



UNIVERSITÀ^{DEGLI STUDI DI}
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Studi in Ingegneria Informatica



Documentazione del sistema software *Travelgram*

Anno Accademico 2019/2020



Pasquale Di Maio - Mat. M63000837
Fabrizio Guillaro - Mat. M63000851
Ivano Iodice - Mat. M63000897
Fabio Maresca - Mat. M63000846

Indice

1 Processo di Sviluppo	2
1.1 Tool e Tecnologie Utilizzate	2
1.1.1 Strumenti per la condivisione del lavoro	2
1.1.2 Strumenti e tecnologie per lo sviluppo	7
1.1.3 Servizi esterni	10
1.2 Tecniche e pratiche agili	12
1.2.1 Altre Tecniche Agili	13
1.3 Stima dei costi	14
1.3.1 Function Points	14
1.3.2 Use Case Points	14
2 Avvio del Progetto	16
2.1 Glossario	16
2.2 Visione	17
2.2.1 Obiettivi Generali	17
2.2.2 Requisiti Generali	18
2.2.3 Vincoli	19
2.2.4 Positioning	19
2.2.5 Parti interessate	20
3 Specifica dei Requisiti	21
3.1 Requisiti Funzionali	21
3.1.1 Casi d'Uso in Formato Breve	21
3.1.2 Casi d'Uso in Formato Dettagliato	23
3.2 Specifiche Supplementari	28
3.3 Mockup e Design Concepts	28
4 Analisi del Sistema	30
4.1 Use Case Diagram	30
4.2 System Sequence Diagrams	32
4.2.1 Pubblica Memory	32
4.2.2 Segna Luogo	33
4.2.3 Segui Traveler	33
4.3 Activity Diagrams	34
4.3.1 Visualizza Memories	34

4.4 Textual Analysis	35
4.5 System Domain Model	36
5 Architettura e Progettazione	40
5.1 Vista Componenti e Connettori	40
5.2 Architettura Client-Server	42
5.3 Architettura MVVM	43
5.4 Data Model	45
5.4.1 Firestore	45
5.4.2 Authentication	46
5.4.3 Cloud Storage	47
5.5 Package Diagram	48
5.6 Class Diagrams	50
5.6.1 Package View	52
5.6.2 Package ViewModel	53
5.6.3 Package Model	54
5.7 Context Diagram with Boundary	56
5.8 Dinamica del Sistema	56
5.8.1 Pubblica Memory	58
5.8.2 Visualizza Memory	62
5.8.3 Sign up	64
5.9 Deployment	67
6 Documentazione dell'implementazione e manuale d'uso	69
6.1 Organizzazione del Codice	69
6.2 Descrizione dei file realizzati	72
6.3 Design Pattern	74
6.3.1 Pattern GRASP	75
6.3.2 Pattern Repository	76
6.3.3 Pattern Observer	76
6.4 Firestore Query	80
6.5 Manuale di Utilizzo	81
6.5.1 Configurazione ed installazione	81
6.5.2 Flow di navigazione ed utilizzo	81
7 Testing	86
7.1 Necessità del testing semi-automatico	86
7.2 Fase di Capture	87
7.3 Generazione dei Test Cases	89
7.3.1 Test Case 1 - Signup	89
7.3.2 Test Case 2 - Login	93
7.3.3 Test Case 3 - Following	95

Introduzione

Quattro giovani studenti del corso di laurea magistrale in Ingegneria Informatica dell’Università degli Studi di Napoli Federico II hanno avuto l’idea di realizzare *Travelgram*: una semplice applicazione smartphone che consente agli utenti di condividere i ricordi dei propri viaggi con gli altri, e di supportarsi a vicenda fornendo recensioni sui luoghi visitati. Ogni utente può trarre vantaggio dall’utilizzo di questa applicazione per orientare e chiarire le scelte dei posti da vedere e dei viaggi da programmare, condividendo le proprie esperienze e opinioni.

Gli utenti sono dei viaggiatori, pertanto ci si riferirà ad essi con il termine *traveler*. Ogni traveler può registrarsi in maniera semplice e gratuita all’applicazione e iniziare ad esplorare la mappa interattiva segnando quali sono i luoghi che intende visitare o che ha già visitato, pubblicando *memory* e recensioni dei propri viaggi. Si può interagire con gli altri traveler aggiungendoli alla lista dei traveler seguiti, ciò consente di visualizzare i ricordi di altri utenti.

Una breve anteprima è presentata al seguente link: <https://www.youtube.com/watch?v=QBkgVoCjopg>

Questo elaborato descrive il processo di sviluppo, la documentazione e le fasi del progetto che il team di sviluppo ha eseguito per produrre *Travelgram*.

Capitolo 1

Processo di Sviluppo

Al fine di poter realizzare il progetto appena descritto, è stato necessario utilizzare strumenti e pratiche ad hoc per organizzare al meglio il lavoro del team di sviluppo.

Per questo motivo si mostrano in questo capitolo tutte le tecnologie ed i tool utilizzati durante lo sviluppo del progetto (figura 1.1), e le pratiche agili adoperate al fine di migliorare la produttività del team.

1.1 Tool e Tecnologie Utilizzate



Figura 1.1: Loghi degli strumenti utilizzati

1.1.1 Strumenti per la condivisione del lavoro

Microsoft Teams: Il primo strumento da citare che ha permesso al team di poter lavorare e comunicare a distanza durante tutta la realizzazione del progetto, dovuta all'impossibilità di incontrarsi in gruppo, è stato *Microsoft Teams* (figura 1.2). Infatti, tramite la piattaforma è stato possibile effettuare delle riunioni virtuali che hanno permesso al team di "incontrarsi" giornalmente in modo da aggiornarsi sullo stato di avanzamento dei lavori.

CAPITOLO 1. PROCESSO DI SVILUPPO

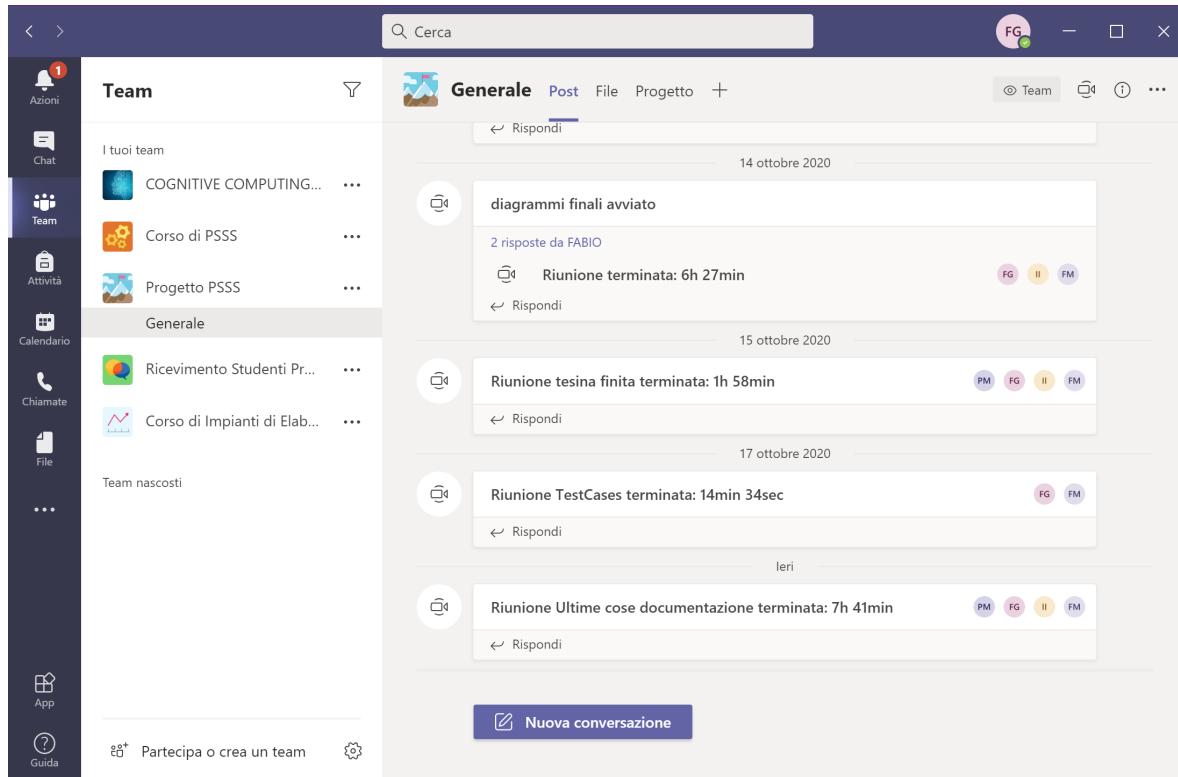


Figura 1.2: Microsoft Teams

Google Docs e Microsoft OneNote: Sin dalle prime riunioni, è stato necessario adoperare molti strumenti per il brainstorming, quali *Google Docs* (figura 1.3) e *OneNote* (figura 1.4), al fine di poter lavorare contemporaneamente su di un unico documento condiviso.

The screenshot shows a Google Docs document titled 'Formato breve'. The document contains the following text:

Attori primari: Utente non registrato, Traveler
Attori di supporto: Follower, Seguito

Il grassetto indica che la descrizione del caso d'uso in formato breve è stata fatta nella fase di ideazione, per i casi d'uso ritenuti principali.

Attore	Nome	Descrizione in formato breve
Traveler	Pubblica Memory	Il traveler aggiunge un nuovo ricordo relativo ad un luogo.
	Segna Luogo	Il traveler contrassegna un luogo di interesse come visitato o da visitare interagendo con la mappa.
	Segui Traveler	Il traveler aggiunge un determinato profilo nella propria lista di utenti seguiti.

Figura 1.3: Google Docs

CAPITOLO 1. PROCESSO DI SVILUPPO

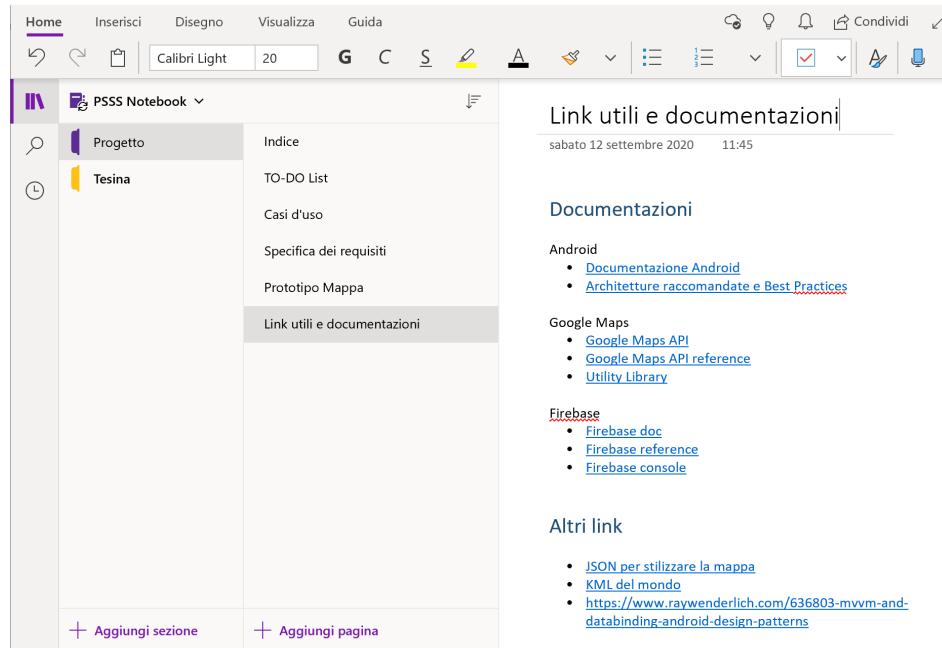


Figura 1.4: OneNote

Trello: Una volta definita l'idea e prodotti i primi documenti di Analisi del progetto, è stato necessario chiarire quali funzionalità avessero la precedenza, e quindi, come dover dividere i compiti da svolgere nelle varie iterazioni e fasi del lavoro. Per questo motivo è stato fondamentale l'utilizzo di *Trello*, uno strumento molto utile per tenere sempre sotto controllo ogni compito da realizzare in ogni Sprint. Infatti esso mette a disposizione delle lavagne virtuali all'interno delle quali è possibile definire delle sezioni personalizzabili. Nel nostro caso sono state create due bacheche, rispettivamente “Travelgram - Fase di Ideazione”, con le sezioni “Cose da fare”, “In esecuzione” e “Fatto”, e “Travelgram - Fase di Elaborazione” (figura 1.5), con le sezioni “Cose da fare”, “Cose da fare - Iterazione corrente”, “In esecuzione - Iterazione corrente” e “Fatto”. Sta quindi ad ogni membro del team spostare manualmente ogni task nelle categorie adatte man mano che si avanza nei lavori.

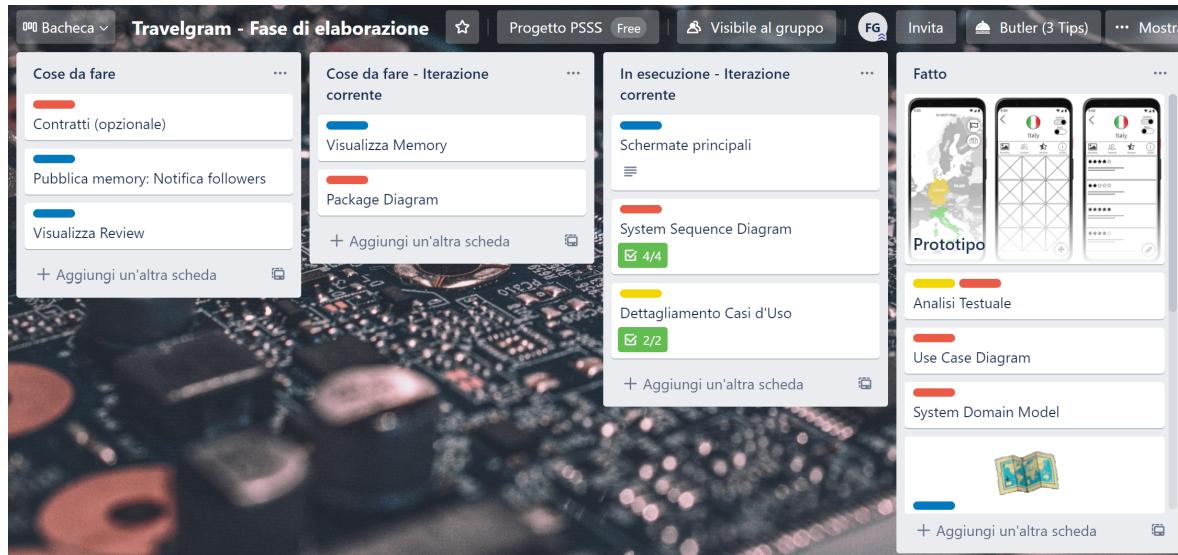


Figura 1.5: Trello - bacheca per la fase di elaborazione

GitHub: Per garantire la gestione del codice, si è fatto uso di un sistema di Version-Control (VCS) capace di supportare il Configuration Management e le sue attività, ossia Version Management, System Building, Change Management e Release Management. Il VCS scelto è stato *Git*, un noto VCS Distribuito con il sistema di hosting e gestione delle repository basato su cloud offerto da *GitHub*^[2] (figura 1.6). Questo è stato utile per lavorare su parti diverse del codice, o in momenti diversi, tenendo traccia di tutte le modifiche effettuate garantendo una semplice risoluzione dei conflitti laddove presenti.

Per semplificare ulteriormente il lavoro, si è fatto uso anche della versione desktop di GitHub, *GitHub Desktop* (figura 1.7).

