che hanno collaborato allo svolgimento dei task:

Nome	Competenza tecnologica	Età	Posizione Lavorativa	Sesso
Giovanna	Bassa	44	Casalinga	Donna
Ernesto	Media	50	Impiegato	Uomo
Paolo	Media	23	Studente	Uomo
Giulia	Alta	26	Studente	Donna
Rocco	Bassa	34	Operaio	Uomo
Francesca	Alta	30	Segretaria	Donna

Figura 16: Tabella utenti scelti

Di seguito i risultati ottenuti dallo svolgimento dei task:

Nome	Task 1	Task 2	Task 3	Task 4	Task 5
Giovanna	✗	✗	✗	✗	✓
Ernesto	✗	✓	✓	✓	✓
Paolo	✓	✓	✓	✗	✓
Giulia	✓	✓	✓	✓	✓
Rocco	✗	✗	✗	✓	✓
Francesca	✓	✓	✓	✓	✓

Figura 17: Tabella risultati

Da come si può notare la creazione dell'itinerario è risultata difficoltosa per alcuni utenti, infatti è stato richiesto un aiuto da parte loro in sede di prova. È stato ritenuto opportuno modificare leggermente la UX della creazione itinerario, ponendo in rilievo le azioni principali attraverso una scelta di colori migliore.

In percentuale, i risultati del test sono stati positivi, ulteriori cambiamenti nei prototipi non hanno comunque stravolto i precedenti.

1.12 Prototipazione funzionale via statechart dell'interfaccia grafica

In questa sezione sono riportati gli statechart relativi ai due casi d'uso significativi dettagliati nella sottosezione precedente.

1.12.1 Segnalazione itinerario

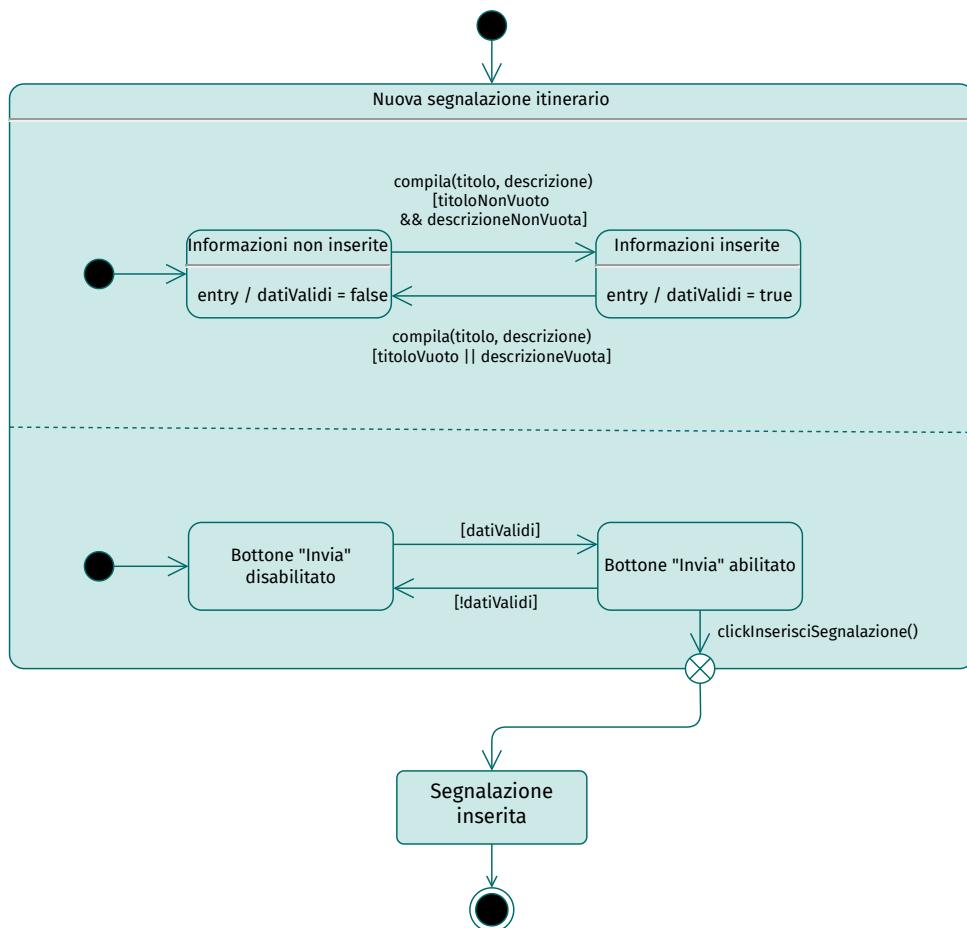


Figura 18: Nuova segnalazione itinerario

1.12.2 Inserimento itinerario

Per garantire una maggiore leggibilità, data l'estensione di questo statechart, si è fatta la scelta di rappresentarlo tramite regioni interne, riportate poi per intero.

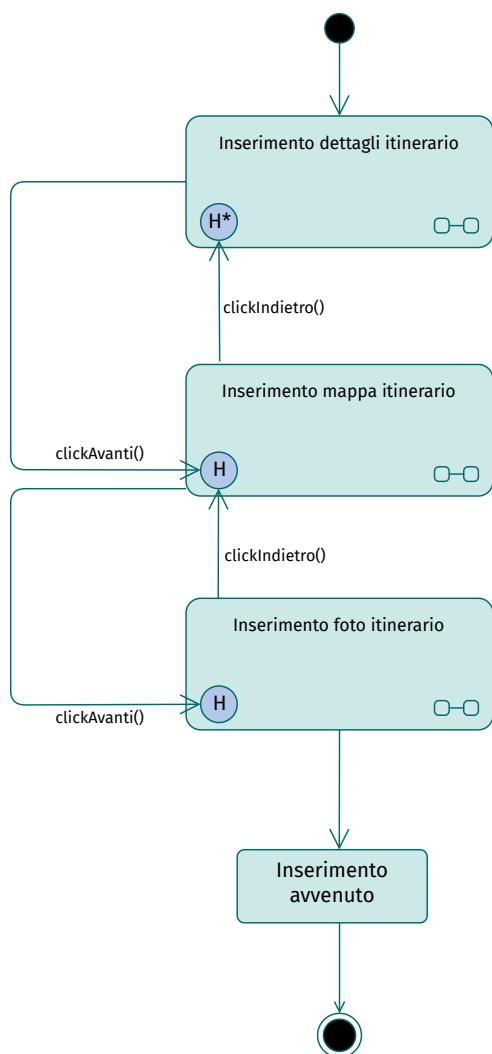


Figura 19: Nuovo itinerario

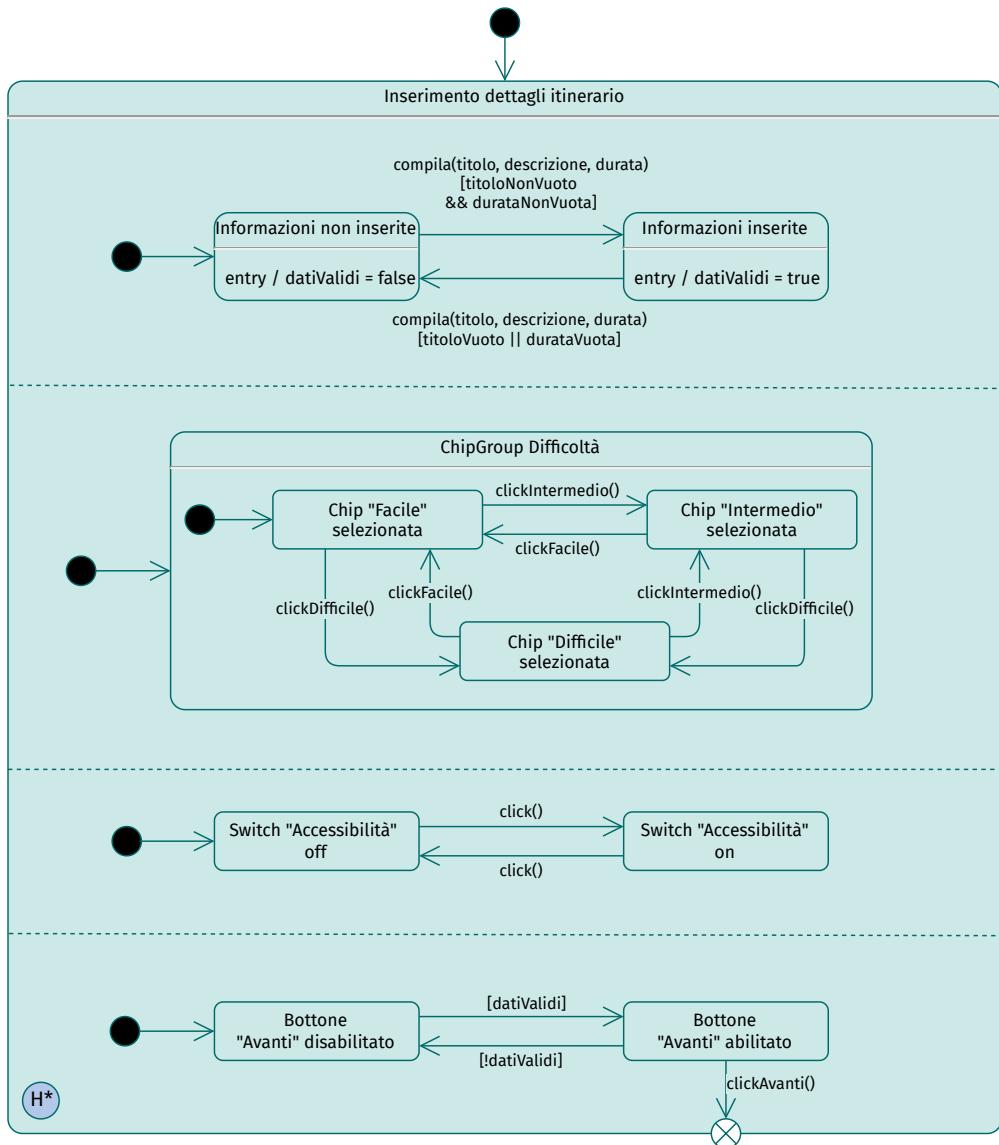


Figura 20: Regione interna - Inserimento dettagli itinerario

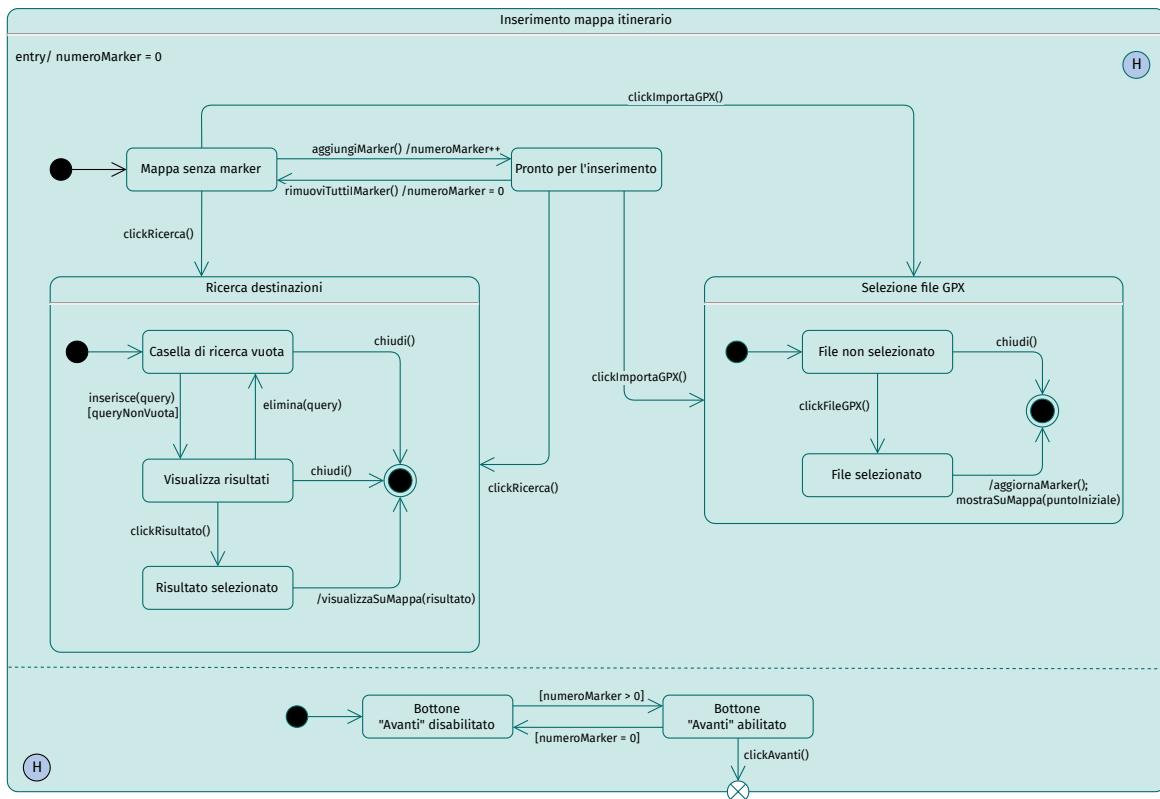


Figura 21: Regione interna - Inserimento mappa itinerario

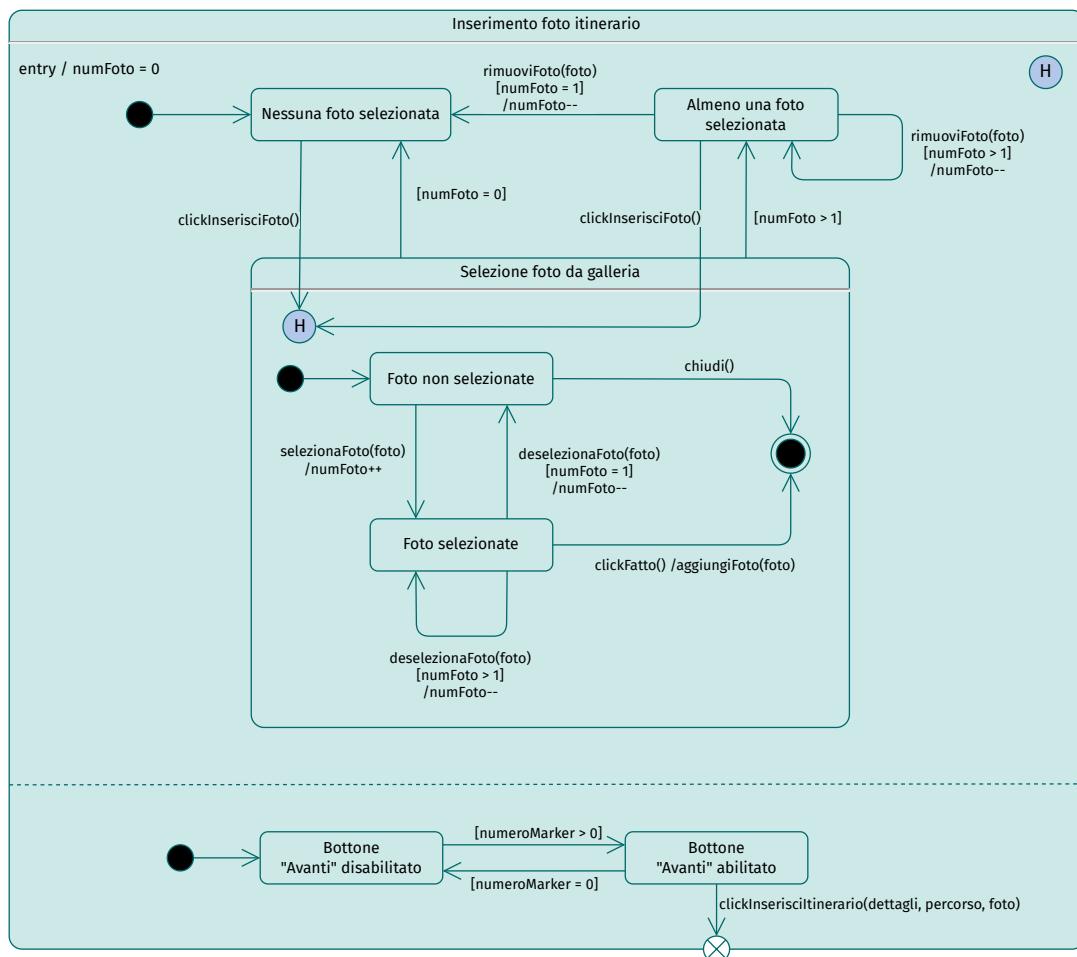


Figura 22: Regione interna - Inserimento foto itinerario

1.13 Classi, oggetti e relazioni di analisi

Nella sezione seguente sono riportati i diagrammi di analisi **Entity Control Boundary**, un pattern architettonale *use-case driven*.

1.13.1 Autenticazione

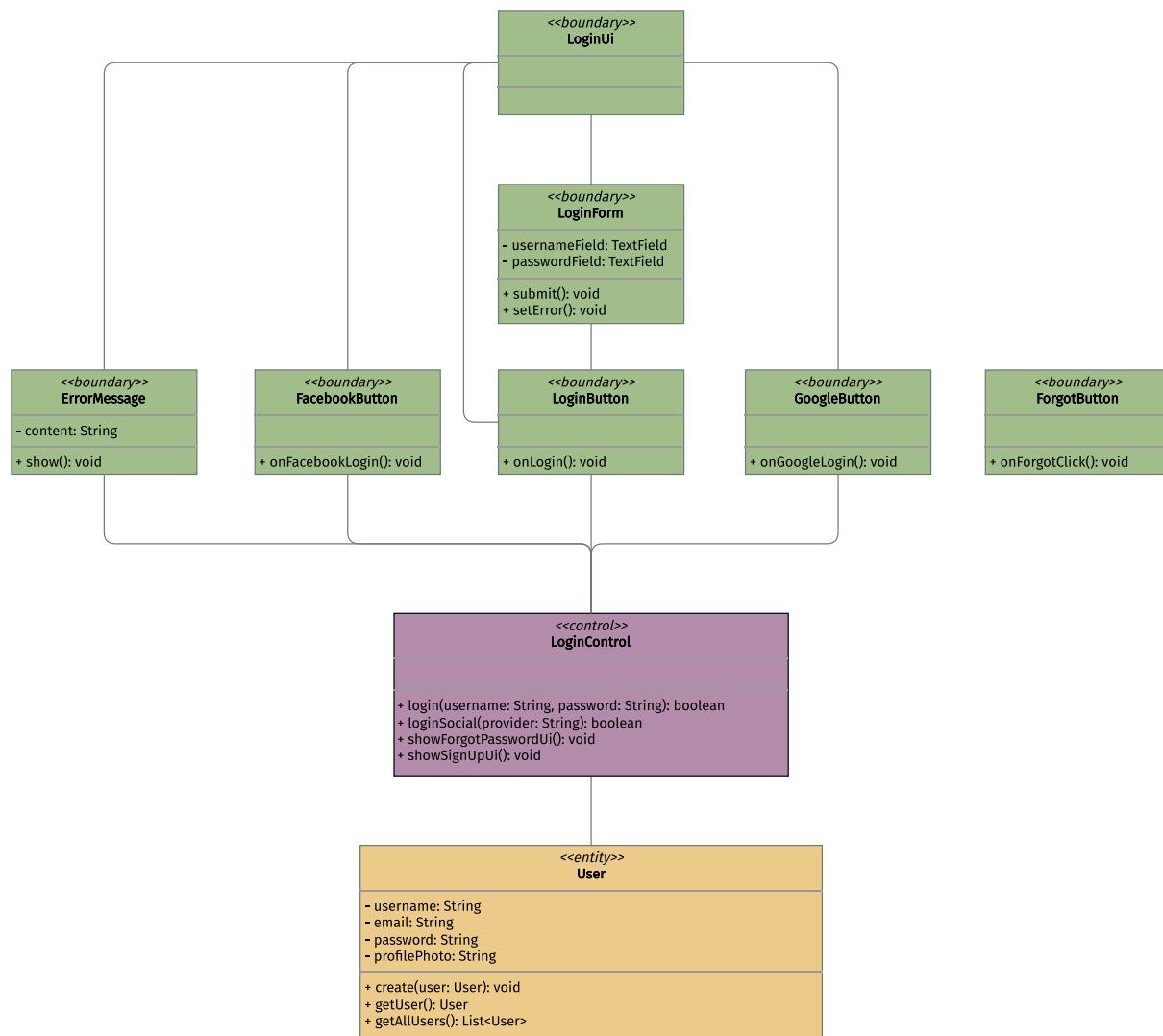


Figura 23: Login e login con social

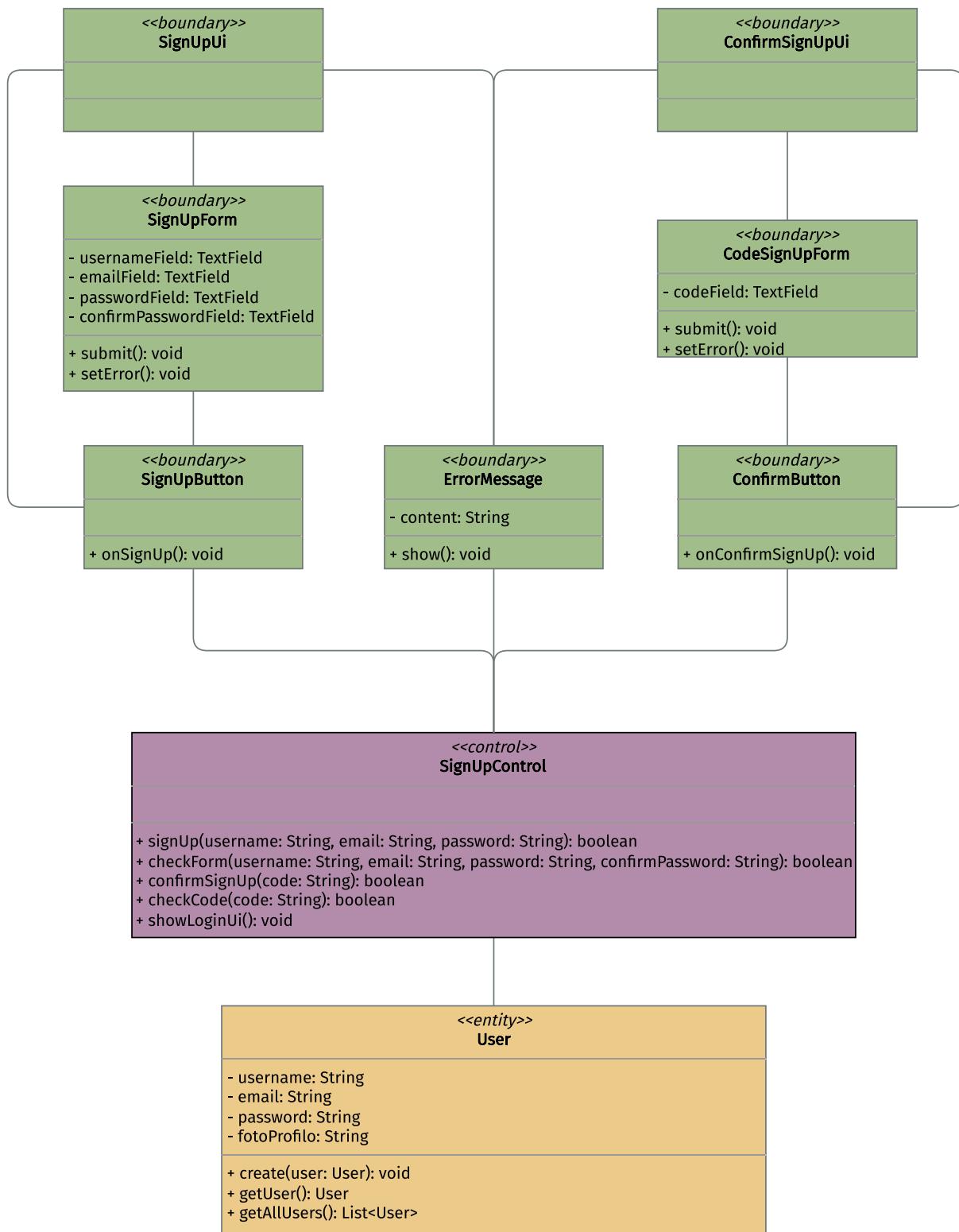


Figura 24: Registrazione e conferma

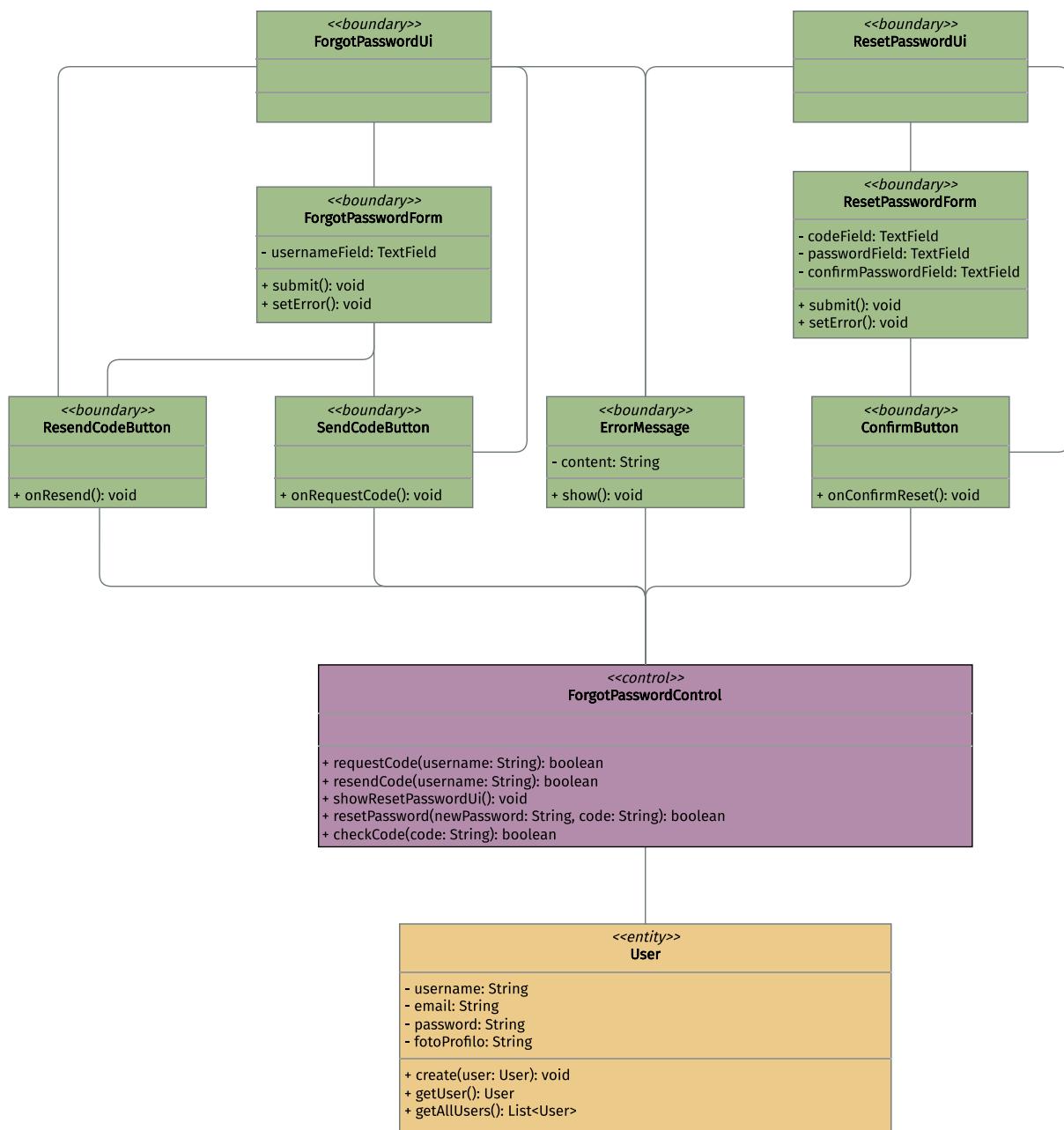


Figura 25: Password dimenticata

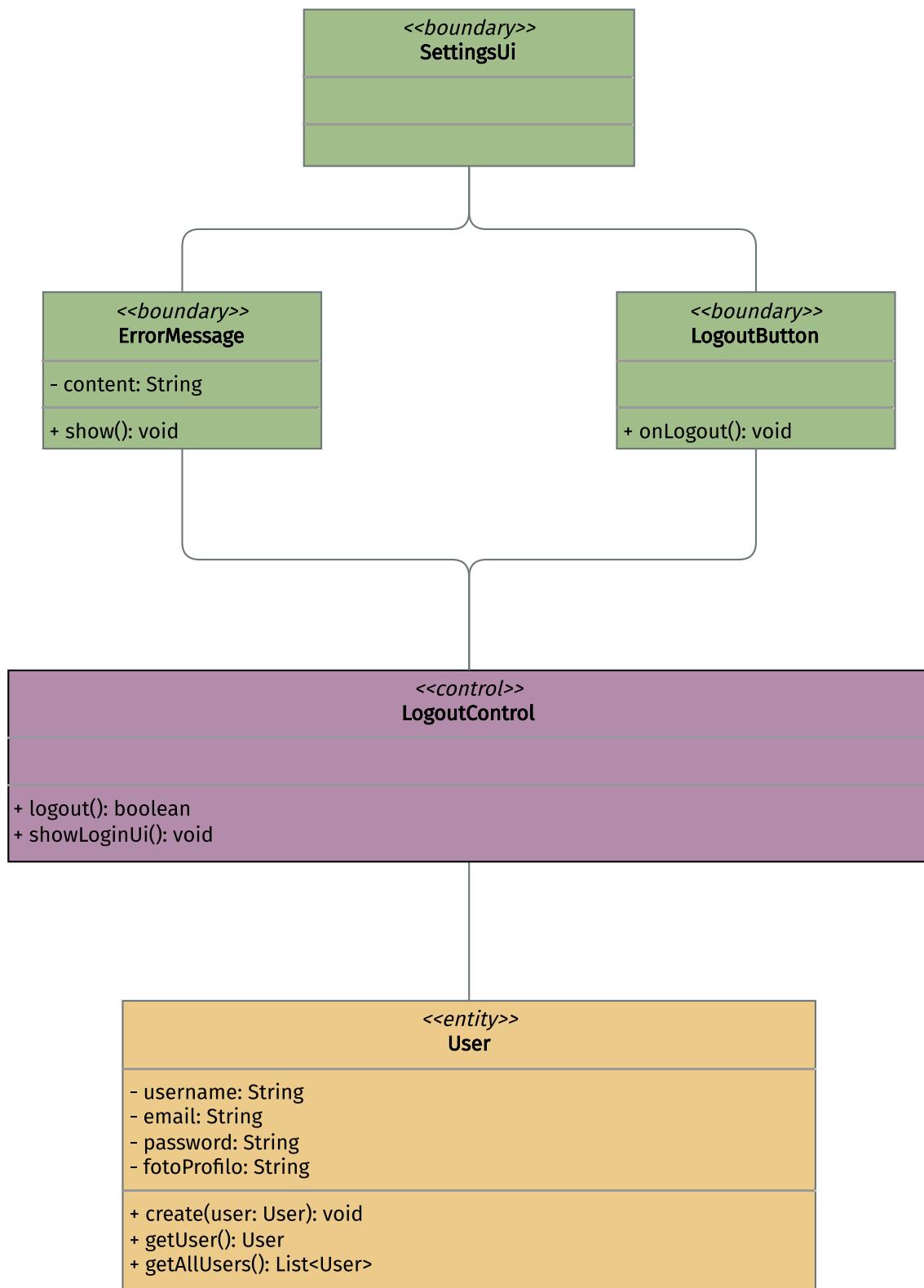


Figura 26: Logout

1.13.2 Interazione con un itinerario

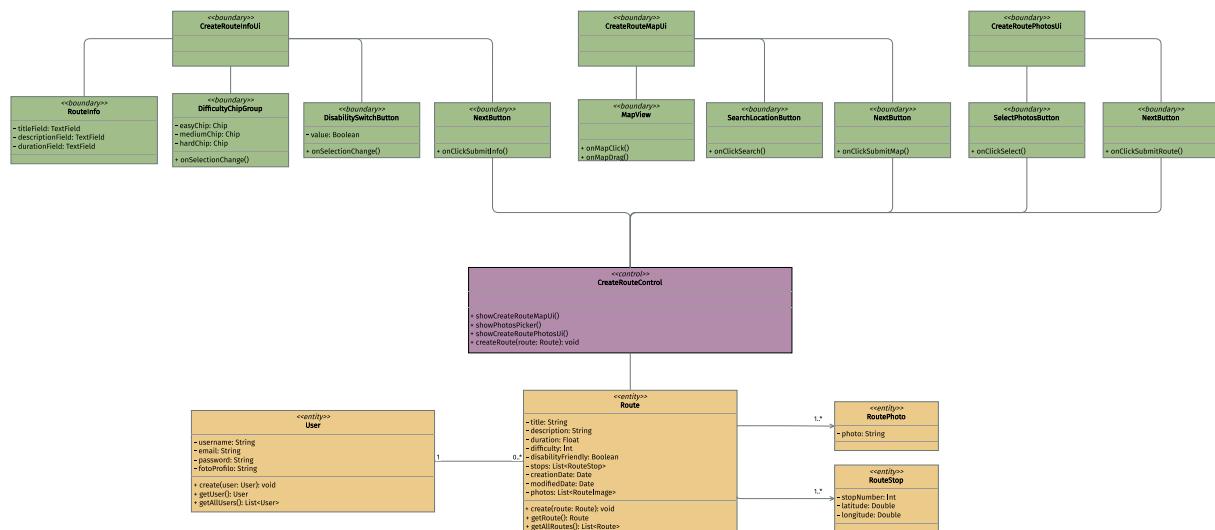


Figura 27: Aggiunta itinerario

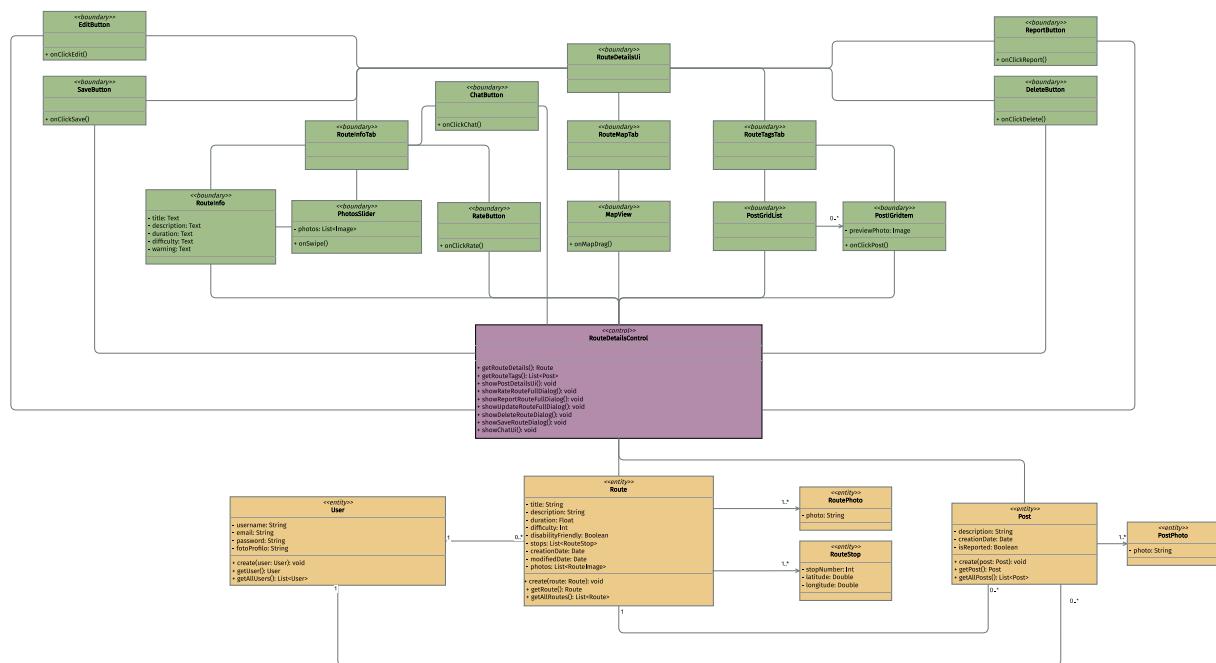


Figura 28: Dettagli itinerario

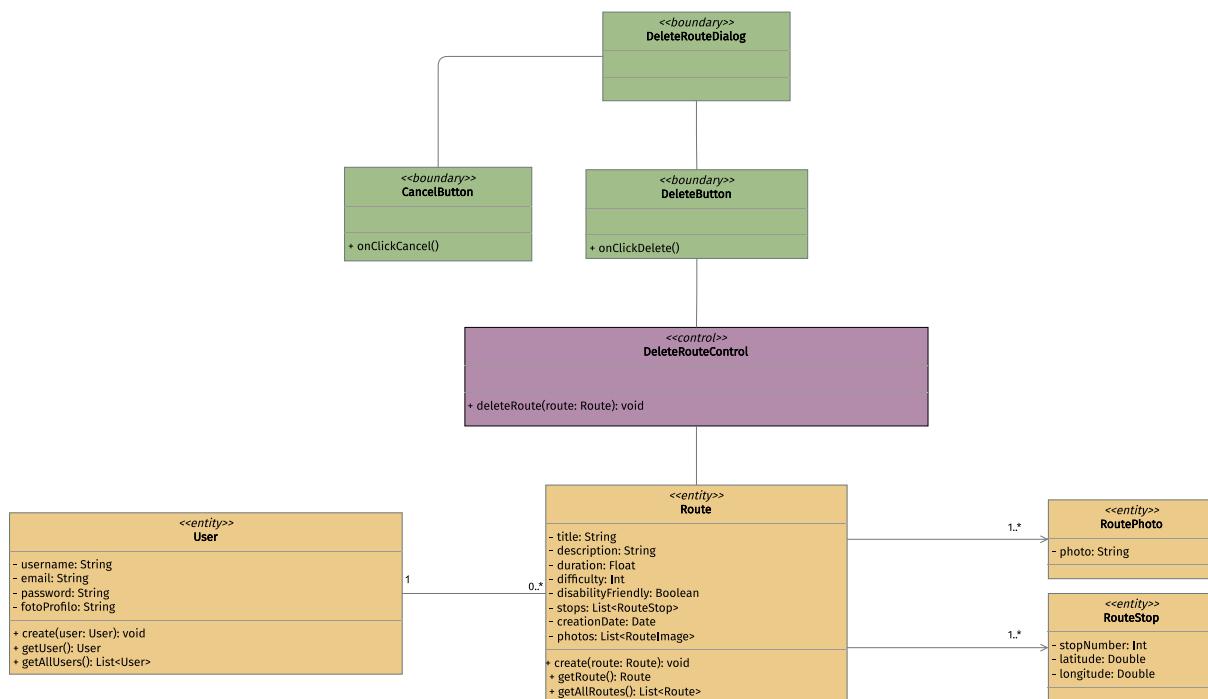


Figura 29: Eliminazione itinerario personale - Eliminazione itinerario altrui (admin)