CAPITOLO 1. PROCESSO DI SVILUPPO

The screenshot shows a GitHub repository page for 'fabiom95 / ProgettoPSSS_Travelgram'. The top navigation bar includes links for Pull requests, Issues, Marketplace, and Explore. Below the header, the repository name is displayed, along with an 'Unwatch' button and a '2' indicating two notifications. A navigation bar below the repository name offers links to Code, Issues, Pull requests, Actions, Projects, Wiki, and Security. The 'Code' link is highlighted with a red underline. Below this, a summary bar shows 'master' (with a dropdown arrow), '1 branch', '0 tags', and buttons for 'Go to file', 'Add file', and 'Code' (which is highlighted in green). A list of recent commits is shown, all made by 'fabrizioguillaro' 12 hours ago, with a total of 151 commits. The commits are: 'update README', 'update README', 'caricamento apk', 'ulteriori aggiustamenti readme', and 'Update README.md'. At the bottom, there is a preview of the 'README.md' file, which contains the following text:

credits go to Fabio Maresca, Pasquale Di Maio, Fabrizio Guillaro, Ivano Iodice; this is an Italian team, so the ReadMe is built in two English sections and the Italian Introduction as well:

1. English summary
2. Repository organization
3. Italian summary

Figura 1.6: GitHub

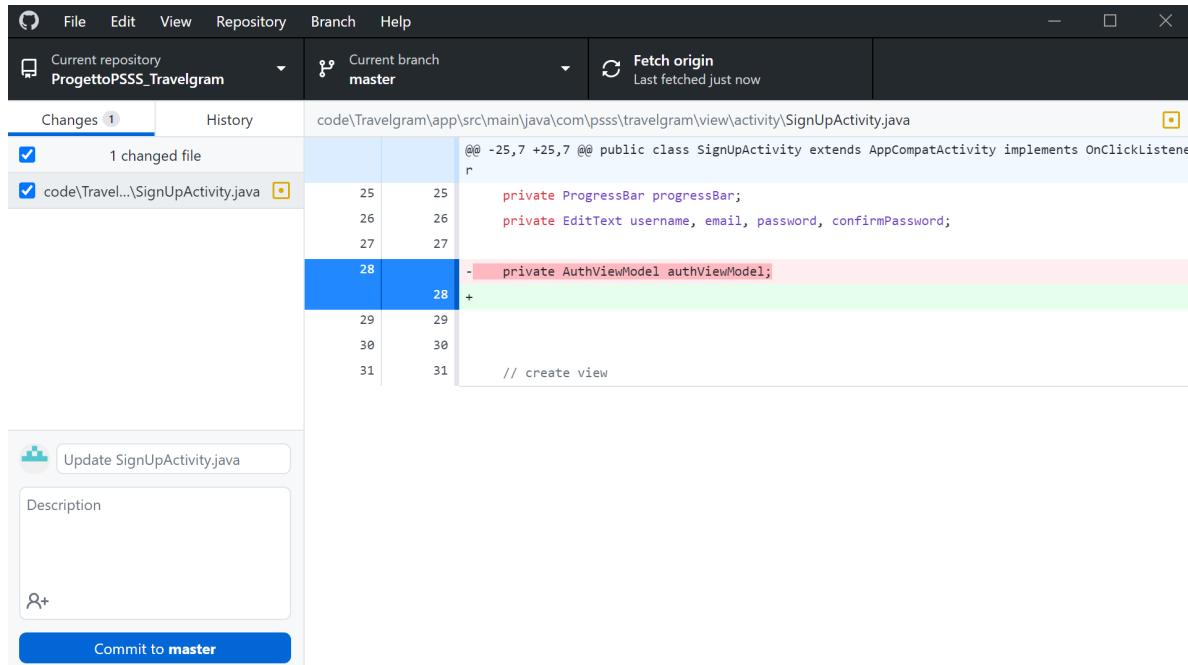


Figura 1.7: GitHub Desktop

1.1.2 Strumenti e tecnologie per lo sviluppo

Visual Paradigm: Per modellare diagrammi aderenti allo standard UML che rappresentassero le viste più adeguate della nostra Architettura Software abbiamo utilizzato lo strumento studiato durante il corso, *Visual Paradigm* (figura 1.8). In particolare, è stata usata inizialmente la versione di prova di Visual Paradigm Enterprise, allo scadere della quale è stata usata la Community Edition.

Sono inoltre stati creati degli account su *Visual Paradigm Online*, piattaforma che ha permesso di usufruire del servizio “Team Collaboration” (figura 1.9), sistema di Version-Control (VCS) che permette ad ogni componente del team di lavorare in proprio su una copia locale dei diagrammi, andando a realizzare così un meccanismo di Configuration Management. Questo strumento semplifica notevolmente la gestione delle versioni dei diagrammi, dal momento che, a differenza di altri VCS come GitHub, non tratta il file .vpp come un unico elemento, ma apprezza le variazioni di ogni singolo componente di ogni diagramma, mostrando di volta in volta le specifiche modifiche e gli eventuali conflitti.

È possibile visualizzare l’integrazione di Team Collaboration nella versione desktop in alto in figura 1.8.

CAPITOLO 1. PROCESSO DI SVILUPPO

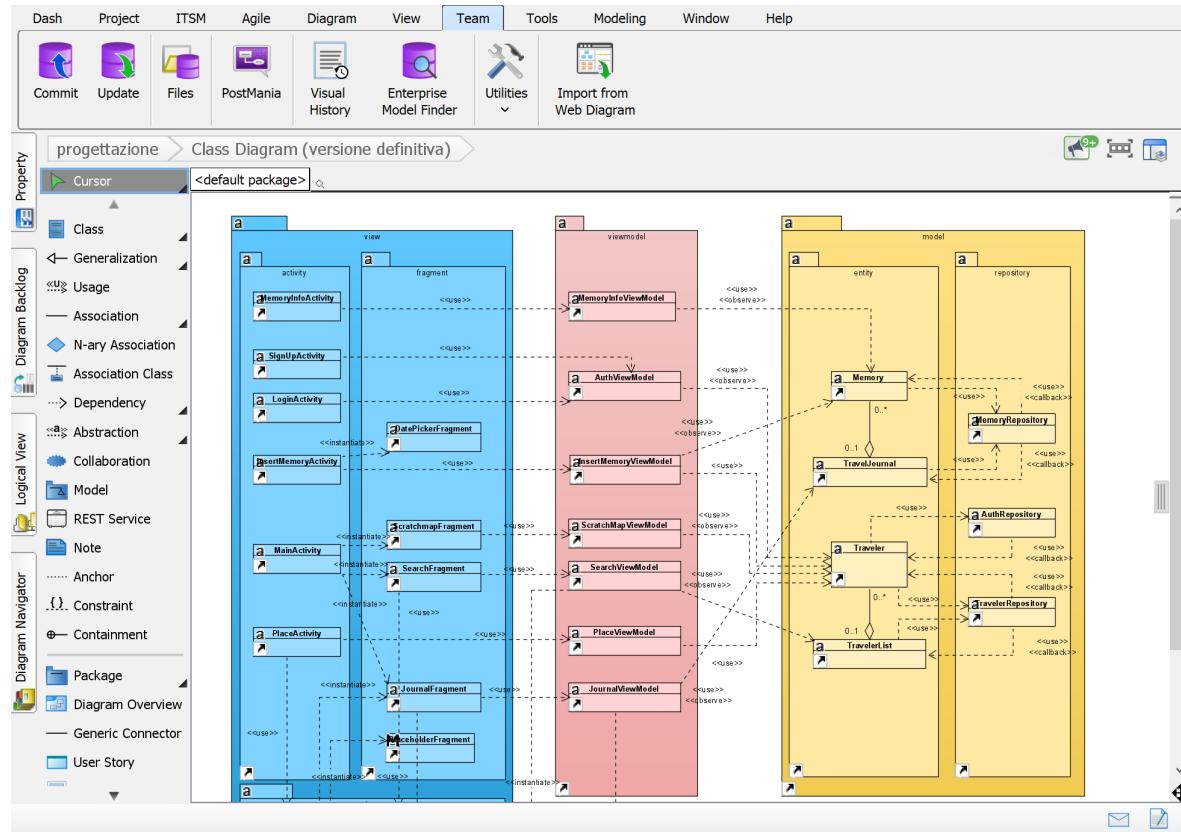


Figura 1.8: Visual Paradigm

	Members	Projects	Last Login	Status	Desktop	Online
<input type="checkbox"/>	Fabio	1	Oct 15, 2020	Active	Community	Express
<input type="checkbox"/>	Fabrizio	1	Oct 19, 2020	Active	Enterprise Ev...	Express
<input type="checkbox"/>	Ivano	1	Oct 14, 2020	Active	Community	Express
<input type="checkbox"/>	Lino	1	Oct 20, 2020	Active	Community	Express

Figura 1.9: Gestione della collaborazione con Visual Paradigm Online

MockFlow: Per realizzare dei mockup che ci guidassero nella progettazione dell’Interfaccia Utente Grafica (GUI) abbiamo utilizzato *MockFlow* con il suo editor *WireframePro* (figura 1.10), uno strumento disponibile online che consente di utilizzare funzionalità legate alla prototipazione ed al design.

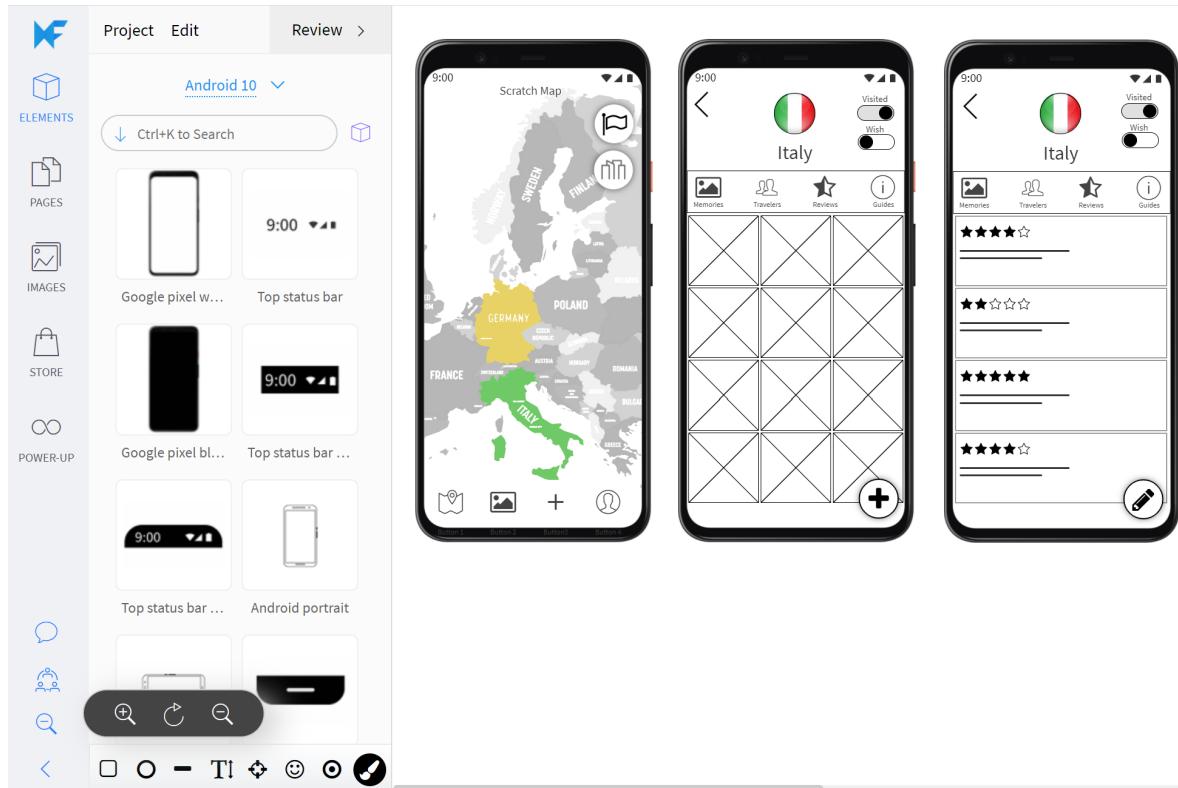


Figura 1.10: WireframePro su MockFlow

Android Studio: Una volta stabilite le funzionalità da realizzare con maggior priorità ed i task che le componessero, è stato il momento di definire quali tecnologie e quale linguaggio di programmazione utilizzare per supportare lo sviluppo dell’applicazione. Abbiamo subito scelto all’unanimità di utilizzare l’IDE proposto da Google per lo sviluppo di Applicazioni Android, *Android Studio*^[3] (figura 1.11), che mette a disposizione molte funzionalità per semplificare la realizzazione di applicazioni, tra cui un emulatore integrato dei possibili dispositivi Android, la possibilità di integrare strumenti di Version-Control, Gradle come sistema di build automatizzato per il progetto e molto altro. Come linguaggio di programmazione, la possibile scelta ricadeva tra *Kotlin* o *Java*; nonostante il primo sia molto consigliato anche dalla community, si è scelto di optare per un linguaggio ormai noto al team che conosce le sue particolarità e i suoi costrutti.

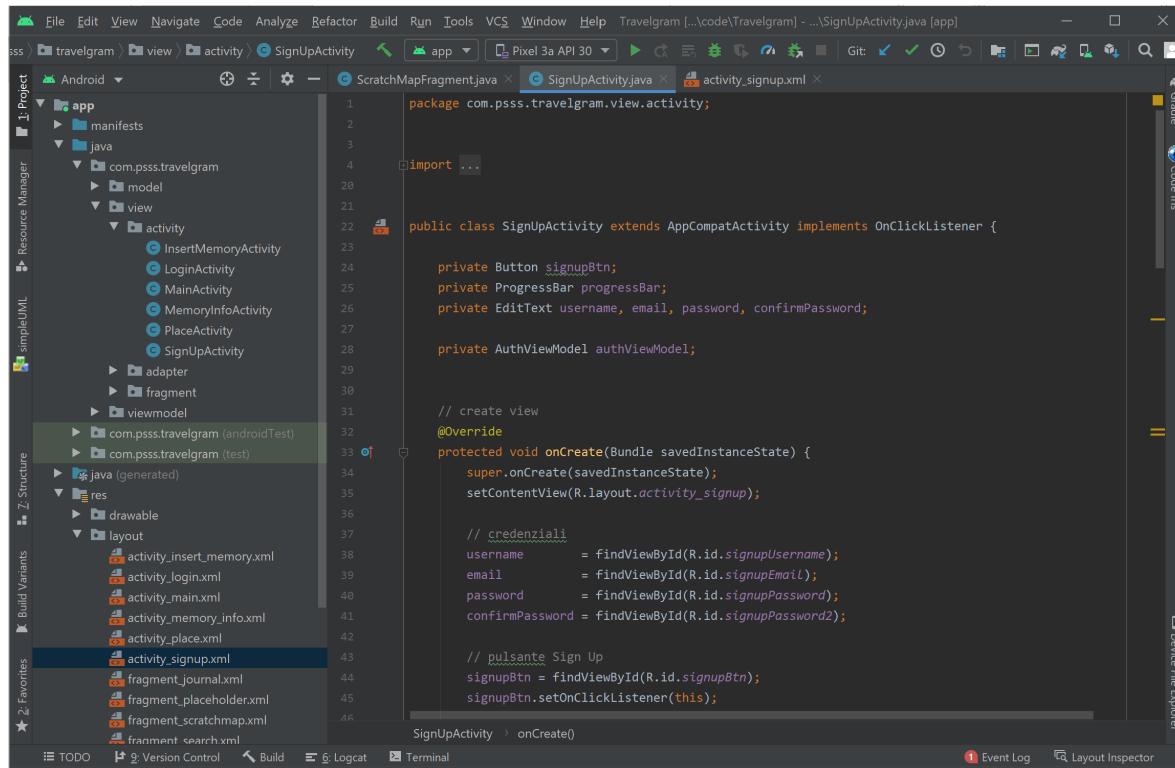


Figura 1.11: Android Studio

1.1.3 Servizi esterni

Google Firebase: Molte delle funzionalità necessarie per il funzionamento dell'applicazione hanno richiesto il supporto di *Firebase*^[4] (figura 1.12), una piattaforma appartenente alla famiglia Google di cui sono stati adoperati i seguenti servizi:

1. *Firebase Authentication*: per gestire l'autenticazione e la registrazione degli utenti dell'applicazione.
2. *Firebase Cloud Firestore*: database che garantisce persistenza e sincronizzazione dei dati.
3. *Firebase Storage*: per l'immagazzinamento delle immagini.

CAPITOLO 1. PROCESSO DI SVILUPPO

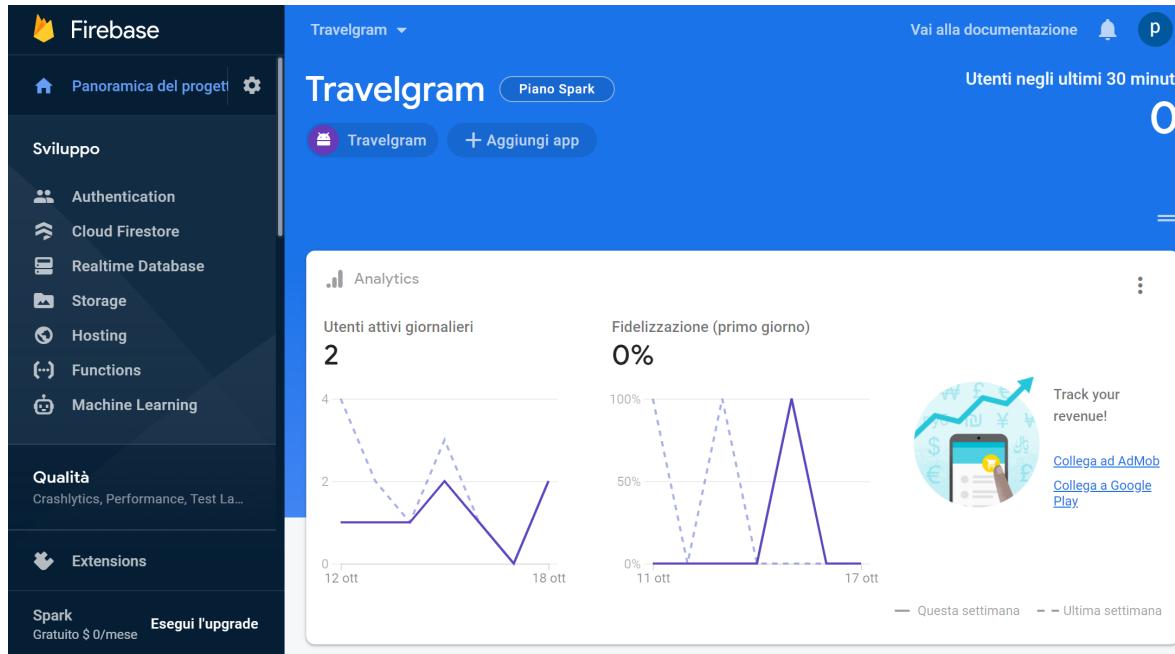


Figura 1.12: Console di Firebase

Google Maps: Infine sono state utilizzate le API di *Google Maps*^[5] al fine di poter gestire una mappa interattiva all'interno dell'applicazione. La gestione delle Google Maps API è gestita tramite *Google Cloud Platform* (figura 1.13).

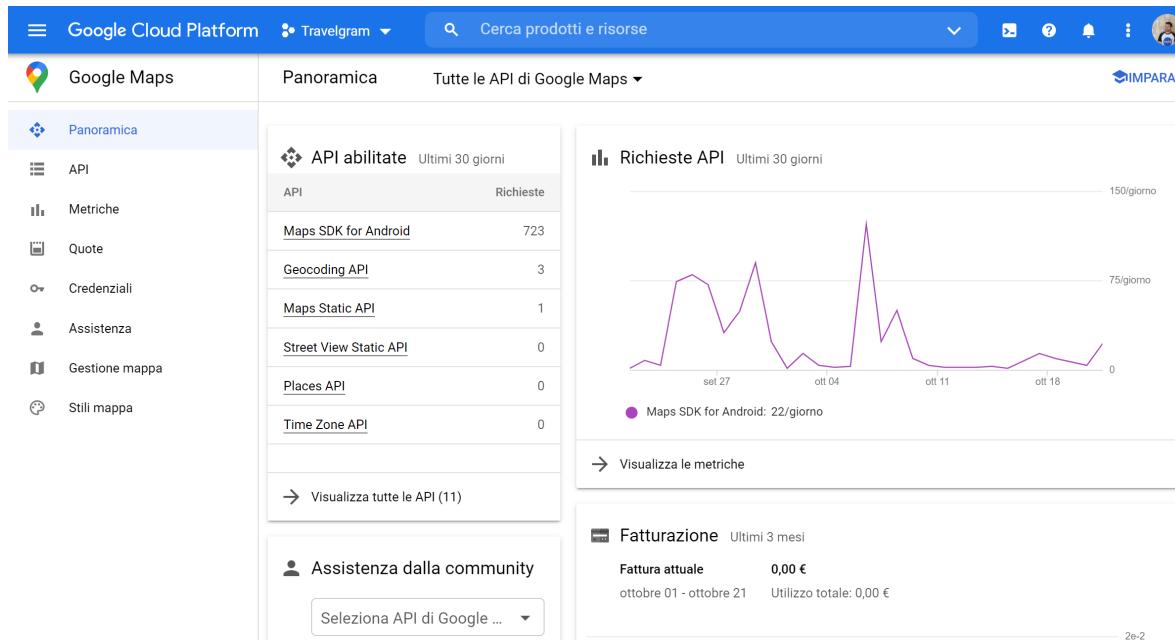


Figura 1.13: Google Cloud Platform

1.2 Tecniche e pratiche agili

La scelta del modello di sviluppo da seguire è ricaduta sullo *Unified Process* (UP) Agile. I motivi dietro questa scelta sono numerosi:

- Usa i principi di modellazione Agile per UML;
- È aperto all'uso di pratiche Agili;
- Prevede una serie di iterazioni time-boxed, con durata che varia dalle 2 alle 6 settimane, che garantiscono di avere sempre qualcosa di funzionante anche nel caso in cui non si riesca a portare a termine l'intero progetto entro una data di consegna predefinita;
- Dà molta attenzione al testing;
- Prevede l'uso di VCS per mantenere sempre una versione funzionante del software sulla Baseline;
- Il codice è pensato per il cambiamento ed è quindi flessibile e modificabile.

Questo modello di sviluppo software prevede quattro fasi durante il processo di sviluppo, sebbene non sia estremamente restrittivo né nella loro realizzazione, né nel numero di iterazioni da compiere in ogni fase:

1. *Fase di Ideazione*: è la fase iniziale di fattibilità e visione
2. *Fase di Elaborazione*: sviluppo iterativo del nucleo del sistema, ovvero gli elementi principali e con rischi maggiori
3. *Fase di Costruzione*: sviluppo iterativo degli elementi rimanenti, a rischio più basso
4. *Fase di Transizione*: beta test e rilascio

Di queste quattro fasi, per motivi legati al tempo, ci siamo limitati a completare la fase di Ideazione in una prima iterazione di due settimane ed a portare a termine due iterazioni complete di Elaborazione in altre due settimane ciascuna, per un periodo totale di sviluppo di sei settimane. La versione dell'applicazione non è ancora pronta al rilascio, ma è vicina all'essere la prima milestone da cui partire per programmare delle minor-release in fase di Costruzione. Si mostra in tabella 1.1 il dettaglio di cosa è stato realizzato nelle singole fasi:

Ideazione	È stato realizzato il documento di visione che mette in luce gli obiettivi e i requisiti generali del progetto. Sono stati individuati i casi d'uso dell'applicazione che sono stati descritti in formato breve. Si è passato al dettagliamento di circa il 10-20% dei casi d'uso precedentemente individuati. Sono state definite le specifiche supplementari.
Elaborazione 1	È stato realizzato un prototipo dell'applicazione da implementare. Si è effettuata l'analisi testuale dei documenti realizzati finora. Si sono realizzati i principali diagrammi di Analisi. Si è iniziato ad implementare i casi d'uso principali. Si sono creati i diagrammi dinamici e modificati di pari passo con l'implementazione. Si sono dettagliati altri casi d'uso per un totale di circa il 40%.
Elaborazione 2	Analogamente alla prima iterazione sono stati implementati tutti i casi d'uso principali che permettono all'applicazione di poter offrire le sue funzionalità più importanti. Sono stati raffinati e completati tutti i diagrammi.

Tabella 1.1: Fasi di UP realizzate

1.2.1 Altre Tecniche Agili

Altre pratiche Agili adoperate che hanno aiutato il team nel processo di sviluppo sono:

- Utilizzo di una Task Board (SCRUM), offerta da Trello per tenere traccia del lavoro svolto.
- Daily Meeting (SCRUM), tramite Microsoft Teams, per far sì che il team si aggiornasse continuamente riguardo al progresso del progetto.
- Continous Integration (XP), integrazione giornaliera del codice effettuata tramite il repository condiviso Git.
- Refactoring (XP), pulizia del codice per migliorarne alcune caratteristiche non funzionali senza modificarne il comportamento esterno.

1.3 Stima dei costi

1.3.1 Function Points

Seguendo la tecnica dei *Function Point*, sono state considerate le 5 caratteristiche proposte e ad ognuna di esse è stato assegnato un valore di complessità per ottenere l'*Unadjusted Function Point Count* (UFC) e il fattore correttivo TFC, ottenendo il *Delivered Function Point* (DFP):

$$UFC = 68$$

$$TFC = 0.65 + (0.01 * 41) = 1.06$$

$$DFP = UFC * TFC = 72.$$

Ora per calcolare quindi l'effort totale, si è visto che in media per Java ogni FP corrisponde a 53 LOC, quindi in totale il numero di LOC è 3816. Supponendo quindi che nel team ci sia una produttività di 10 FP al mese, ossia 530 LOC per persona, con una durata della giornata lavorativa di 8 ore. Considerando una media su tutti i membri del team, si giunge al risultato presentato in tabella 1.2.

	Effort	Giorni lavorativi
Singolo membro	408h/p	51
Totle (4 persone)	1632h/p	204

Tabella 1.2: Costi

1.3.2 Use Case Points

Seguendo le linee guida del calcolo con *Use Case Points* invece, abbiamo analizzato:

- La complessità dei casi d'uso, assegnando un peso di 5 ai casi d'uso semplici (2), 10 ai mediamente complessi (1) e 15 ai complessi (3) per ottenere un *Unadjusted Use Case Weight* (UUCW) pari a 65.
- La complessità degli attori, individuandone uno semplice con peso 1 ed uno complesso con peso 4, ottenendo un *Unadjusted Actor Weight* (UAW) pari a 5.
- Abbiamo ottenuto quindi l'*Unadjusted Use Case Point* (UUCP) sommando i due valori precedenti.
- La complessità dei fattori tecnici è stata stimata in un *Tfactor* pari a 42 che ci ha permesso di ottenere un *Technical Complexity Factor* (TCF) pari a 1.12.
- La complessità ambientale è stata stimata in un *Efactor* pari a 15 che ci ha permesso di ottenere un *Environment Factor* (EF) pari a 0.95.

CAPITOLO 1. PROCESSO DI SVILUPPO

Fatto ciò quindi abbiamo ottenuto un valore per gli Use Case Points pari a:

$$UCP = UUCP * TCF * EF = 70 * 1.12 * 0.95 = 75$$

Stimando un numero di ore per use case point che va dalle 20 alle 30 si ottiene quindi che saranno necessarie per svolgere il lavoro dalle 1500 alle 2250 ore. Supponendo quindi che il team lavori 224 ore a settimana, e quindi 448 ore ad iterazione, abbiamo un numero di iterazioni minime di 3 e massimo di 5.

Capitolo 2

Avvio del Progetto

In questo capitolo andremo a descrivere la Visione del progetto. Le finalità del documento di Visione sono quelle di trovare e definire i bisogni e le caratteristiche generali dell'applicazione Travelgram e delle parti interessate. Inoltre, poiché nelle pagine seguenti saranno presenti ripetutamente termini specifici riguardanti il progetto, se ne è fatta una descrizione accurata tramite un Glossario.

2.1 Glossario

In tabella 2.1 è mostrato il glossario dei termini.

Termine	Alias	Descrizione	Formato
Travelgram		Nome dell'applicazione	
Memory	Ricordo	Post di un utente atto a conservare una propria esperienza	Si compone di Stringhe riguardanti UID utente, città, paese, descrizione, data e link.
Review	Recensione	Recensione testuale di un luogo, con annessa valutazione (stelle)	
Traveler	Utente, Viaggiatore	Utente registrato che usa l'applicazione	Si compone di Stringhe riguardanti username e UID, ed Array legati a paesi visitati e preferiti, followers e following
Scratch Map	Mappa	Elemento grafico rappresentante la mappa del mondo	File json
Travel Journal	Diario di Viaggio	Collezione di ricordi legati ai luoghi visitati	Array di Memory
GDPR	General Data Protection Regulation	Regolamento dell'Unione europea in materia di trattamento dei dati personali e di privacy	
Follower		Utente dell'applicazione che segue il Traveler	
Following		Utente dell'applicazione seguito dal Traveler	

Tabella 2.1: Glossario

2.2 Visione

2.2.1 Obiettivi Generali

L'applicazione **Travelgram** è destinata agli amanti dei viaggi, per memorizzare ricordi (*memories*) e condividerli con gli amici.

Gli obiettivi principali sono:

- tenere traccia dei luoghi visitati
- condividere ricordi con gli amici
- aiutare i viaggiatori a scegliere una meta

Il primo obiettivo riguarda la sfera personale: l'applicazione è pensata in primis per coloro che desiderano mantenere un Diario di Viaggio (*Travel Journal*) al fine di conservare ricordi, pensieri ed esperienze relativi a luoghi visitati.

Il secondo obiettivo che ci si è posti è quello di dare all'applicazione un'impronta social, orientandola alla condivisione delle esperienze e alla creazione di una community di utenti con interessi in comune.

Il terzo obiettivo è quello di aiutare gli utenti (*travelers*) a scoprire nuovi luoghi interessanti e ad orientare utenti indecisi verso l'alternativa che potrebbe soddisfarli di più grazie ai feedback di migliaia di viaggiatori.

2.2.2 Requisiti Generali

L'applicazione consente all'utente di conservare in un Diario di Viaggio i propri ricordi relativi ad un luogo visitato sotto forma di immagini, con eventuali descrizioni, e di recensione dell'esperienza complessiva.

Ogni utente registrato (*traveler*) ha la possibilità di seguire altri utenti e visualizzare le loro memories e recensioni, così da restare al corrente delle loro esperienze e magari scoprire nuovi luoghi da visitare. Quando l'utente pubblica un contenuto, tutti gli utenti che lo seguono vengono notificati.

L'applicazione permette inoltre all'utente di selezionare uno specifico luogo (stato o città) attraverso una mappa interattiva (*Scratch Map*), al fine di:

- indicare di averlo già visitato o di volerlo visitare
- aggiungerlo tra i luoghi preferiti
- visualizzare i ricordi e recensioni relativi a tale luogo e pubblicati dai traveler seguiti o dall'utente stesso
- pubblicare ricordi o recensioni relativi a tale luogo
- consultare delle guide di viaggio che consigliano i punti di interesse da visitare nello stato/città selezionato

Infine, ogni utente ha a disposizione una pagina del profilo che è possibile personalizzare indicando i propri interessi relativi ai viaggi (natura, cultura, cucina, ...) per migliorare l'esperienza ed inoltre essere visualizzata dagli altri utenti registrati all'applicazione.

D'altro canto, l'applicazione deve anche garantire:

- sicurezza

- facilità d'uso
- compatibilità con più dispositivi
- affidabilità
- prestazioni
- Interoperabilità
- license

Una descrizione dettagliata dei requisiti funzionali e non funzionali sarà esposta nel capitolo 3 (Specifiche dei Requisiti), attraverso la definizione dei casi d'uso a diversi livelli di dettaglio e delle specifiche supplementari, in maniera conforme allo sviluppo UP.

2.2.3 Vincoli

Per poter usare l'applicazione, l'utente deve registrarsi. L'applicazione è supportata sulle piattaforme Android dalla versione 5.0 in poi. L'applicazione deve essere sviluppata rispettando le regole imposte dal GDPR (General Data Protection Regulation) per il rispetto della privacy dell'utente e i termini contrattuali relativi al Google Play Store.

2.2.4 Positioning

Il problema di	capire quali mete soddisfano le proprie esigenze di viaggio
Affligge	gli utenti desiderosi di viaggiare
Il suo impatto è	lo spreco di tempo e/o la scelta di una meta non soddisfacente
Una soluzione sarebbe	avere a portata di mano i feedback di migliaia di viaggiatori

Tabella 2.2: Positioning

In tabella 2.2 è presentato il positioning dell'applicazione Travelgram.

L'**unicità** del nostro prodotto è dovuta al fatto che:

- Differisce dai social media convenzionali per il fatto che questi sono orientati alla condivisione di immagini o video di diverso genere per divertimento, piuttosto che alla ricerca di mete di viaggio
- Differisce dalle altre piattaforme di viaggi per il fatto che queste permettono di recensire specifiche attrazioni, musei, ristoranti, hotel e così via, mentre la nostra applicazione permette di recensire qualsiasi tipo di luogo, tra cui città e paesi

2.2.5 Parti interessate

In tabella 2.3 sono indicate le principali parti interessate e le loro descrizioni.