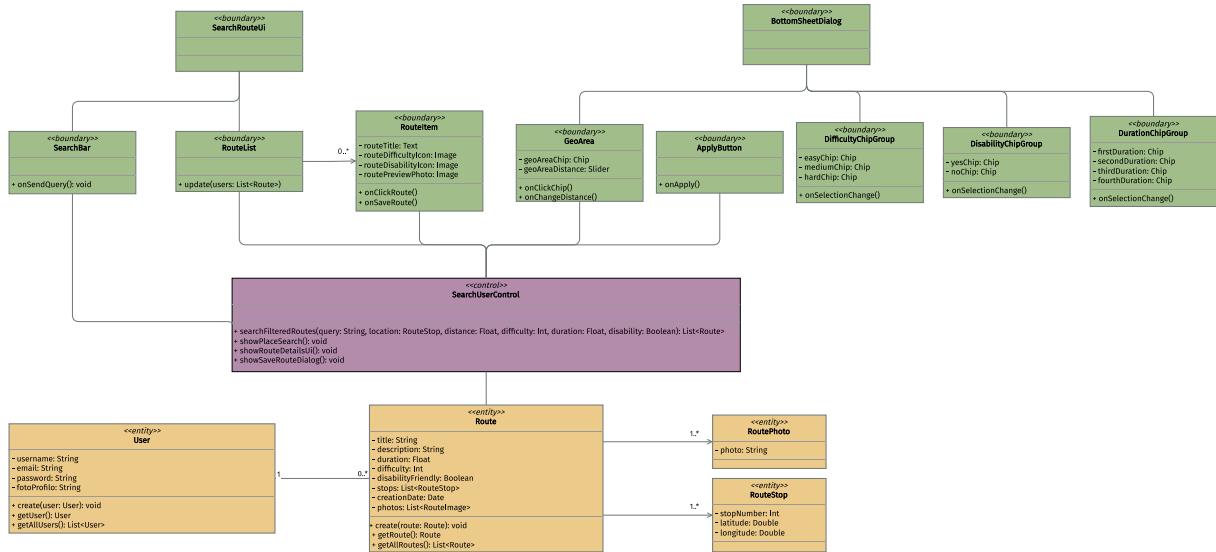


Figura 30: Ricerca itinerario

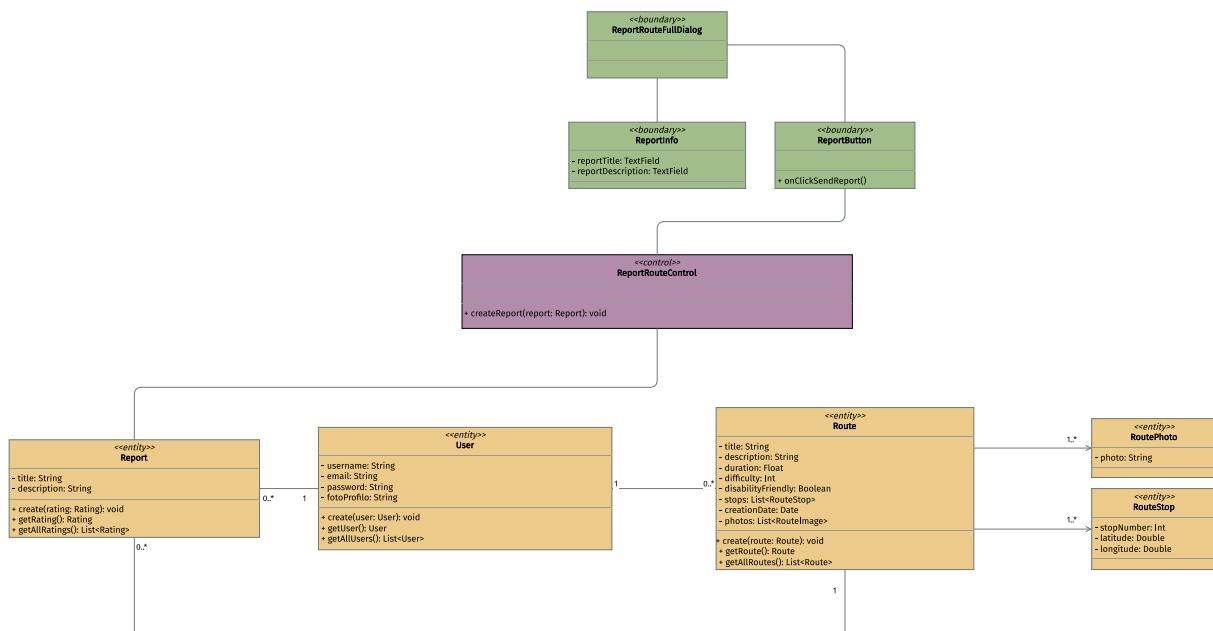


Figura 31: Segnalazione itinerario

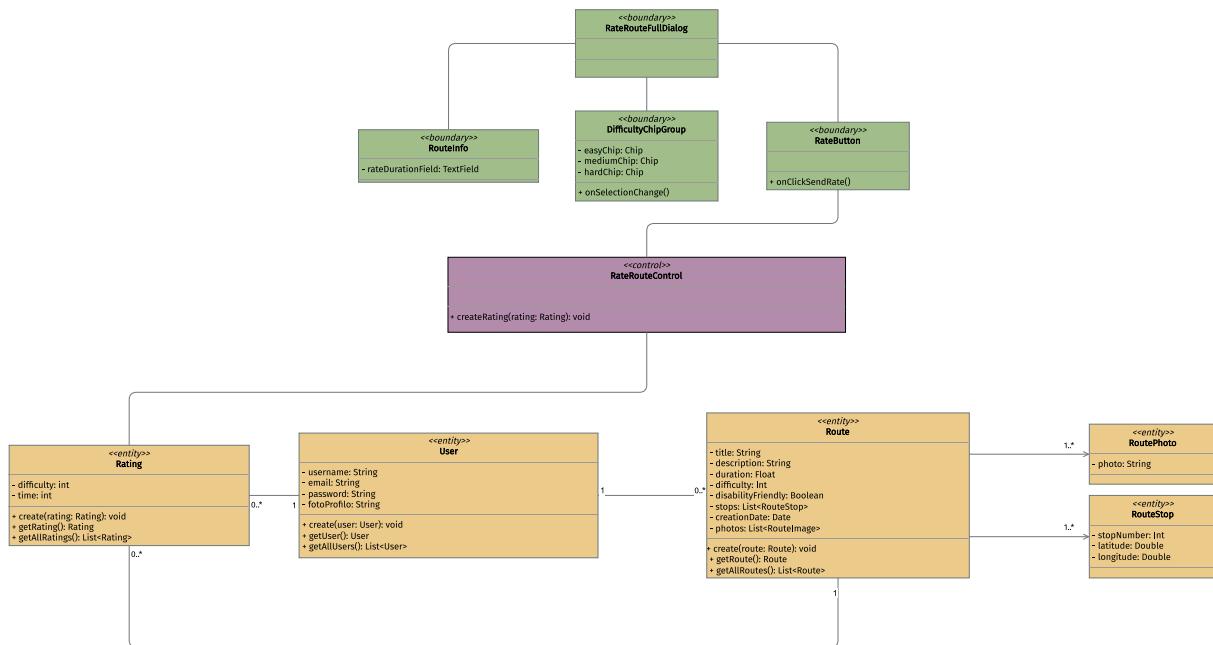


Figura 32: Valuta itinerario

1.13.3 Interazione con un post

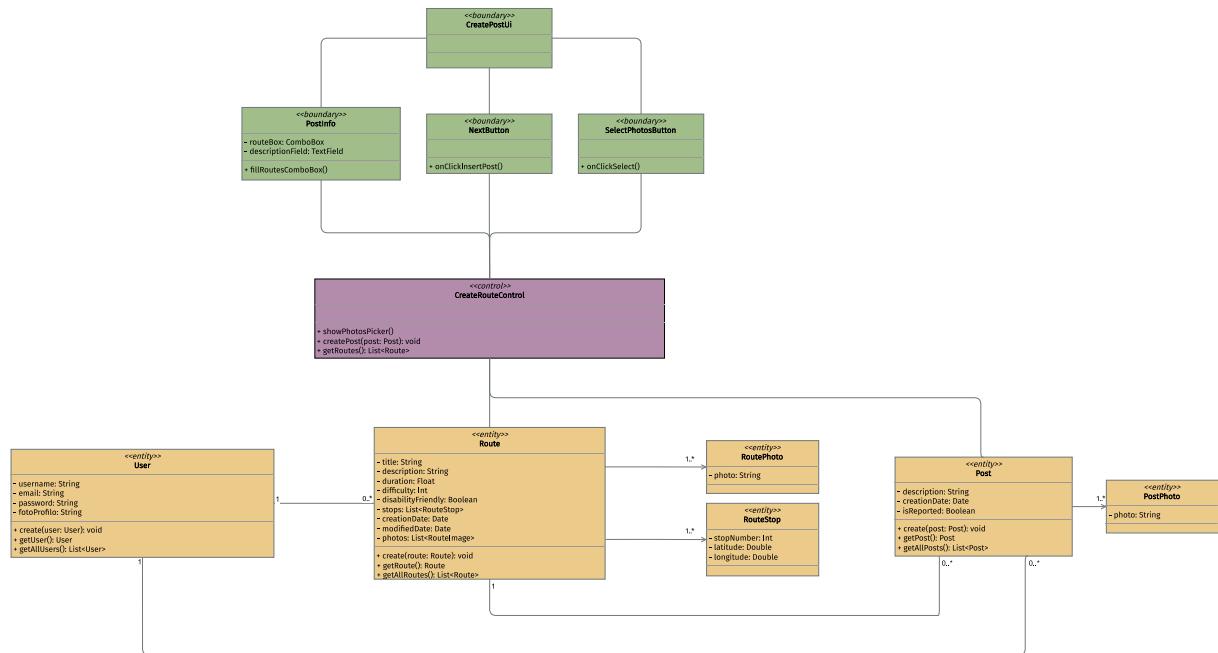


Figura 33: Aggiunta post

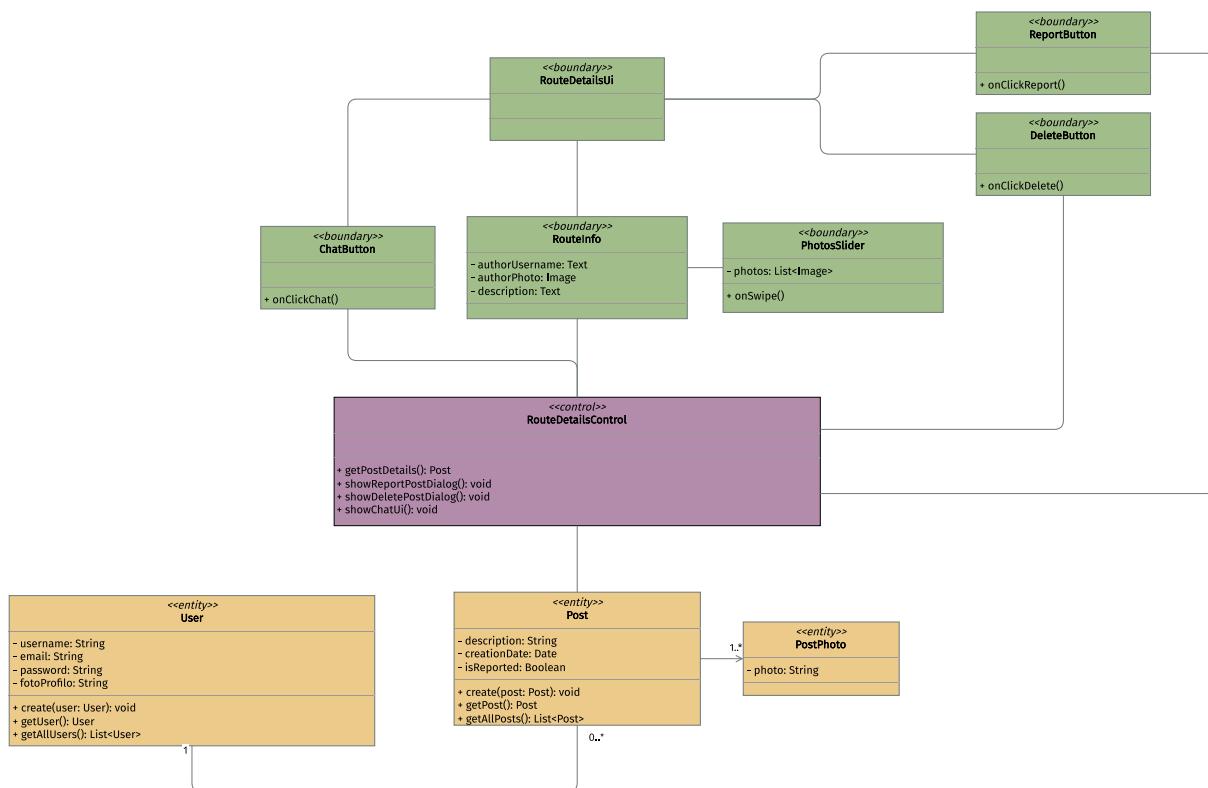


Figura 34: Dettagli post

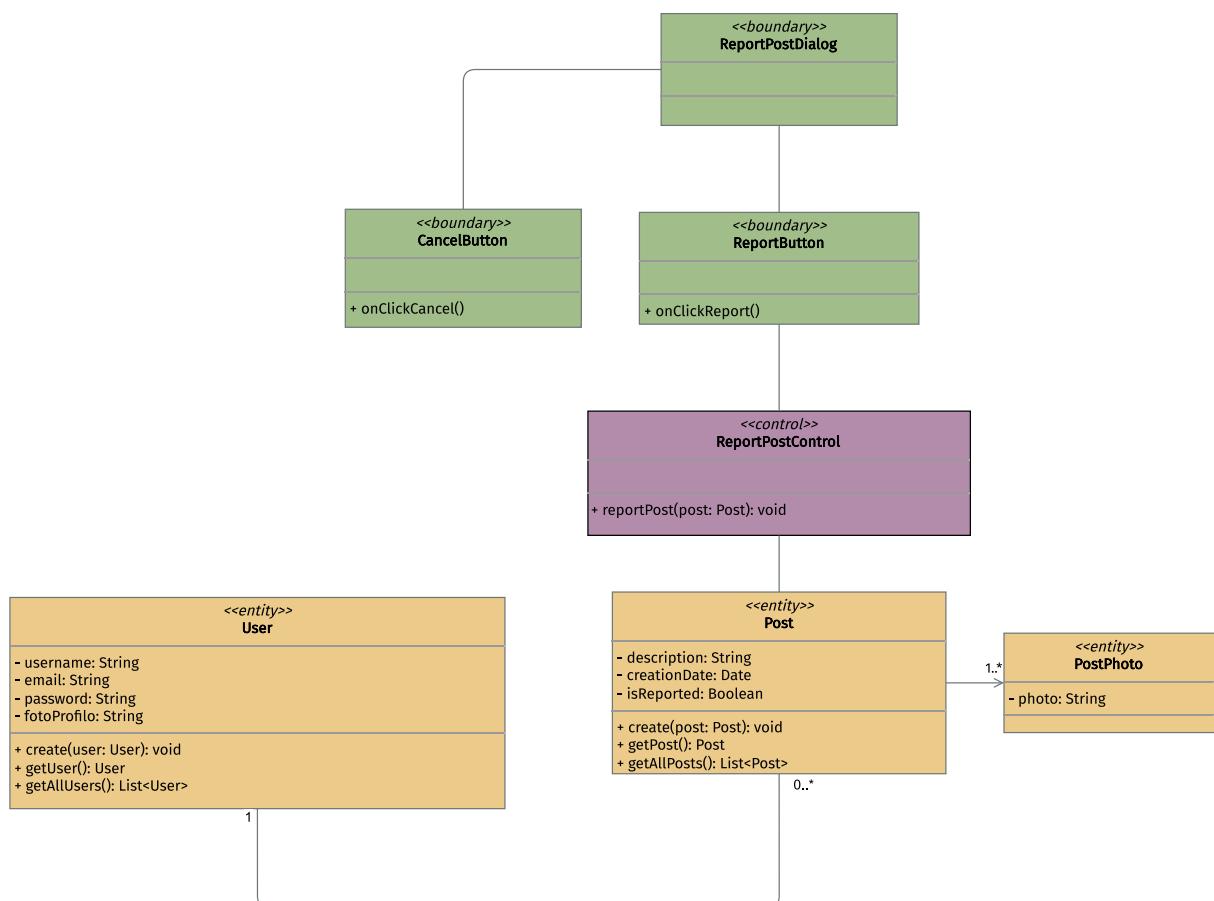


Figura 35: Segnala post

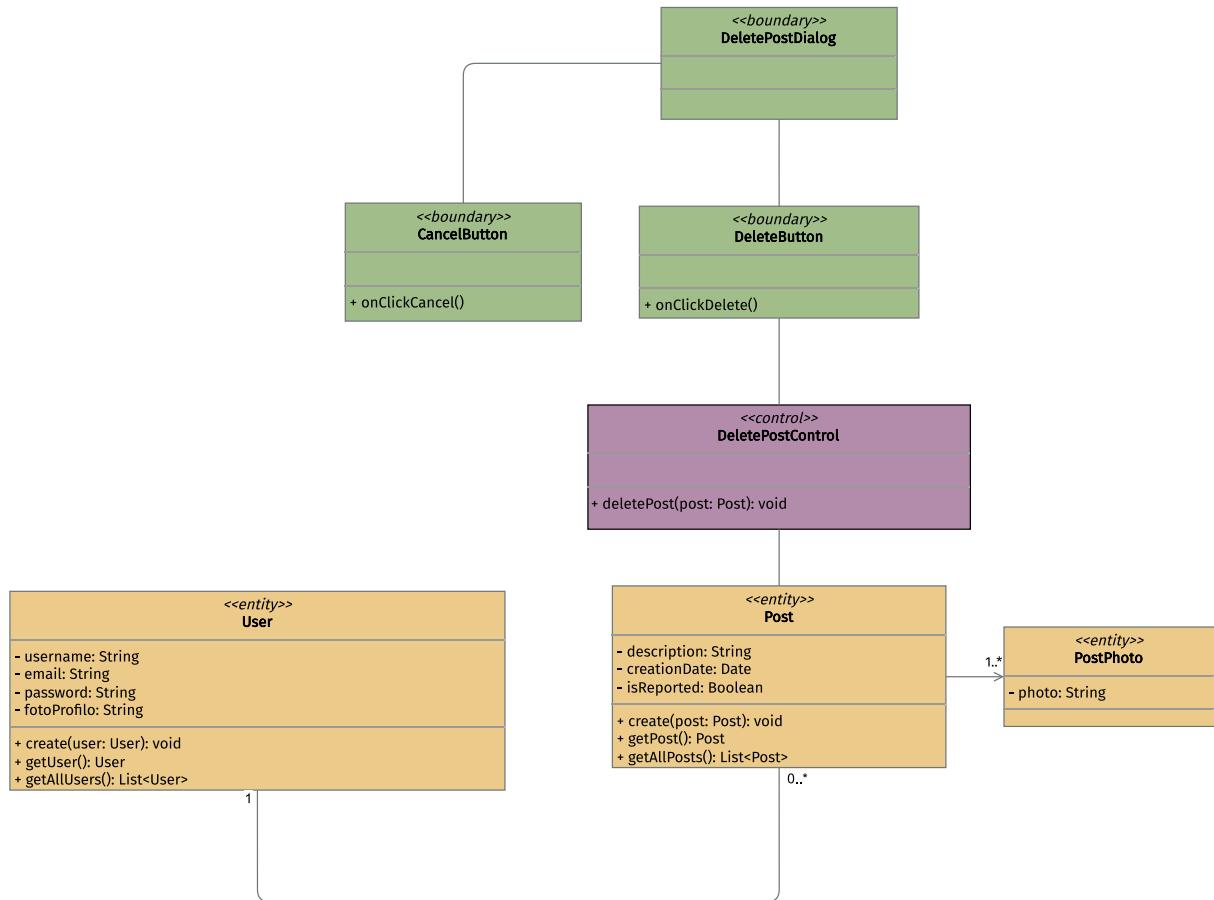


Figura 36: Eliminazione post personale

1.13.4 Interazione con una compilation

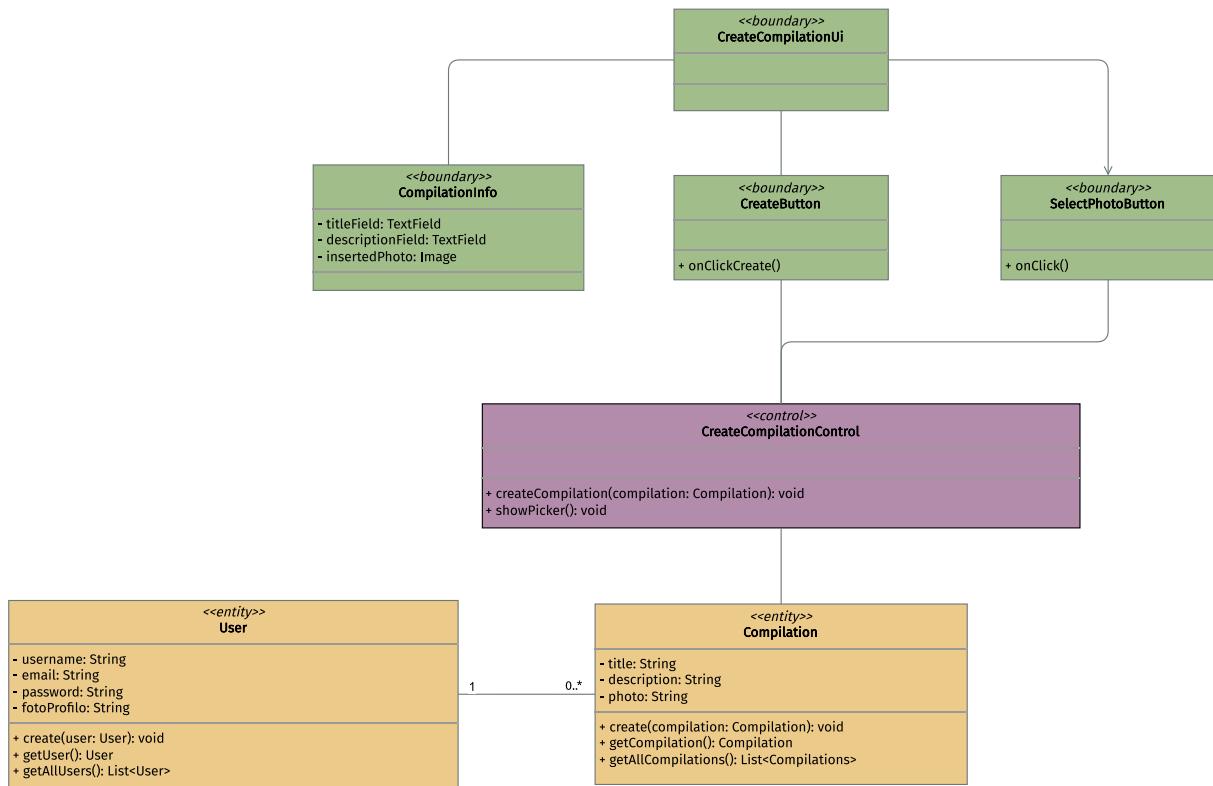


Figura 37: Aggiunta compilation

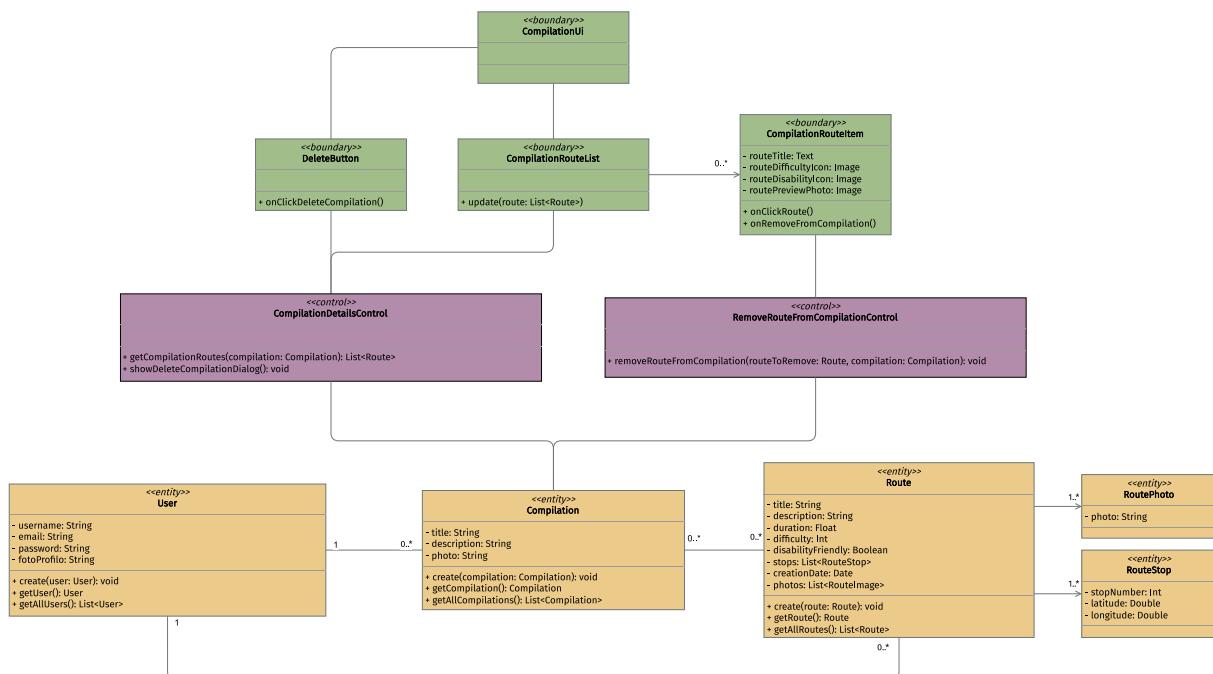


Figura 38: Dettagli compilation e rimozione itinerario da compilation

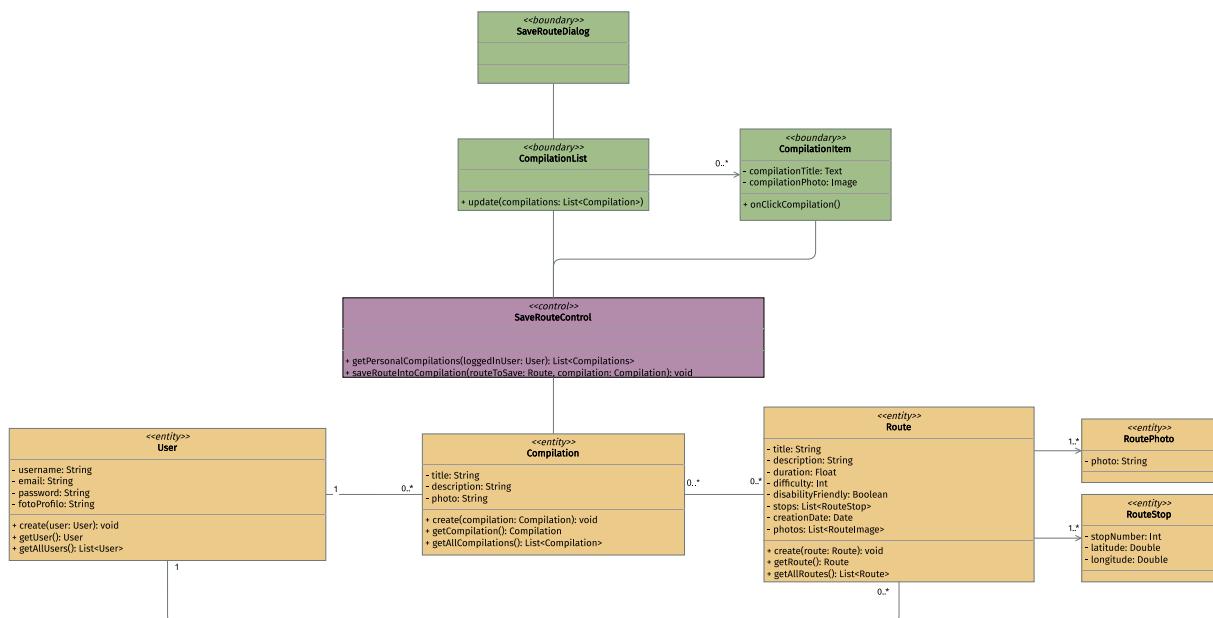


Figura 39: Salvataggio itinerario in compilation

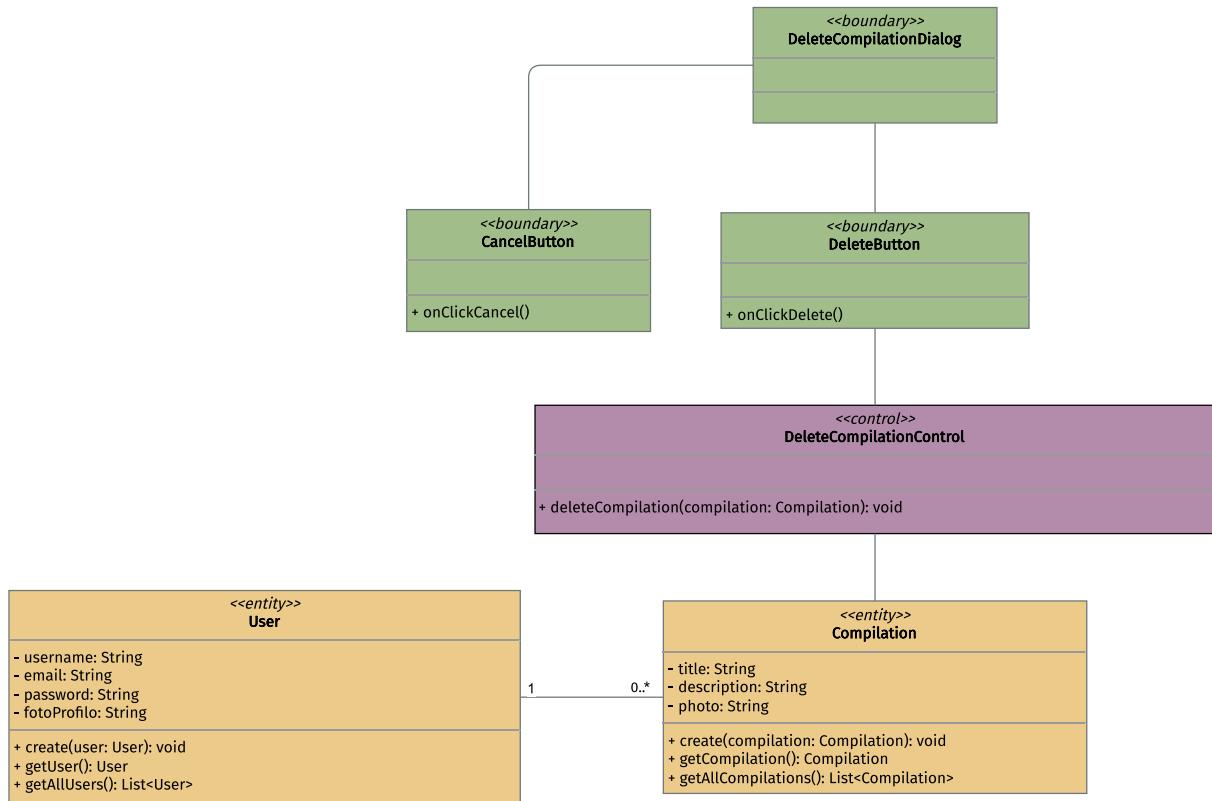


Figura 40: Eliminazione compilation personale

1.13.5 Gestione profilo e interazione con gli utenti

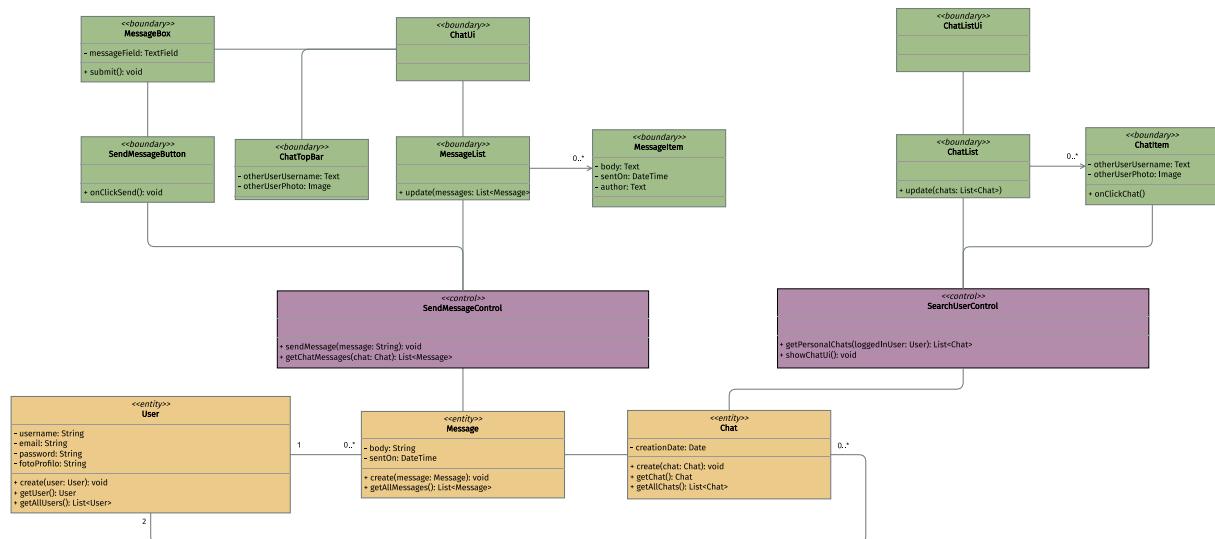


Figura 41: Storico conversazioni e invio messaggio privato

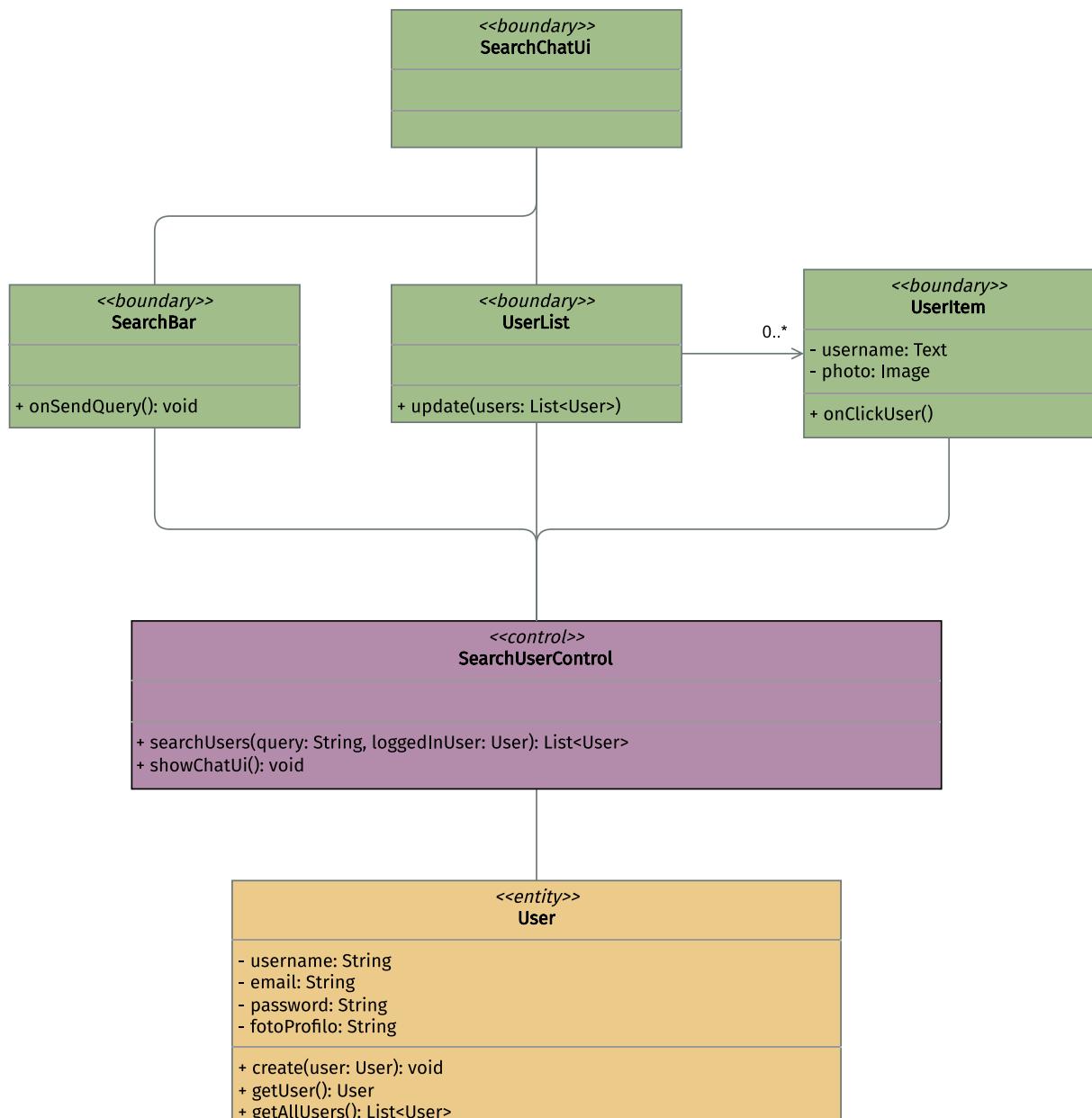


Figura 42: Ricerca destinatario messaggio

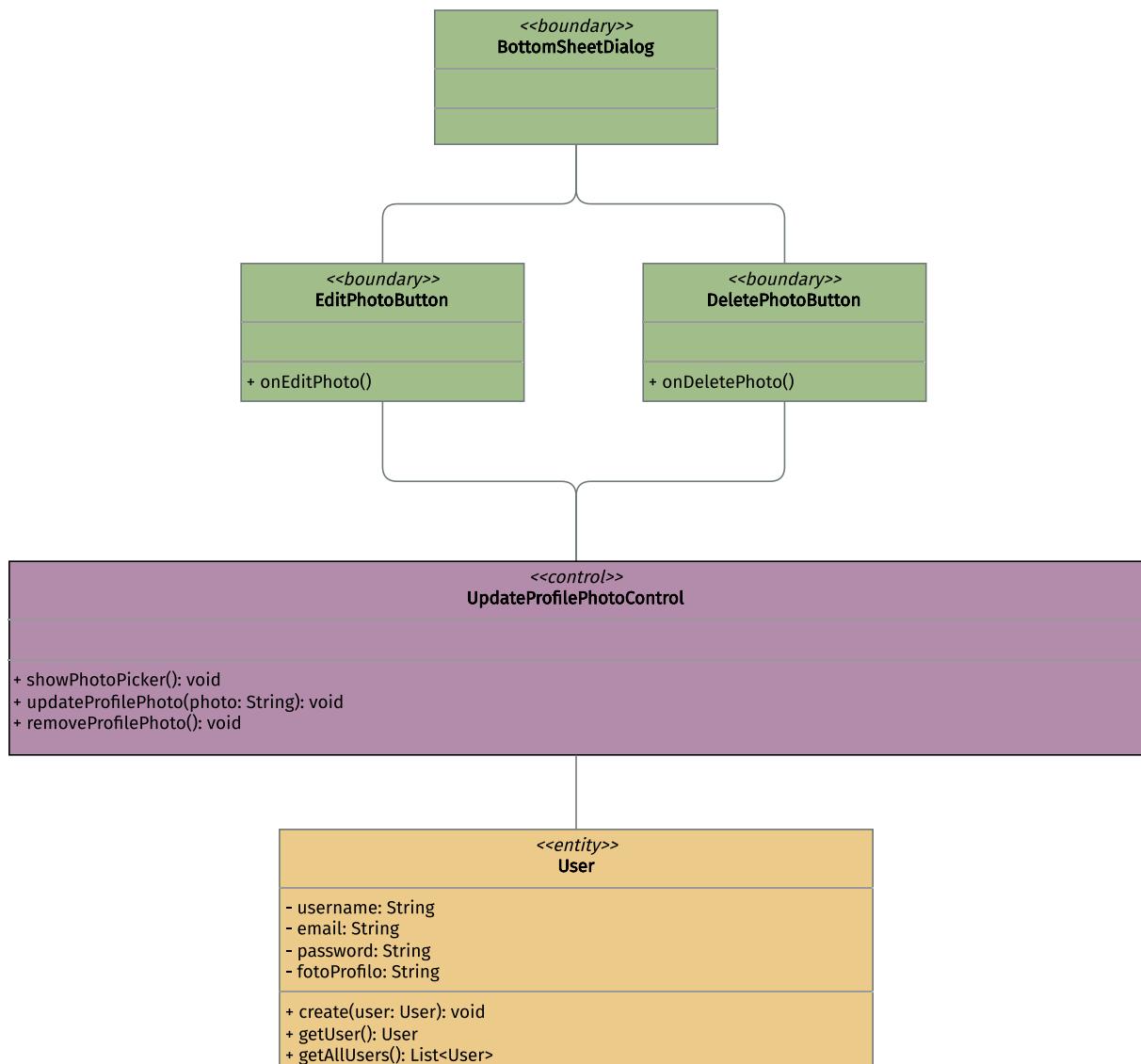


Figura 43: Aggiornamento foto profilo

1.13.6 Funzionalità riservate agli amministratori

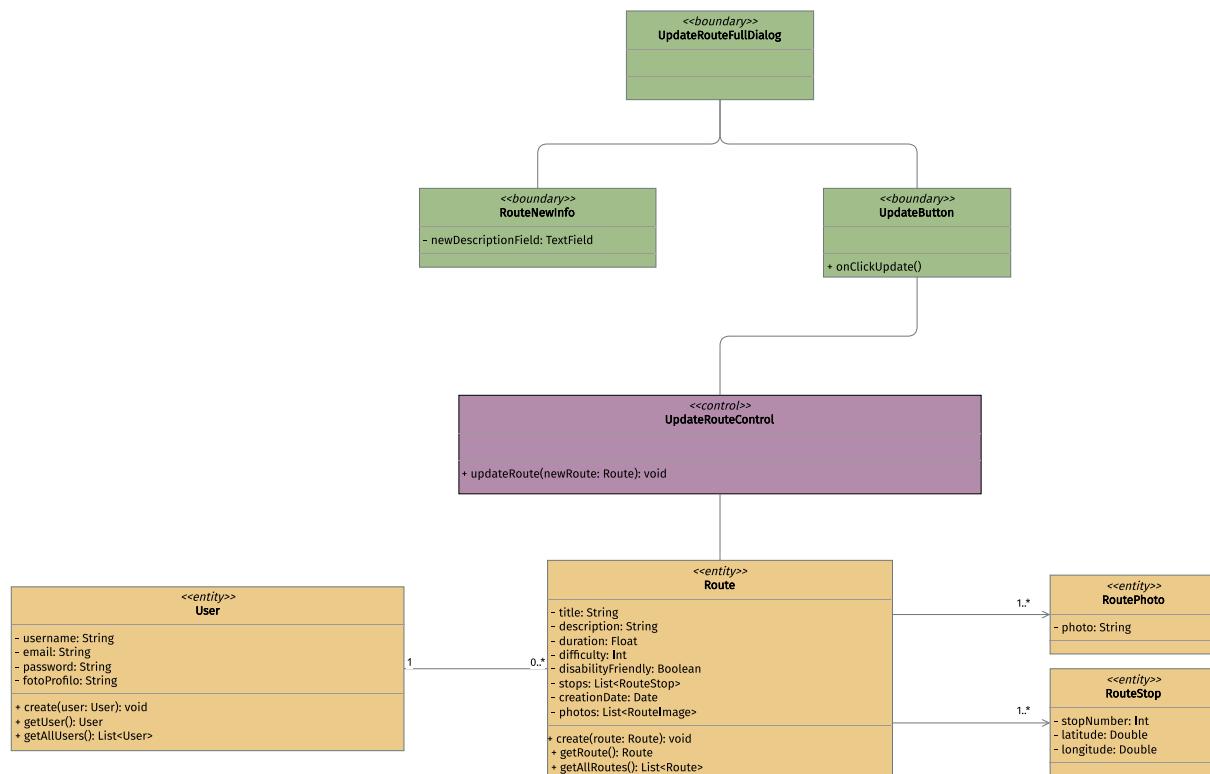


Figura 44: Modifica itinerario

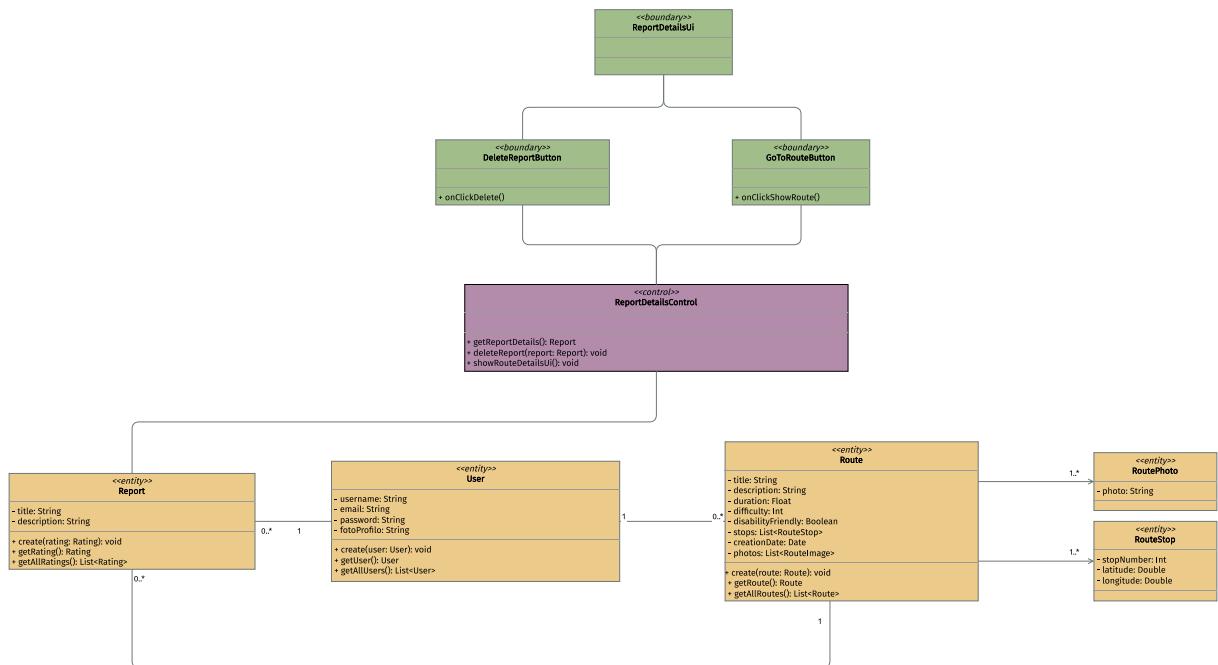


Figura 45: Dettagli segnalazione e eliminazione segnalazione

1.14 Diagrammi di sequenza di analisi

Sono presentati nella seguente sezione i Sequence Diagram relativi a due funzionalità offerte dall'applicazione: la segnalazione di un itinerario e la ricerca di un itinerario.