Parti interessate	Descrizione
Project manager	Monitora l'andamento del progetto
Team di sviluppo	Si occupa di: definizione dei requisiti progettazione implementazione testing manutenzione
Traveler	è l'utente che usa l'applicazione

Tabella 2.3: Parti interessate

Capitolo 3

Specifiche dei Requisiti

Questo capitolo presenta i passaggi fondamentali della fase di specifica dei requisiti, durante la quale tutti i requisiti raccolti e descritti nel documento di visione vengono analizzati e specificati. Secondo UP, il processo di sviluppo agile adottato, la specifica dei requisiti è un'attività che attraversa tutto il ciclo di sviluppo del sistema, in quanto ogni requisito viene modificato e raffinato man mano che il prodotto software prende forma; infatti, nelle prime iterazioni del processo, la specifica dei requisiti si riduce ad individuare e dettagliare i requisiti più importanti per l'utente e solo nelle iterazioni più avanzate vengono analizzati e specificati gli altri requisiti di contorno. Ciò evita di congelare tutti i requisiti del sistema prima dell'avvio del progetto, e di conseguenza agevola il team di sviluppo nell'adattare il sistema ai continui cambiamenti dei requisiti voluti dall'utente. Pertanto, i diagrammi prodotti in questa fase sono stati oggetto di continue modifiche e raffinamenti dovute all'evoluzione che i requisiti hanno avuto durante tutto il ciclo di lavoro.

3.1 Requisiti Funzionali

Tra i modelli indicati a descrivere i requisiti funzionali si è scelto di utilizzare il Modello dei Casi d'Uso, come proposto da UP. Questo si presenta in due formati (breve e dettagliato) che variano per la loro dimensione ed accuratezza. Ogni formato è adatto ad una fase diversa del processo di sviluppo con metodologia UP Agile.

3.1.1 Casi d'Uso in Formato Breve

Durante la fase di Ideazione si è fornita una semplice e breve descrizione testuale che sintetizza l'intero caso d'uso. Abbiamo in particolare individuato 2 Attori e 14 Casi d'Uso rappresentati nella seguente tabella. I casi d'uso effettivamente implementati sono indicati in grassetto.

CAPITOLO 3. SPECIFICA DEI REQUISITI

Attore	Nome	Descrizione in formato breve
Traveler	Sign Up	L'utente non ancora registrato inserisce le credenziali di registrazione. Il sistema salva i dati inseriti.
	Login	L'utente registrato si autentica inserendo le proprie credenziali. Il sistema verifica la validità.
	Pubblica Memory	Il traveler aggiunge un nuovo ricordo relativo ad un luogo.
	Segna Luogo	Il traveler contrassegna un luogo di interesse come visitato o da visitare interagendo con la mappa.
	Segui Traveler	Il traveler aggiunge un determinato profilo nella propria lista di utenti seguiti.
	Visualizza Memories	Il traveler visualizza le Memories (proprie o di chi segue).
	Pubblica Review	Il traveler scrive una Review relativa ad un luogo visitato, optionalmente esprime un giudizio.
	Invia Notifica	Il sistema notifica un utente quando viene seguito, o i follower di un Traveler in seguito alla pubblicazione di una memory o di una review.
	Visualizza Reviews	Il traveler visualizza le Reviews.

Attore	Nome	Descrizione in formato breve
	Personalizza Profilo	L'utente può modificare il proprio profilo aggiornando i suoi interessi.
	Modifica o Elimina Memory	L'utente può cancellare o modificare una memory pubblicata.
	Modifica o Elimina Review	L'utente può cancellare o modificare una review pubblicata.
	Consulta Guida Turistica	L'utente può controllare guide turistiche relative ad un determinato luogo.
	Visualizza profilo di un Traveler	L'utente può visualizzare il profilo di altri travelers.

3.1.2 Casi d'Uso in Formato Dettagliato

Durante la prima fase di Ideazione, e nelle successive fasi di Elaborazione, si è passato al dettagliamento dei Casi d'Uso principali, che nella tabella precedente sono stati marcati in grassetto.

Pubblica Memory:

NOME DEL CASO D'USO: Pubblica Memory
DESCRIZIONE IN FORMATO BREVE: Il Traveler aggiunge un nuovo ricordo relativo ad un luogo.
PORTATA: Applicazione Travelgram
LIVELLO: Obiettivo Utente
ATTORE PRIMARIO: Traveler
PARTI INTERESSATE: Traveler (vuole pubblicare una nuova Memory) Follower (vuole essere notificato della pubblicazione)
PRE-CONDIZIONI: Il Traveler deve aver effettuato il Login.
POST-CONDIZIONI: La Memory è salvata. Messaggio di conferma mostrato. Luogo contrassegnato come visitato. La mappa viene aggiornata.
SCENARIO PRINCIPALE: <ol style="list-style-type: none">1. Il Traveler accede alla schermata di inserimento Memory2. Il Traveler inserisce le informazioni della Memory, aggiungendo una o più immagini dalla galleria, lo stato, la città, descrizione e data3. Il sistema salva la Memory4. Il sistema notifica la pubblicazione ai follower usando <u>Invia Notifica</u>
SCENARI ALTERNATIVI: 3.a Luogo non ancora visitato <ol style="list-style-type: none">1. Il sistema usa <u>Segna Luogo</u>

Segna Luogo:

NOME DEL CASO D'USO:
Segna Luogo
DESCRIZIONE IN FORMATO BREVE:
Il traveler contrassegna un luogo di interesse come visitato o da visitare interagendo con la mappa.
PORTATA:
Applicazione Travelgram
LIVELLO:
Obiettivo Utente
ATTORE PRIMARIO:
Traveler
PARTI INTERESSATE:
Traveler (vuole segnare un luogo come visitato o da visitare)
PRE-CONDIZIONI:
Il Traveler deve aver effettuato il Login.
POST-CONDIZIONI:
Luogo contrassegnato come visitato o da visitare. La mappa viene aggiornata.
SCENARIO PRINCIPALE:
<ol style="list-style-type: none">1. Il Traveler accede alla schermata di un luogo.2. Il Traveler indica il luogo come visitato o da visitare.3. Il sistema aggiorna lo stato del luogo.
SCENARI ALTERNATIVI:

Segui Traveler:

NOME DEL CASO D'USO:
Segui Traveler
DESCRIZIONE IN FORMATO BREVE:
Il Traveler aggiunge un determinato profilo nella propria lista di utenti seguiti.
PORTATA:
Applicazione Travelgram
LIVELLO:
Obiettivo Utente
ATTORE PRIMARIO:
Traveler
PARTI INTERESSATE:
Traveler (vuole seguire altri utenti per essere aggiornato delle loro pubblicazioni)
PRE-CONDIZIONI:
Il Traveler deve aver effettuato il Login.
POST-CONDIZIONI:
L'utente seguito viene aggiunto alla lista dei seguiti del traveler Il Traveler viene aggiunto alla lista dei follower dell'utente seguito
SCENARIO PRINCIPALE:
<ol style="list-style-type: none">1. Il Traveler accede alla schermata di ricerca2. Il Traveler ricerca l'altro utente che vuole seguire3. Il Traveler aggiunge l'utente cercato ai suoi seguiti4. Il sistema aggiorna i profili Il sistema notifica l'utente seguito del nuovo follower usando <u>Invia Notifica</u>
SCENARI ALTERNATIVI:

Visualizza Memories:

NOME DEL CASO D'USO:
Visualizza Memories
DESCRIZIONE IN FORMATO BREVE:
Il traveler visualizza tutte le Memories (proprie o di chi segue).
PORTATA:
Applicazione Travelgram
LIVELLO:
Obiettivo Utente
ATTORE PRIMARIO:
Traveler
PARTI INTERESSATE:
Traveler (vuole visualizzare memories pubblicate da lui o da un traveler)
PRE-CONDIZIONI:
Il Traveler deve aver effettuato il Login. Il traveler per visualizzare memory di altri utenti deve aver prima effettuato il follow.
POST-CONDIZIONI:
Viene presentata una nuova schermata che contiene tutte le memories del traveler corrente, o dei suoi following.
SCENARIO PRINCIPALE:
<ol style="list-style-type: none">1. Il traveler accede alla schermata di visualizzazione delle memories.2. Il sistema mostra le memories pubblicate.
SCENARI ALTERNATIVI:
2.a Visualizza memories dell'utente corrente: <ol style="list-style-type: none">1. Il sistema mostra tutte le memories pubblicate dall'utente
2.b Visualizza memories di un Luogo: <ol style="list-style-type: none">1. Il sistema mostra tutte le memories pubblicate dall'utente relative ad un Luogo specifico.
2.c Visualizza memories dei following: <ol style="list-style-type: none">1. Il sistema mostra le memories pubblicate dai following dell'utente relative ad un Luogo specifico.

3.2 Specifiche Supplementari

Altri requisiti funzionali da implementare, ma non dettagliati, sono il *Login* di un Traveler registrato e la *Sign Up* di un nuovo utente.

Oltre a quanto visto finora, come accennato nel Capitolo 2 (Avvio del Progetto), l'applicazione deve garantire anche alcuni requisiti non funzionali.

Primo fra tutti è la **facilità d'uso** dell'applicazione mobile nei confronti degli utenti, per questo l'applicazione deve essere progettata garantendo la user friendliness proponendo un'interfaccia intuitiva ed accattivante.

Altro punto fondamentale è la **sicurezza**, infatti bisogna che l'applicazione tenga sicuri i dati e le informazioni personali degli utenti senza che esse vengano accedute da persone non autorizzate.

Coerentemente l'app deve anche essere **affidabile** e permettere quindi che a seguito di malfunzionamenti i dati non vengano persi, ma possano essere ripristinati.

Considerando sempre i bisogni dell'utente, altro punto fondamentale sono le **prestazioni**, infatti l'app deve essere reattiva agli input degli utenti per far sì che questi non si stufino nell'utilizzo; inoltre queste prestazioni devono essere garantite tenendo conto della **scalabilità**, quindi all'aumentare sia dei dati che dei Travelers iscritti, il tutto minimizzando il più possibile i consumi delle risorse.

Da un punto di vista maggiormente tecnico invece, l'applicativo deve essere **compatibile** con la maggior parte dei dispositivi Android per ottenere una diffusione maggiore tra gli utenti, e deve interagire con gli altri sistemi al fine di ottenere un corretto funzionamento. Per questo si è scelto di utilizzare come target SDK il livello API 30, in modo da supportare al meglio i nuovi dispositivi (Android 11), ma si è impostato come min SDK la versione 21, in modo da rendere l'applicazione compatibile dalla versione Android 5.0 (Lollipop), raggiungendo quindi circa il 95% dei dispositivi in circolazione.

3.3 Mockup e Design Concepts

In fase di Inception ci è stato utile visualizzare i requisiti funzionali dell'applicazione, descritti nei casi d'uso, tramite un supporto grafico. Per questo motivo abbiamo Realizzato, tramite MockFlow, dei Mockup (figura 3.1) dell'Applicazione con le sue funzionalità previste, in modo da avere un'idea del risultato finale di alcune scelte fatte.

CAPITOLO 3. SPECIFICA DEI REQUISITI



Figura 3.1: Mockup realizzato con MockFlow

Capitolo 4

Analisi del Sistema

In questo capitolo viene discussa la fase di **Analisi** del sistema, che ha come scopo il chiarimento e la documentazione delle funzioni e dei servizi che vengono offerti. Questa fase è corredata da diversi diagrammi realizzati in notazione UML e mediante l'uso dei concetti tipici dello sviluppo orientato agli oggetti. Si tratta di un'analisi di alto livello che funge da punto cruciale per gettare le basi su cui condurre la fase di progettazione. I diagrammi sono stati modificati e raffinati durante il corso delle iterazioni del processo di sviluppo adottato fino a raggiungere un certo grado di conformità con i requisiti e con il progetto di dettaglio.

4.1 Use Case Diagram

Il diagramma dei casi d'uso (figura 4.1) è stato utile per una sintesi grafica dei casi d'uso definiti in fase di specifica dei requisiti e da essi derivato. Esso mette in luce le funzionalità offerte dal sistema, così come percepite e utilizzate dagli attori, ponendo l'accento sugli obiettivi e le esigenze degli utenti. Infatti, seguendo il modello di Jacobson, ogni caso d'uso del diagramma racchiude una serie di scenari che l'attore attraversa per raggiungere un preciso e definito obiettivo.

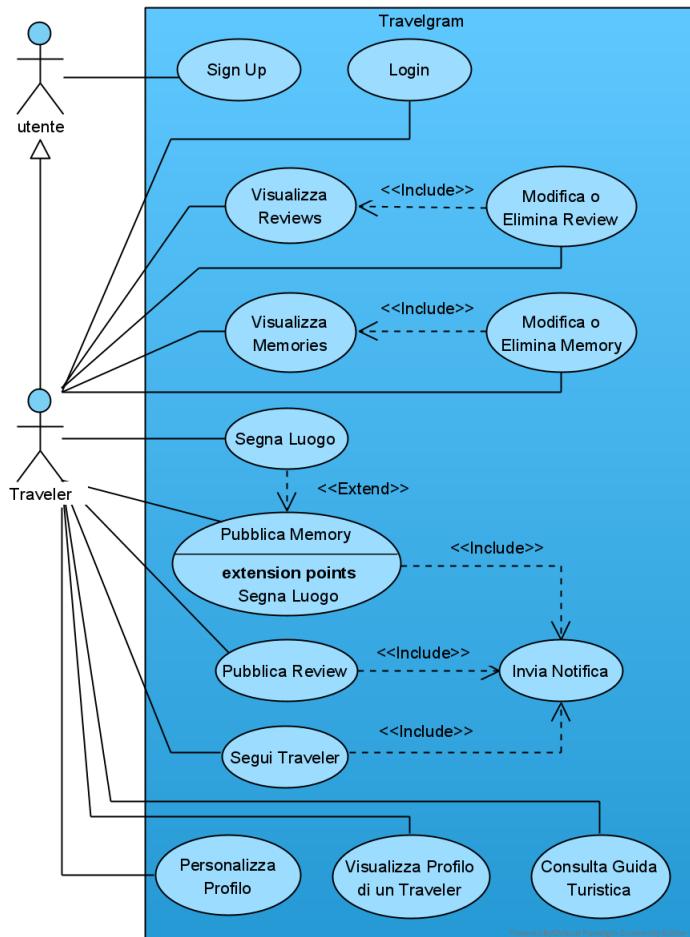


Figura 4.1: Use Case Diagram

Per mostrare in maniera grafica l'insieme completo di servizi che il sistema dovrebbe offrire, il diagramma raffigura sia i casi d'uso implementati, sia quelli non implementati (che non sono stati dettagliati in fase di specifica).

Da notare che:

- I casi d'uso *Modifica o Elimina Review* e *Modifica o Elimina Memory* includono rispettivamente i casi d'uso *Visualizza Reviews* e *Visualizza Memories* poiché si immagina che l'attore debba necessariamente visualizzare l'elenco di Memories/Reviews prima di modificarne o eliminarne qualcuna
- Il caso d'uso *Segna Luogo* estende il caso d'uso *Pubblica Memory* dato che il luogo viene impostato come visitato se vi si pubblica una memory
- I casi d'uso *Segui Traveler*, *Pubblica Memory* e *Pubblica Review* includono *Invia Notifica* dato che prevedono l'invio di una notifica
- L'attore *Traveler* è un tipo di attore *Utente* che ha usato il caso d'uso *Registrati* nel suo scenario principale, portando a termine l'obiettivo di registrazione al sistema

4.2 System Sequence Diagrams

Per i casi d'uso implementati, che quindi sono stati dettagliati in fase di specifica, è stata fornita un'altra rappresentazione grafica che mette in luce le interazioni tra attore e sistema relative agli scenari principali di ciascun caso d'uso. Il diagramma utilizzato è il System Sequence Diagram che modella ogni scenario come una sequenza di interazioni. I casi d'uso *Pubblica Memory*, *Segna Luogo* e *Segui Traveler* si prestano bene ad essere rappresentati con questo tipo di diagramma, in quanto gli scenari sono costituiti da semplici azioni in sequenza.

4.2.1 Pubblica Memory

In Figura 4.2 è riportato l'SSD del caso d'uso *Pubblica Memory*. Il sistema salva i dati della memory inserita dal traveler, setta il luogo a visitato se non lo era già, e notifica ogni follower dell'utente. Per una maggiore chiarezza della rappresentazione dell'operazione di invio della notifica, è stato aggiunto l'attore *follower* per indicare un'altra istanza dell'attore *traveler*.