1.14.1 Segnalazione itinerario

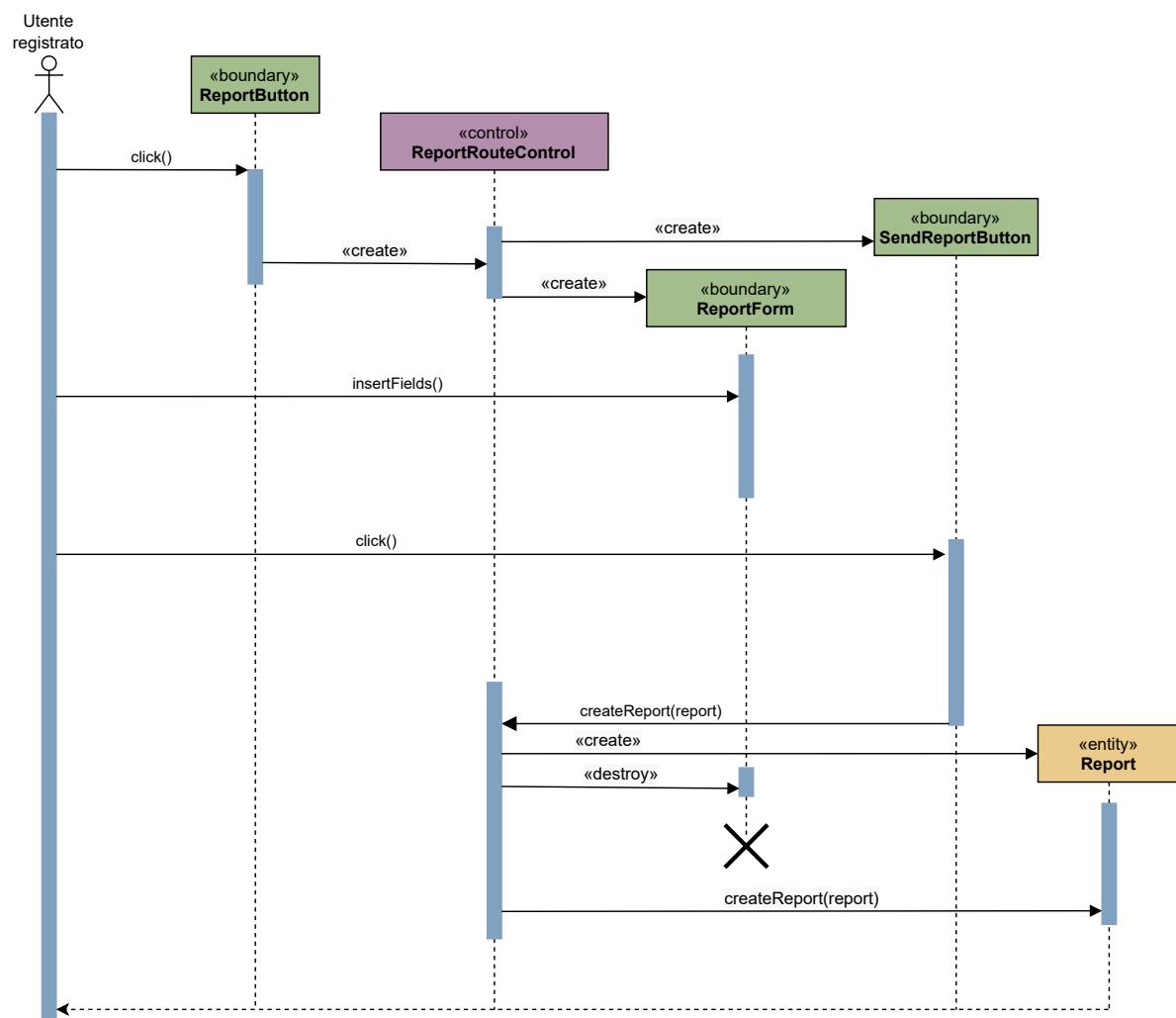


Figura 46: Sequence Diagram 1 - Segnalazione di un itinerario

1.14.2 Ricerca itinerario

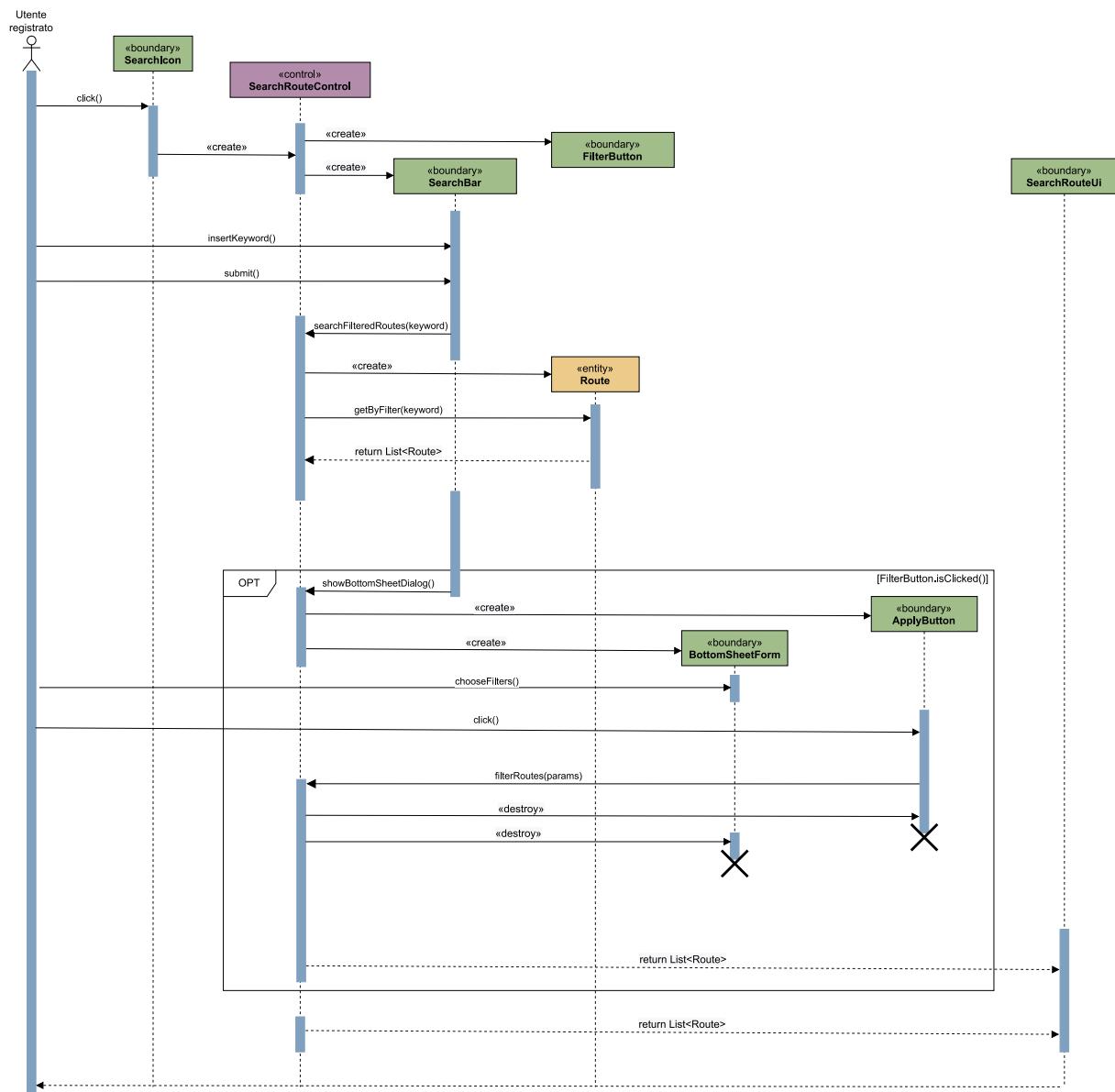


Figura 47: Sequence Diagram 2 - Ricerca di un itinerario

1.15 Diagrammi di attività

1.15.1 Autenticazione

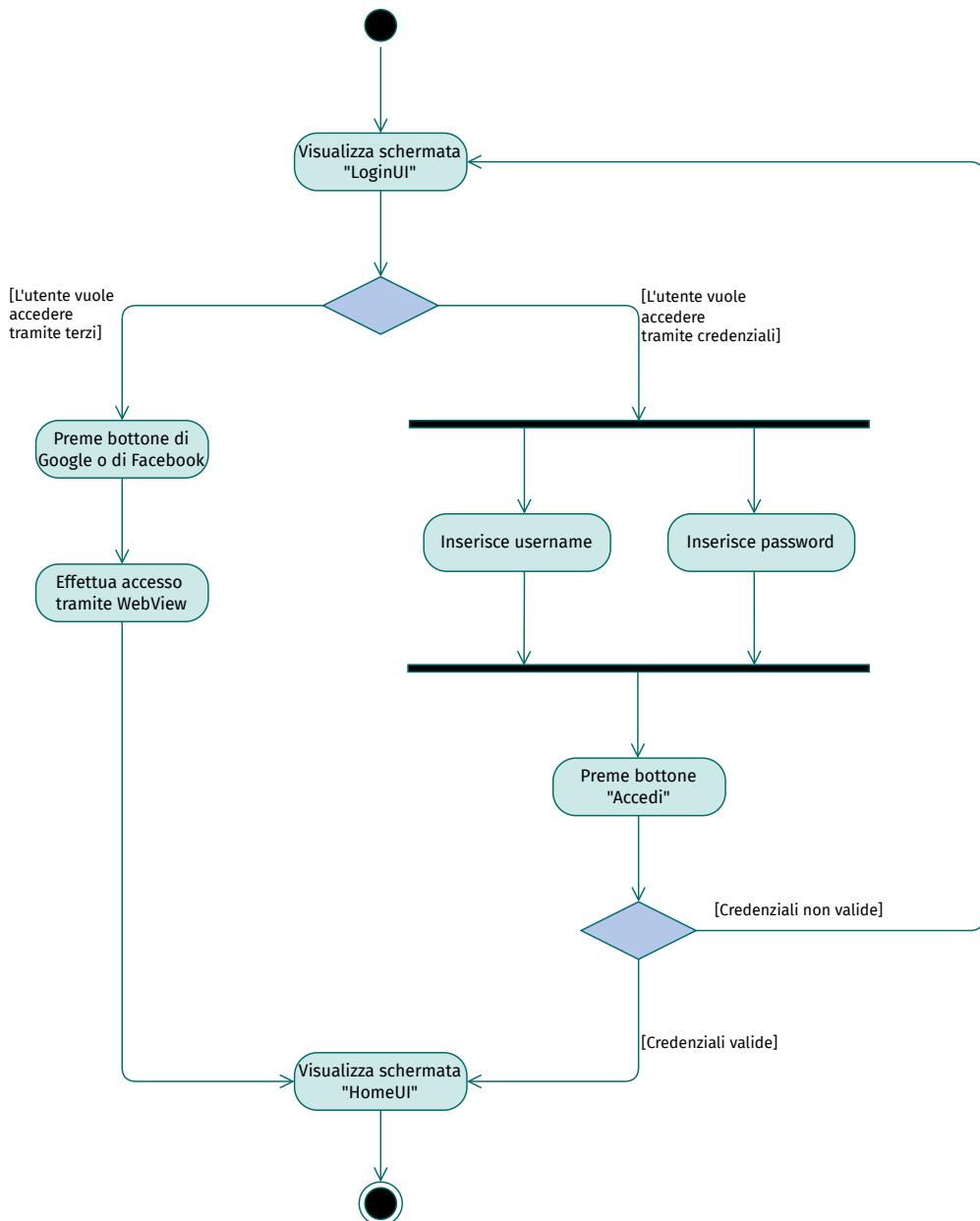


Figura 48: Login utente

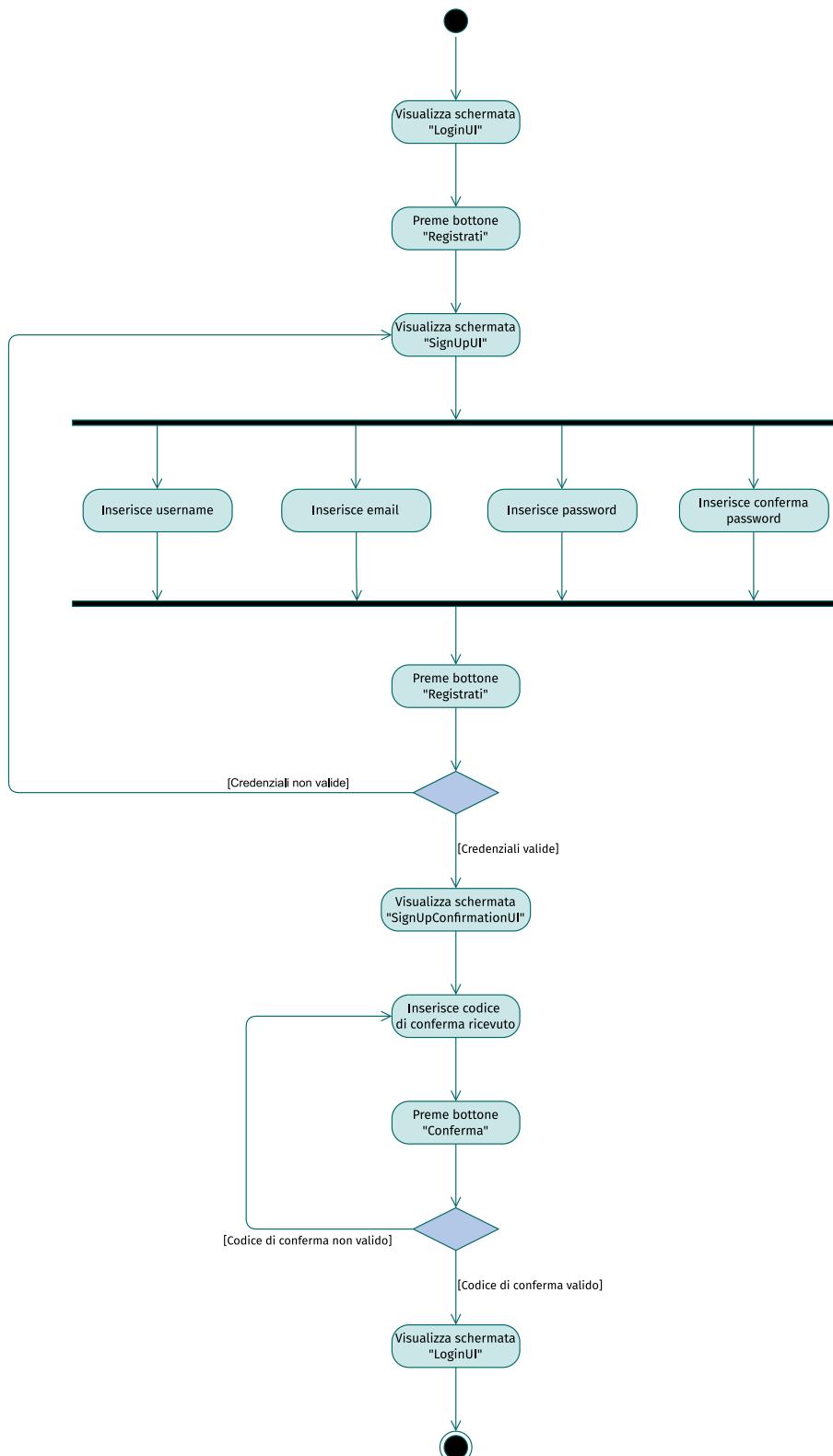


Figura 49: Registrazione utente

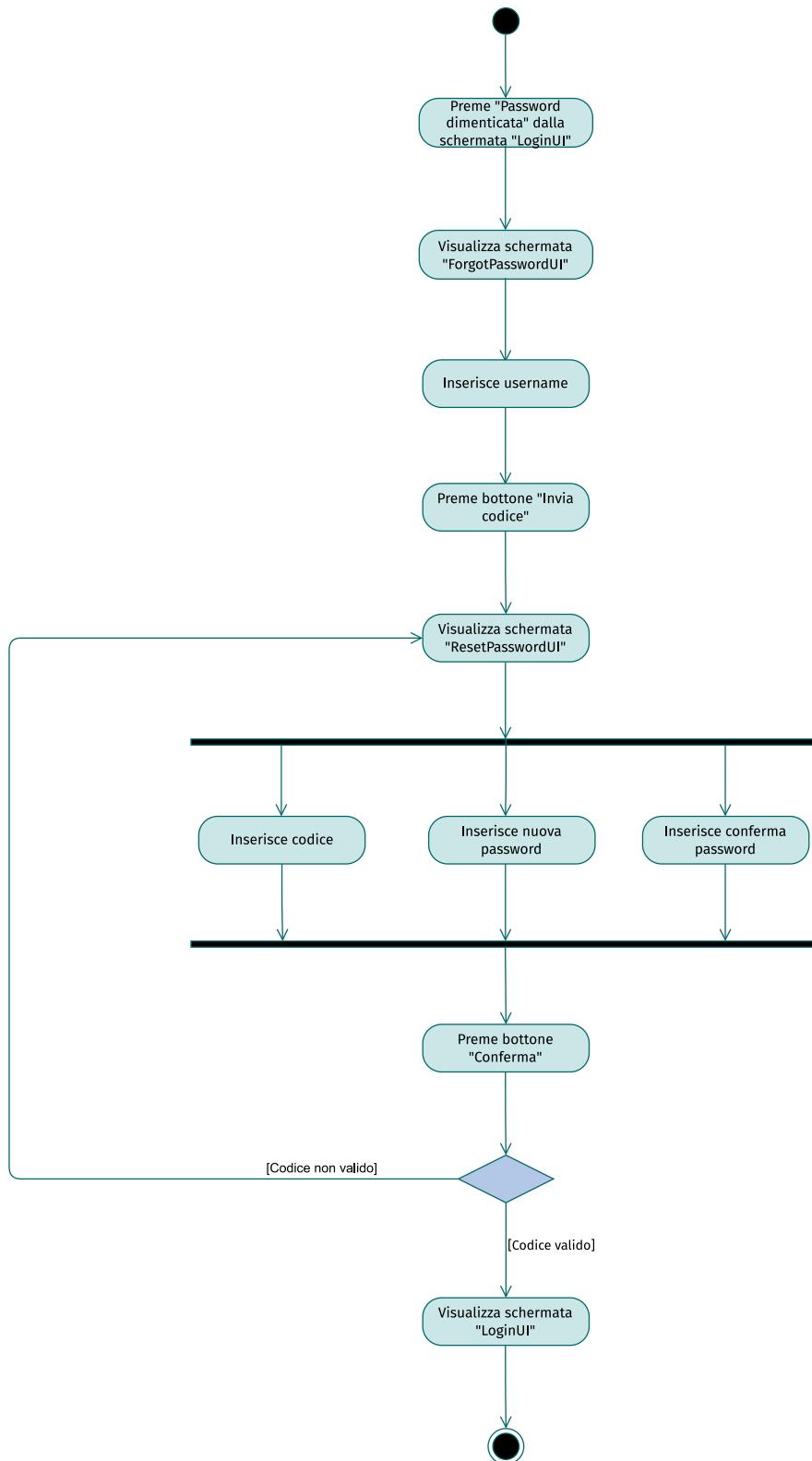


Figura 50: Reset password dimenticata

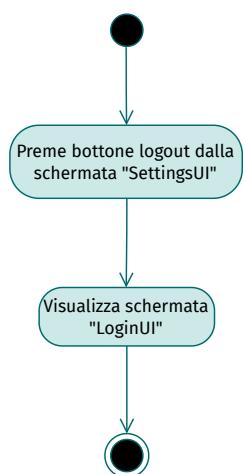


Figura 51: Logout

1.15.2 Interazione con un itinerario

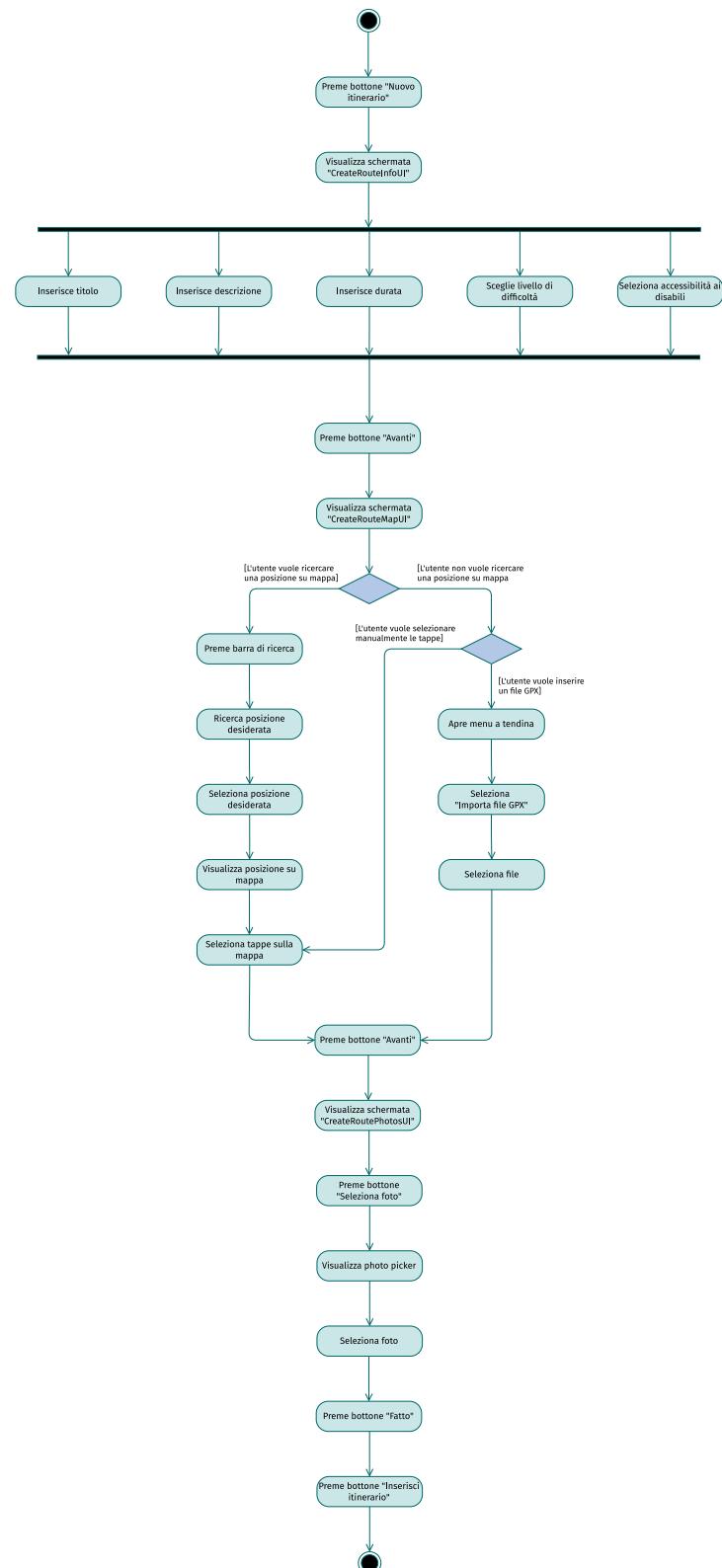


Figura 52: Aggiunta itinerario

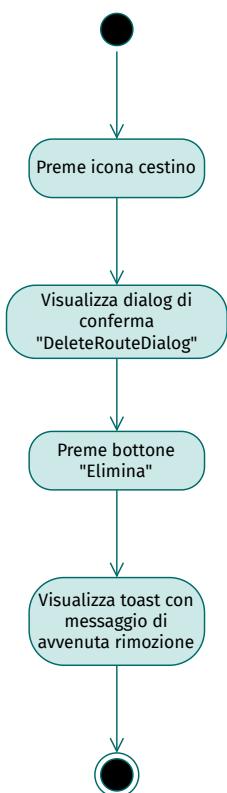


Figura 53: Rimozione itinerario

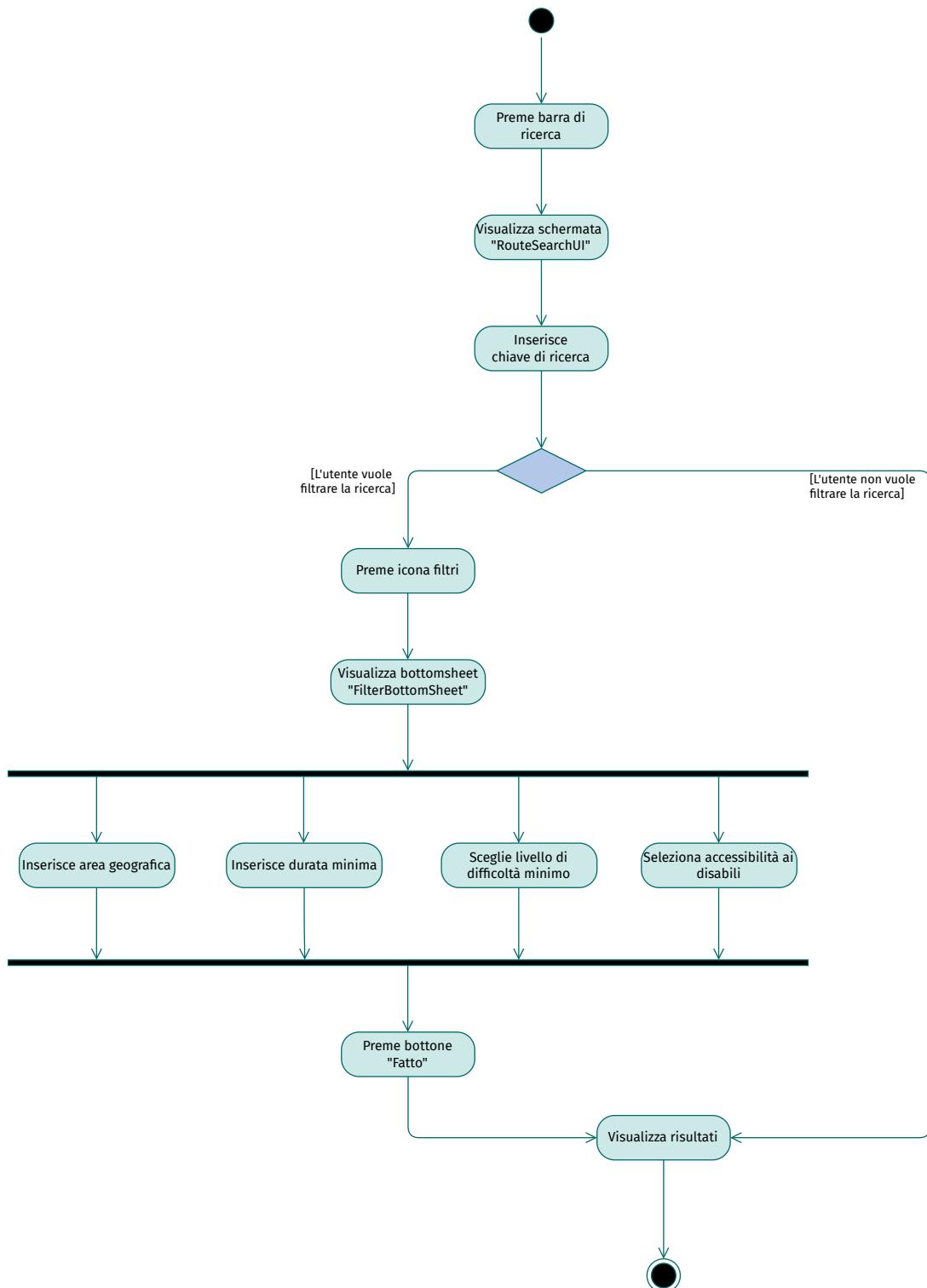


Figura 54: Ricerca itinerario

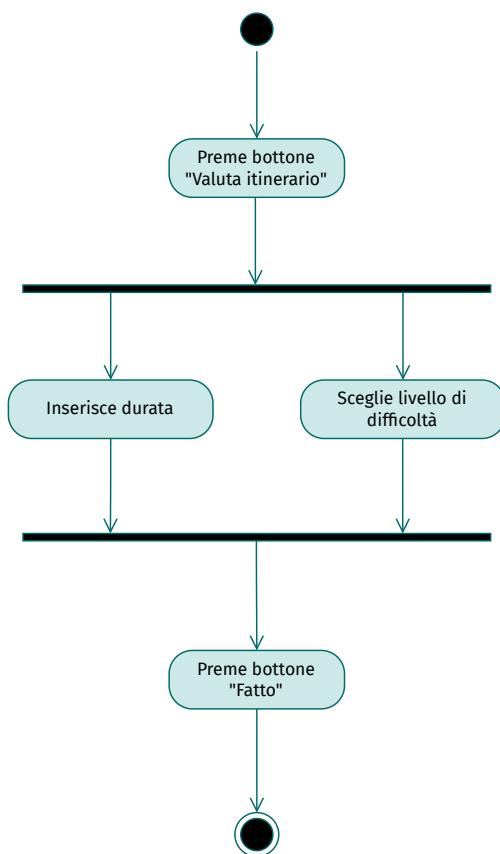


Figura 55: Valutazione itinerario



Figura 56: Salvataggio itinerario

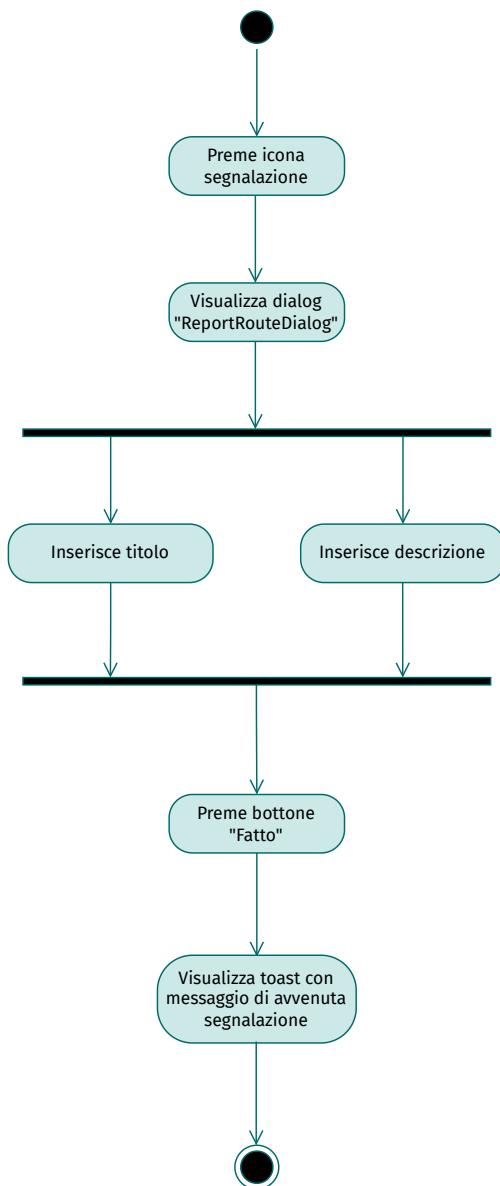


Figura 57: Segnalazione itinerario

1.15.3 Interazione con un post

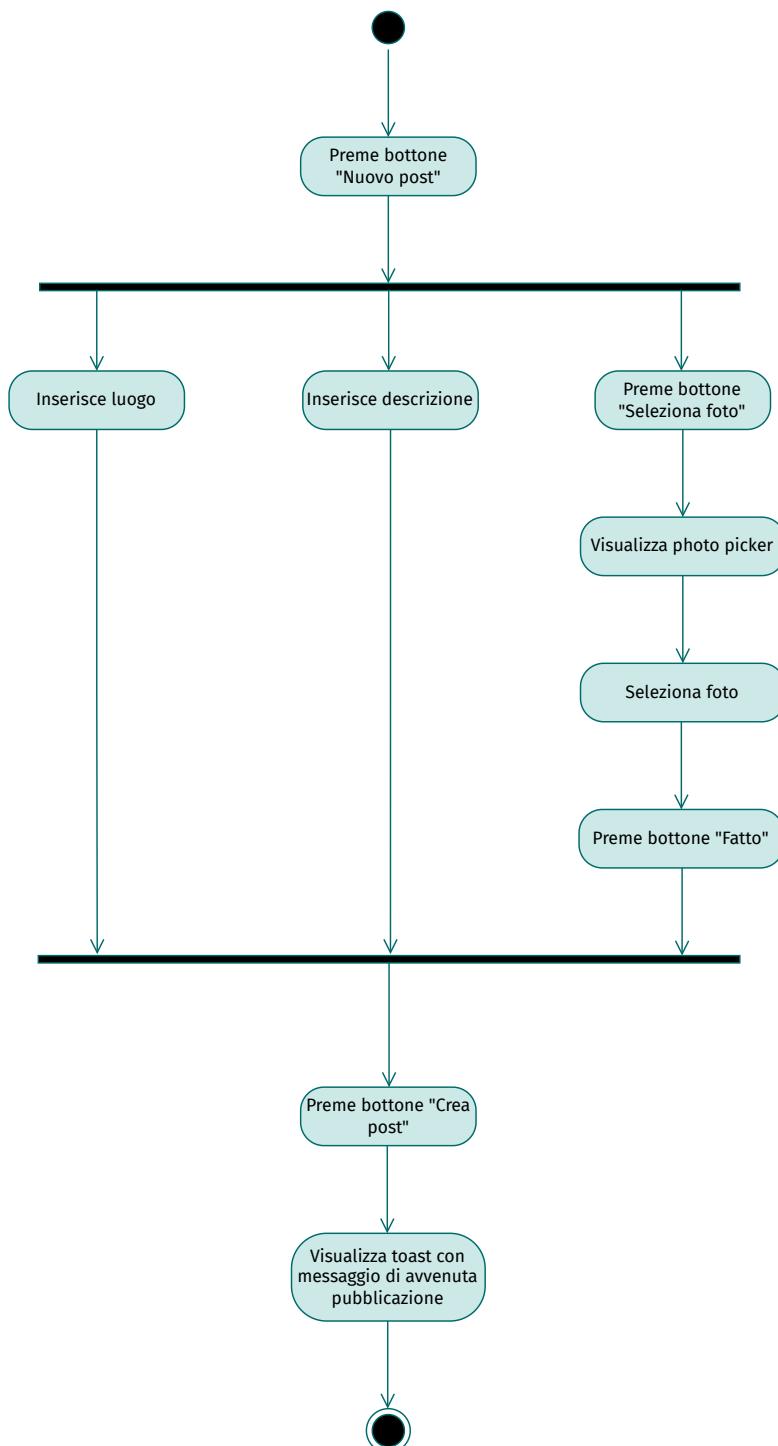


Figura 58: Aggiunta post

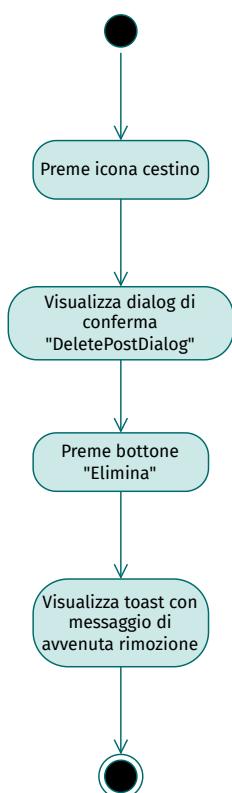


Figura 59: Rimozione post

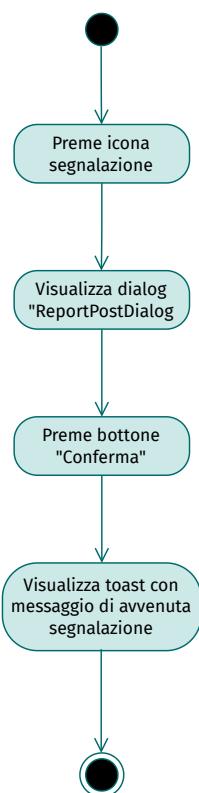


Figura 60: Segnalazione post

1.15.4 Interazione con una compilation

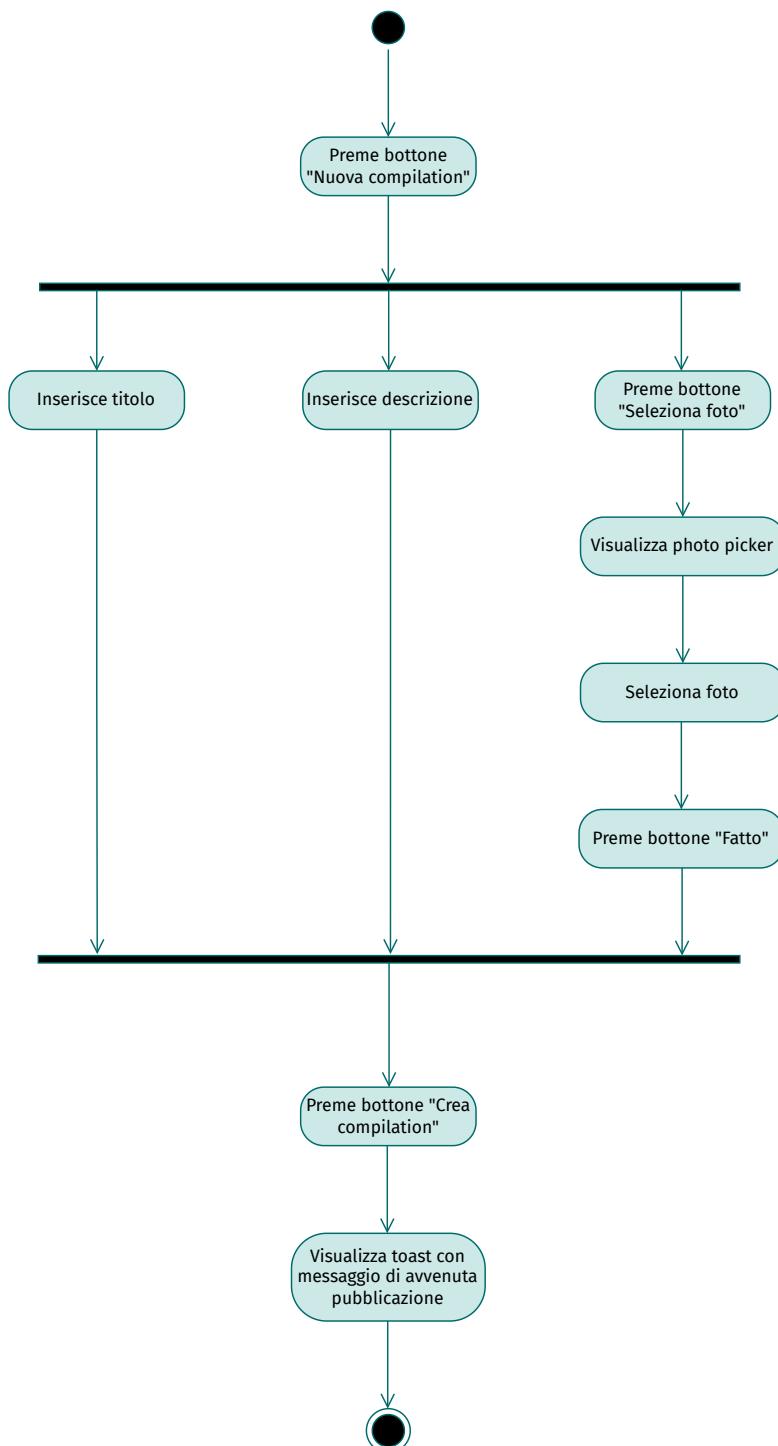


Figura 61: Aggiunta compilation

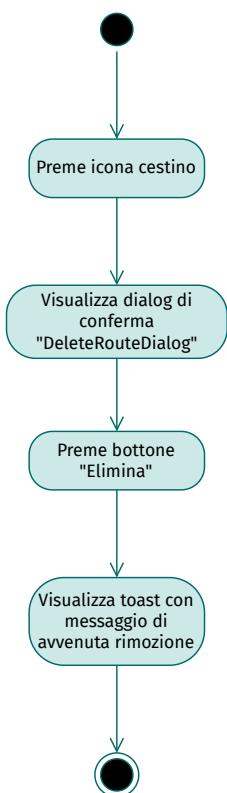


Figura 62: Rimozione compilazione

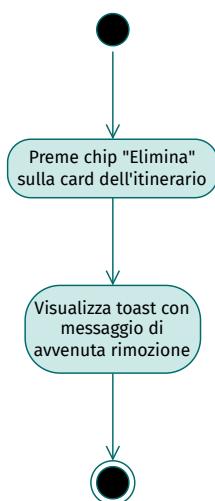


Figura 63: Rimozione itinerario da compilation

1.15.5 Gestione profilo e interazione con utenti

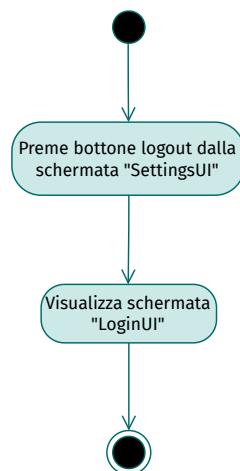


Figura 64: Invio messaggio

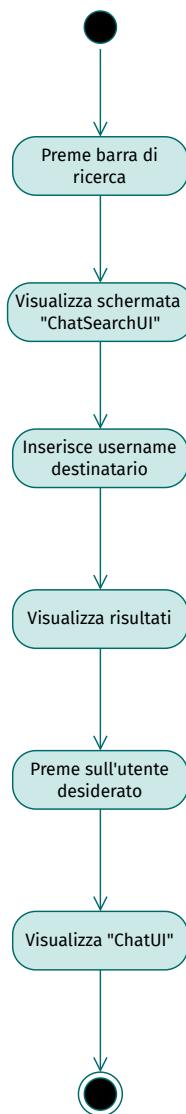


Figura 65: Ricerca destinatario messaggio

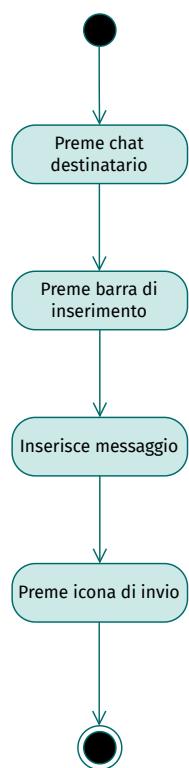


Figura 66: Avvio conversazione con l'autore di un post o itinerario

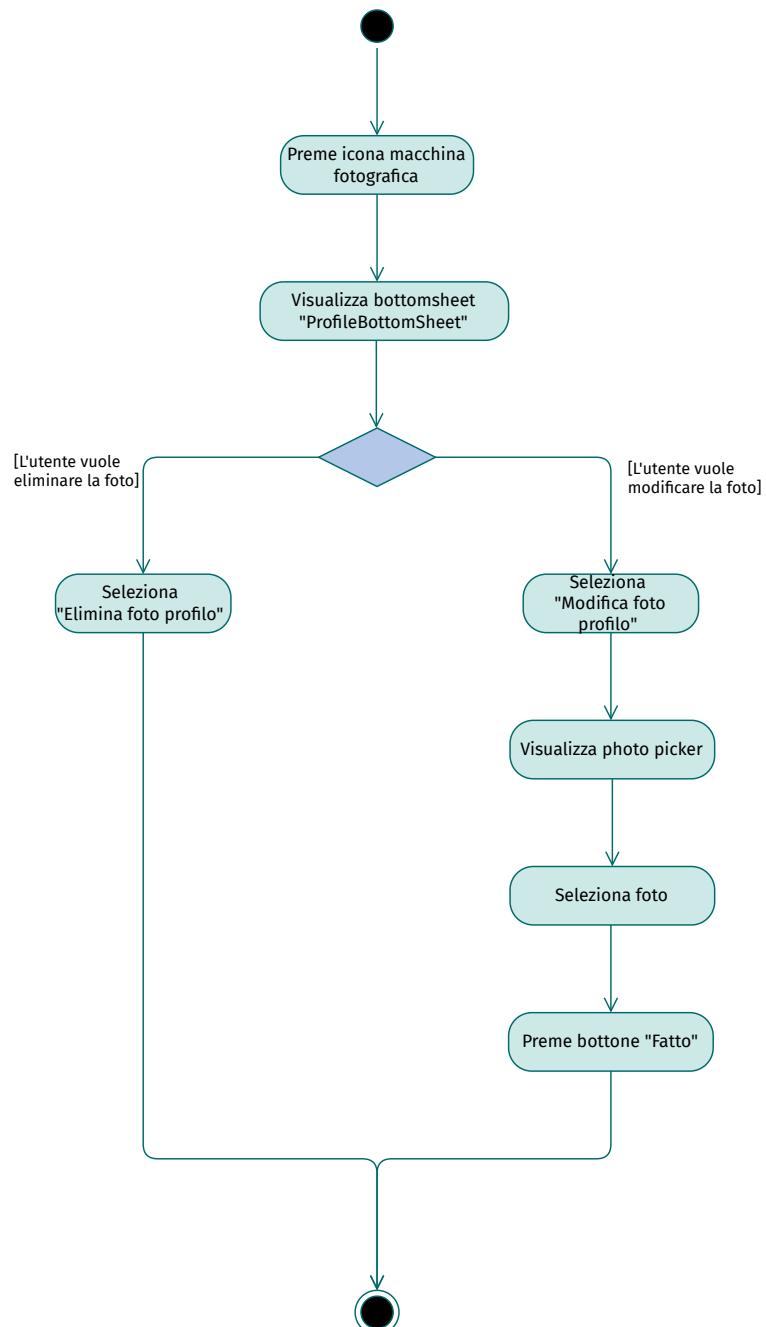


Figura 67: Modifica foto profilo

1.15.6 Funzionalità riservate agli amministratori

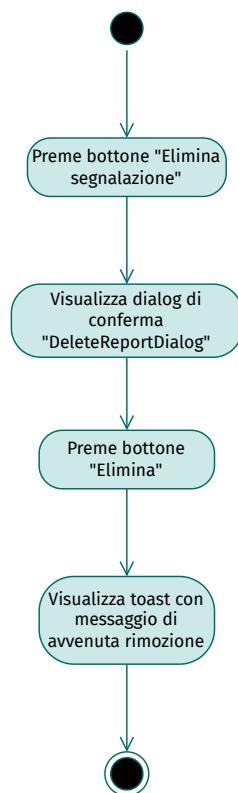


Figura 68: Rimozione segnalazione

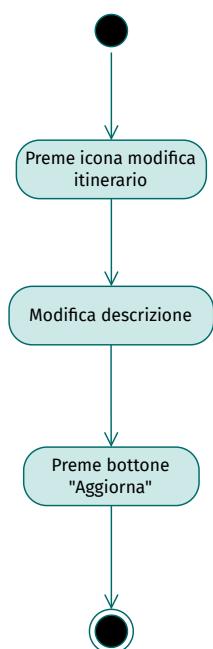


Figura 69: Modifica itinerario

2 Documento di Design del sistema

2.1 Analisi dell'architettura e criteri di design

2.1.1 Diagramma di design del sistema

In questa sezione viene presentato un diagramma realizzato con lo scopo di rappresentare la struttura generale dell'architettura di sistema realizzata; in esso sono stati messi in evidenza i servizi di cui si è usufruito nello sviluppo del software.

Le specifiche di ciascuna scelta implementativa sono dettagliate nelle sezioni successive.

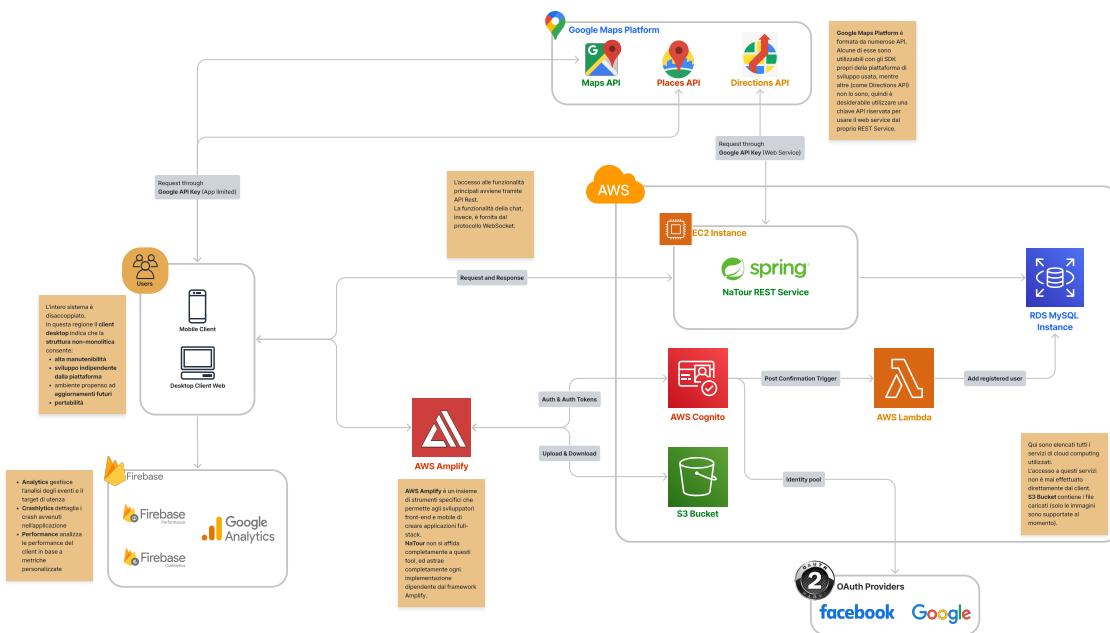


Figura 70: Diagramma del sistema

2.1.2 Software deployment

L'architettura del software realizzato pone le sue fondamenta nel concetto di **cloud computing**.

Il cloud computing, o computazione cloud, consiste nella distribuzione on-demand di risorse IT, con una tariffazione basata sul consumo.

Tra i diversi provider, è stato scelto di usufruire dei servizi offerti dalla piattaforma proprietaria del gruppo Amazon **AWS, Amazon Web Services**. AWS offre infatti cloud services ideali per creare applicazioni in modo scalabile, flessibile e affidabile; esso presenta inoltre notevoli vantaggi, ritenuti come caratteristiche desiderabili per la progettazione di qualunque software.

La scelta della piattaforma AWS è stata compiuta, inoltre, per le seguenti qualità:

- **Agilità** - la possibilità di aumentare a seconda delle necessità risorse quali servizi infrastrutturali, calcolo, storage e database;
- **Elasticità** - la possibilità di evitare l'allocazione anticipata di una quantità maggiore di risorse di quante siano necessarie, così da gestire i picchi nei livelli di attività future;
- **Distribuzione globale di contenuti** - l'infrastruttura AWS offre una copertura globale: fare in modo che le applicazioni siano vicino agli utenti finali riduce la latenza e migliora la loro esperienza.

AWS offre una grande varietà di servizi; ciascuno di essi va incontro a una specifica necessità dello sviluppatore.

Nello sviluppo del software, per implementare determinate funzionalità si è scelto di ricorrere all'utilizzo dei seguenti servizi:

- **Cognito** - Amazon Cognito permette di aggiungere strumenti di registrazione degli utenti, accesso e controllo degli accessi alle app Web e per dispositivi mobili. Esso supporta l'accesso degli utenti tramite l'uso di provider di identità social quali Facebook, Google. Tale risorsa è stata sfruttata realtivamente all'autenticazione utente, tenendo particolarmente in considerazione l'alto fattore di sicurezza che essa conferisce;
- **EC2** - l'**Elastic Compute Cloud** (Amazon EC2) fornisce capacità di calcolo scalabile in AWS Cloud. Esso offre ambienti di elaborazione virtuale, noti come *istanze*, varie configurazioni di CPU, memoria, archiviazione e capacità di rete per le istanza note come *tipi di istanza*. La scelta di tale servizio è stata finalizzata al deploy dell'applicativo Spring, nella realizzazione del Rest Service.
- **RDS** - Amazon RDS ha permesso la configurazione e l'utilizzo del database relazionale alla base del software prodotto;
- **S3** - il **Simple Storage Service** (Amazon S3) permette l'archiviazione di oggetti in modo scalabile. Esso è stato adoperato in merito alla preservazione permanente dei file immagine caricati dagli utenti;

- **Lambda** - AWS Lambda è un servizio di calcolo basato su eventi serverless. La scelta del servizio Lambda è stata incentivata dalla sua possibilità di integrazione con il servizio **Cognito**, e dettata dalla volontà di conferire *consistenza* al pool utenti del sistema. Nello specifico, si è sfruttata la possibilità offerta dal servizio di creare dei *trigger* (in particolare trigger di *post conferma*), al fine di evitare - in seguito al processo di registrazione - la possibile presenza di inconsistenze tra database e registrazioni effettivamente portate a termine.

Si è ritenuto inoltre fondamentale l'utilizzo del framework **Amplify**.

In particolare, Amplify è stato impiegato per realizzare:

- Un servizio di autenticazione particolarmente sicuro tramite, come citato, il servizio *Cognito*;
- L'archiviazione di file (nella prima versione del software solo di file immagine) tramite il servizio *S3*.

È importante specificare che il framework Amplify consente anche di creare back-end o applicativi serverless. Ciononostante, si sottolinea che nella realizzazione del software è stata fatta la scelta di utilizzarlo *solo* in funzione dei servizi da essi offerti, soprattutto in relazione alla deprecazione dell'SDK standard AWS per Android.

2.1.3 Google Maps Platform

Una delle caratteristiche principali del software risulta essere la forte componente geolocalizzata degli itinerari presenti in piattaforma.

Gli utenti, infatti, nell'inserimento e nella visualizzazione dei sentieri si trovano ad interfacciarsi direttamente con mappe interattive. Per garantire un'esperienza ottimale da questo punto di vista sono stati utilizzati i servizi della piattaforma Google Maps.

La **Google Maps Platform** è un insieme di API e SDK che permette di integrare in applicazioni mobile Google Maps, e di recuperare dati da esso stesso. L'esperienza utente con le funzionalità sopracitate è stata supportata dall'utilizzo delle seguenti API:

- MapsAPI - per la visualizzazione interattiva di mappe statiche e dinamiche;
- PlacesAPI - per il recupero di informazioni sui posti tramite richieste HTTP;
- DirectionsAPI - per il calcolo del percorso tra diverse tappe; utilizzano una richiesta HTTP per ritornare le direzioni tra località in formato JSON o XML. Le DirectionsAPI, in quanto web service, sono state integrate nel Rest Service proprio del software.

2.1.4 Architettura del REST Service

Nella realizzazione del REST Service è stato scelto di utilizzare il framework **Spring**. Spring fornisce un'*infrastruttura di supporto* per lo sviluppo di applicazioni: esso prevede una *modularizzazione* dell'architettura come segue:

- Presentation layer - strato più esterno, che gestisce la presentazione del contenuto e l'interazione con l'utente;

- Business logic layer - strato centrale, che gestisce la logica;
- Data access layer - strato più interno, che gestisce il recupero di dati dalle diverse sorgenti.

Ciascuno di questi strati dipende da quello sottostante per far sì che l'applicazione funzioni. In altre parole, lo strato di presentazione comunica con quello della business logic, che a sua volta comunica con lo strato di data access. Ogni strato ha quindi bisogno di questa *dipendenza* per eseguire il proprio compito correttamente

La scelta di utilizzare il framework Spring è stata fatta in relazione alla volontà di ottenere un *loose coupling*, ossia un accoppiamento largo. Senza l'utilizzo di Spring, ci sarebbe stata la possibilità che il codice avesse potuto causare *tight coupling*, che non è considerato essere una buona pratica di programmazione. Il loose coupling è ideale, in quanto le componenti largamente accoppiate sono indipendenti: ciò implica che, a seguito di possibili cambiamenti futuri, questi non influenzano le altre componenti.

Al cuore del framework Spring si trova la **dependency injection**.

La dependency injection è un pattern di programmazione che permette agli sviluppatori di costruire architetture disaccoppiate. Ciò vuol dire che Spring comprende le *annotazioni* poste in capo alle classi; in seguito alla creazione di un'istanza, dunque, il framework si assicura che le istanze siano state create con le opportune dependencies.

Nello specifico, per l'implementazione del software, è stata utilizzata un'estensione del framework Spring, chiamata **Spring Boot**.

Spring Boot ha delle caratteristiche specifiche che rendono la gestione dell'applicazione più semplice.

Tra alcuni dei vantaggi che hanno portato alla scelta di Spring Boot, si ricordano:

- Creazione di applicazioni Spring stand-alone;
- Dotazione di dependencies "starter" per semplificare la configurazione di build;
- Configurazione automatica di Spring e di librerie di terze parti, ove possibile;
- Nessuna generazione di codice e nessun requisito per la configurazione XML;
- Inclusione di un web server embedded (nello specifico *Apache Tomcat*) senza la necessità di ulteriori configurazioni.

Di seguito viene presentato un diagramma che dettaglia la struttura e il funzionamento del REST Service.

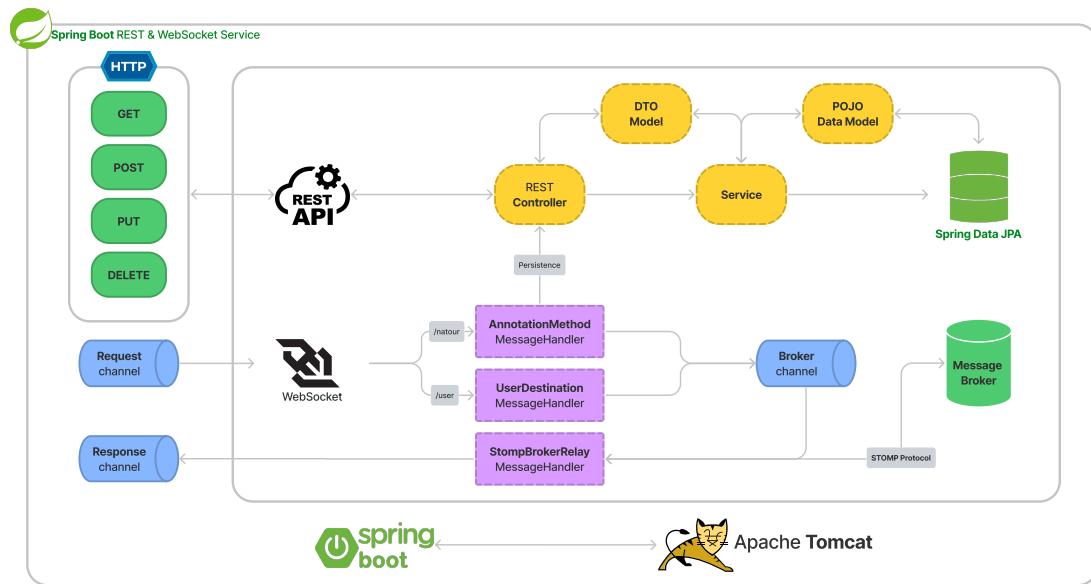


Figura 71: REST + WS Service realizzato con Spring Boot

2.2 Architettura dell'applicativo

L'intera architettura del software è stata progettata seguendo le linee guida della cosiddetta **Clean Architecture**.

Il concetto di architettura *pulita* si basa sui principi enunciati da Robert C. Martin nel libro "Clean Architecture". In seguito alle informazioni acquisite dalla lettura del libro, infatti, questo approccio è stato ritenuto quello più valido.

L'idea chiave è quella di utilizzare il principio di inversione di dipendenza per porre dei boundaries tra componenti di alto livello e componenti di basso livello. Ciò crea un'architettura *plug-in*, che conferisce al sistema un'elevata flessibilità e un'elevata manutenibilità. Un'architettura *pulita* inizia da un codice *pulito*. Classi pulite derivano da componenti puliti, che, a loro volta, determinano un *sistema pulito*.

È stato quindi ritenuto di fondamentale importanza applicare i principi della Clean Architecture e del Clean Code in maniera uniforme e costante, nell'implementazione di qualsiasi componente.

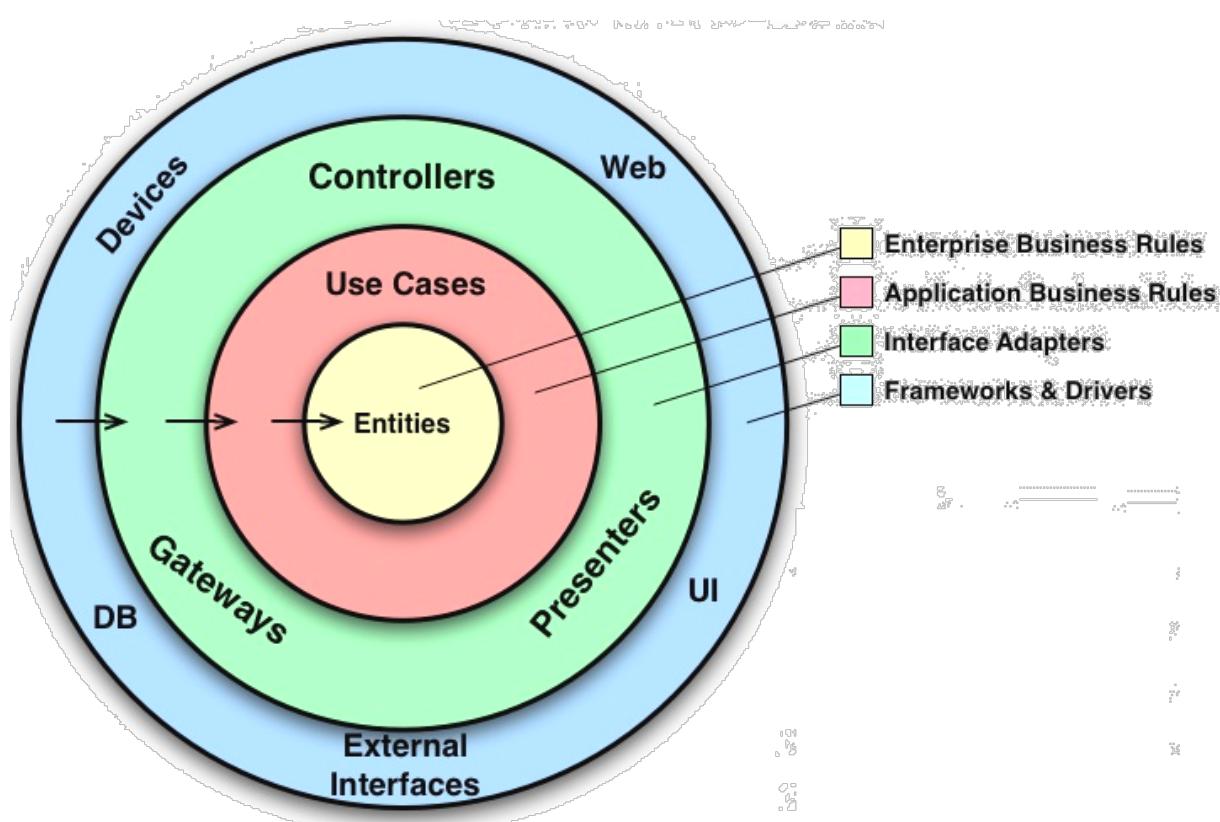


Figura 72: Clean Architecture

Il concetto di *Clean Code* sfruttato nello sviluppo del software segue i principi **SOLID**. Tali principi sono stati applicati in modo da ottimizzare l'architettura e ciascuna delle sue componenti, le quali risultano dunque essere caratterizzate dal:

- **Single responsibility principle** – una classe dovrebbe avere uno ed un solo motivo per cambiare;
- **Open-closed principle** – una classe dovrebbe essere aperta all'estensione ma chiusa alle modifiche;

- **Liskov's substitution principle** – gli oggetti di un programma dovrebbero essere sostituibili con istanze dei propri sottotipi senza alterare la correttezza del programma stesso;
- **Interface segregation principle** – più interfacce client-specific sono meglio di un'unica interfaccia general-purpose;
- **Dependency inversion principle⁴** – i moduli di alto livello non dovrebbero dipendere da moduli di basso livello; entrambi dovrebbero dipendere da *astrazioni*.

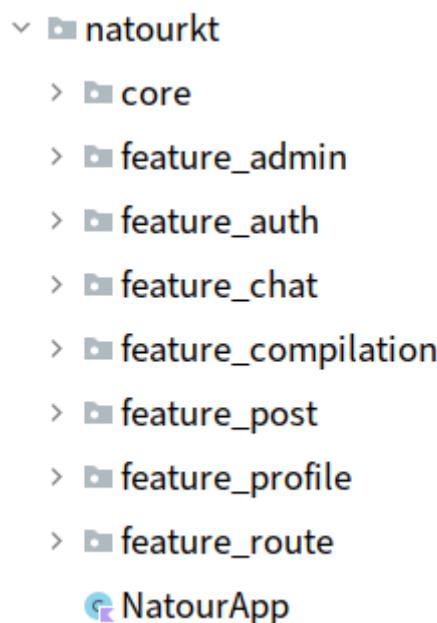
Il concetto di Clean Architecture si basa sulla *modularità* dell'architettura stessa.

Nel caso della progettazione del software prodotto è stata seguita una suddivisione del seguente tipo:

- Presentation layer – responsabile della presentazione a schermo e della gestione delle interazioni degli utenti;
- Domain layer – rappresentazione formale del dominio applicativo
- Data layer – contenitore delle implementazioni delle sorgenti di dati e dei repository, i quali coordinano i dati provenienti da esse.

È bene sottolineare, però, che nell'effettiva implementazione dell'applicativo non è stato pedissequamente seguito lo schema appena proposto.

È stato ritenuto più opportuno adottare un approccio *per funzionalità*; ciò è risultato nell'applicazione del concetto di **Screaming Architecture**. Di seguito ne è mostrato un esempio di implementazione, tratto dal software stesso.



⁴Il principio di inversione di dipendenza è stato applicato, in particolare, in relazione agli UseCase (nel Domain Layer): la loro indipendenza dagli altri moduli fa in modo che essi possano dipendere solo da *astrazioni* dei loro strati.

L'espressione "Screaming Architecture" è stata coniata dal sopracitato Robert C. Martin; essa viene utilizzata nelle situazioni in cui, rivolgendo un solo sguardo ad un progetto, si riesce ad avere un'idea di base su cosa esso faccia e su cosa riguardi.

Da ciò deriva la possibilità di comprendere esattamente la struttura al cuore del codice implementato: la suddivisione nei diversi package "urla" al lettore l'approccio utilizzato, che risulta dunque chiaro e comprensibile già a primo impatto.

La Clean Architecture non preclude, però, la possibilità di adattare all'applicativo un proprio design pattern architettonicale.

Come descritto nella sezione successiva, tale approccio è stato ritenuto adatto alle esigenze di implementazione.

2.3 Le scelte implementative

2.3.1 Applicazioni mobile native e ibride

Un'applicazione mobile nativa è un'applicazione software sviluppata per funzionare su uno specifico tipo di dispositivo o piattaforma. Le app native consentono performance migliori, un'esecuzione veloce e un alto grado di precisione. Ciononostante, esse presentano alcune problematiche:

- Le app native richiedono codice sorgente *esclusivo*, poiché ogni dispositivo deve avere la sua versione specifica dell'app;
- Richiedono un costo maggiore, e sono richiesti più sviluppatori per creare e gestire il codice per ogni piattaforma;
- È richiesto molto tempo per la creazione differenziata per i diversi sistemi operativi in ogni aggiornamento delle funzionalità.

Le app ibride combinano gli elementi di applicazioni web e native; i linguaggi utilizzati fanno parte di tecnologie web come HTML, CSS e JavaScript.

- Hanno un'interfaccia utente multipiattaforma;
- Possono essere sviluppate più velocemente e richiedono meno costi di sviluppo e manutenzione;
- Non hanno bisogno di codice sorgente specifico.
- Penalizzano la UX in alcuni casi.

La scelta stata è influenzata da diversi fattori. Tra i requisiti del sistema sono richieste prestazioni di buon livello, e il target di clienti è particolarmente esigente. È stato adottato un approccio nativo, precisamente su sistema operativo **Android**, scegliendo **Kotlin** come linguaggio di programmazione.

2.3.2 Perchè Kotlin?

Java detiene da anni un ruolo chiave nello sviluppo di applicazioni, tuttavia alcuni requisiti *platform-dependant* possono rendere tedioso l'utilizzo di questo linguaggio. Esistono altri linguaggi che operano sulla JVM: tra questi, Kotlin ha guadagnato molti punti a favore nello sviluppo di applicazioni native Android.

Kotlin presenta molti vantaggi; tra essi si sottolineano:

- Elevata **concisione**, raggiunta attraverso l'inferenza di tipi, lo *smart-cast* e le *data class*: tali fattori conferiscono al codice una maggiore leggibilità e manutenibilità. Ciò avviene anche in relazione al fatto che - sfruttando queste tecnologie - il codice necessario per l'implementazione di una feature risulta essere di un numero di righe di gran lunga minore rispetto allo stesso codice sviluppato in Java;
- Il type system di Kotlin mira ad eliminare le *NullPointerException*, garantendo controlli a tempo di compilazione e offrendo operatori di *null-safety*.

Tra le funzionalità più interessanti volte a migliorare la produttività dello sviluppatore è presente una forte integrazione del paradigma di programmazione funzionale, oltre alla possibilità di creare particolari funzioni note come *estensioni*, che consentono l'aggiunta di nuovi metodi a classi pre-esistenti.

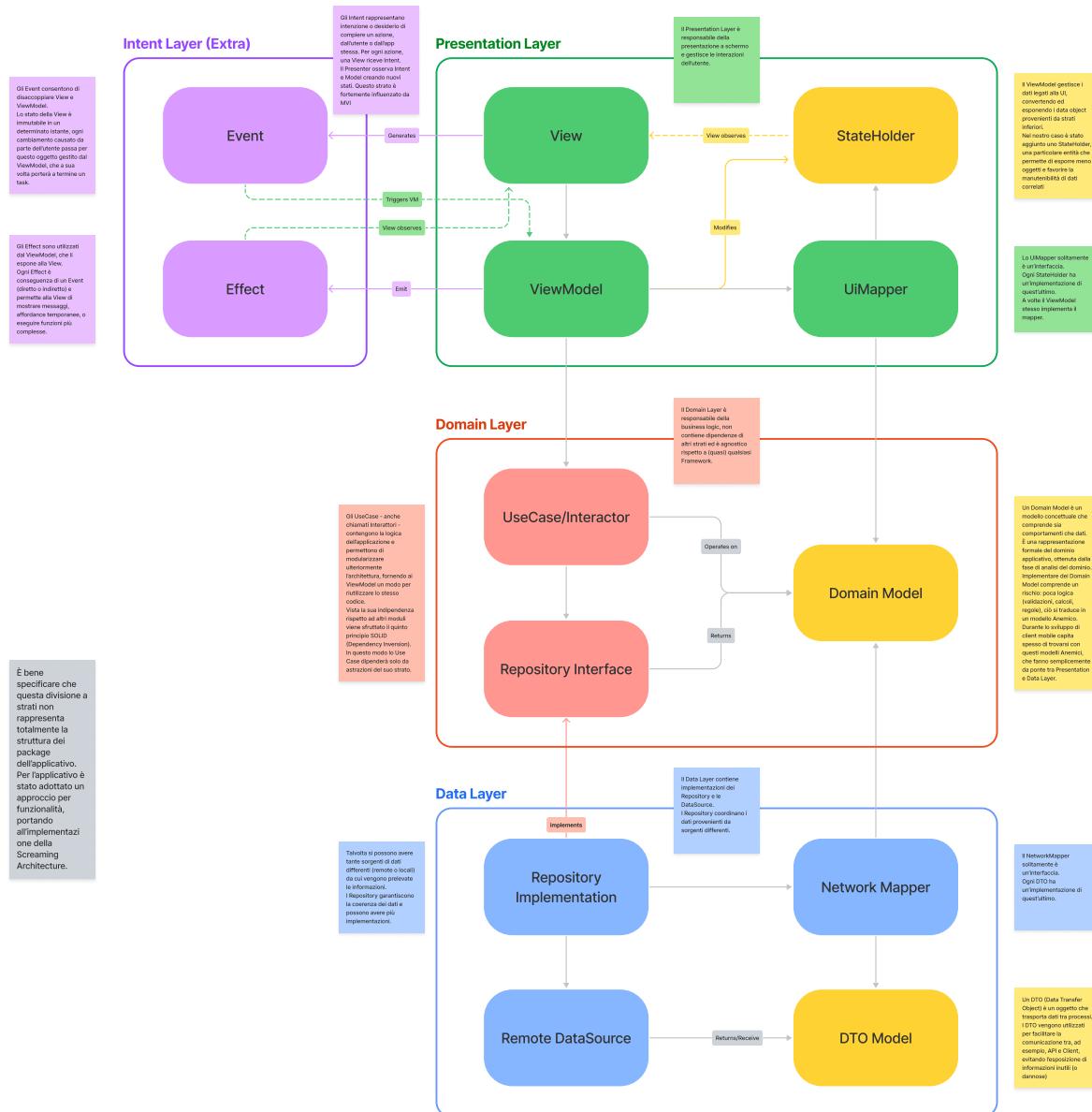
Inoltre, Kotlin non preclude la possibilità di utilizzare codice Java, ma questa possibilità è bi-direzionale: Java e Kotlin possono coesistere all'interno dello stesso progetto.

Alla luce delle informazioni fornite, Kotlin è stato ritenuto poter essere un valido alleato nello sviluppo dell'applicativo.

2.3.3 Diagramma di design dell'applicativo

In questa sezione viene presentato un diagramma realizzato con lo scopo di rappresentare la struttura generale dell'applicativo.

Le specifiche di ciascuna scelta implementativa sono dettagliate nelle sezioni successive.



2.3.4 Pattern architetturale utilizzato

Il design pattern architetturale potrebbe essere definito come *ibrido*: ciò è dovuto alla *fusione* di due pattern differenti utilizzati nella progettazione.

Tali pattern sono **MVVM** e **MVI**; quest'ultimo è stato utilizzato per ovviare ai problemi sofferti dal primo. Procediamo ad un'analisi più dettagliata.

MVVM è un'architettura del tipo Model-View-ViewModel che evita l'accoppiamento stretto tra ciascuna componente. Più nello specifico, in questo tipo di architettura, i figli non hanno un riferimento diretto al padre, ma hanno solo riferimenti tramite *observables*. Esso è organizzato secondo tre strati:

- Model - rappresenta i dati e la business logic dell'applicazione;
- View - consiste del codice relativo alle interfacce utente;
- ViewModel - definisce una sorta di ponte tra i due modelli sopracitati.

Il modello MVVM, per quanto affermato, pone però alcune problematiche, quali ad esempio:

- la difficoltà di riusabilità sia per le View che per i ViewModel;
- la registrazione a molteplici observables nello stesso ViewModel, che ne aumenta le relative responsabilità;
- View e ViewModel possono essere soggetti ad avere un coupling stretto.

A questo scopo, è stata realizzata un'integrazione con il modello MVI, la cui differenza principale si risolve nell'assenza della moltitudine di observables citati in precedenza.

MVI sta per Model-View-Intent. Si tratta di uno dei pattern architetturali più recenti per applicazioni Android.

- Model - rappresenta uno stato. I Model dovrebbero essere immutabili, in modo da assicurare un flow di dati unidirezionale con gli altri strati;
- View - rappresenta le View, che possono essere implementate in Activity o Fragment
- Intent - rappresenta un'intenzione o una volontà di eseguire un'azione, sia dell'utente che dell'app stessa. Gli Intent si traducono in *Event* ed *Effect*.

L'integrazione dei due pattern architetturali sopracitati culmina nella nascita di un nuovo pattern, chiamato dagli sviluppatori Android **UDF**.

Alcune delle caratteristiche di UDF possono essere così sintetizzate:

- Presenza di *StateHolder*, necessari per ogni ViewModel nella maggior parte dei casi e non necessariamente unici;
- Richiamo alle macchine di stato e agli **Statechart** definiti in precedenza;
- *Event* e *Effect*, che si interpongono tra View e ViewModel.