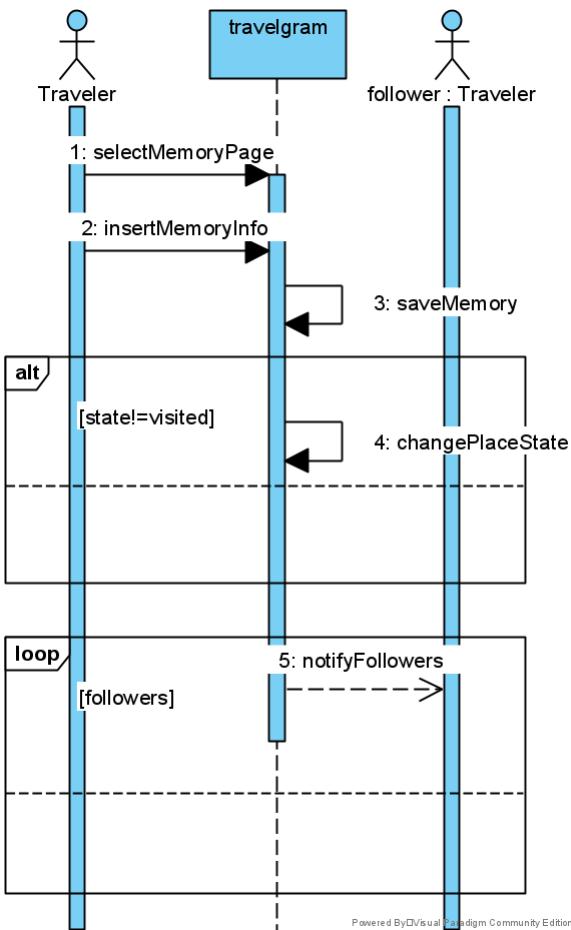


Figura 4.2: SSD di Pubblica Memory

4.2.2 Segna Luogo

In Figura 4.3 è riportato l'SSD del caso d'uso Segna Luogo. Le interazioni tra traveler e sistema sono semplici e portano quest'ultimo ad aggiornare lo stato del luogo.

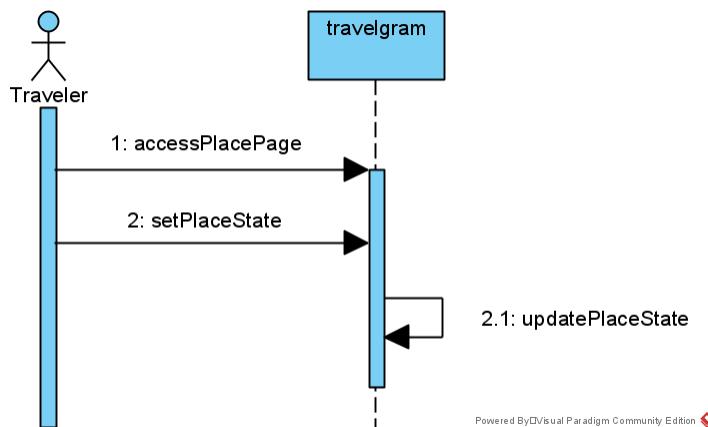


Figura 4.3: SSD di Segna Luogo

4.2.3 Segui Traveler

In Figura 4.4 è riportato l'SSD del caso d'uso Segui Traveler. Il traveler interagisce con il sistema per aggiungere un altro traveler ai suoi seguiti. Il sistema aggiorna le liste dei following e dei follower di entrambi gli utenti e notifica il traveler seguito del nuovo follower. Per una maggiore chiarezza della rappresentazione dell'operazione di invio della notifica, è stato aggiunto l'attore *Traveler seguito* per indicare un'altra istanza dell'attore traveler.

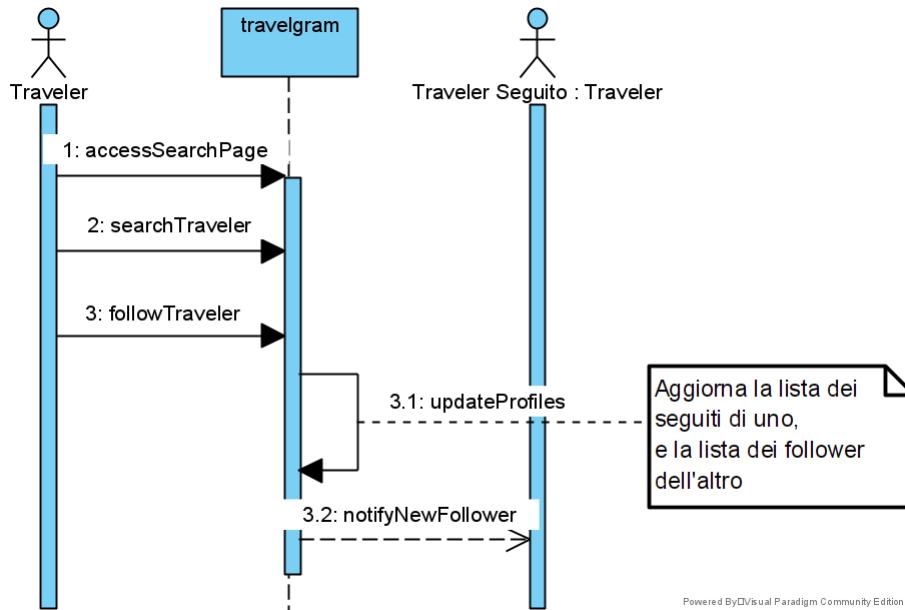


Figura 4.4: SSD di Segui Traveler

4.3 Activity Diagrams

4.3.1 Visualizza Memories

Il caso d'uso Visualizza Memories è stato rappresentato mediante l'impiego di un Activity Diagram. Questo modello, rispetto a un Sequence Diagram, consente di schematizzare più efficacemente i diversi scenari alternativi di questo caso d'uso; infatti come si può notare in Figura 4.5 è stato impiegato il costrutto *branch-merge* per questo motivo.

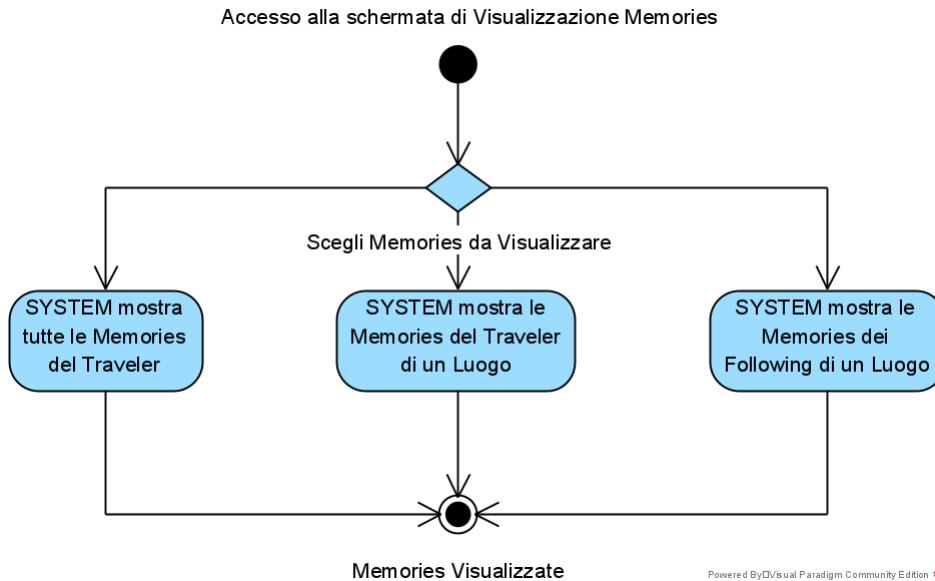


Figura 4.5: Activity Diagram di Visualizza Memory

4.4 Textual Analysis

È stata realizzata un'operazione di Textual Analysis (figura 4.6) a partire dai requisiti descritti nel documento di visione (sezione 2.2) al fine di trovare le principali entità di dominio.

L'applicazione consente all'utente di conservare in un **Diario di Viaggio** i propri **ricordi** relativi ad un **luogo** visitato sotto forma di immagini, con eventuali descrizioni, e di **recensione** dell'esperienza complessiva.

Ogni **utente** registrato (**traveler**) ha la possibilità di seguire altri utenti e visualizzare le loro **memories** e **recensioni**, così da restare al corrente delle loro esperienze e magari scoprire nuovi **luoghi** da visitare. Quando l'utente pubblica un contenuto, tutti gli utenti che lo seguono (**follower**) vengono **notificati**.

L'applicazione permette inoltre all'utente di selezionare uno specifico **luogo** (stato o città) attraverso una **mappa interattiva** (**Scratch Map**), al fine di:

- indicare di averlo già visitato o di volerlo visitare
- aggiungerlo tra i **luoghi** preferiti
- visualizzare i **ricordi** e **recensioni** relativi a tale **luogo** e pubblicati dai **traveler** seguiti o dall'utente stesso
- pubblicare **ricordi** o **recensioni** relativi a tale **luogo**
- consultare delle **guide di viaggio** che consigliano i punti di interesse da visitare nello **stato/città** selezionato

Infine, è possibile personalizzare il **profilo** indicando i propri interessi relativi ai viaggi (natura, cultura, cucina, ...) per migliorare l'esperienza.

Figura 4.6: Analisi testuale dei requisiti

Le entità scoperte sono mostrate in figura 4.7 e sono state usate per generare il System Domain Model.

No.	Candidate Class	Extracted Text	Type	Description	Occur...	Highlight
1	notificati	notificati	→ Message		1	
2	mappa	mappa	Class	Synonym: ScratchMap	1	
3	ScratchMap	Scratch Map	Class		1	
4	Memory	memories	Generated Model Element		1	
5	recensioni	recensioni	Class	Synonym: Review	3	
6	ricordi	ricordi	Class	Synonym: Memory	3	
7	Review	recensione	Generated Model Element		1	
8	TravelJournal	Diario di Viaggio	Generated Model Element		1	
9	luoghi	luoghi	Class	Synonym: Luogo	2	
10	Luogo	luogo	Generated Model Element		4	
11	stato	stato	Class		2	
12	città	città	Generated Model Element		2	
13	guide di viaggio	guide di viaggio	Generated Model Element		1	
14	follower	follower	Actor		1	
15	utente	utente	Generated Model Element	Synonym: Traveler	1	
16	Traveler	traveler	Generated Model Element		2	
17	Traveler	profilo	Generated Model Element		1	

Figura 4.7: Output dell'analisi testuale

4.5 System Domain Model

Al fine di porre delle basi significative alla fase di progettazione, è utile individuare le principali entità del dominio del sistema, stabilendo quali sono i loro attributi significativi e come sono associate tra di loro, senza badare, almeno in questa fase di analisi, alle eventuali responsabilità e operazioni.

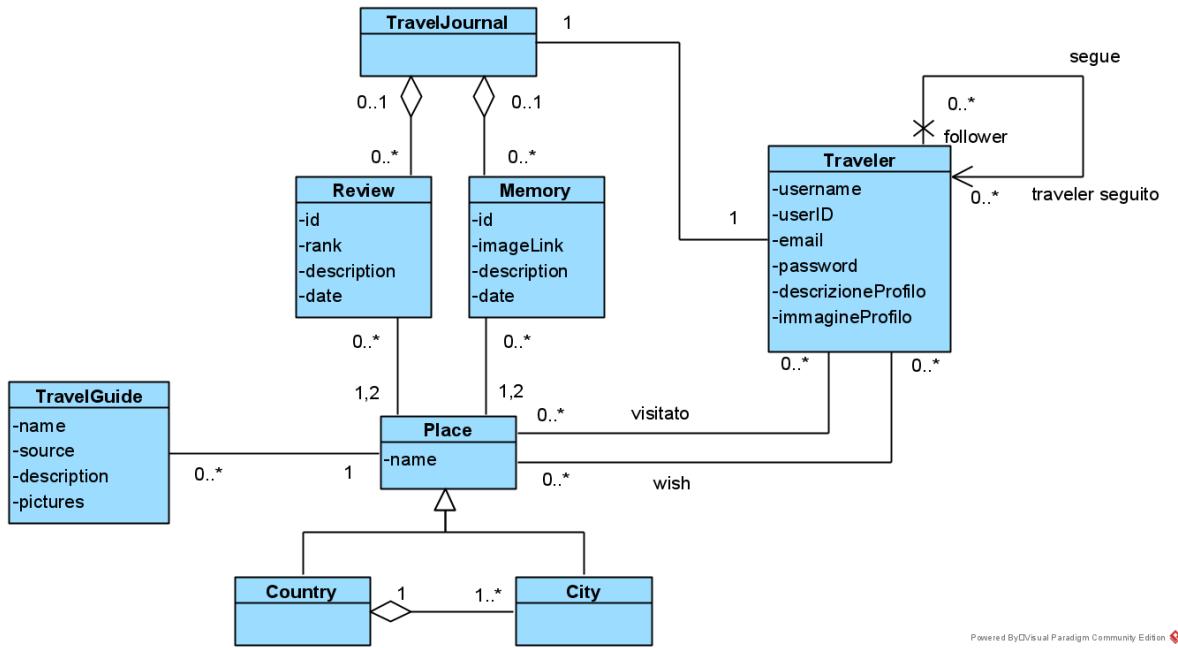


Figura 4.8: System Domain Model

Di seguito si riporta una descrizione delle singole classi concettuali individuate nel dominio del sistema Travelgram, riportate nel System Domain Model di Figura 4.8:

- **Traveler**: Rappresenta un utente registrato, pertanto è l'attore fondamentale del sistema; è identificato da un ID, possiede uno username, una email e una password forniti al momento della registrazione, e in più, una descrizione e un'immagine del profilo per arricchire la propria esperienza di utilizzo dell'applicazione. Può pubblicare memory e review, seguire altri Traveler e navigare tra i luoghi della ScratchMap. Inoltre, può stabilire di voler visitare un luogo o di averlo visitato
- **Memory**: Rappresenta un ricordo di viaggio che può pubblicare un Traveler. Possiede un ID, una descrizione, un'immagine e eventualmente la data del relativo viaggio. In più, contiene il paese e, optionalmente, la città a cui si riferisce.
- **Review**: Rappresenta una recensione di un viaggio che può pubblicare un Traveler. Possiede un ID, una descrizione, un voto e eventualmente la data del relativo viaggio. In più, contiene il paese e, optionalmente, la città a cui si riferisce.
- **TravelJournal**: Rappresenta l'insieme di tutte le memory e review pubblicate da uno specifico Traveler, il così detto diario di viaggio
- **Place**: Rappresenta un luogo che un Traveler può visitare e riguardo al quale può pubblicare una memory o una review. Ha il solo attributo nome, ed è la classe padre di una gerarchia

- **City:** Rappresenta una città che il Traveler può visitare e che può opzionalmente inserire tra i riferimenti della propria memory o review. Essendo un tipo di luogo, è figlia della classe Place
- **Country:** Rappresenta un paese che il Traveler può visitare e che inserisce tra i riferimenti delle proprie memory o review. Essendo un tipo di luogo, è una classe figlia di Place; inoltre, contiene un insieme di città
- **TravelGuide:** Rappresenta una guida turistica che fornisce informazioni riguardo uno specifico luogo. Possiede un nome, una descrizione, delle immagini, e il link sorgente da cui consultarla; inoltre, contiene il nome del luogo a cui si riferisce

In seguito invece, un elenco delle associazioni individuate tra le classi di cui sopra:

- **Traveler - Traveler:** un traveler può seguire zero o più traveler, ed avere zero o più follower
- **TravelJournal - Traveler:** Un traveler ha uno e un solo diario di viaggio, il quale è relativo certamente a un solo traveler
- **Traveler - Place:** Un traveler può impostare più luoghi come *visitati* o *da visitare*. Ogni luogo può essere impostato in questi modi da zero o più traveler diversi
- **TravelJournal - Memory/Review:** Il diario di viaggio è un insieme di memory e review, e può non contenerne nessuna, mentre una memory, o una review, può essere contenuta al più nel diario di viaggio del traveler che l'ha pubblicata
- **Memory/Review - Place:** Ogni memory o review è relativa almeno a un luogo, cioè il paese, e al massimo a due luoghi, cioè al paese e alla città, dato che la seconda è opzionale. A un luogo possono riferirsi più memory e review, o nessuna
- **TravelGuide - Place:** Una guida turistica si riferisce ad uno e un solo luogo, mentre un luogo può avere a supporto più guide turistiche
- **City - Country:** Un paese è un insieme di più città (almeno una, perché non esistono paesi senza), mentre una città appartiene a uno e un solo paese

Infine, i motivi della scelta di una gerarchia tra classi per rappresentare il legame tra memory/review e il luogo, che sia esso una città o un paese, è il seguente:

- Si vuole che una memory/review contenga almeno il paese
- Si vuole che una memory/review contenga opzionalmente una città

- Si vuole che la città opzionalmente contenuta nella memory/review sia effettivamente appartenente al paese specificato

Associare la memory/review ad almeno un luogo e massimo due, dove luogo è il padre di una gerarchia i cui figli sono paese e città, e in cui un paese è l'insieme delle città che vi appartengono, assicura il rispetto dei requisiti di cui sopra. Infatti, l'associazione di contenimento tra città e paese consente di tenere traccia delle città effettivamente appartenenti al paese specificato nella memory. Un'alternativa sarebbe stata quella di usare due classi distinte Country e City a cui collegare direttamente la memory/review e mantenere la relazione di contenimento; tuttavia ciò avrebbe comportato un eccessivo accoppiamento in quanto memory/review avrebbe avuto un'associazione sia con la città sia con il paese, e il paese avrebbe avuto l'associazione con la città.