2.4 Diagramma delle classi di design

2.4.1 Autenticazione

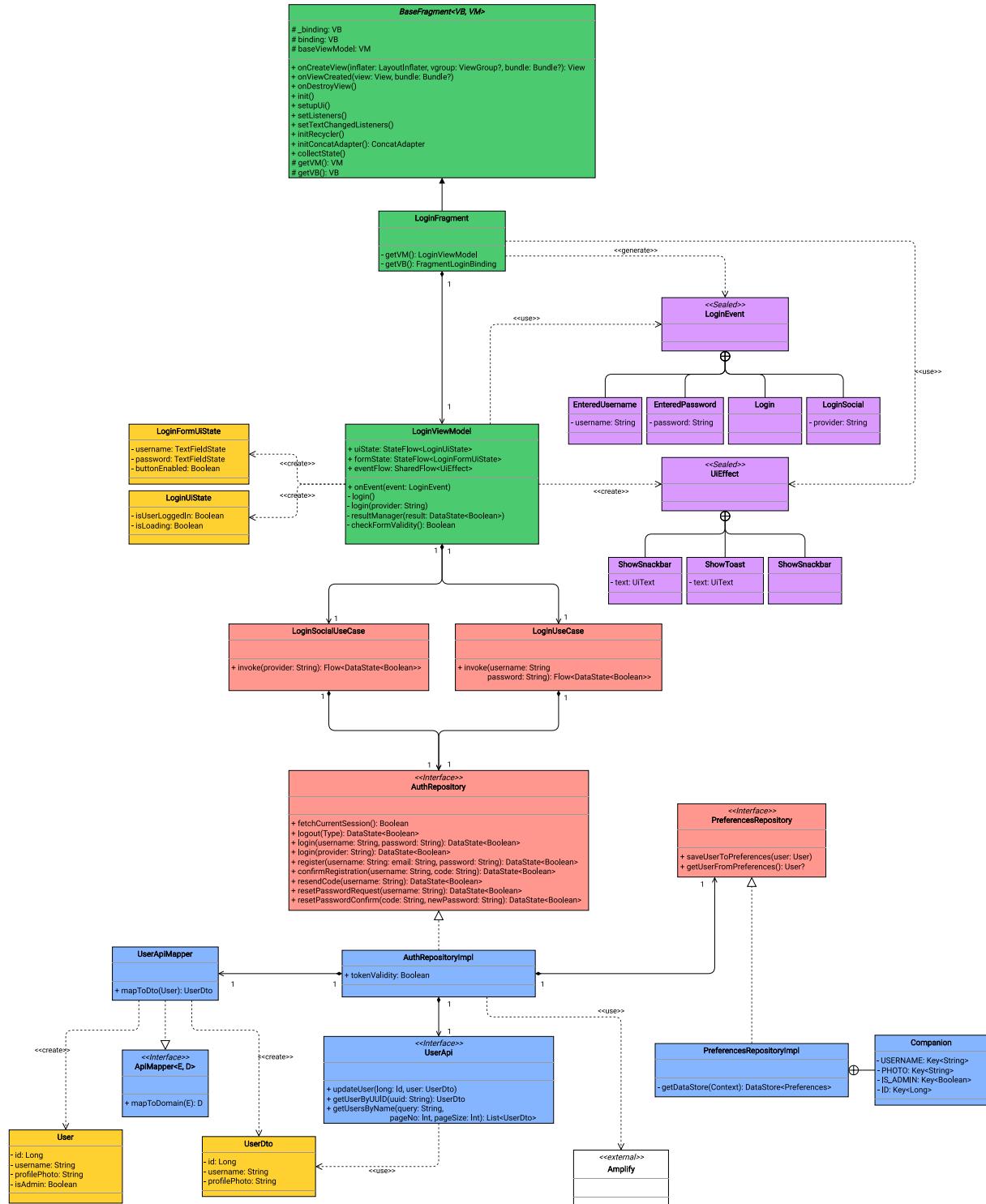


Figura 73: Login e login con social

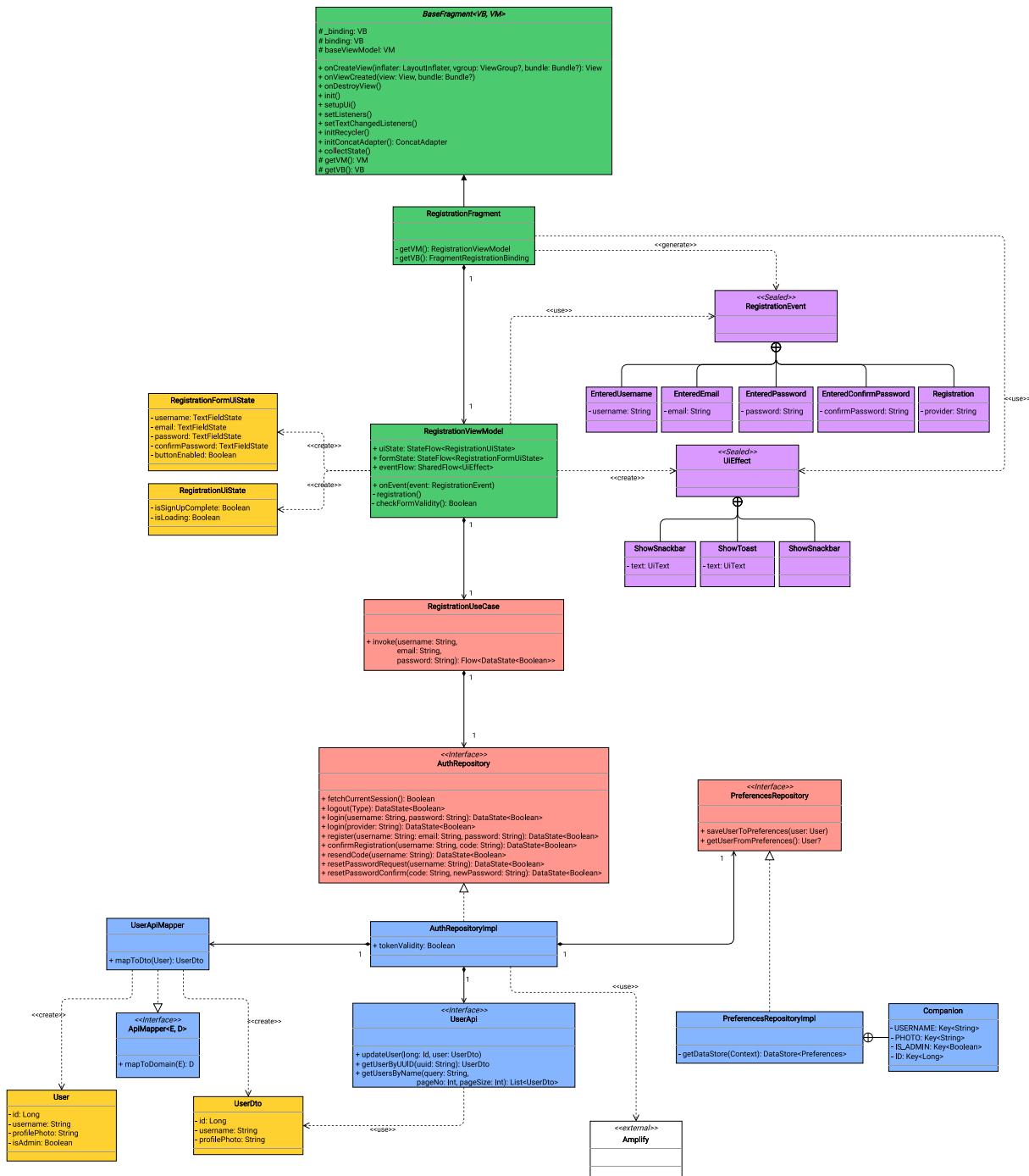


Figura 74: Registrazione

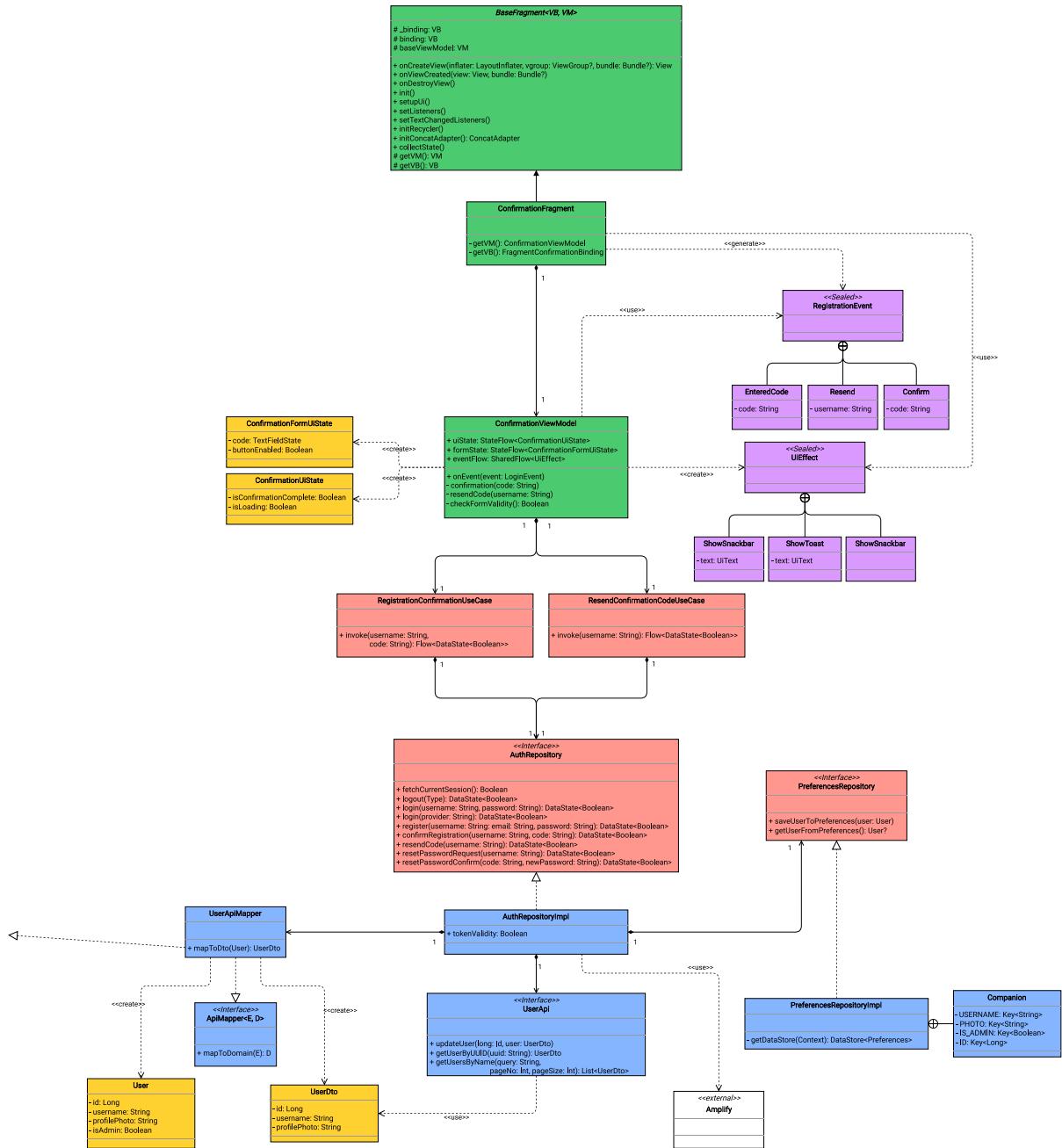


Figura 75: Conferma registrazione

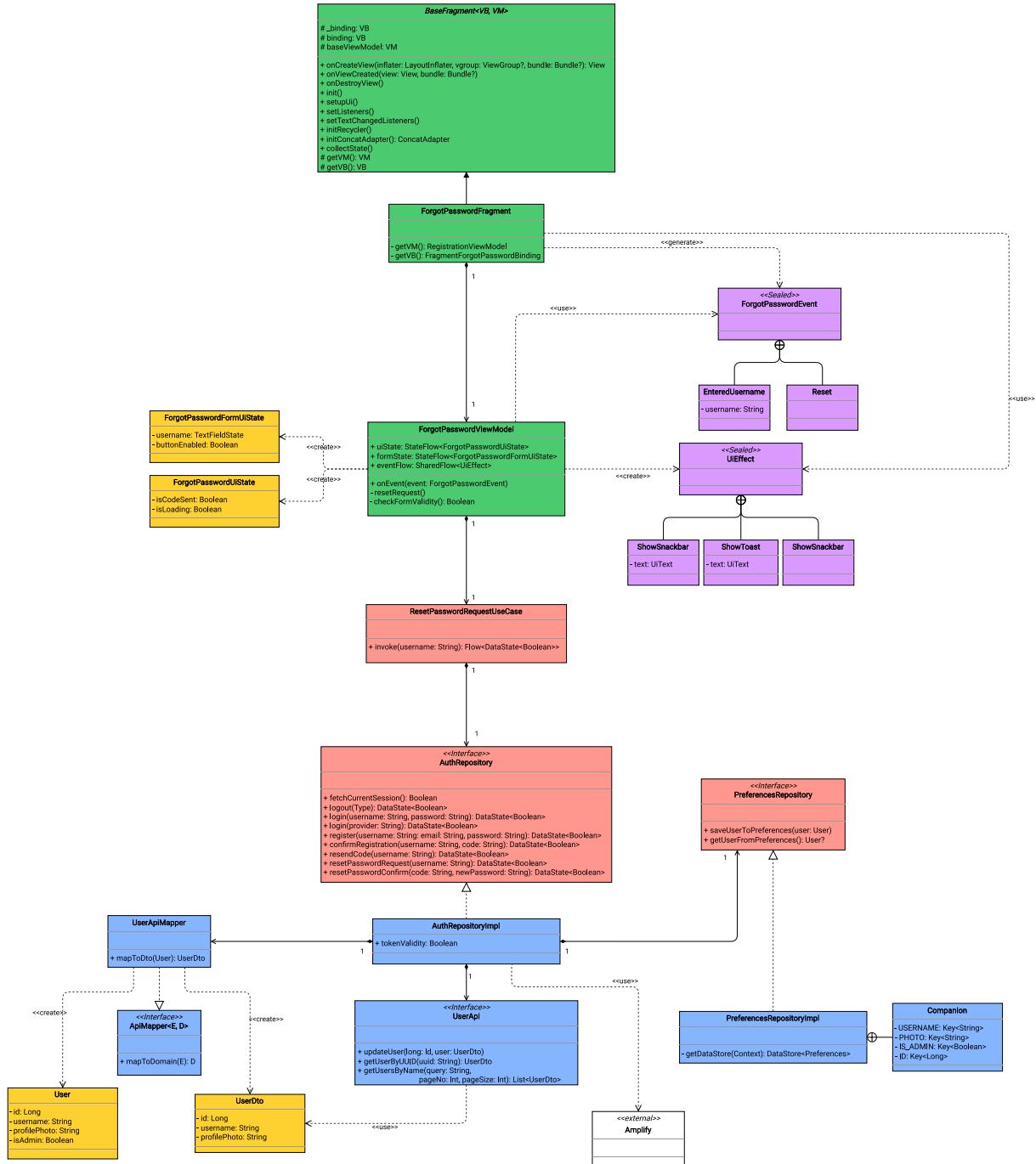


Figura 76: Password dimenticata

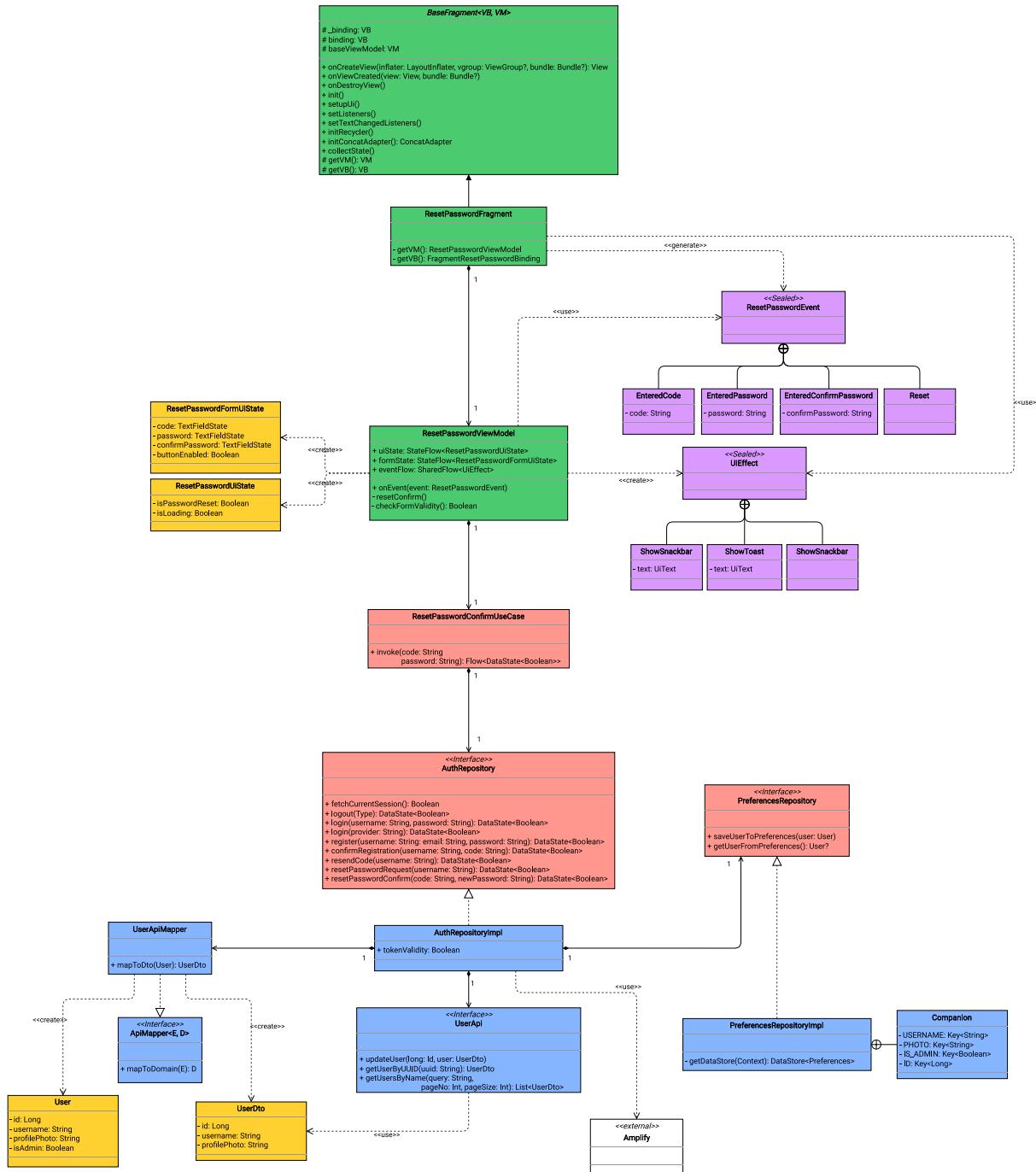


Figura 77: Reimposta password dopo la richiesta

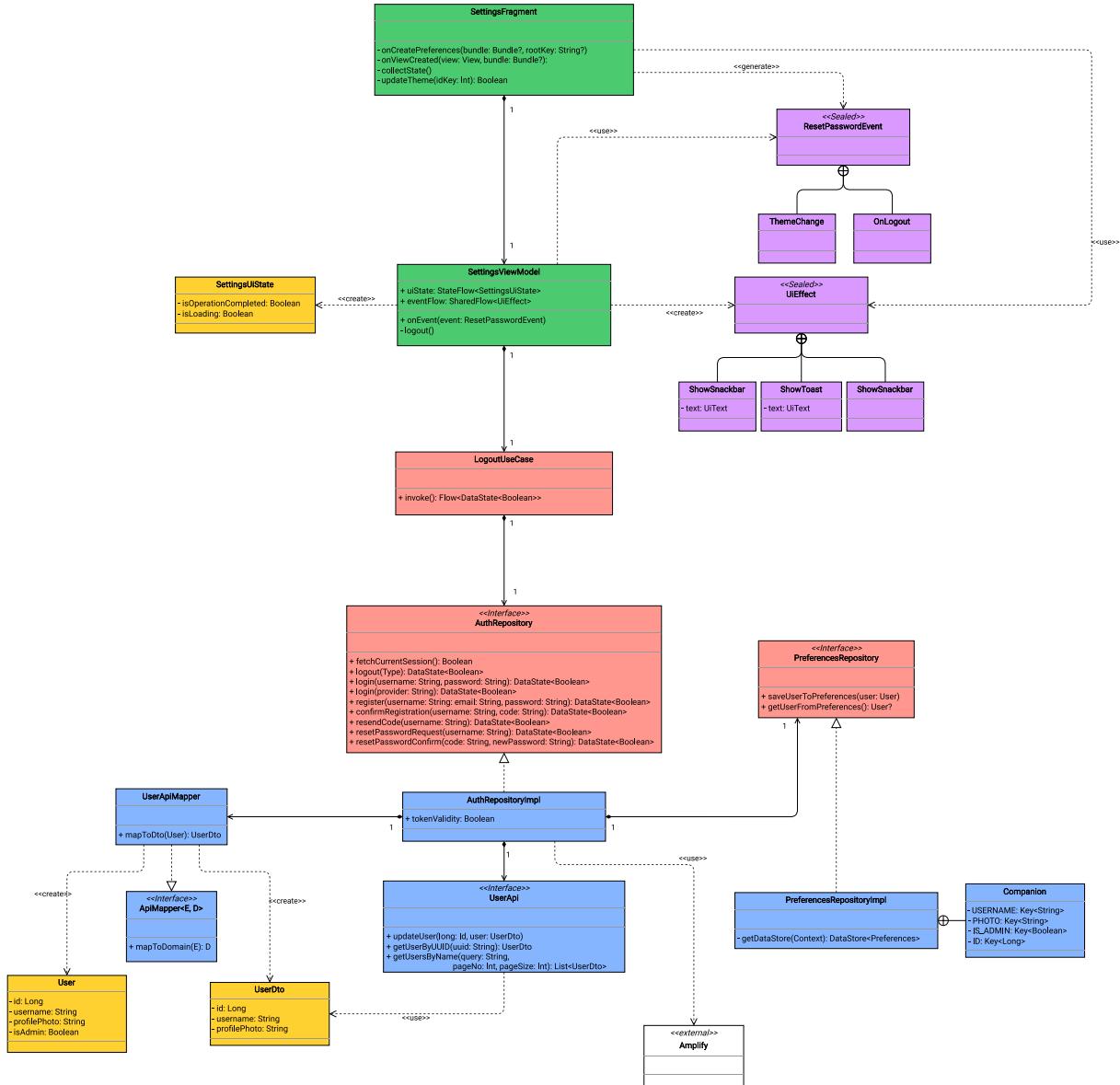


Figura 78: Logout

2.4.2 Interazione con un itinerario

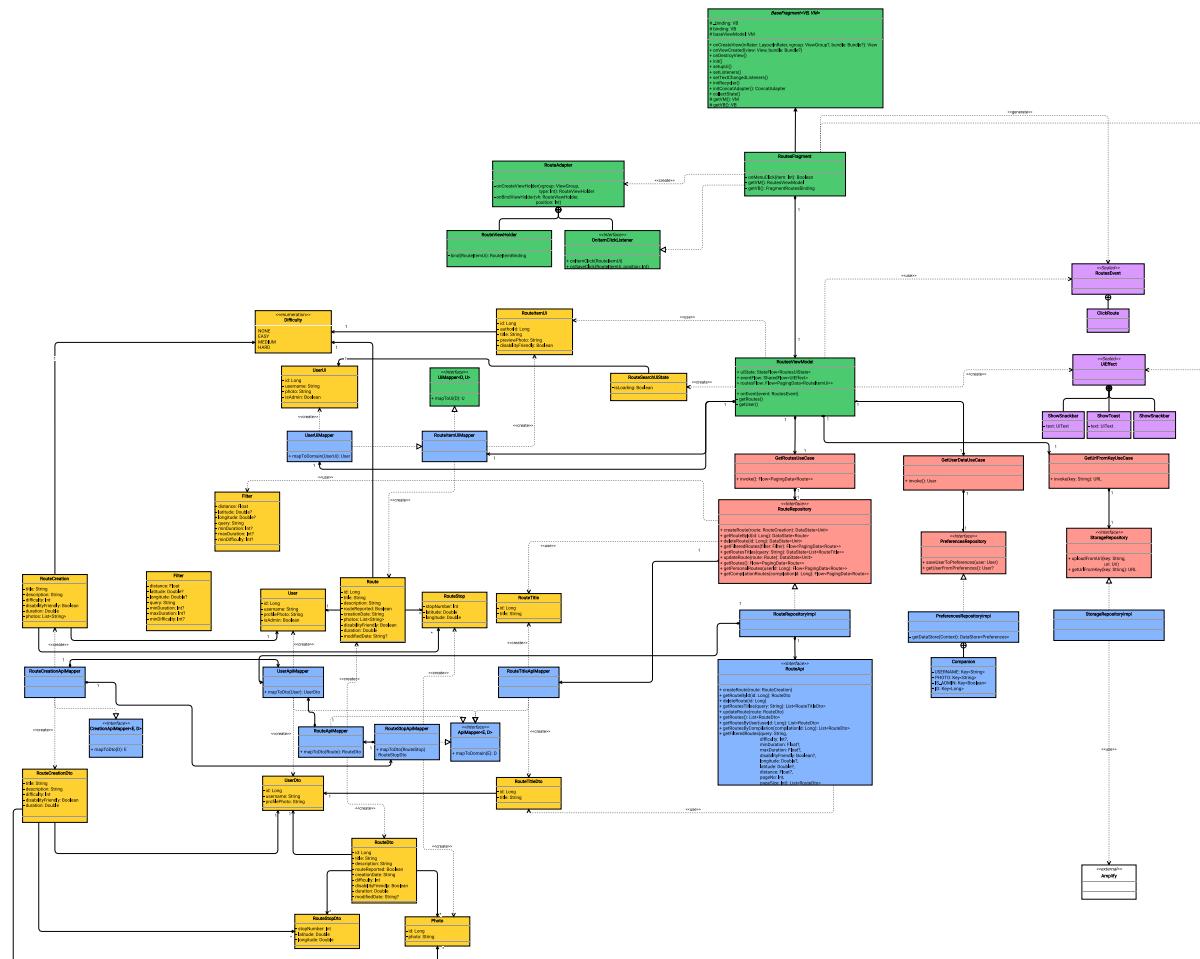


Figura 79: Itinerari più recenti

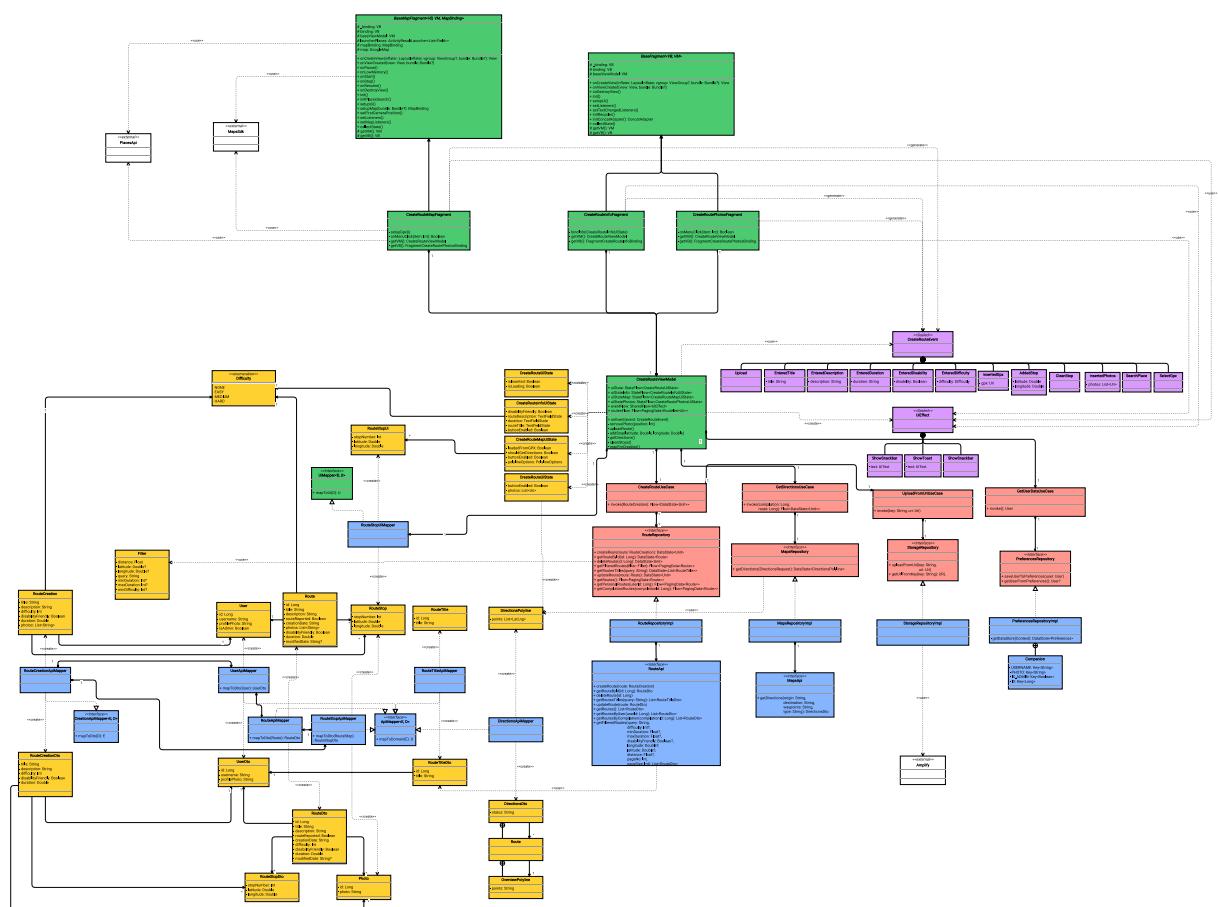


Figura 80: Aggiunta itinerario

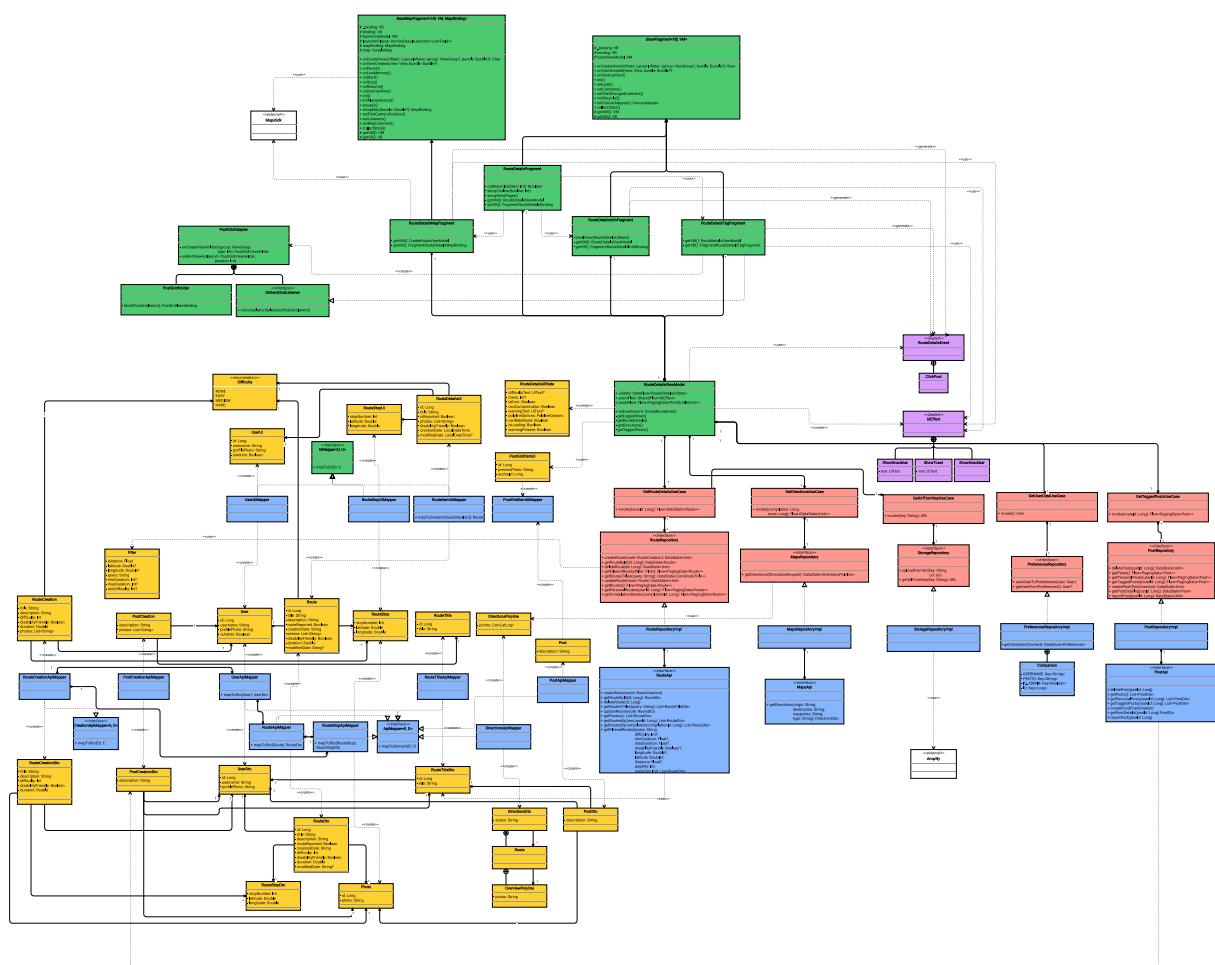


Figura 81: Dettagli itinerario

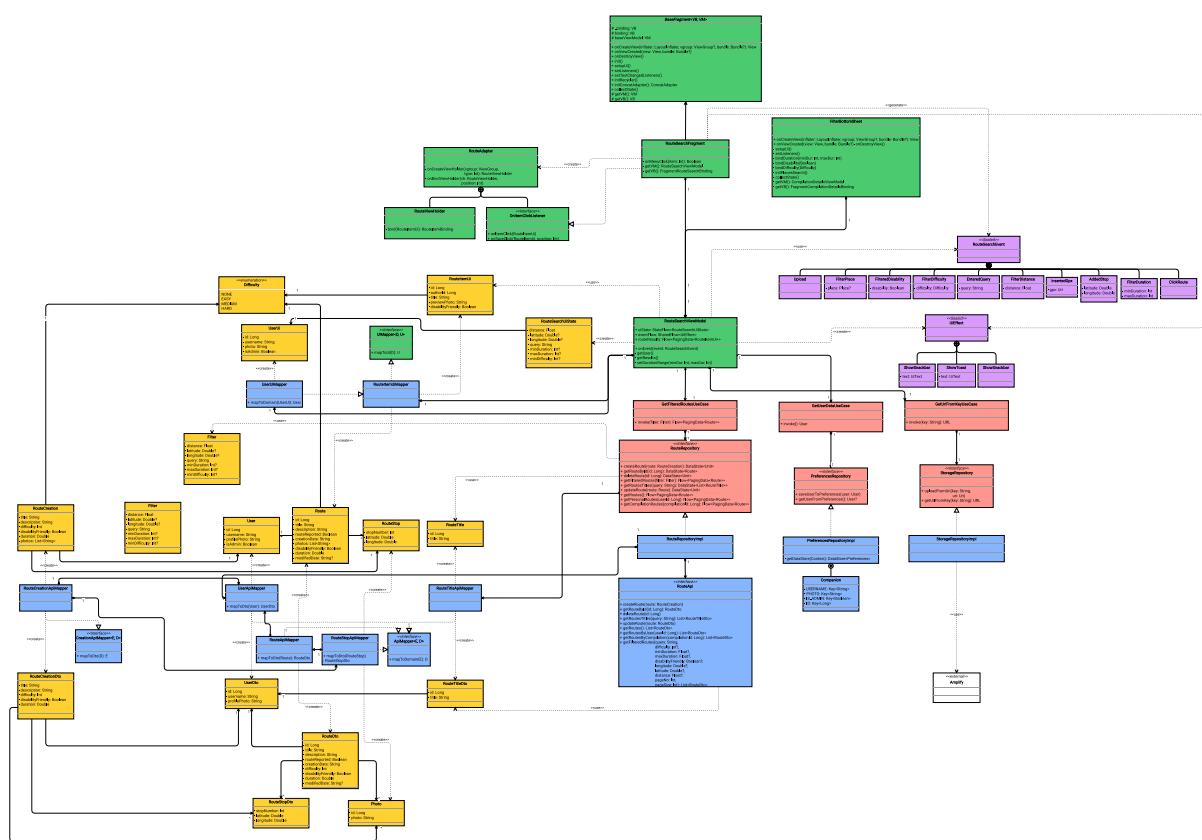


Figura 82: Ricerca itinerario

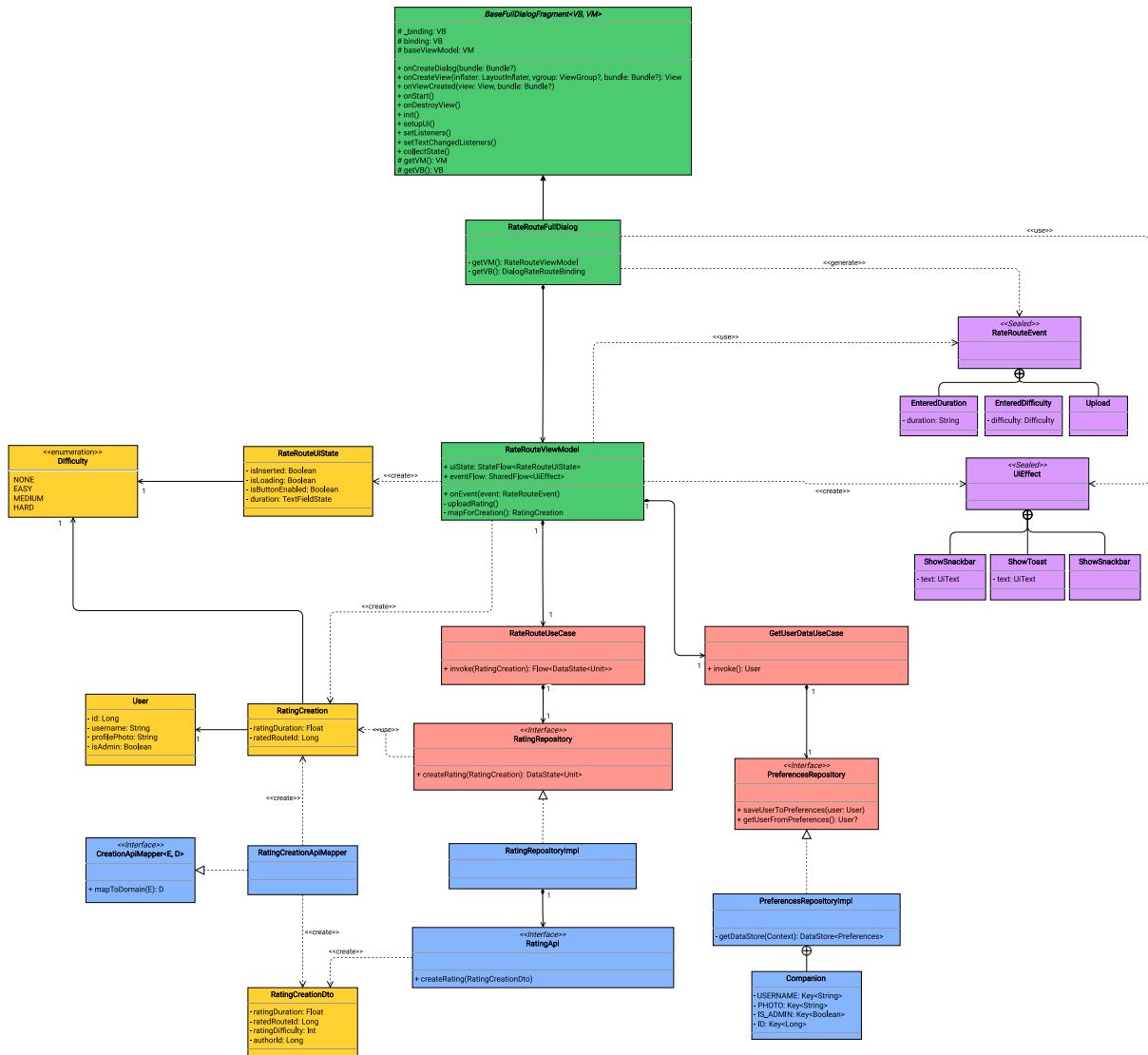


Figura 83: Valutazione itinerario

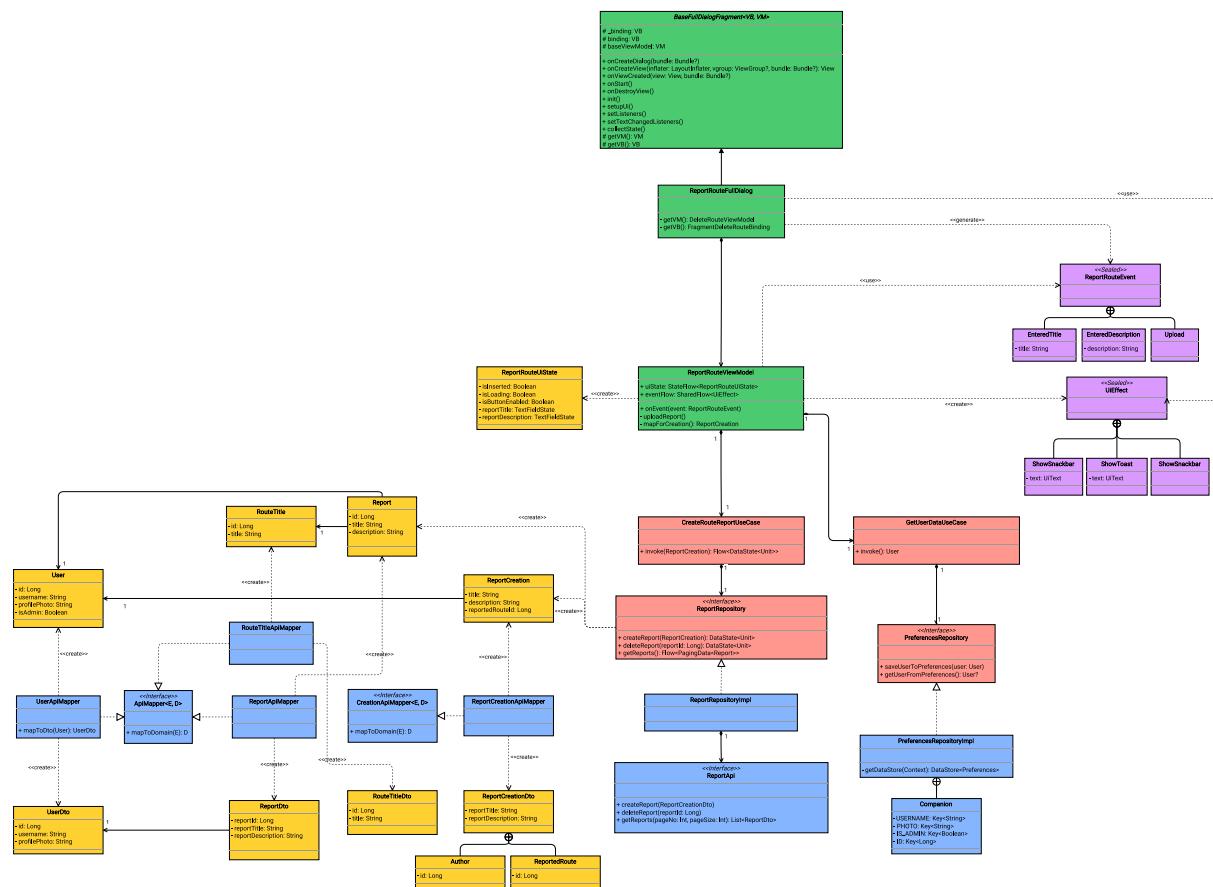


Figura 84: Segnalazione itinerario

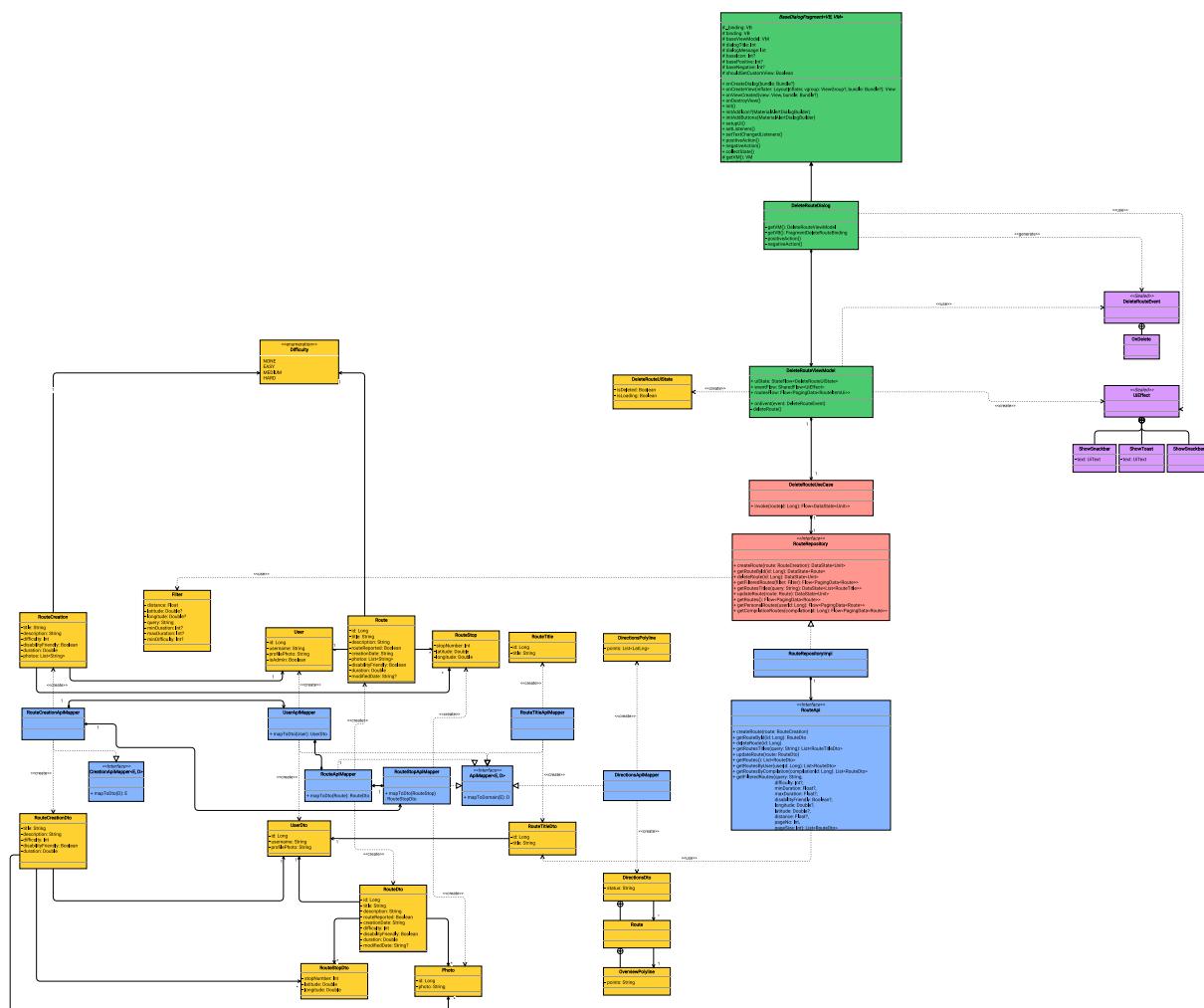


Figura 85: Eliminazione itinerario

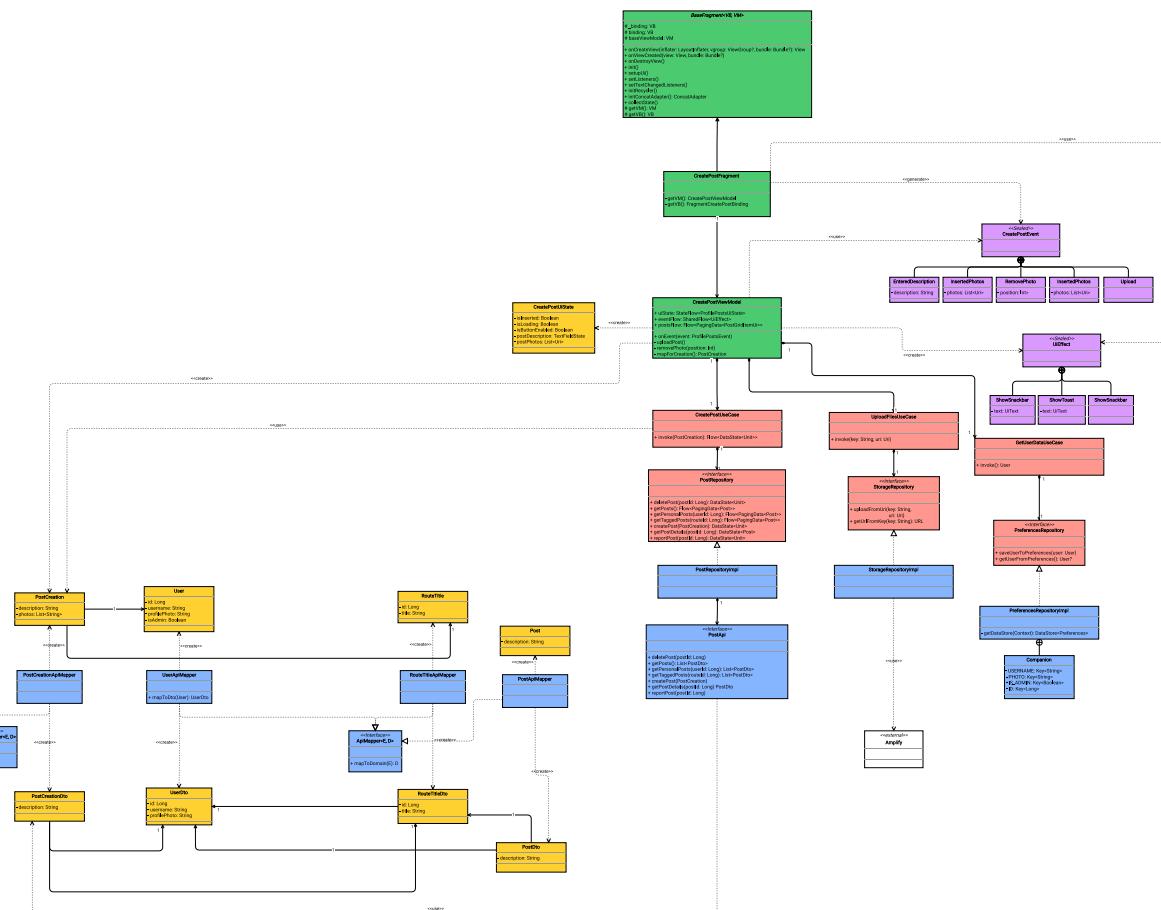


Figura 87: Aggiunta post

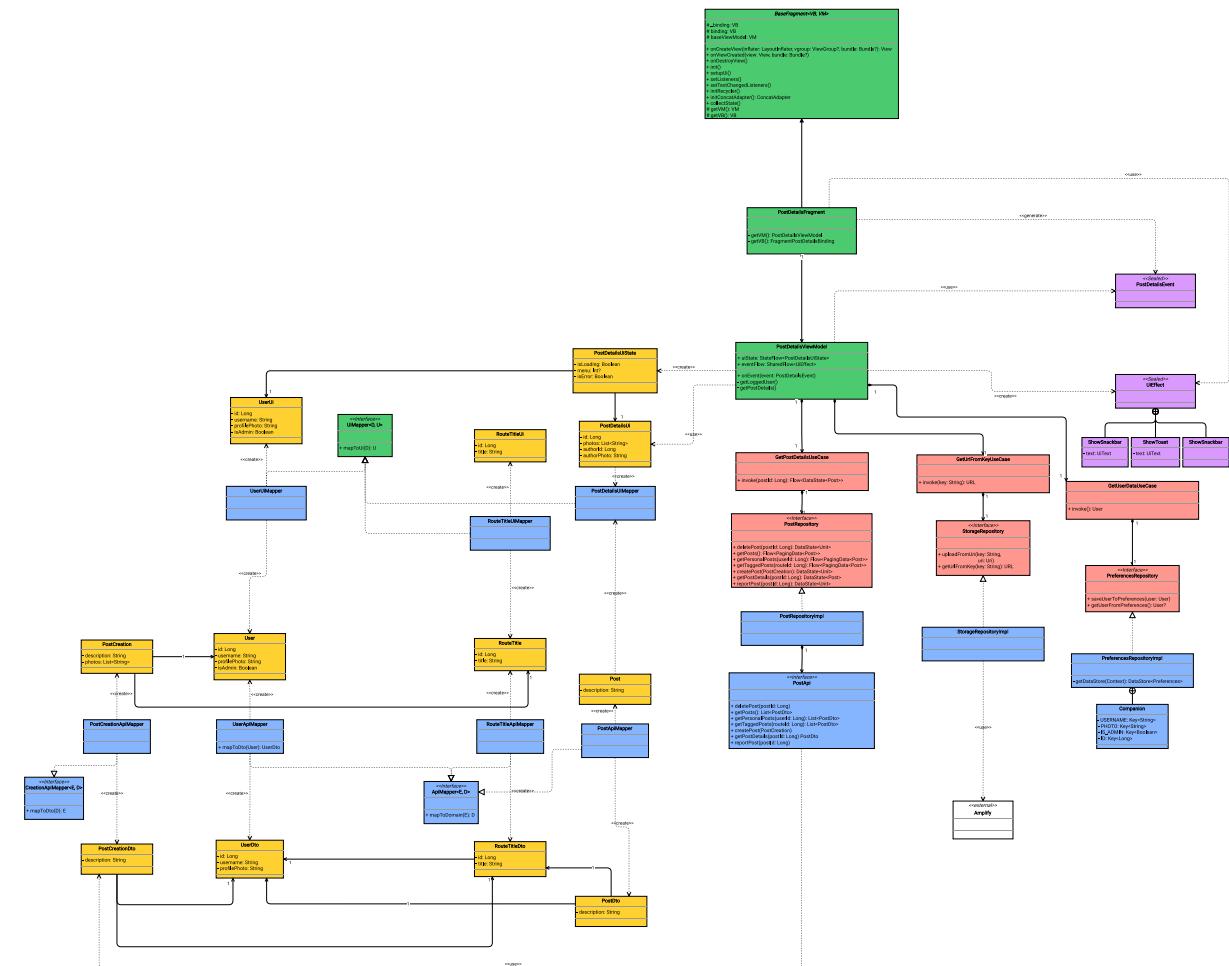


Figura 88: Dettagli post

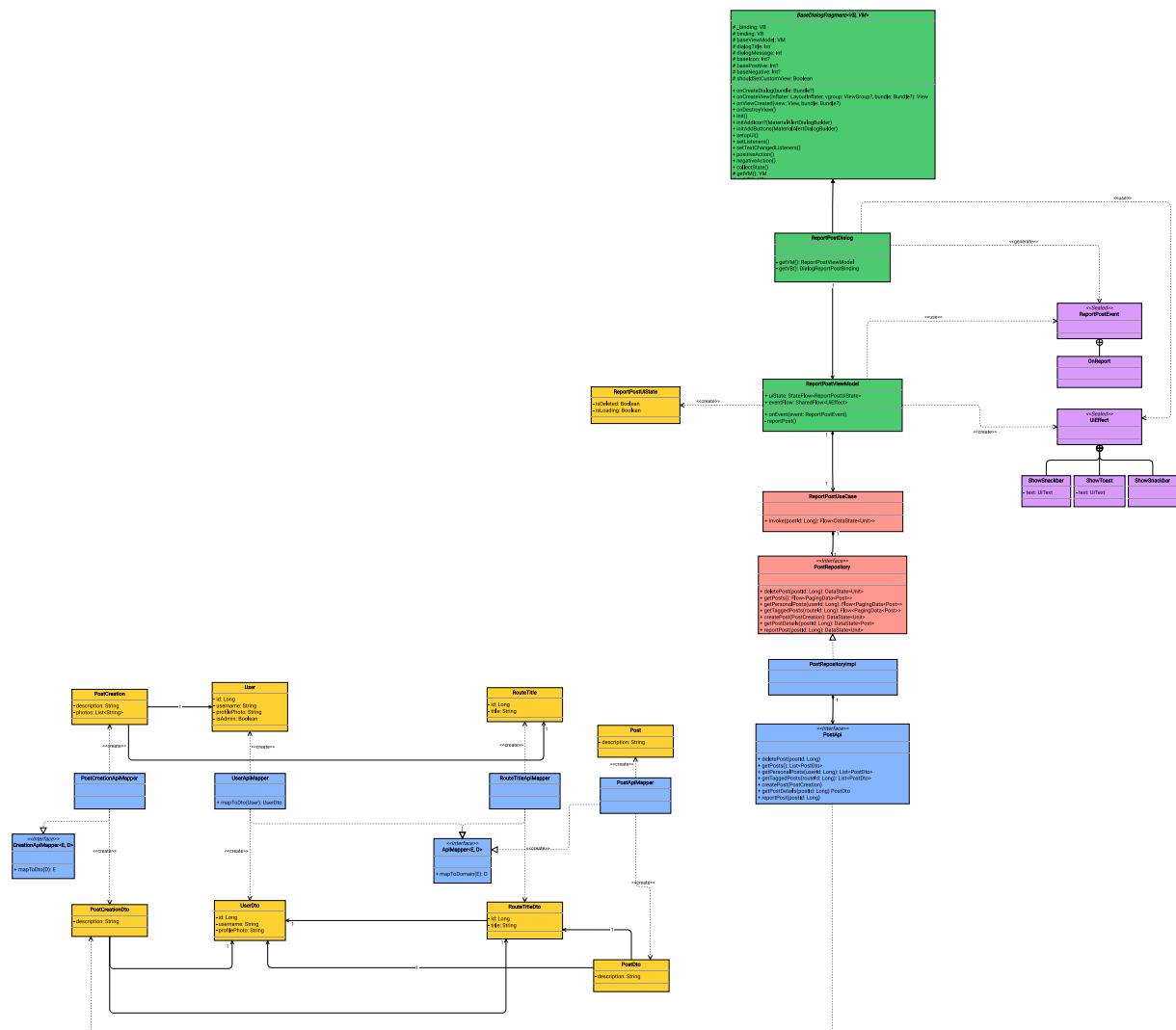


Figura 89: Segnalazione post

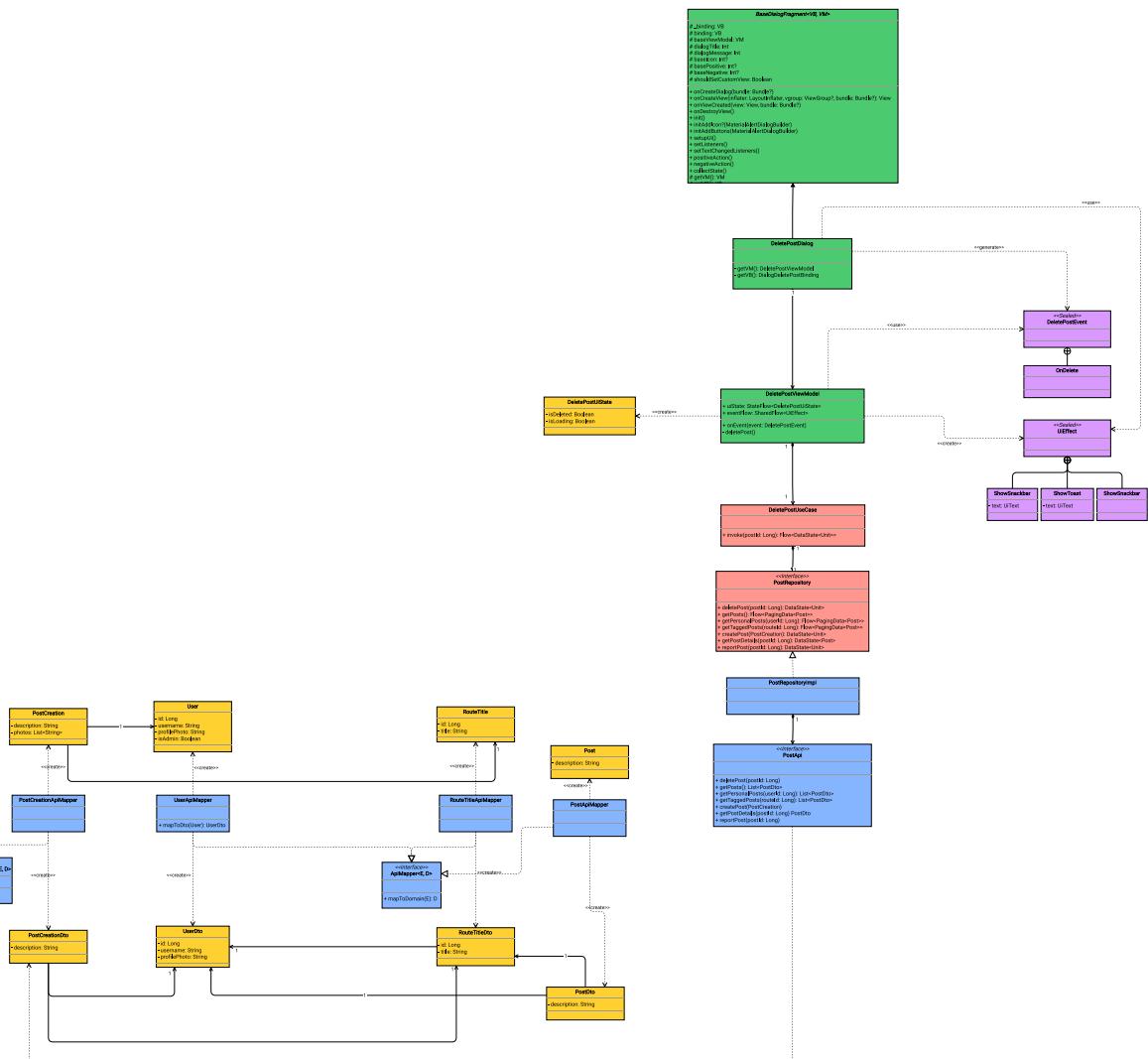


Figura 90: Eliminazione post

2.4.4 Interazione con una compilation

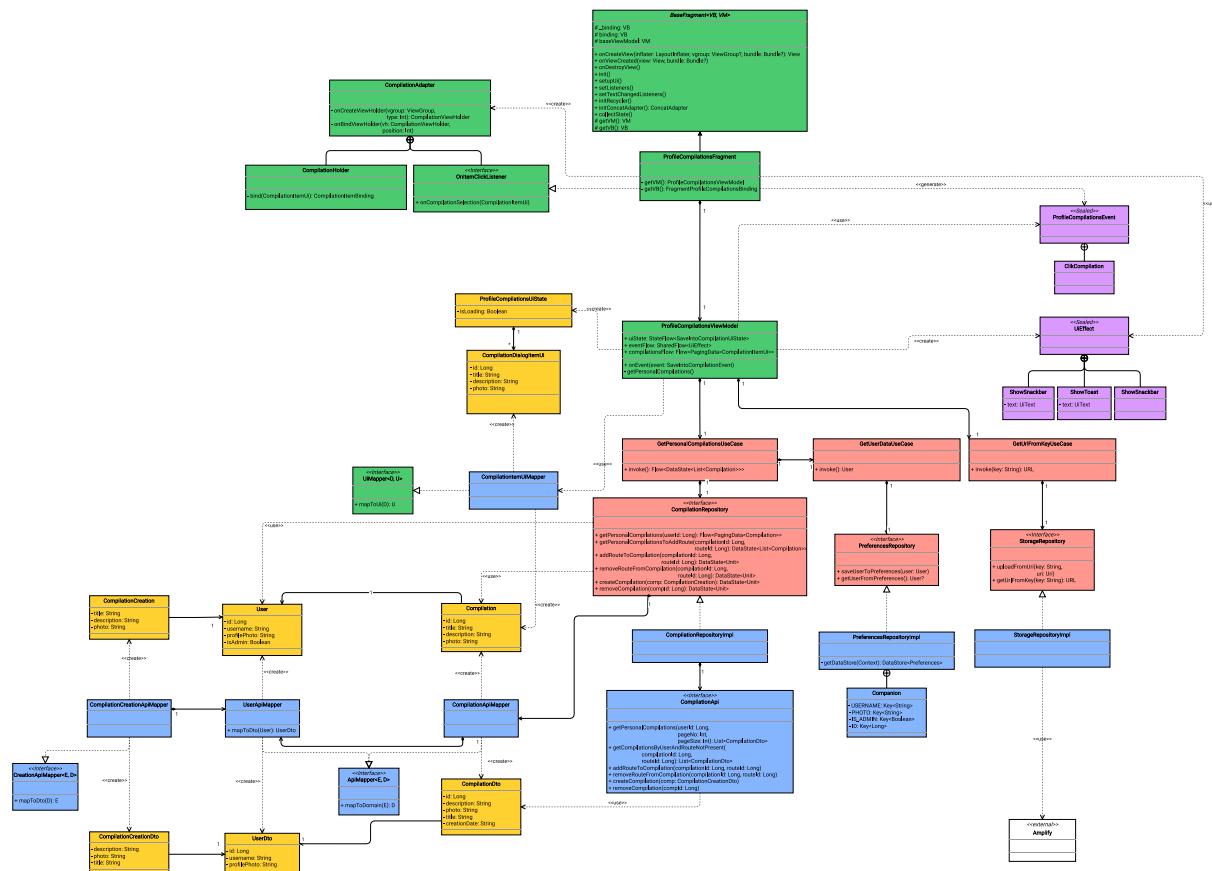


Figura 91: Visualizza compilation personali

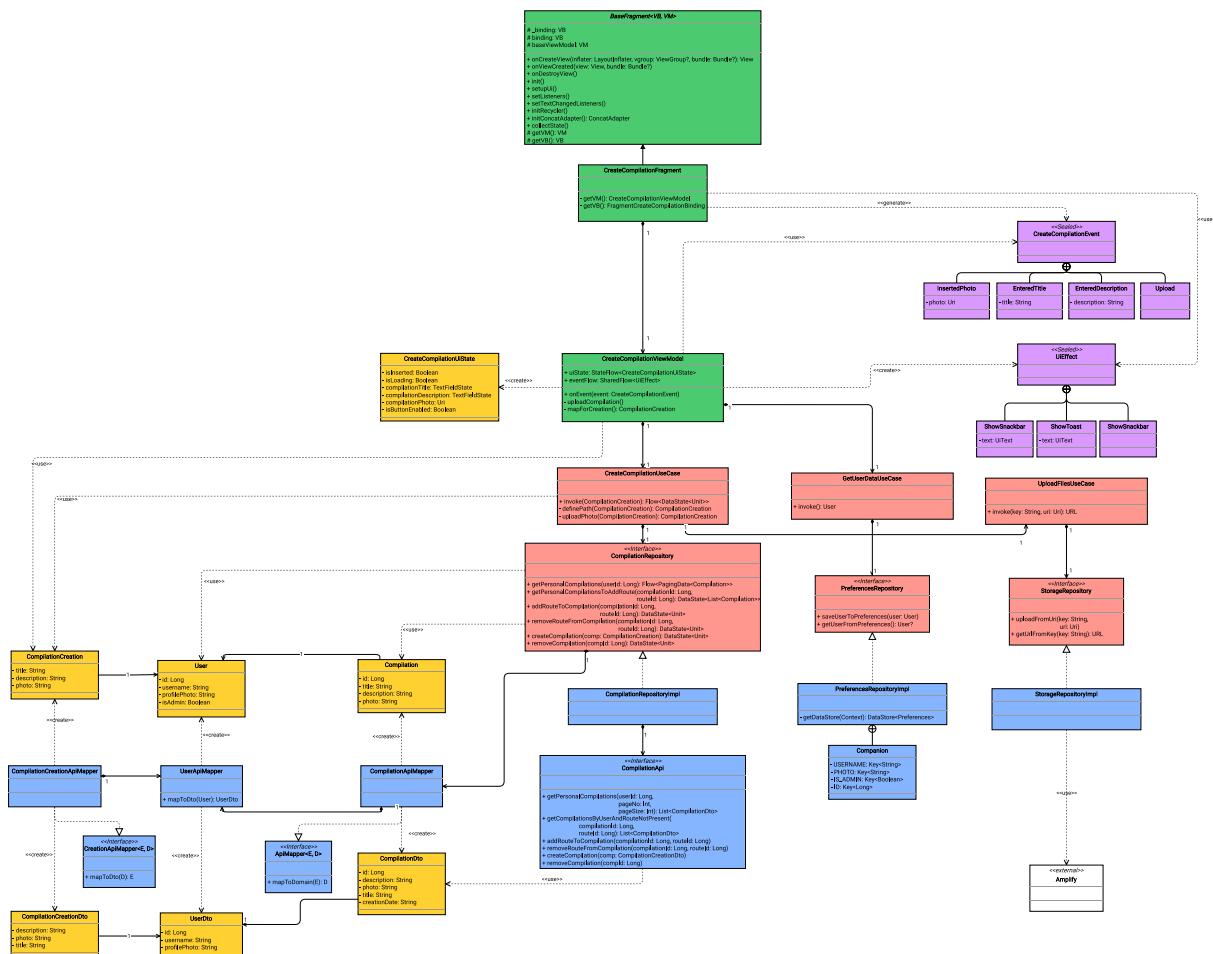


Figura 92: Aggiunta compilation

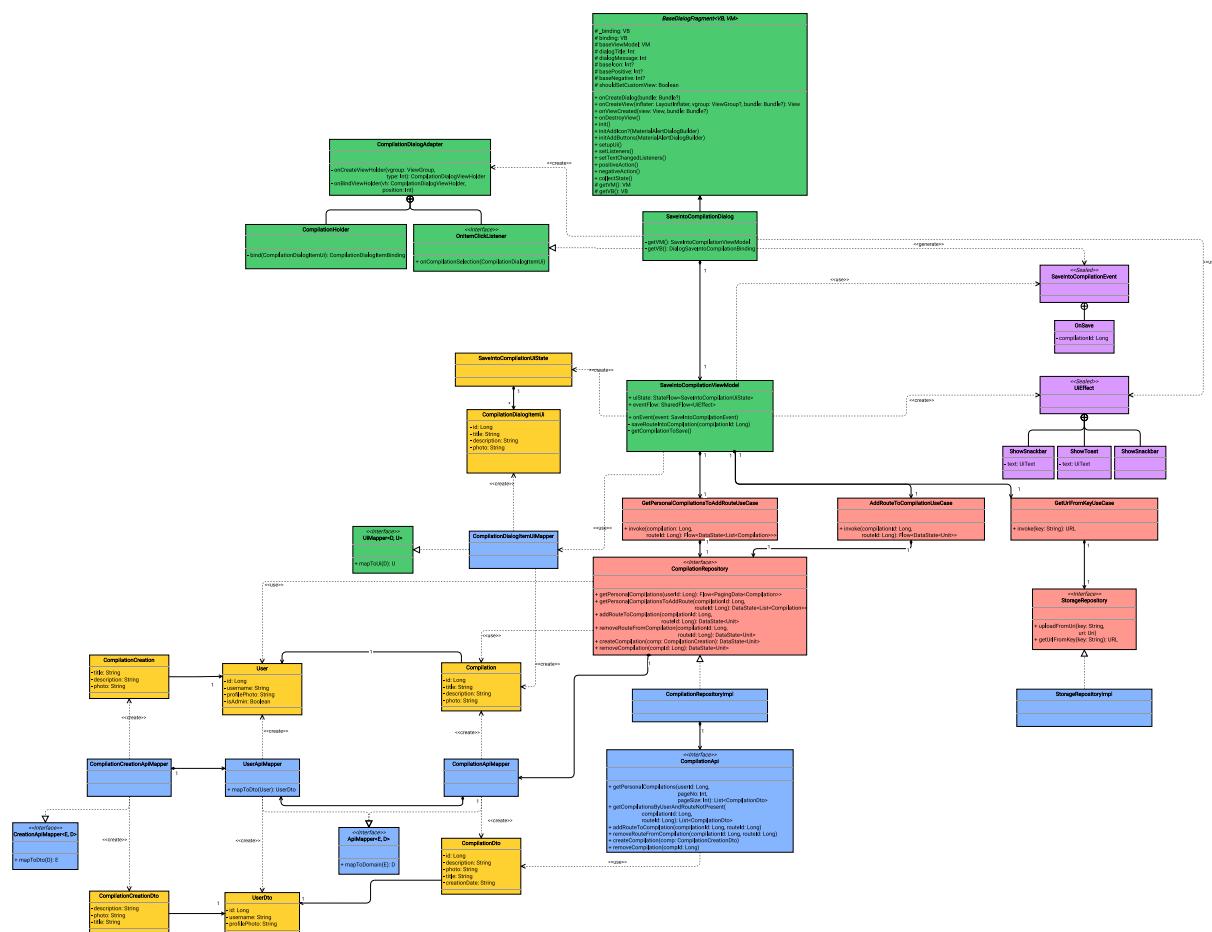


Figura 94: Aggiunta itinerario a compilation

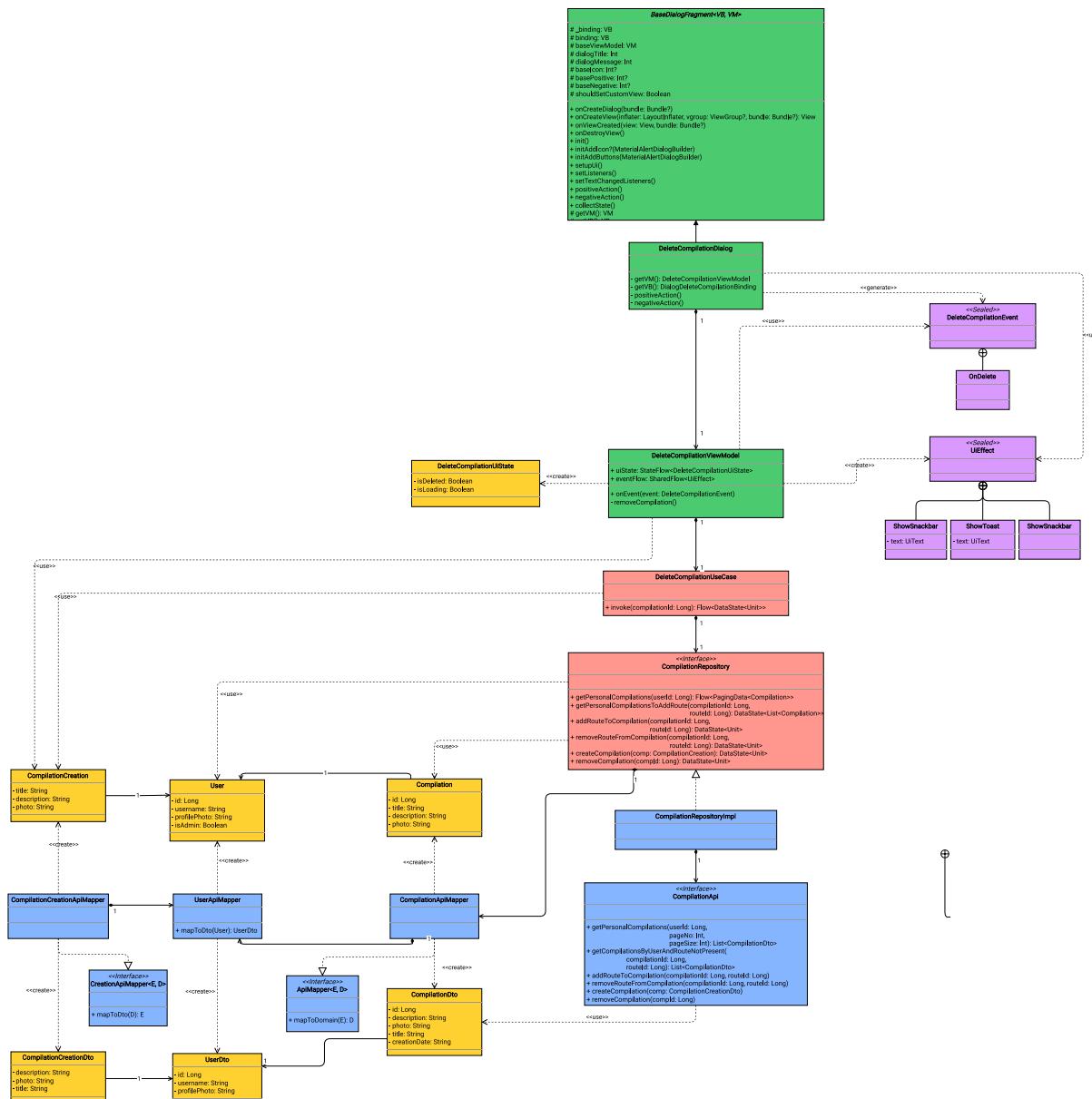


Figura 95: Eliminazione compilation

2.4.5 Interazione con altri utenti