Capitolo 5

Architettura e Progettazione

Ogni qualvolta un insieme di requisiti viene specificato e dettagliato, esso va implementato. Quindi, al fine di agevolare e snellire il processo di implementazione, è necessaria una fase di design del sistema, individuando un'architettura complessiva che evidensi quali sono i componenti del sistema e come sono collegati, e quali sono le dinamiche che descrivono il comportamento di questi componenti quando interagiscono tra loro per realizzare le funzionalità da implementare. Tuttavia, secondo il processo di sviluppo agile adottato, non esiste un'architettura ben definita già all'inizio del progetto, in quanto i requisiti evolvono in continuazione, quindi il progetto iniziale va raffinato e modificato continuamente durante tutto il ciclo di sviluppo. Di conseguenza, in virtù dei cambiamenti che subiscono le scelte di design, ogni diagramma di progetto presentato in questo capitolo è frutto di un lavoro di continuo raffinamento.

5.1 Vista Componenti e Connitori

Un primo step della fase di design è quello di definire l'architettura di massima del sistema: come sarà strutturato l'insieme dei componenti che interagiscono per realizzare le funzionalità analizzate e specificate nelle fasi precedenti, in relazione anche al rispetto dei requisiti non funzionali richiesti.

Di seguito è riportato un diagramma (figura 5.1) che schematizza una vista *Componenti & Connitori* del sistema, cioè la struttura è immaginata guardando il sistema dal punto di vista dei possibili componenti principali e dei connettori che consentono la comunicazione di questi componenti. Per il team di sviluppo, questa vista è stata fondamentale per stabilire quale pattern architettonico utilizzare per delineare la struttura effettiva del sistema.

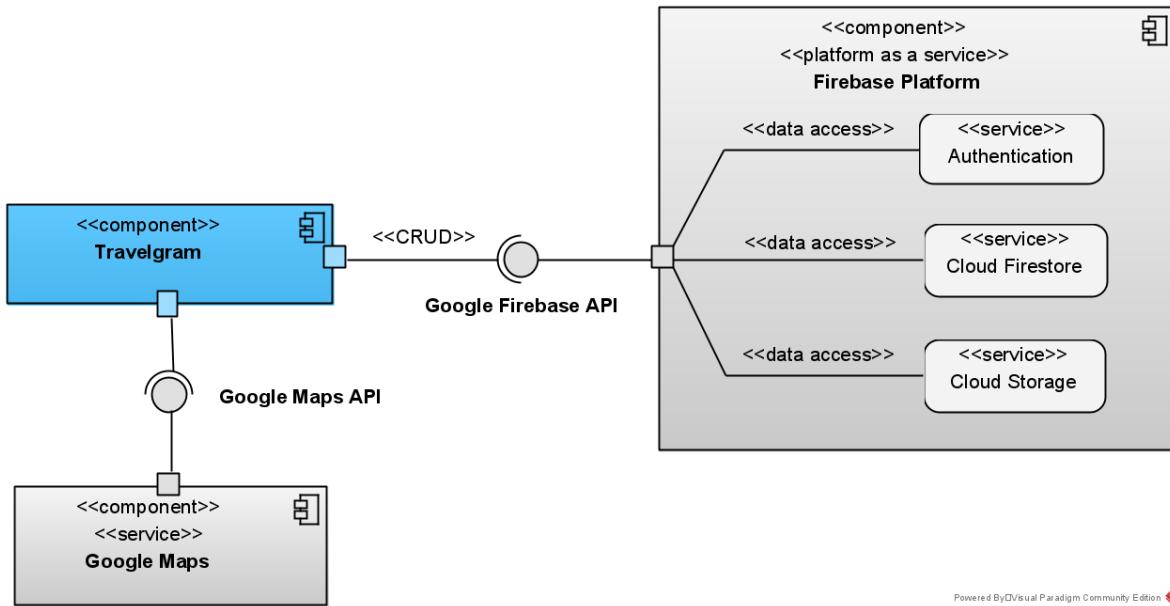


Figura 5.1: Vista componenti e connettori

Analisi dei **componenti** previsti:

- **Firebase Platform**: stereotipato come **<<platform as a service>>** è una piattaforma esterna che offre una serie di servizi utili al funzionamento complessivo del sistema quali:
 1. **Authentication**: realizza tutto il corredo di operazioni legato alla registrazione e autenticazione di un utente. Le interazioni tra le entità esterne e questo servizio di Firebase riguardano la creazione e il prelievo dei dati relativi all'autenticazione di un utente, pertanto il servizio è collegato al porto di uscita del componente mediante una relazione di tipo **<<data access>>**
 2. **Cloud Firestore**: è il servizio di archiviazione di tutte le informazioni utilizzate dal sistema. Anche per questo servizio, le interazioni con le entità esterne sono di tipo **<<data access>>**
 3. **Cloud Storage**: è il servizio di archiviazioni dei dati multimediali che utilizza l'applicazione, in particolare sarà usato per la memorizzazione delle immagini associate alle memory. Anche in tal caso, l'interazione con questo tipo di servizio è di tipo **<<data access>>**
- **Google Maps**: stereotipato come **<<service>>** è il componente che rappresenta il servizio di gestione delle mappe, fondamentale per realizzare la mappa interattiva con cui possono interagire gli utenti del sistema
- **Travelgram**: è il componente che rappresenta il sistema, il quale è costituito da un unico prodotto software

Analisi dei **connettori** previsti:

- *Travelgram – Google Maps*: l’interazione tra questi componenti è stata rappresentata mediante la notazione di *interfaccia richiesta* e *interfaccia fornita*. Infatti Travelgram richiede l’interfaccia di accesso e di gestione delle mappe, Google Maps fornisce esattamente questa interfaccia. Pertanto, il connettore che realizza questo tipo di comunicazione consiste nelle *Google Maps API*, che da un lato vengono fornite, dall’altro vengono utilizzate
- *Travelgram – Firebase Platform*: anche in questo caso è stata utilizzata la notazione lollipop, in quanto Travelgram richiede l’interfaccia di accesso e di gestione delle informazioni archiviate nel database remoto, e Firebase fornisce le funzioni di questa interfaccia. Pertanto, il connettore che realizza questo tipo di comunicazione consiste nelle *Firebase API*. Travelgram utilizza queste API per eseguire le sue operazioni CRUD

5.2 Architettura Client-Server

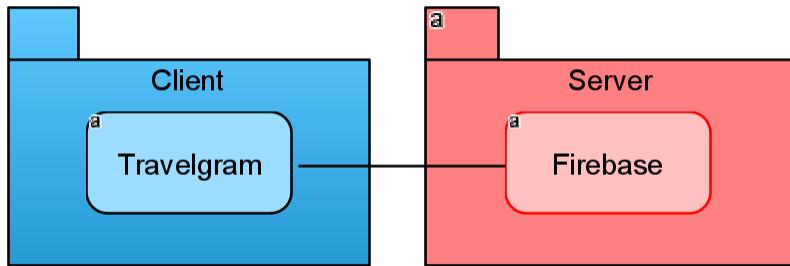


Figura 5.2: Architettura Client-Server

Come mostrato in figura 5.2, il sistema Travelgram realizza il componente Client di un’architettura di tipo Client-Server. È stato infatti scelto il pattern suggerito in figura 5.3, in cui la piattaforma Firebase si occupa interamente della dinamica lato Server, permettendo agli sviluppatori di concentrarsi solo sul codice Client^[6].

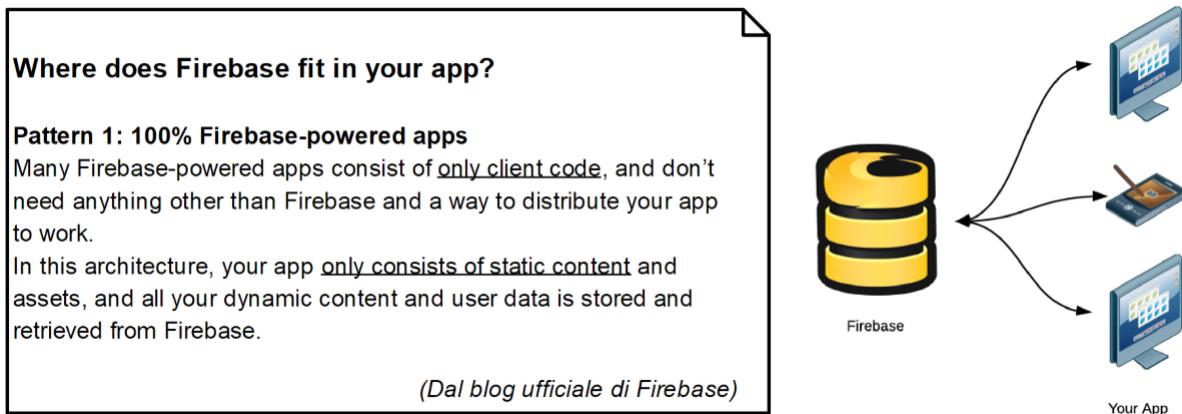


Figura 5.3: Pattern suggerito da Firebase

5.3 Architettura MVVM

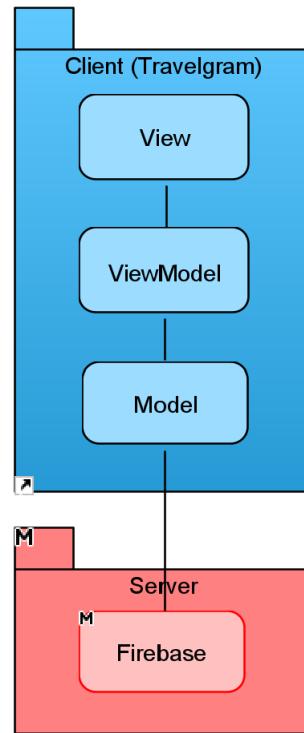


Figura 5.4: Architettura MVVM

Per la realizzazione dell'applicazione Android si è scelto di adottare il pattern architettonico MVVM (Model - View - ViewModel)^[7] (figura 5.4).

La principale motivazione per cui è stato selezionato MVVM è che questo pattern è indicato dalla guida ufficiale per gli sviluppatori^[8] come una Best Practice,

e quindi come architettura raccomandata per la costruzione di applicazioni Android. Si tratta di un'architettura molto usata per applicazioni basate su molte interazioni con l'utente.

I componenti principali di MVVM sono:

- **View:** comprende la logica UI, ovvero tutto ciò che occorre per gestire l'aspetto di ciò che vede l'utente e catturare le sue interazioni.
- **ViewModel:** è un intermediario tra View e Model. Fornisce dati provenienti dal Model ad uno o più componenti UI e trasforma gli input provenienti dalla View in azioni sul Model. Si occupa anche di trasformare i dati provenienti dal Model per prepararli alla presentazione, attraverso, ad esempio, conversioni di formato.
- **Model:** comprende la logica di dominio e di accesso al database.

Il pattern MVVM impone alcuni semplici **vincoli** da rispettare:

- La View non deve contenere logica diversa da quella UI
- La View conosce il ViewModel, ma nessun elemento del ViewModel deve conoscere la View
- La View e il Model possono interagire tra loro solo attraverso la mediazione del ViewModel

Un esempio di come interagiscono tra loro i vari elementi di MVVM è presentato nella sezione 5.8 (dinamica del sistema).

Vantaggi di MVVM: Un vantaggio è che si basa quindi sul principio **separazione di interessi** (*separation of concerns*), secondo cui le classi relative all'interfacciamento con l'utente non devono in alcun modo contenere una logica diversa da quella di gestione dell'interfaccia. Alcuni vantaggi del *separation of concerns* sono:

- Le classi UI risultano molto più leggere e performanti
- Si riduce la probabilità di errore
- Favorisce il riuso, la leggibilità e la manutenzione del codice

MVVM è, inoltre, nativamente supportato da Android attraverso classi apposite. Il binding dei dati tra ViewModel e View avviene in Android tramite un wrapper chiamato *LiveData*, attraverso cui il ViewModel può presentare alla View un flusso continuo di informazioni che viene aggiornato in tempo reale.

5.4 Data Model

5.4.1 Firestore

Il servizio **Cloud Firestore** di Firebase mette a disposizione un database NoSQL di tipo documentale. In questo tipo di database ogni elemento è un documento json e un insieme di documenti prende il nome di collezione. I concetti di documento e collezione sono simili a quelli di tupla e tabella del mondo relazionale, con la differenza che un documento è semi-strutturato, ovvero diversi documenti possono anche avere un insieme di campi differente.

Per realizzare le funzionalità presentate nell'elaborato si è fatto uso di due collezioni: "Travelers" (figura 5.5) e "Memories" (figura 5.6).

The screenshot shows the Firebase Cloud Firestore interface. On the left, there's a sidebar with a project icon labeled 'travelgram-psss'. Below it are two buttons: '+ Avvia raccolta' and '+ Aggiungi documento'. Underneath these are two collections: 'Memories' and 'Travelers'. The 'Travelers' collection is selected, indicated by a grey border around its name and a right-pointing arrow. To the right of the sidebar, the main area displays the 'Travelers' collection. It shows a single document with the ID 'Nmwt33fBWh6yt7XPZV8hKFDhq42'. This document contains several fields: 'followers' (with two items: 'arafRk4rExT9or1RcJXzVPil4r82' and 'd2gEC09J1yV3Ecxlcaeh08xfDk1'), 'following' (with one item: 'arafRk4rExT9or1RcJXzVPil4r82'), 'username' (set to 'ivano'), 'visited_countries' (with two items: 'Italy' and 'France'), and 'wished_countries' (with one item: 'France').

Figura 5.5: Collezione Travelers

travelgram-psss	Memories	GSsjTrA17f569sPEMc1z
+ Avvia raccolta	+ Aggiungi documento	+ Avvia raccolta
Memories >	00Is07sAAyvCJrMV9	+ Aggiungi campo
Travelers	6Y1FxfVMQs2zhqgy1f	city: "Berlino"
	BABUX3l1ei7XQPbETg	country: "Germany"
	GSsjTrA17f569sPEMc	date: "6/10/2020"
	nNonM395imHtY2FJ4j	description: "Porta di Brandeburgo"
	uk5Jyq7fuamCNqnrt	imageLink: "https://firebasestorage.google
	v1YGygTlyhgURoJpP8	pssss.appspot.com/o/arafRk4rE
	zjk7NTp5cPiDuCt70Y	alt=media&token=6c956e64-2f
		▼ traveler
		UID: "arafRk4rExT9or1RcJXzVPil4r82"
		username: "fabrizio"

Figura 5.6: Collezione Memories

Si noti come nella collezione "Memories" si usa la **ridondanza** dei dati, tipica dei database NoSQL. In particolare, ogni Memory contiene anche alcuni dati relativi al Traveler che l'ha pubblicata, poiché accade spesso che assieme alle informazioni della Memory debbano essere mostrate anche questa porzione di informazioni del Traveler. Questo permette di ottenere tutte le informazioni che servono con un'unica query, soluzione molto importante dal momento che il prezzo servizio di Firestore è proporzionale al numero di letture (conceitto approfondito nel capitolo 6).

5.4.2 Authentication

I dati relativi agli utenti autenticati sono gestiti da Firebase nella tabella di figura 5.7. Per motivi di sicurezza, Firebase non mostra le password nella tabella.

CAPITOLO 5. ARCHITETTURA E PROGETTAZIONE

Identificatore	Provider	Data creazione	Accesso eseguito	UID utente ↑
fabio@gmail.com	✉	7 ott 2020	16 ott 2020	18BvNeqn40WmZs0PkgnC...
ivano@gmail.com	✉	6 ott 2020	9 ott 2020	Nmwtt33fBWh6yt7XPZV8hK...
espresso@gmail.com	✉	16 ott 2020	16 ott 2020	UpEeM2VL0gWR56p9tiRlm...
fabrizio@gmail.com	✉	6 ott 2020	14 ott 2020	arafRk4rExT9or1RcJXzVPil4...
lino2@gmail.com	✉	10 ott 2020	12 ott 2020	d2gEC09J1yV3Ecxlcaeah08...

Figura 5.7: Utenti registrati

5.4.3 Cloud Storage

Cloud Storage è il servizio usato per salvare le immagini degli utenti. Si è scelto di creare una cartella diversa per ogni utente (figura 5.8), ognuna contenente le immagini pubblicate da quell'utente (figura 5.9).

<input type="checkbox"/>	Nome	Dimensioni	Tipo	Ultima modifica
<input type="checkbox"/>	Nmwtt33fBWh6yt7XPZV8hKFDhq42/	—	Cartella	—
<input type="checkbox"/>	arafRk4rExT9or1RcJXzVPil4r82/	—	Cartella	—
<input type="checkbox"/>	d2gEC09J1yV3Ecxlcaeah08xfDk1/	—	Cartella	—

Figura 5.8: Cartelle degli utenti

	Nome	Dimensioni	Tipo	Ultima modifica
<input type="checkbox"/>	59029	4.99 MB	image/jpeg	7 ott 2020
<input type="checkbox"/>	59534	5.36 MB	image/jpeg	7 ott 2020
<input type="checkbox"/>	62535	118.33 KB	image/jpeg	6 ott 2020
<input type="checkbox"/>	62536	141.51 KB	image/jpeg	6 ott 2020
<input type="checkbox"/>	62538	96.64 KB	image/jpeg	6 ott 2020
<input type="checkbox"/>	62540	75.56 KB	image/jpeg	6 ott 2020

62535 X



Nome
[62535](#)

Dimensioni
121.165 byte

Tipo
image/jpeg

Data di creazione
6 ott 2020, 16:34:05

Data di aggiornamento
6 ott 2020, 16:34:05

Figura 5.9: Contenuto della cartella di un utente

5.5 Package Diagram

Nel package diagram in figura 5.10, sono mostrati i package e le loro principali dipendenze.

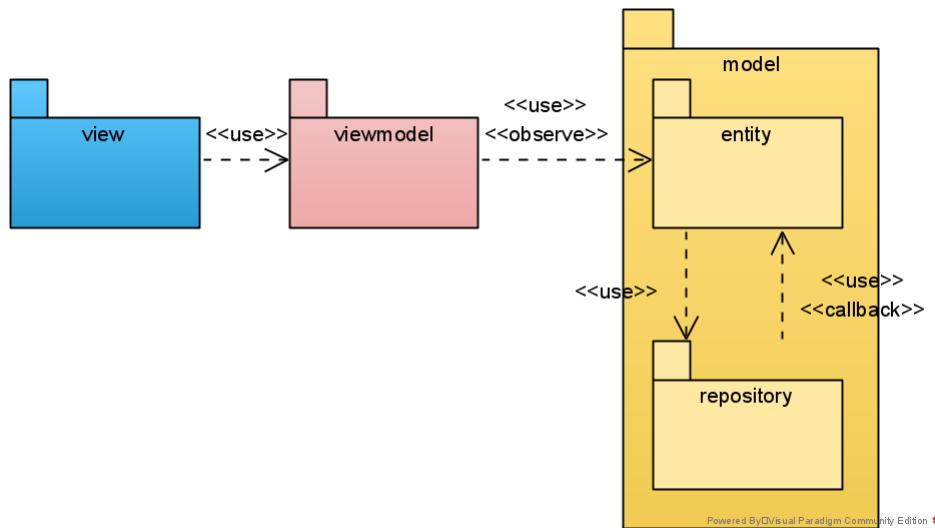


Figura 5.10: Package Diagram

Come si evince dal diagramma, si è scelto di suddividere il Model in due sotto-package al fine di separare la logica di dominio (package Entity) dalla logica di accesso al database (package Repository). Questo porta vari vantaggi, tra cui quello di disaccoppiare maggiormente le classi del Model e di poter, eventualmente, sostituire facilmente le classi Repository che accedono a Firebase con altre classi che accedono a servizi di storage diversi.

La comunicazione tra i vari package, al di là delle normali invocazioni di metodi, avviene:

- Tra View e ViewModel, attraverso un meccanismo di **data binding** (non visibile nel package diagram perché realizzato con librerie native di Android).
- Tra ViewModel e Model, attraverso un **pattern Observer** (indicato con lo stereotipo <<observe>> nel diagramma), che sarà descritto nel dettaglio in seguito.
- Tra i sotto-package Entity e Repository, tramite un meccanismo di **callback** (indicato con lo stereotipo <<callback>> nel diagramma). Il motivo per cui è stato usato tale meccanismo sarà motivato nella sezione 5.8 (dinamica del sistema).

Nella figura 5.11 è mostrata una versione più dettagliata del package diagram, in cui sono mostrate tutte le dipendenze presenti tra le classi, evidenziando in particolar modo anche i sotto-package del package View. Questi sotto-package sono stati usati per separare le classi relative ai concetti di Activity, Fragment e Adapter di Android.

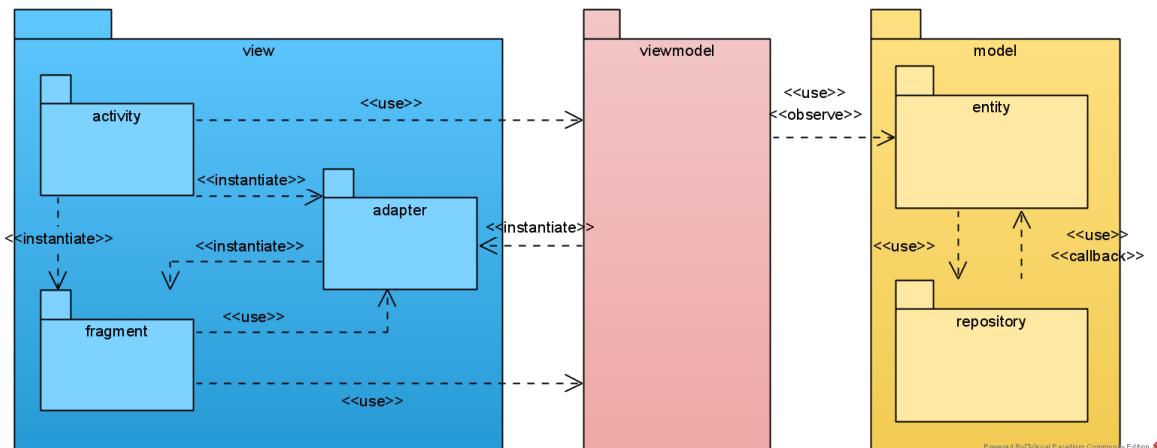


Figura 5.11: Packag Diagram dettagliato

5.6 Class Diagrams

Mostriamo ora il Class Diagram. Per evidenziare in maniera molto più chiara e semplice le relazioni tra le classi, si è scelto di nascondere il contenuto delle classi (attributi e metodi) e lasciarne solo il nome. Una descrizione dettagliata delle classi con tutto il contenuto è invece presentata nei paragrafi successivi.

In figura 5.12 è mostrata la prima versione realizzata del Class Diagram. Per le funzionalità selezionate ai fini dell'elaborato, non sono state implementate tutte le classi relative alle entità di dominio descritte nel System Domain Model, ma solo quelle mostrate in questo diagramma. In questa prima versione, sono state tradotte fedelmente le associazioni trovate nel SDM.

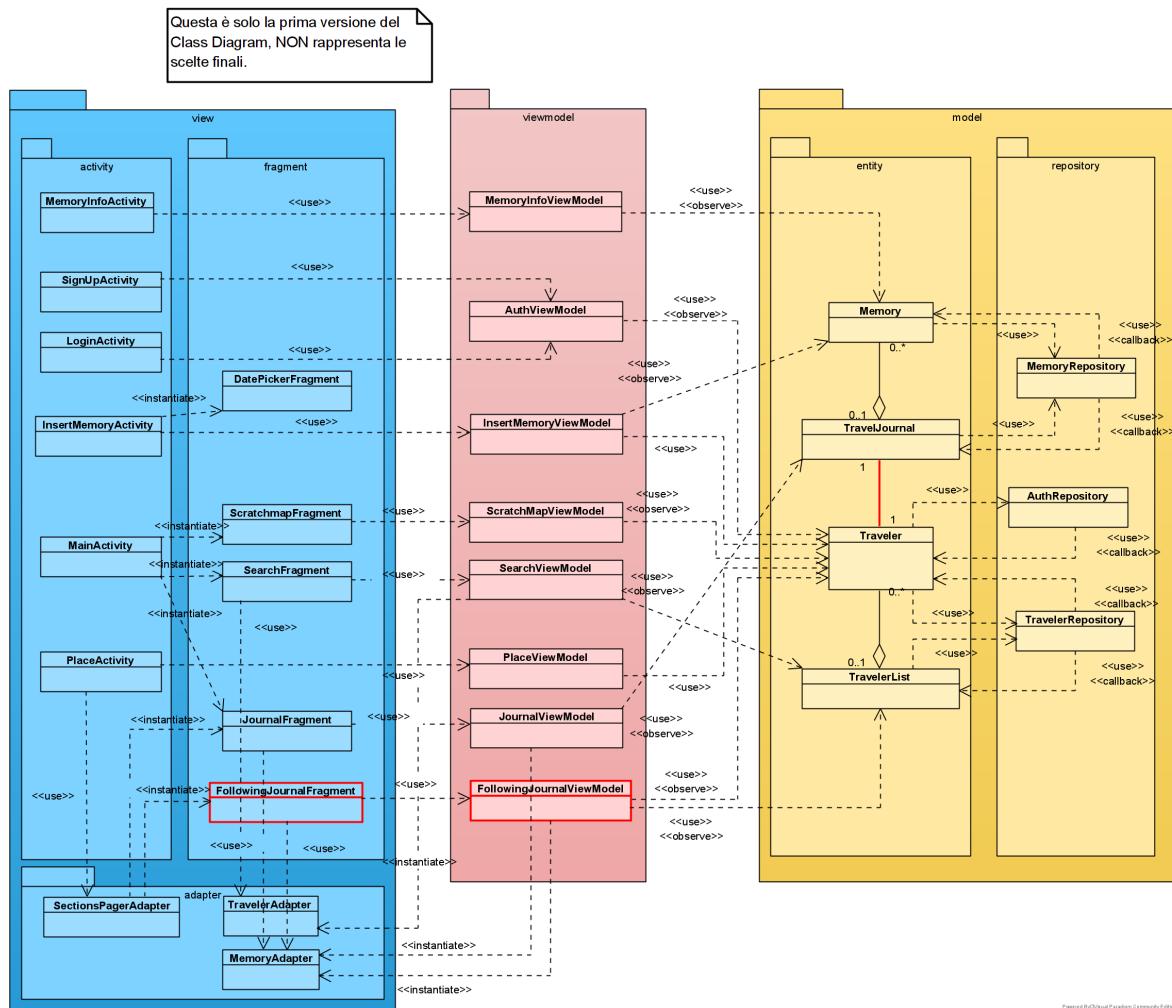


Figura 5.12: Class Diagram (prima versione)

Si noti che sono presenti relazioni d'uso anche tra gli information expert (TravelerList e TravelJournal) e i rispettivi oggetti Repository (rispettivamente TravelerRepository e MemoryRepository). Senza tale relazione d'uso, infatti, sarebbe stato necessario effettuare un numero di query pari alla grandezza della lista degli elementi dei contenitori.

In figura 5.13 è invece mostrata la versione definitiva del Class Diagram. Rispetto alla versione precedente sono stati eliminati:

- L'associazione 1 a 1 tra Traveler e TravelJournal
- La classe FollowingJournalFragment del package view.fragment
- La classe FollowingJournalViewModel del package.viewmodel

Si noti che la rimozione dell'associazione 1 a 1 appena citata cambia leggermente senso alla entità TravelJournal del System Domain Model. A seguito della rimozione, un oggetto di tipo TravelJournal non rappresenta più l'insieme di Memory pubblicate da uno specifico Traveler (come nel SDM), ma rappresenta un generico information expert usato qualora si volesse gestire un insieme di Memory, a prescindere da chi le ha pubblicate.

Il problema della prima soluzione (figura 5.12) sta nel fatto che, per visualizzare le Memory di tutti i traveler seguiti (caso d'uso Visualizza Memories, scenario 2.c), il FollowingJournalViewModel deve: - Recuperare l'elenco dei traveler seguiti (1 query) - Per ogni Traveler seguito, caricare il suo TravelJournal (N query, dove N è il numero di traveler seguiti) - Mostrare i risultati solo dopo che tutte le query sono terminate

Nella seconda soluzione (figura 5.13), basta effettuare una sola query che contiene tutte le informazioni necessarie. In questa soluzione, le classi FollowingJournalFragment e FollowingJournalViewModel non sono necessarie, perché è possibile riusare le classi JournalFragment e JournalViewModel.

I motivi per cui la seconda soluzione è migliore della prima saranno opportunamente spiegati nel capitolo 6, ma possiamo anticipare che sono legati a motivi economici e al modo in cui si è scelto di effettuate le query.

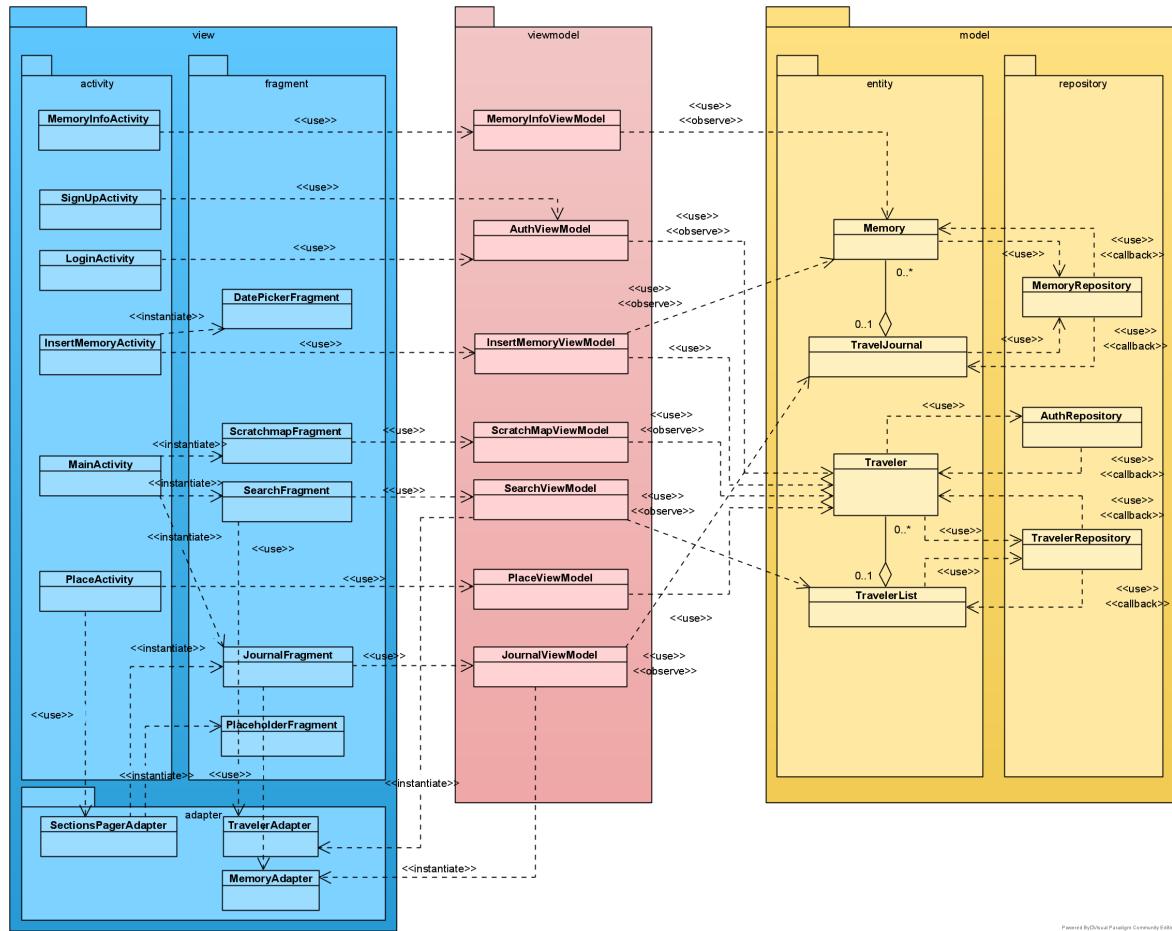


Figura 5.13: Class Diagram (versione definitiva)

Saranno ora analizzati in dettaglio i singoli package.