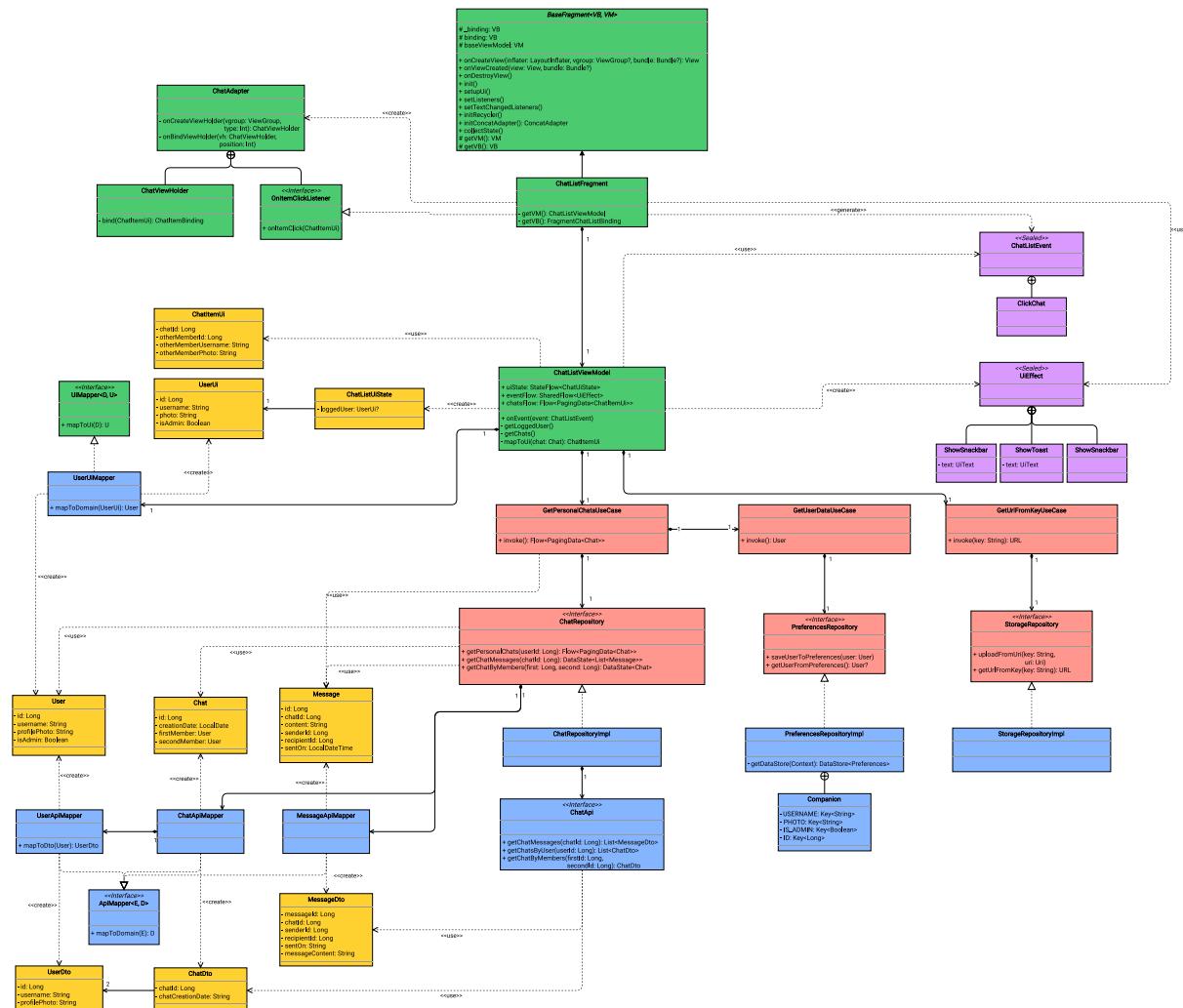


Figura 96: Lista conversazioni

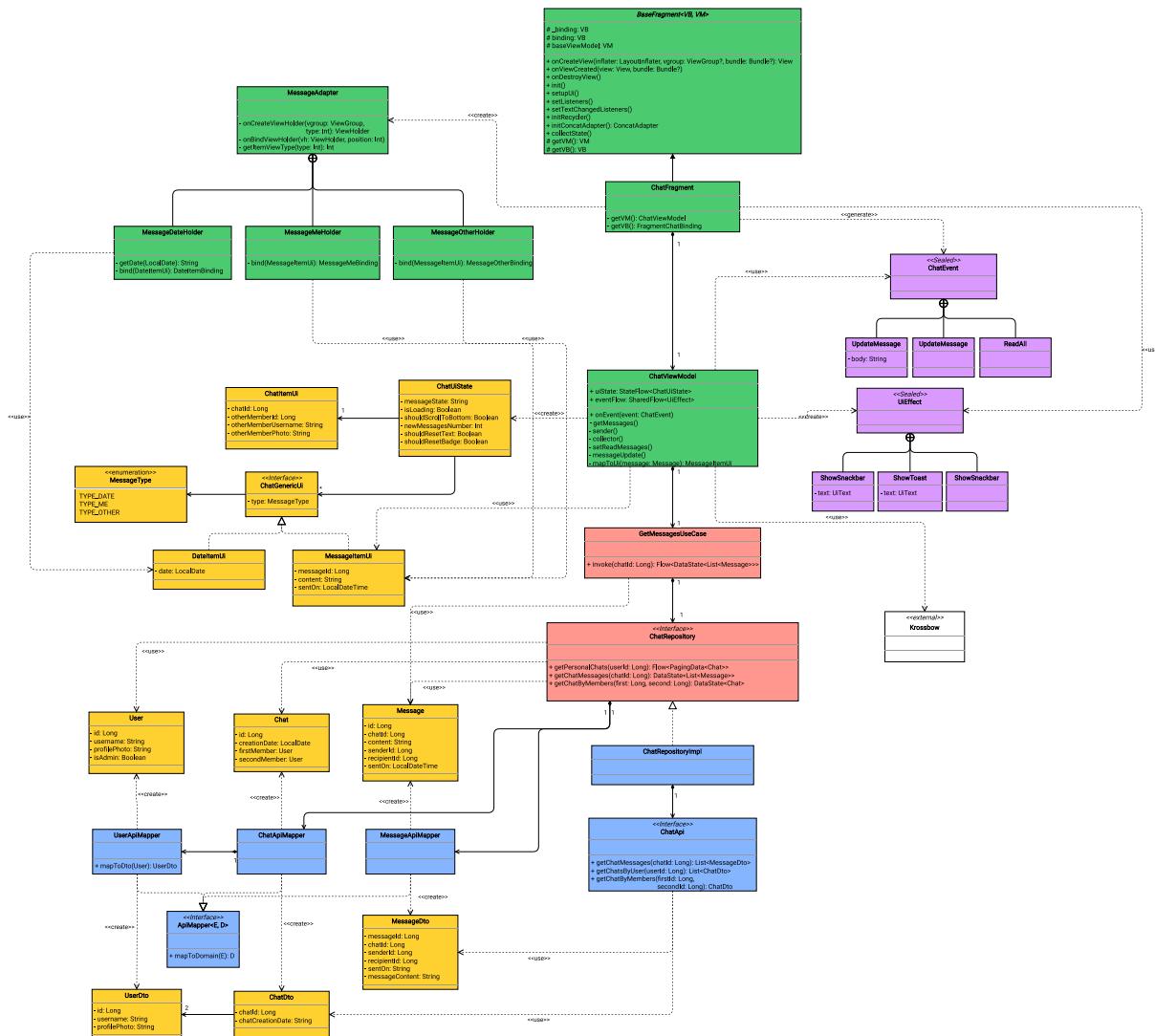


Figura 97: Conversazione privata

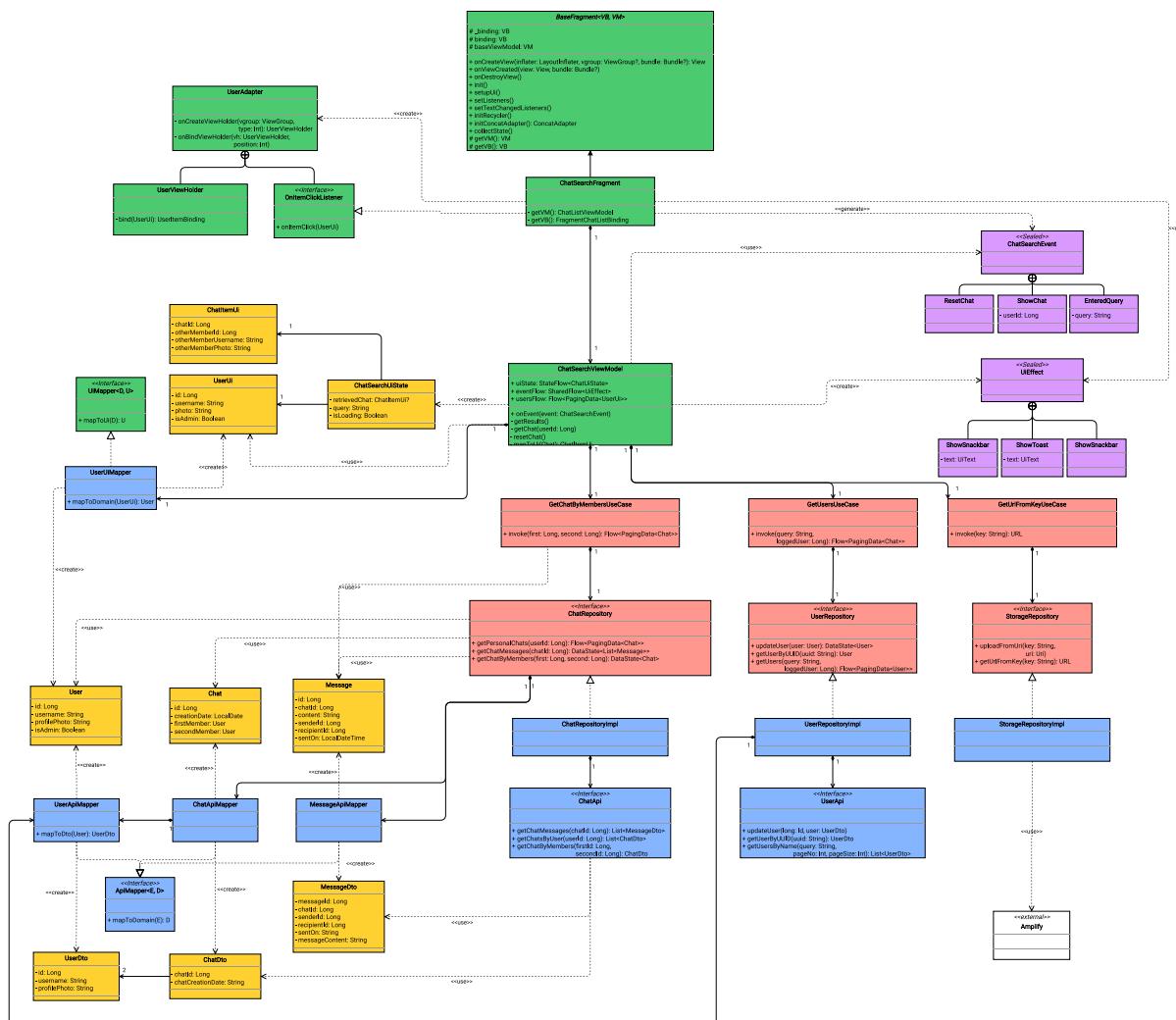


Figura 98: Ricerca destinatario messaggio

2.4.6 Gestione profilo

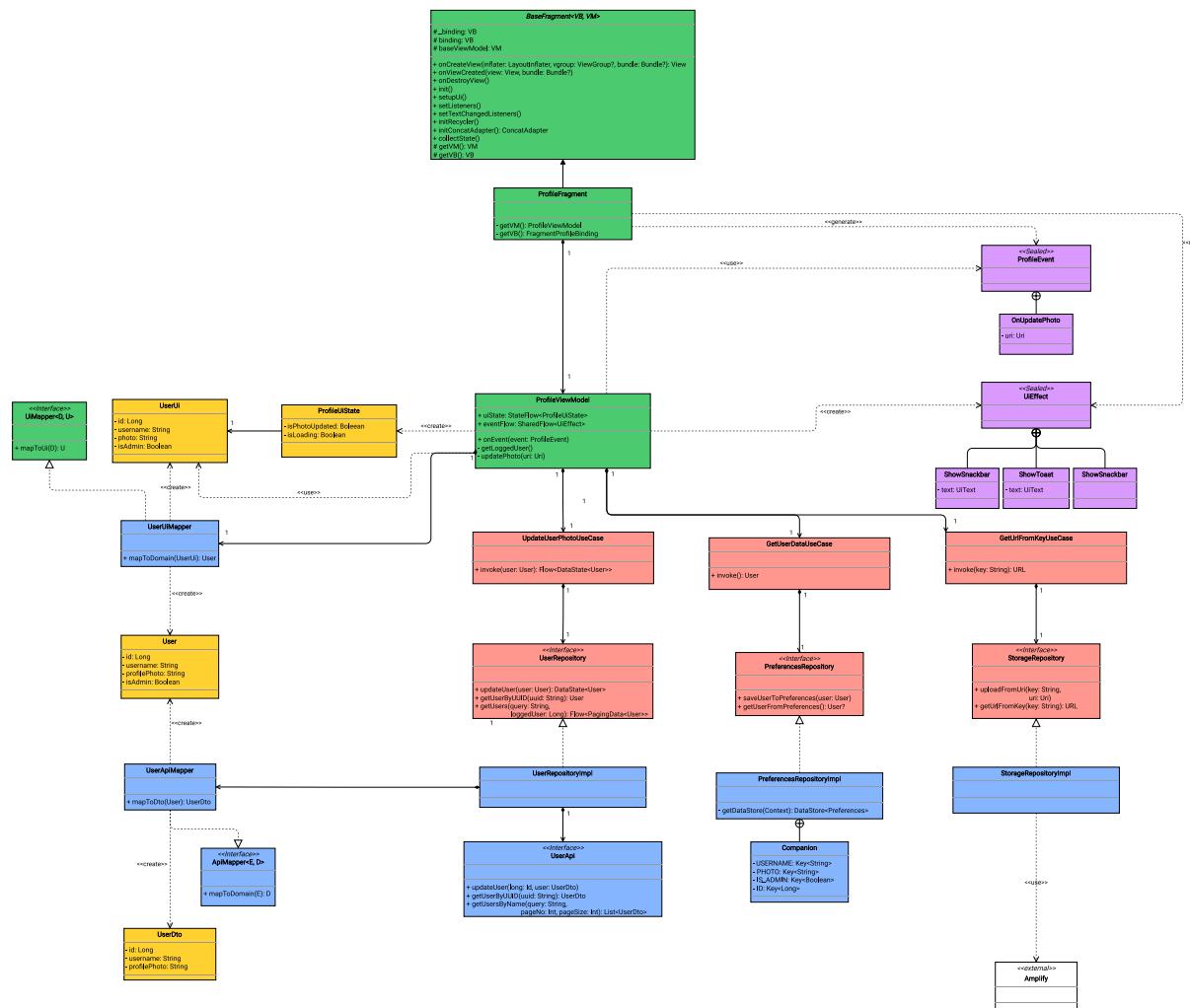


Figura 99: Modifica foto profilo

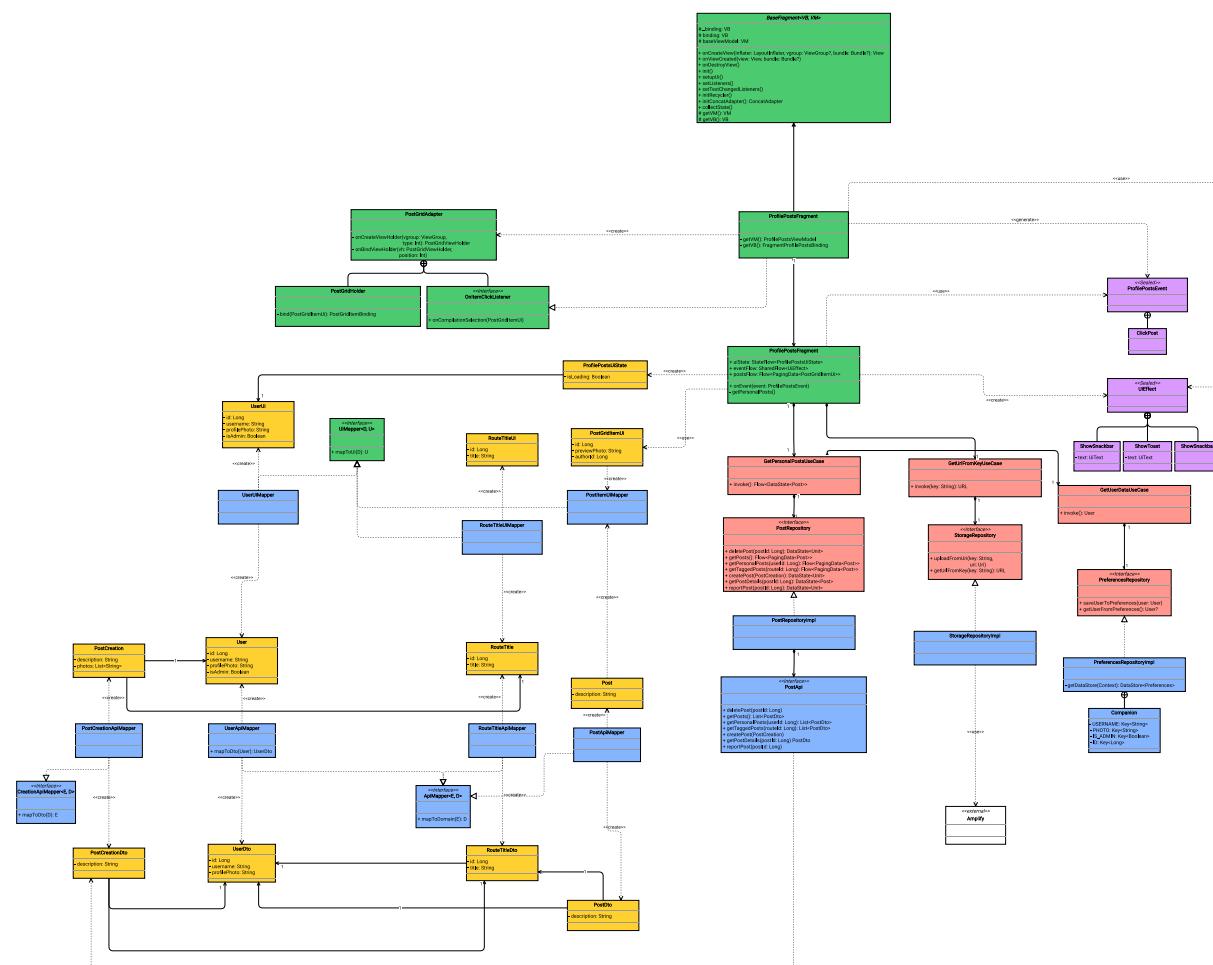


Figura 100: Post personali

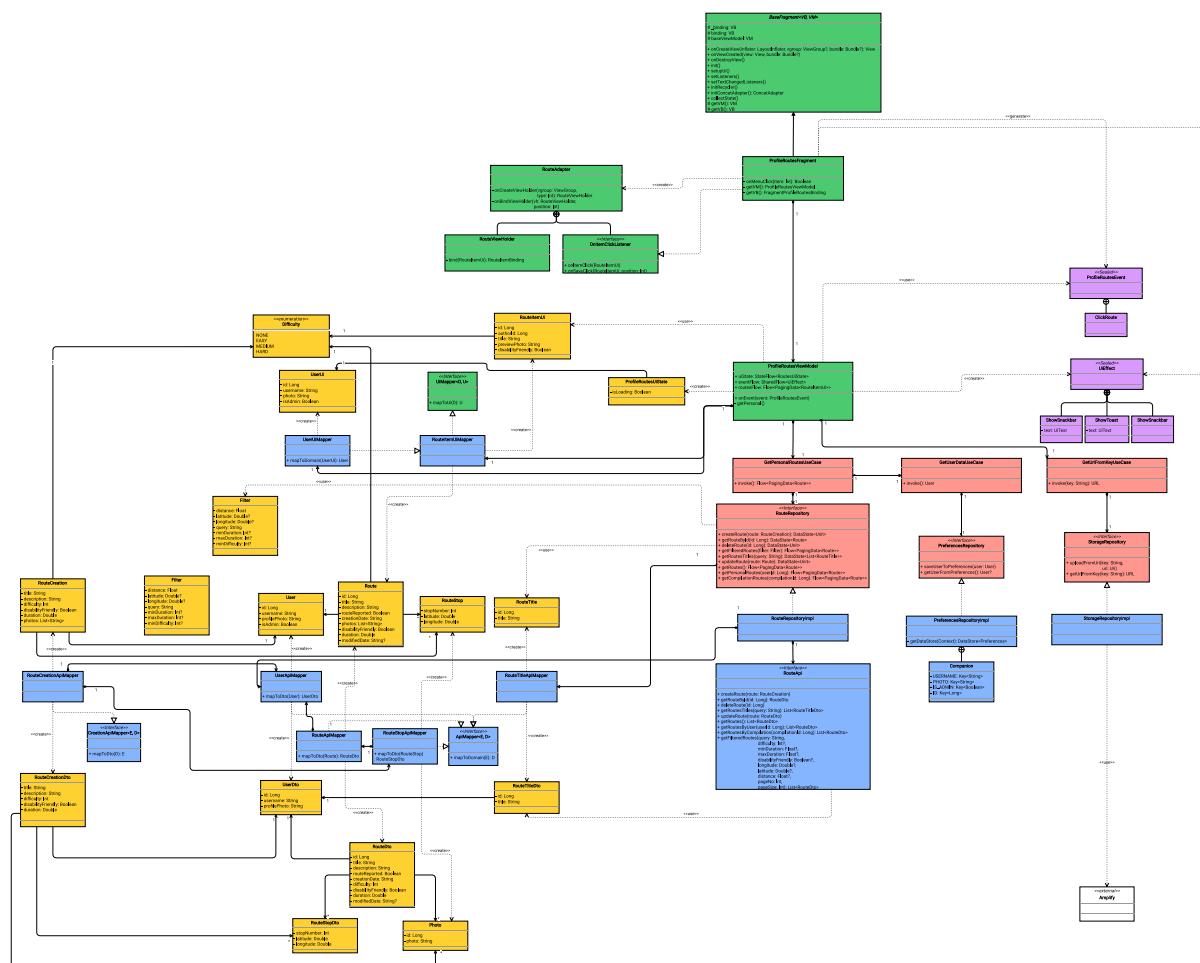


Figura 101: Itinerari personali

2.4.7 Amministratore

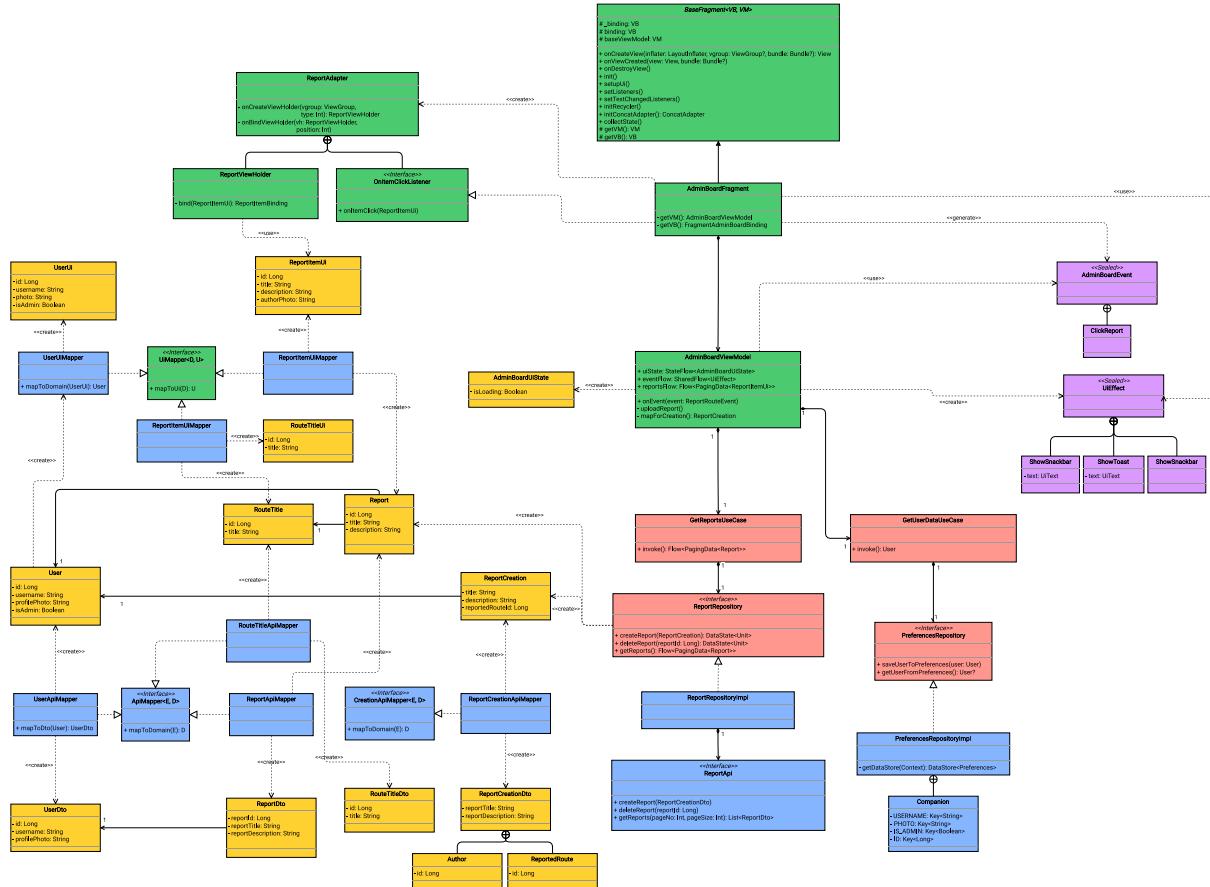


Figura 102: Lista segnalazione

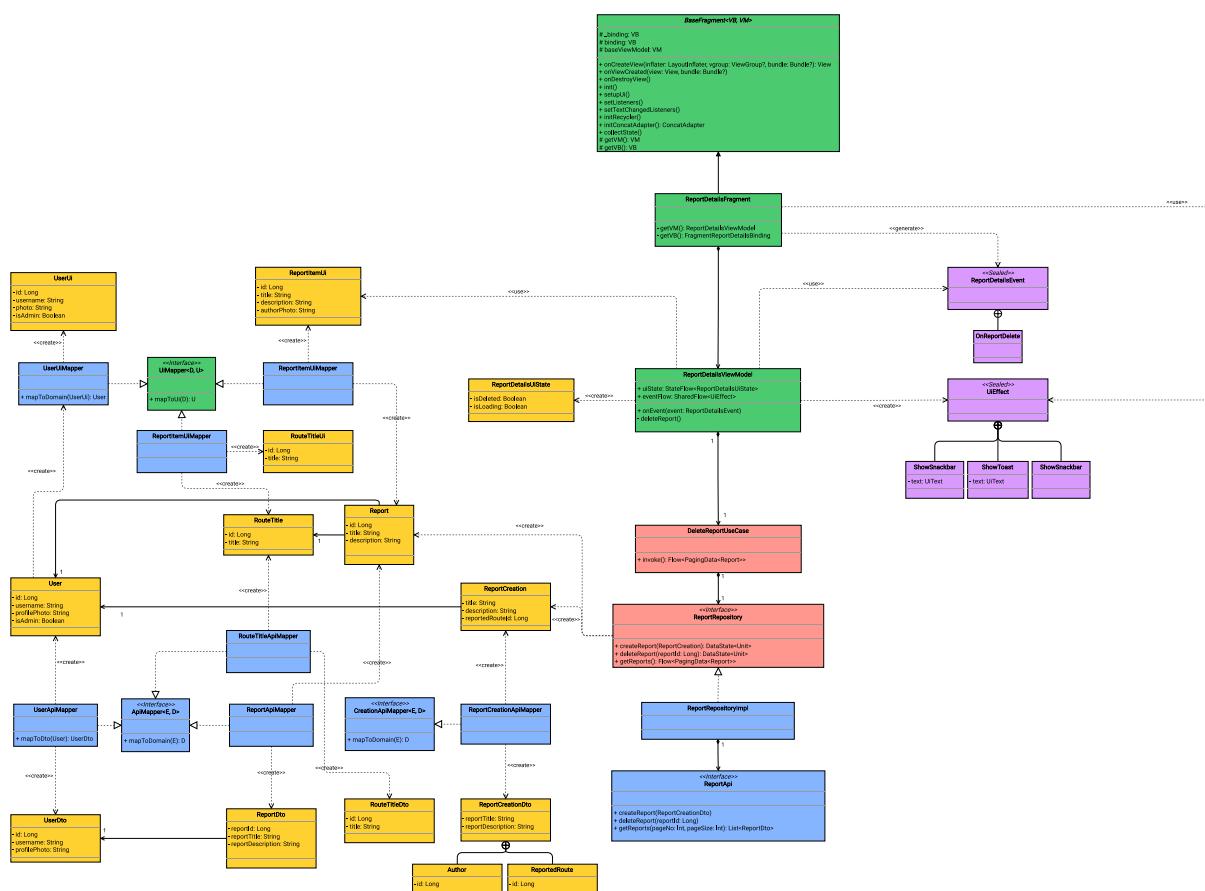


Figura 103: Dettagli segnalazione e eliminazione segnalazione

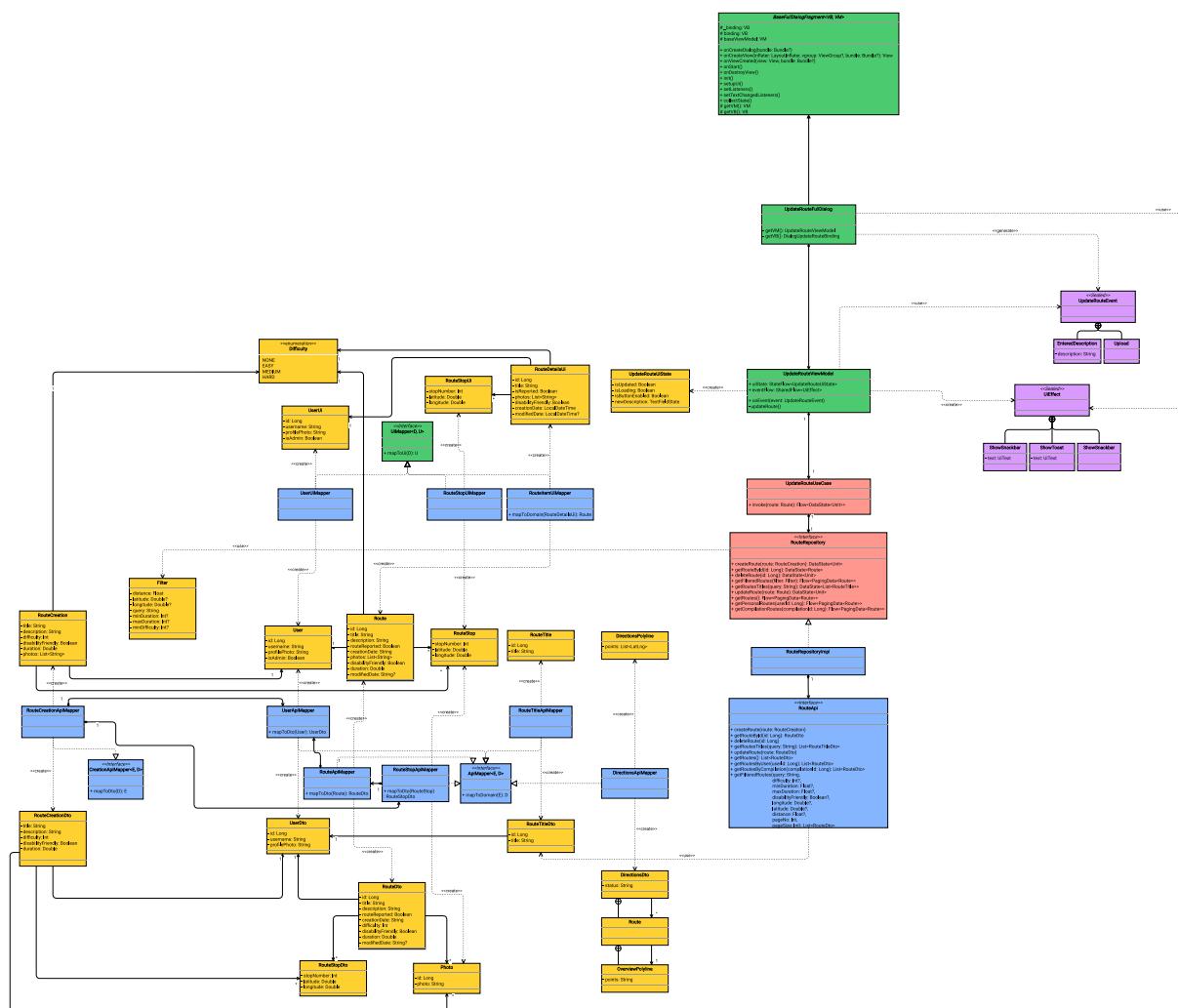


Figura 104: Modifica itinerario

2.5 Diagrammi di sequenza di design

Sono di seguito presentati i diagrammi di sequenza di design per due casi d'uso significativi.

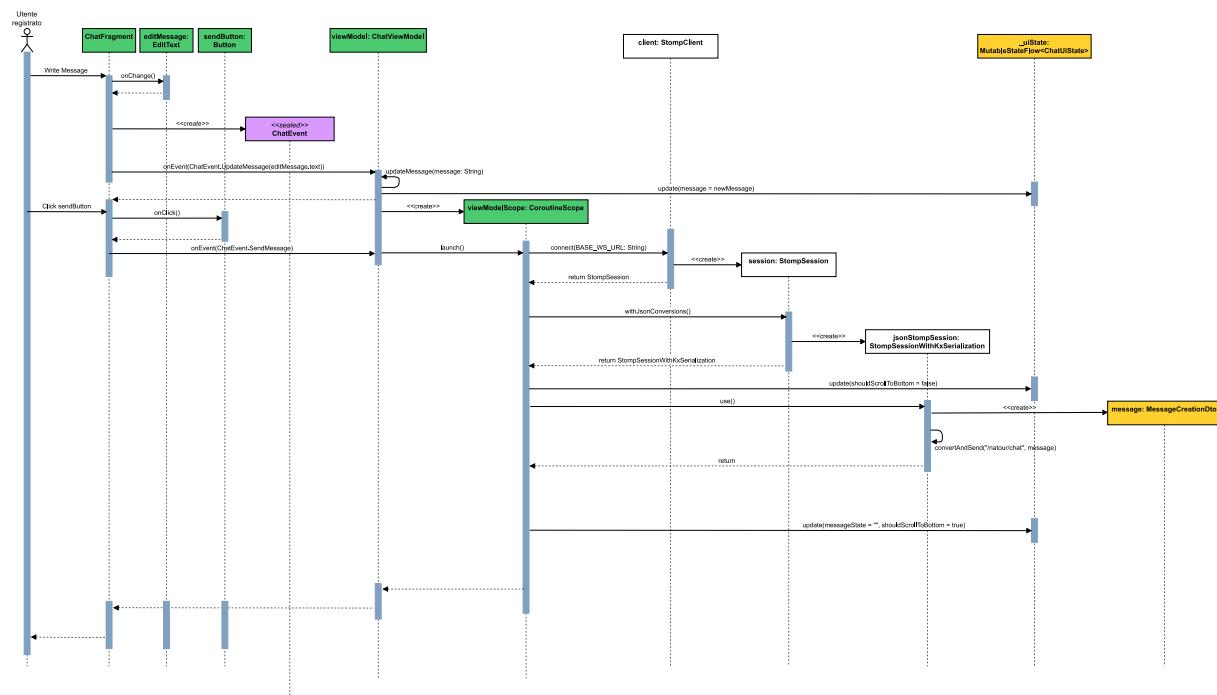


Figura 105: Invio messaggio

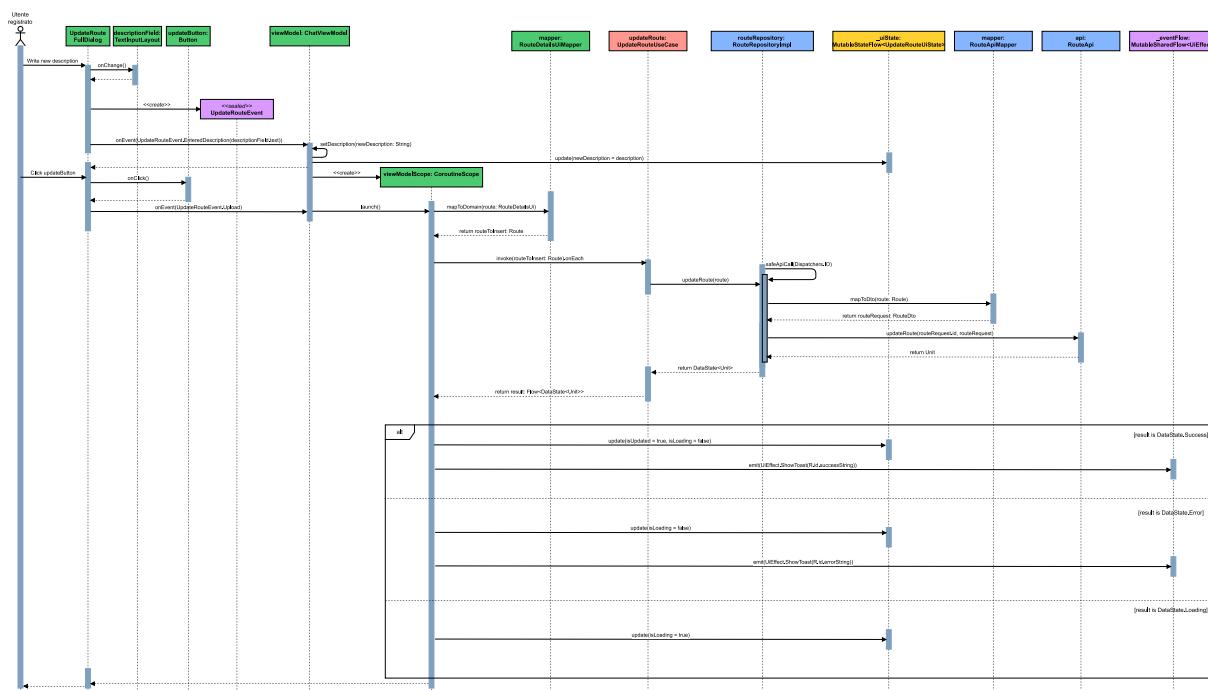
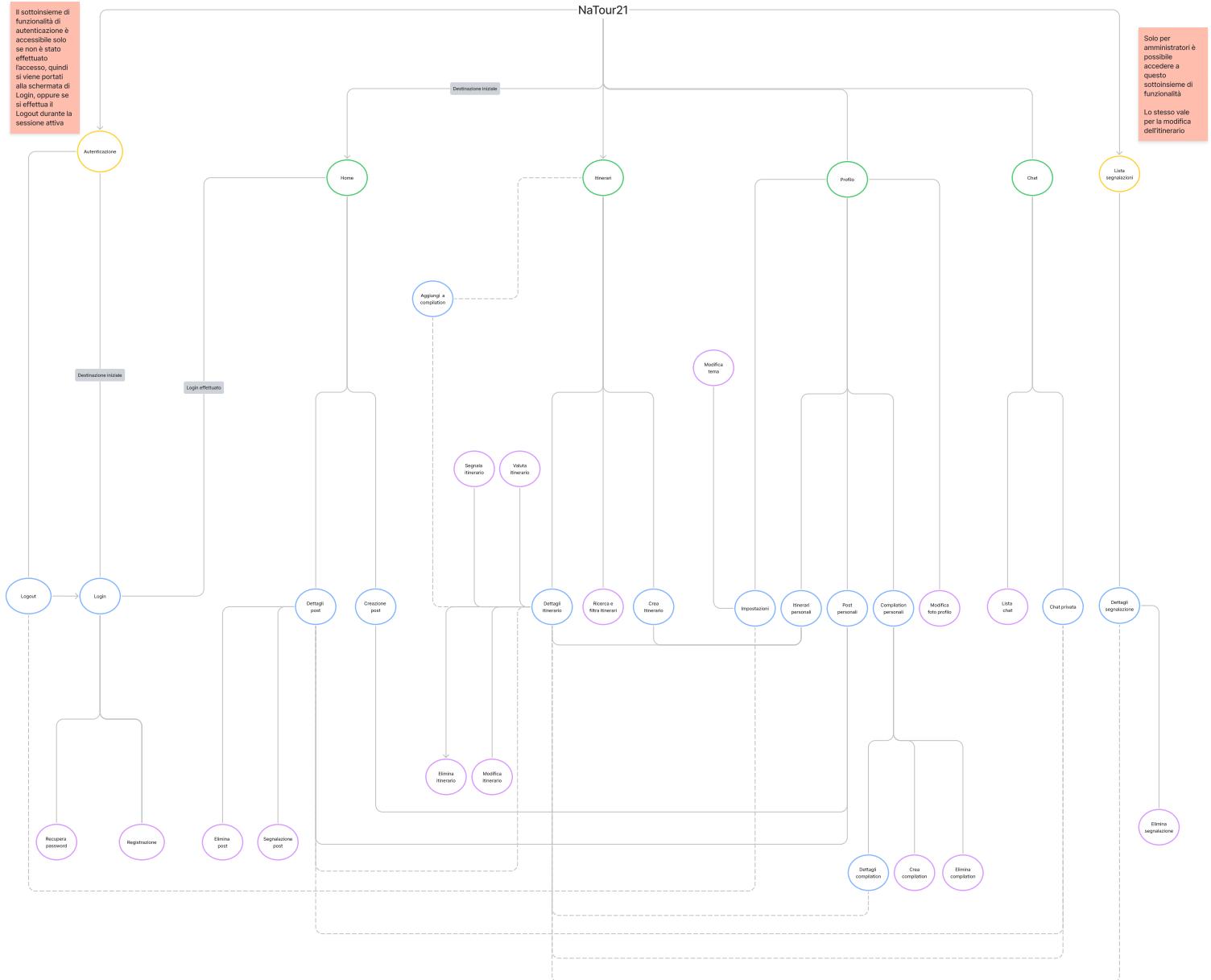


Figura 106: Modifica itinerario (amministratore)

2.6 Definizione delle gerarchie funzionali



3 Definizione di un piano di testing e valutazione sul campo dell'usabilità.

3.1 Codice xUnit per testing di 3 metodi

In questa sezione saranno trattati dei casi di test per 3 metodi non banali, adottando strategie specifiche per ognuno di questi. È bene precisare che - per effettuare alcuni test - è stato necessario utilizzare il framework open-source **Mockito**, che consente la creazione di *test double* (informalmente detti *mock*).

Mockito ha permesso di rimpiazzare le dipendenze delle classi dove sono presenti 2 dei 3 metodi sotto test, evitando la creazione di implementazioni fittizie per le dipendenze.

3.1.1 Metodo *form Validator*

Il metodo *form Validator* permette di validare gli input di un form di registrazione, di cui viene escluso il campo di conferma password perché verificato a priori.

Il metodo richiede tre parametri:

- **E-mail:** identifica un indirizzo email inserito;
- **Username:** identifica uno username inserito;
- **Password:** identifica la password inserita;

Per il metodo è stata adottata una strategia **black-box**. Di seguito le classi di equivalenza identificate:

Nome CE	Parametro	Valore
CE1	E-mail	Pattern <code>prefix@example.net</code> (valido)
CE2	E-mail	Prefisso con caratteri speciali +._%- (valido)
CE3	E-mail	Più di una @ (non valido)
CE4	E-mail	Due punti consecutivi prima del top-level domain (non valido)
CE5	E-mail	Punto dopo @ (non valido)
CE6	E-mail	Punto prima di @ (non valido)
CE7	Username	Contiene spazi (non valido)
CE8	Username	Almeno 3 caratteri (valido)
CE9	Username	Meno di 3 caratteri (non valido)
CE10	Password	Almeno 8 caratteri (valido)
CE11	Password	Meno di 8 caratteri (non valido)

Tabella 36: Classi di equivalenza individuate per *form Validator*

Nello specifico è stato adottato un approccio **WECT**, scelto in quanto, non essendovi correlazioni tra i parametri forniti, la copertura offerta è stata ritenuta sufficiente.

La minima copertura fornita da **WECT** in questo caso è vantaggiosa, è stato infatti possibile evitare tutte le combinazioni possibili, ottimizzando il numero di test per questo metodo.

Sono stati quindi individuati sei metodi di test:

ID Test	Email	Username	Password	Output
WECT 1	mattia@rossi.org	kotlin	coroutines	true
WECT 2	mari-o.%o@live.it	torvalds4ever	redhatlinuxenterprise	true
WECT 3	bianca@@unina.com	bc	ingsw	false
WECT 4	martin.fowler@tdd..us	mfowler	domaindrivendesign	false
WECT 5	kentbeck@.yahoo.com	kn	refactoring-man	false
WECT 6	robertc.martin.@clean.code	uncle bob	deathmarchphase	false

Tabella 38: RF.12

```

1 package com.unina.natourkt.core.domain.use_case.auth
2
3 import com.unina.natourkt.core.domain.repository.
4     AuthRepository
5
6 import org.junit.Before
7 import org.junit.Test
8 import org.junit.runner.RunWith
9 import org.mockito.InjectMocks
10 import org.mockito.Mock
11 import org.mockito.junit.MockitoJUnitRunner
12
13 @RunWith(MockitoJUnitRunner::class)
14 class RegistrationUseCaseTest {
15
16     @InjectMocks
17     private lateinit var registrationUseCase:
18         RegistrationUseCase
19
20     @Mock
21     private lateinit var repository: AuthRepository
22
23     @Before
24     fun setUp() {
25         registrationUseCase = RegistrationUseCase(repository)
26     }
27
28     @Test
29     fun `should register user`() {
30         // Given
31         val email = "test@example.com"
32         val password = "password123"
33
34         // When
35         val result = registrationUseCase.registerUser(email, password)
36
37         // Then
38         assertEquals("User registered successfully", result)
39     }
40
41     @Test
42     fun `should fail to register user if email is invalid`() {
43         // Given
44         val email = "invalid_email"
45         val password = "password123"
46
47         // When
48         val result = registrationUseCase.registerUser(email, password)
49
50         // Then
51         assertEquals("Email is invalid", result)
52     }
53
54     @Test
55     fun `should fail to register user if password is too short`() {
56         // Given
57         val email = "valid_email@example.com"
58         val password = "short"
59
60         // When
61         val result = registrationUseCase.registerUser(email, password)
62
63         // Then
64         assertEquals("Password must be at least 8 characters long", result)
65     }
66
67     @Test
68     fun `should fail to register user if both fields are empty`() {
69         // Given
70         val email = ""
71         val password = ""
72
73         // When
74         val result = registrationUseCase.registerUser(email, password)
75
76         // Then
77         assertEquals("Both fields cannot be empty", result)
78     }
79
80     @Test
81     fun `should fail to register user if both fields are null`() {
82         // Given
83         val email: String? = null
84         val password: String? = null
85
86         // When
87         val result = registrationUseCase.registerUser(email, password)
88
89         // Then
90         assertEquals("Both fields cannot be null", result)
91     }
92 }
```

```

25     }
26
27     @Test
28     fun `WECT 1 - Simple pattern, username with length
29         greater than or equal to 3, password with length
30         greater than or equal to 8`() {
31         val result = registrationUseCase.formValidator(
32             email = "mattia@rossi.org",
33             username = "kotlin",
34             password = "coroutines"
35         )
36
37         assertTrue(result)
38     }
39
40     @Test
41     fun `WECT 2 - Email's with special characters, username
42         length greater than or equal to 3, password with
43         length greater than or equal to 8`() {
44         val result = registrationUseCase.formValidator(
45             email = "mari-o.%o@live.it",
46             username = "torvalds4ever",
47             password = "redhatlinuxenterprise"
48         )
49
50         assertTrue(result)
51     }
52
53     @Test
54     fun `WECT 3 - More than one @ in email, username with
55         length less than 3, password with length less than
56         8`() {
57         val result = registrationUseCase.formValidator(
58             email = "bianca@unina.com",
59             username = "bc",
60             password = "ingsw"
61         )
62
63         assertFalse(result)
64     }
65
66     @Test
67     fun `WECT 4 - Two consecutive dots before top-level
68         domain, username with length greater than or equal to
69         3, password with length greater than or equal to 8`() {
70         val result = registrationUseCase.formValidator(

```

```

63             email = "martin.fowler@tdd..us",
64             username = "mfowler",
65             password = "domaindrivendesign"
66         )
67
68         assertFalse(result)
69     }
70
71     @Test
72     fun 'WECT 5 - One dot after @, username with length less
73         than 3, password with length greater than or equal to
74         8'() {
75         val result = registrationUseCase.formValidator(
76             email = "kentbeck@yahoo.com",
77             username = "kn",
78             password = "refactoring-man"
79         )
80
81         assertFalse(result)
82     }
83
84     @Test
85     fun 'WECT 6 - One dot before @, username with spaces,
86         password with greater than or equal to 8'() {
87         val result = registrationUseCase.formValidator(
88             email = "robertc.martin.@clean.code",
89             username = "uncle_bob",
90             password = "deathmarchphase"
91         )
92     }

```

✓ ✓ Test Results	225 ms
✓ com.unina.natourkt.core.domain.use_case.auth.RegistrationUseCaseTest	225 ms
✓ WECT 4 - Two consecutive dots before top-level domain, username with length greater than or equal to 3, password with length greater than or equal to 8	224 ms
✓ WECT 1 - Simple pattern, username with length greater than or equal to 3, password with length greater than or equal to 8	1 ms
✓ WECT 6 - One dot before @, username with spaces, password with greater than or equal to 8	0 ms
✓ WECT 5 - One dot after @, username with length less than 3, password with length greater than or equal to 8	0 ms
✓ WECT 3 - More than one @ in email, username with length less than 3, password with length less than 8	0 ms
✓ WECT 2 - Email's with special characters, username length greater than or equal to 3, password with length greater than or equal to 8	0 ms

 Figura 107: Risultati dei test sul metodo *formValidator*

3.1.2 Metodo *getChatMyMembersUseCase*

Il metodo *getChatByMembersUseCase* - che è in realtà implementato attraverso l'override dell'operatore *invoke()* della class *GetChatByMembersUseCase* - ha due parametri: gli id di due utenti registrati. Questo metodo permette il recupero di un'oggetto **Chat**, dati gli id degli utenti che la compongono, e il suo controllo.

L'implementazione utilizza le Coroutines di Kotlin, di conseguenza rende possibile *l'emissione di valori* al suo observer in qualsiasi momento.

- Se gli id non sono validi, il metodo emette *DataState.Cause.NotAcceptable* e poi ritorna;
- Se gli id corrispondono, il metodo emette *DataState.Cause.BadRequest* e poi ritorna;
- Una volta effettuata la chiamata al metodo presente nella dipendenza **ChatRepository**, viene effettuato un ulteriore controllo dell'oggetto recuperato, più precisamente sugli id dei "membri" dell'oggetto di tipo **Chat**. Se gli id corrispondono il metodo emette l'oggetto ottenuto sotto forma di *DataState.Success*, altrimenti emette *DataState.Cause.NotFound*.⁵

È stata adottata una strategia white-box, nello specifico con branch coverage, per assicurare che tutti i branch siano testati. La scelta deriva anche dalla natura dell'implementazione, dipendente dalle API del linguaggio di programmazione utilizzato.

I cammini possibili sono:

- **PATH 1** - 1, 2, 3, 4
- **PATH 2** - 1, 2, 5, 6, 7
- **PATH 3** - 1, 2, 5, 8, 9, 10, 11
- **PATH 4** - 1, 2, 5, 8, 9, 10, 12

Analizzati i cammini è stata individuata una lista di possibili input, utilizzando Mockito per testare il metodo sulla base del valore ritornato dalla dipendenza (predefinito, per prescindere dall'implementazione, visto che si tratta di uno Unit Test).

⁵In questo caso il ritorno è implicito

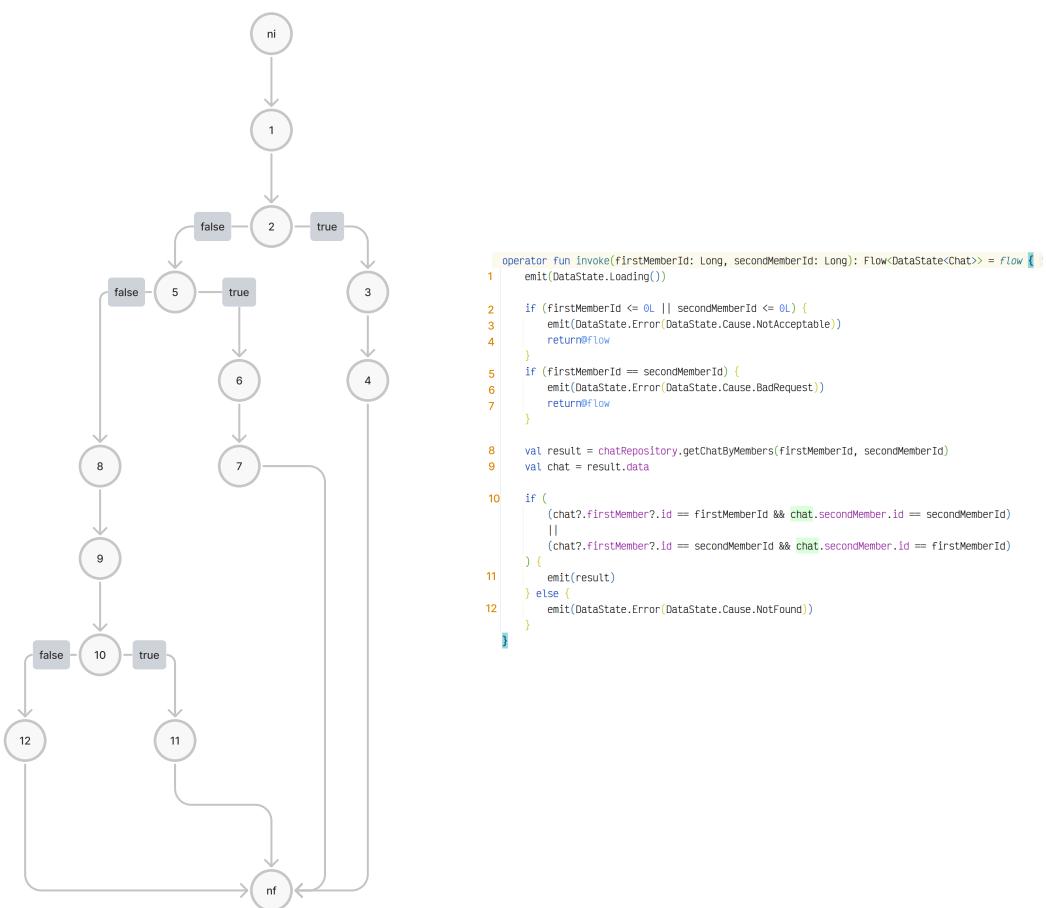


Figura 108: CFG del metodo *getChatByMembersUseCase*

```
1 package com.unina.natourkt.core.domain.use_case.chat
2
3 import com.unina.natourkt.core.domain.model.Chat
4 import com.unina.natourkt.core.domain.model.User
5 import com.unina.natourkt.core.domain.repository.
    ChatRepository
6 import com.unina.natourkt.core.util.DataState
7 import kotlinx.coroutines.ExperimentalCoroutinesApi
8 import kotlinx.coroutines.flow.last
9 import kotlinx.coroutines.test.runTest
10 import org.hamcrest.MatcherAssert.assertThat
11 import org.hamcrest.core.IsEqual.equalTo
12 import org.junit.Before
13 import org.junit.Test
14 import org.junit.runner.RunWith
15 import org.mockito.InjectMocks
16 import org.mockito.Mock
17 import org.mockito.junit.MockitoJUnitRunner
18 import org.mockito.kotlin.verify
19 import org.mockito.kotlin.whenever
20 import java.time.LocalDate
21
22 @ExperimentalCoroutinesApi
23 @RunWith(MockitoJUnitRunner::class)
24 class GetChatByMembersUseCaseTest {
25
26     // Class to Test
27     @InjectMocks
28     private lateinit var getChatByMembers:
        GetChatByMembersUseCase
29
30     // Mocked dependency
31     @Mock
32     private lateinit var repository: ChatRepository
33
34     // Utilities
35     private lateinit var firstMember: User
36     private lateinit var secondMember: User
37     private lateinit var brokenUser: User
38     private lateinit var dummyChat: Chat
39
40     @Before
41     fun setUp() {
42         getChatByMembers = GetChatByMembersUseCase(repository
43             )
44     }
```

```

45     @Test
46     fun `PATH 1 - when the given IDs are minus or equal to
        zero, it should return NotAcceptable as last flow
        value`() =
47         runTest {
48             firstMember = User(-1, "marietto", false, "")
49             secondMember = User(2, "bianca", true, "")
50
51             val result = getChatByMembers(firstMember.id,
52                 secondMember.id).last()
53             assertThat(result.error, equalTo(DataState.Cause.
54                 NotAcceptable))
55         }
56
57     @Test
58     fun `PATH 2 - when the given IDs are equal, it should
        return BadRequest as last flow value`() =
59         runTest {
60             firstMember = User(2, "marietto", false, "")
61             secondMember = firstMember
62
63             val result = getChatByMembers(firstMember.id,
64                 secondMember.id).last()
65             assertThat(result.error, equalTo(DataState.Cause.
66                 BadRequest))
67         }
68
69     @Test
70     fun `PATH 3 - when the IDs are good to go, the result
        data should be a Chat entity`() {
71         runTest {
72             firstMember = User(2, "marietto", false, "")
73             secondMember = User(3, "mattia", isAdmin = false,
74                 "")
75             dummyChat = Chat(34, LocalDate.now(), firstMember,
76                 secondMember)
77
78             val request = repository.getChatByMembers(
79                 firstMember.id, secondMember.id) // Make the
80                 request through repository dependency
81             whenever(request).thenReturn(DataState.Success(
82                 dummyChat)) // Mockito allows the return of
83                 default values when a method is executed
84
85             val result = getChatByMembers(firstMember.id,
86                 secondMember.id).last()
87         }
88     }

```

```

77         verify(repository).getChatByMembers(firstMember.
78             id, secondMember.id) // Helps to ensure
79             that the method is executed ONLY one time
80     assertThat(result.data, equalTo(dummyChat))
81   }
82
83   @Test
84   fun 'PATH 4 - when the IDs are good to go but the chat is
85       not found, the result error should be a NotFound'() {
86     runTest {
87       firstMember = User(2, "marietto", false, "")
88       secondMember = User(3, "mattia", false, "")
89       brokenUser = User(5, "bianca", true, "")
90       dummyChat = Chat(34, LocalDate.now(), firstMember
91           , brokenUser)
92
93       val request = repository.getChatByMembers(
94           firstMember.id, secondMember.id) // Make the
95           request through repository dependency
96       whenever(request).thenReturn(DataState.Success(
97           dummyChat)) // Mockito allows the return of
98           default values when a method is executed
99     }
100   }
101 }
```

Test Results		321ms
com.unina.natourkt.core.domain.use_case.chat.GetChatByMembersUseCaseTest		321ms
PATH 4 - when the IDs are good to go but the chat is not found, the result error should be a NotFound		319ms
PATH 3 - when the IDs are good to go, the result data should be a Chat entity		1ms
PATH 1 - when the given IDs are minus or equal to zero, it should return NotAcceptable as last flow value		0ms
PATH 2 - when the given IDs are equal, it should return BadRequest as last flow value		1ms

Figura 109: Risultati dei test sul metodo *getChatByMembersUseCase*

3.1.3 Metodo *retrofitSafeCall*

Il metodo *retrofitSafeCall* è il fulcro delle richieste HTTP effettuate dall'applicativo, è stato ritenuto uno dei metodi più interessanti e particolari da testare data la notevole riduzione di codice *boilerplate* ottenuta. Il metodo ha come parametri:

- Un attributo di tipo **CoroutineDispatcher**: *dispatcher*, utilizzato all'interno delle *Couroutines*;
- Un attributo di tipo **Long**: *timeout*, per indicare un tempo massimo di timeout per la richiesta effettuata;
- L'ultimo attributo in realtà è una funzione lambda sospensiva, che restituisce un tipo generico *T*

I valori di ritorno dovrebbero essere contenuti nell'oggetto *DataState*, questi dipendono da:

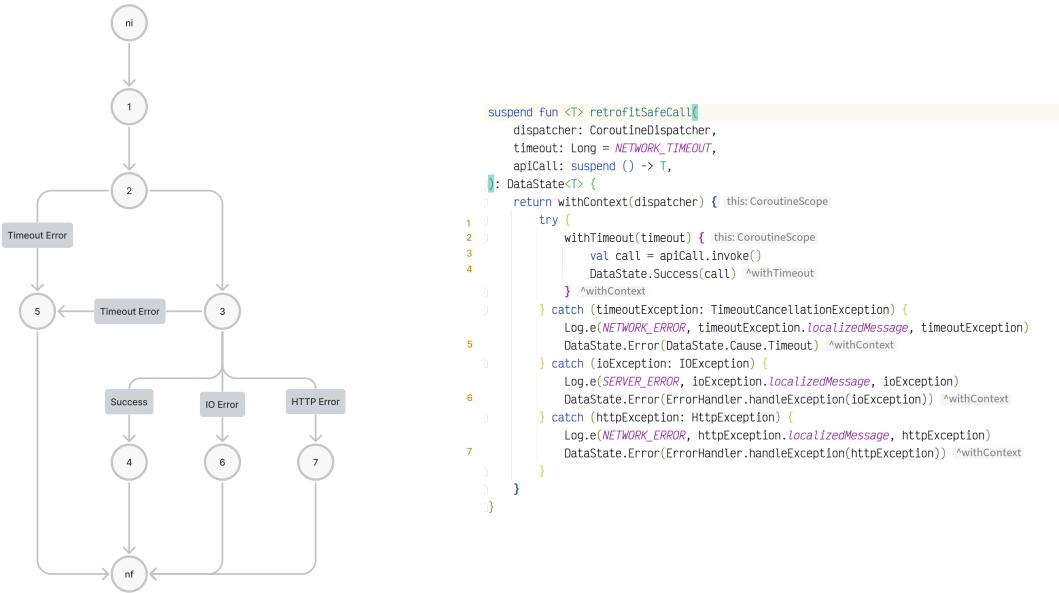
- Timeout fornito in input: un timeout minore o uguale a zero potrebbe tirare un'eccezione prima del previsto; qualora non fosse così, la lambda fornita potrebbe comunque far scattare il timeout;
- Condizioni di errore della chiamata a funzione.

È stata adottata una strategia white-box, nello specifico con branch coverage, per assicurare che tutti i branch siano testati. Rispetto al precedente metodo l'individuazione dei branch è più sottile, questo è dovuto alla presenza del blocco *try-catch*.

I cammini possibili sono:

- **PATH 1** - 1, 2, 5
- **PATH 2** - 1, 2, 3, 4
- **PATH 3** - 1, 2, 3, 5
- **PATH 4** - 1, 2, 3, 6
- **PATH 5** - 1, 2, 3, 7

Analizzati i cammini è stata individuata una lista di possibili input, utilizzando Mockito per testare il metodo sulla base del valore ritornato dalla dipendenza (predefinito, per prescindere dall'implementazione, visto che si tratta di uno Unit Test).


 Figura 110: CFG del metodo *retrofitSafeCall*

```

1 package com.unina.natourkt.core.data.util
2
3 import com.unina.natourkt.core.util.DataState
4 import kotlinx.coroutines.ExperimentalCoroutinesApi
5 import kotlinx.coroutines.delay
6 import kotlinx.coroutines.test.TestDispatcher
7 import kotlinx.coroutines.test.UnconfinedTestDispatcher
8 import kotlinx.coroutines.test.runTest
9 import kotlinx.coroutines.withTimeout
10 import okhttp3.MediaType.Companion.toMediaType
11 import okhttp3.MediaType.Companion.toMediaTypeOrNull
12 import okhttp3.ResponseBody.Companion.toResponseBody
13 import okhttp3.internal.wait
14 import org.hamcrest.MatcherAssert.assertThat
15 import org.hamcrest.core.IsEqual.equalTo
16 import org.junit.Before
17 import org.junit.Rule
18 import org.junit.Test
19 import retrofit2.HttpException
20 import retrofit2.Response
21 import java.io.IOException
22
23 @ExperimentalCoroutinesApi
24 class RetrofitHelperExtension {
25

```

```
26     @get:Rule
27     val coroutineRuleForTest: CoroutineTestRule =
28         CoroutineTestRule()
29
30     private lateinit var dispatcher: TestDispatcher
31
32     @Before
33     fun setup() {
34         dispatcher = UnconfinedTestDispatcher()
35     }
36
37     @Test
38     fun 'PATH 1 - when the TimeoutCancellationException is
39         thrown due to timeout less than or equal to 0, it
40         should emit Timeout Error'() {
41         runTest {
42             val result = retrofitSafeCall(dispatcher =
43                 dispatcher, timeout = -1L) { }
44
45             assertThat(DataState.Cause.Timeout, equalTo(
46                 result.error))
47         }
48     }
49
50     @Test
51     fun 'PATH 2 - when the lambda function returns without
52         error after a network call, it should emit success
53         with generic type data corresponding to the one given
54         as parameter'() =
55         runTest {
56             val lambdaExpected = "String\u2020expect\u2020due\u2020to\u2020no\u2020
57                 operation\u2020by\u2020lambda"
58             val result = retrofitSafeCall(dispatcher =
59                 dispatcher, timeout = 5L) { lambdaExpected }
60             // NOTE: We are already passing the expected
61             // value to this lambda, the variable is not
62             // important for this test!
63
64             assertThat(lambdaExpected, equalTo(result.data))
65         }
66
67     @Test
68     fun 'PATH 3 - when the TimeoutCancellationException is
69         thrown due request expiration, it should emit Timeout
70         Error'() {
71         runTest {
```

```

58         val result = retrofitSafeCall(dispatcher =
59             dispatcher, timeout = 2L) { delay(2L) } // Delay for time = timeout, so we can trigger
60             timeout exceeded
61     }
62 }
63
64 @Test
65 fun `PATH 4 - when an IOException is thrown in the lambda
66 , it should emit Network Error`() {
67     runTest {
68         val result =
69             retrofitSafeCall(dispatcher = dispatcher,
70                 timeout = 5L) { throw IOException() } // Directly throws the "desired" exception
71     }
72 }
73
74 @Test
75 fun `PATH 5 - when an HttpException is thrown in the
76 lambda, it should emit HTTPGeneric`() {
77     runTest {
78         val body =
79             "{\"Request\u00a9not\u00a9processable\"]}".
80             toResponseBody("application/json".
81             toMediaType()) // The body is required
82             for HttpException
83
84         val result =
85             retrofitSafeCall(dispatcher = dispatcher,
86                 timeout = 5L) {
87                 throw HttpException( // Directly
88                     throws the "desired" exception
89                     Response.error<Any>(
90                         422,
91                         body
92                     )
93                 }
94     }
95
96     assertThat(DataState.Cause.HTTPGeneric, equalTo(

```

```
        result.error))  
91    }  
92  }  
93 }
```

Test Results		125 ms
✓	com.unina.natourkt.core.data.util.RetrofitHelperExtension	125 ms
✓	PATH 1 - when the TimeoutCancellationException is thrown due to short timeout value or this timeout is exceeded, it should emit Timeout Error	66 ms
✓	PATH 5 - when an HTTPException is thrown in the lambda, it should emit HTTPGeneric	49 ms
✓	PATH 3 - when the TimeoutCancellationException is thrown due request expiration, it should emit Timeout Error	7 ms
✓	PATH 4 - when an IOException is thrown in the lambda, it should emit Network Error	2 ms
✓	PATH 2 - when the lambda function returns without error after a network call, it should emit success with generic type data corresponding to the one given as parameter	1 ms

Figura 111: Risultati dei test sul metodo *retrofitSafeCall*

3.2 Valutazione dell'usabilità sul campo