5.6.1 Package View

Come già accennato, si è scelto di separare le Activity, i Fragment e gli Adapter in tre package diversi al fine di rendere più comprensibile il loro utilizzo. Gli Adapter sono usati per il corretto utilizzo di RecyclerView e del FragmentPagerAdapter di Android. Il class diagram è mostrato in figura 5.14.

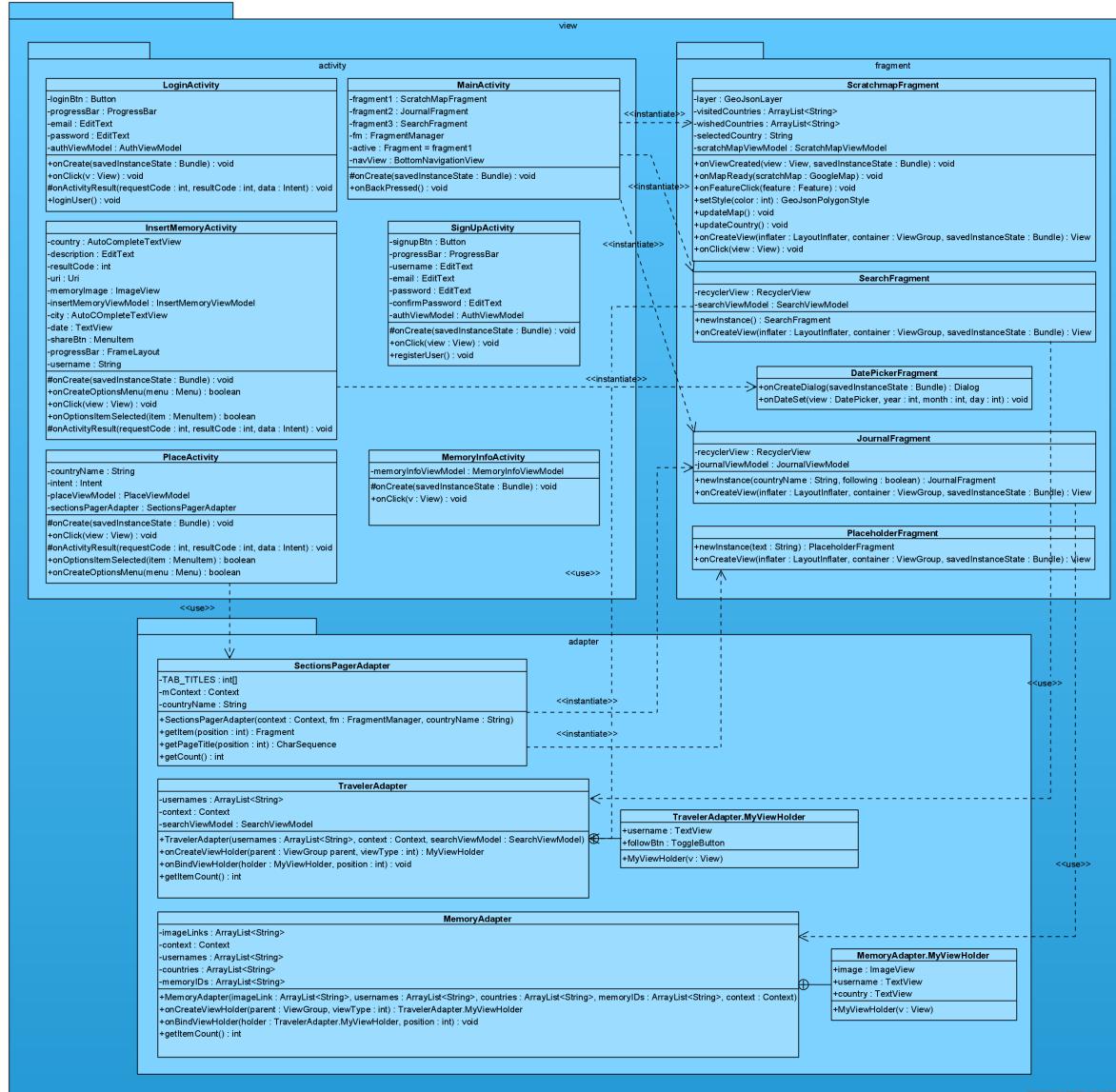


Figura 5.14: Class Diagram del package View

5.6.2 Package ViewModel

Non è stato ritenuto necessario suddividere il ViewModel in ulteriori package. Il class diagram è mostrato in figura 5.15.

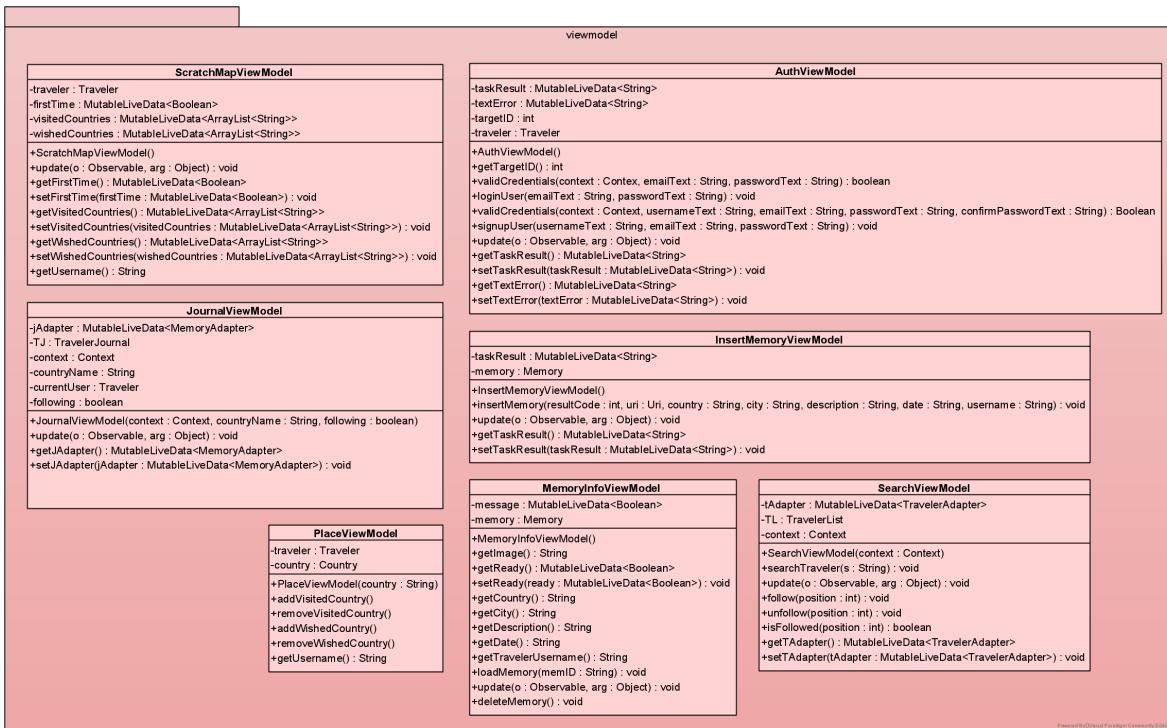


Figura 5.15: Class Diagram del package ViewModel

5.6.3 Package Model

Il package model (figura 5.16) prevede due sotto-package:

- **Entity**: per gestire le entità di dominio.
- **Repository**: per gestire l'accesso al database. Ad eccezione di AuthRepository, che si occupa unicamente del servizio di autenticazione utenti, le classi Repository sono state concepite come un'astrazione del database documentale FireStore:
 - TravelerRepository si occupa unicamente dell'accesso alla collezione "Travelers" su FireStore
 - MemoryRepository si occupa unicamente dell'accesso alla collezione "Memory" su FireStore

Si noti che, a differenza di altre note architetture (come BCED), il package Repository è visto come un'astrazione del database, piuttosto che dell'Entity. In altre parole, abbiamo una classe per ogni collezione sul database, non una classe per ogni entity del dominio. Questo può portare ad un accoppiamento leggermente peggiore, ma favorisce anche una maggiore coesione di tali classi.

Il motivo per cui i package sono stati chiamati Entity e Repository al posto di Entity e Database è anche quello di sottolineare la differenza con un'architettura BCED, la quale, tra l'altro, è a livelli. Nel caso dell'applicazione Travelgram,

invece, non c'è alcun vincolo che impone al package Repository di non poter usare i servizi del package Entity.

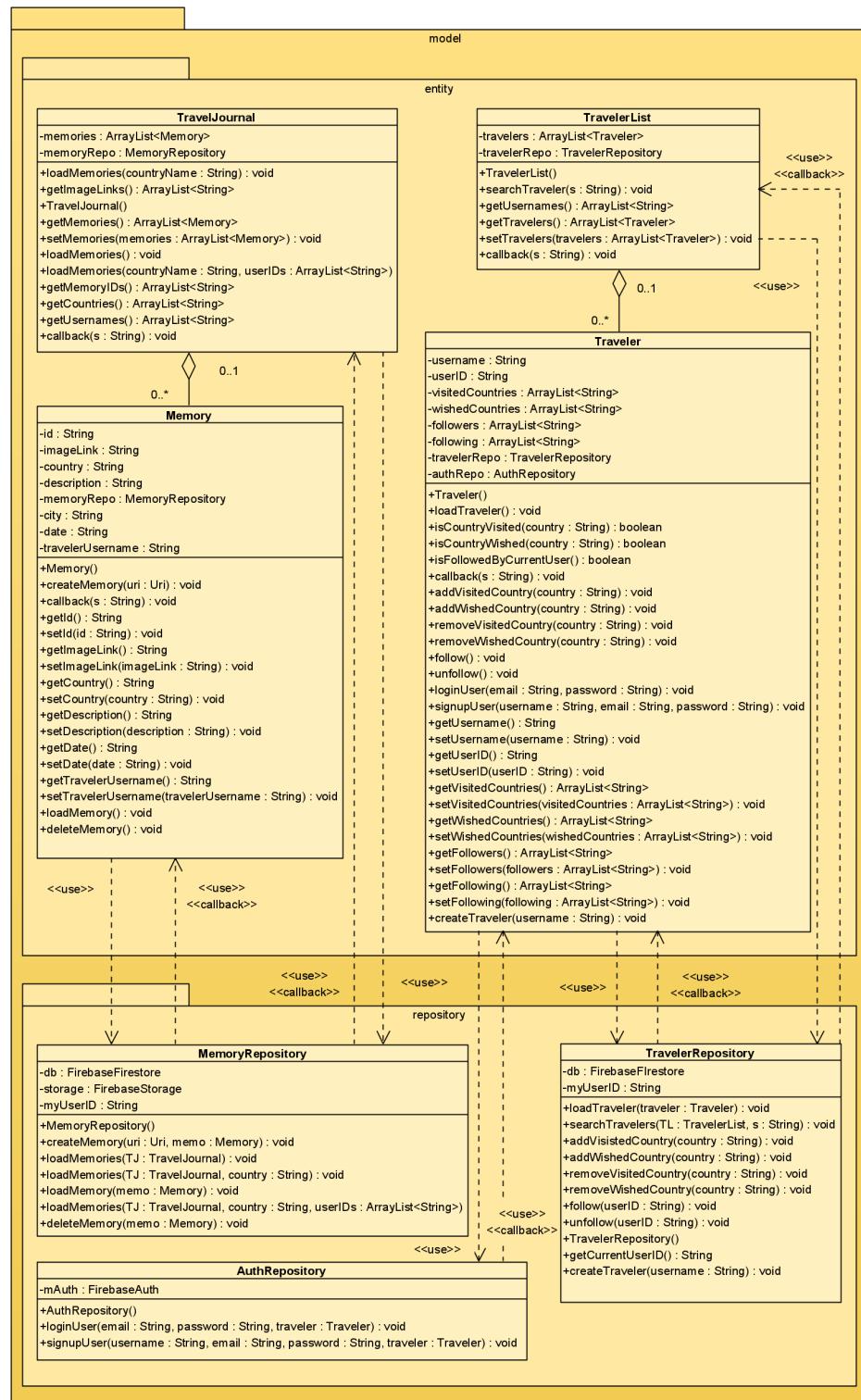


Figura 5.16: Class Diagram del package Model

5.7 Context Diagram with Boundary

Un ulteriore passaggio cruciale della fase di progettazione è quello di stabilire tutte le relazioni che il sistema può avere con l'esterno, evidenziando tutti i possibili attori, sottosistemi e servizi con cui può interagire, e definendo le classi che utilizza per questi tipi di interfacciamento (*boundary*). A tal proposito, i risultati di questo tipo di analisi sono stati schematizzati mediante l'utilizzo di un Context Diagram with Boundary (figura 5.17), in cui il sistema è rappresentato come una scatola chiusa di cui vediamo solo le classi di interfacciamento con l'esterno, dato che non ci interessano i dettagli interni per comprendere queste relazioni, e le relative entità esterne.

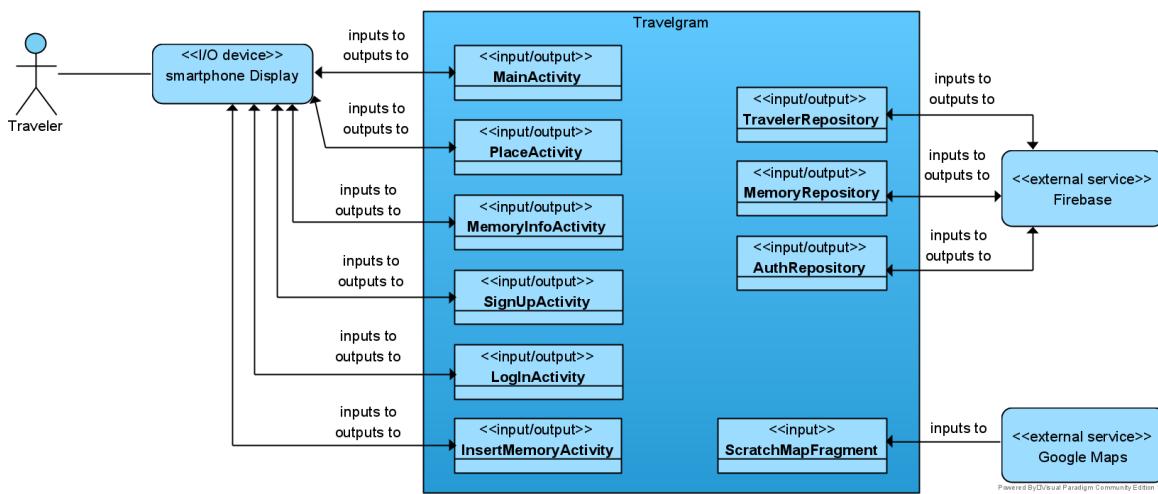


Figura 5.17: Context Diagram con Boundary

L'unico attore esterno è il *Traveler*, che si interfaccia con il sistema mediante l'utilizzo del *display dello smartphone* stereotipato come *<<I/O device>>*. Le classi di interfacciamento con l'utente che utilizza il sistema sono quelle del package activity, ognuna di esse può sia ricevere input dall'utente, sia mostrare degli output, per questo sono state etichettate come *<<input/output>>*.

Il servizio esterno *Google Maps* interagisce con il sistema mediante la sua classe *ScratchMapFragment* la quale può solo ricevere degli input da questo servizio.

Il servizio esterno *Firebase* interagisce con il sistema mediante le sue classi appartenenti al package repository; ognuna di esse può sia ricevere degli input che mandare degli output a Firebase, quindi sono state etichettate come *<<input/output>>*.

5.8 Dinamica del Sistema

Nella sezione 5.5 (Package Diagram) è stato accennato il modo in cui i package comunicano tra loro. Il seguente schema (figura 5.18) mostra, in maniera molto

semplificata ma anche molto esplicativa, la dinamica di una generica richiesta che parte dall'utente e attraversa il sistema Travelgram.

Si tenga conto del fatto che i primi componenti mostrati nello schema sono quattro generiche classi rispettivamente dei package View, ViewModel, Model.Entity e Model.Repository. L'ultimo componente astrae invece la piattaforma Firebase.

Il diagramma mette in evidenza la comunicazione tra i package, in particolare:

- Gli step 5 e 15 mostrano l'interazione tra View e ViewModel tramite **LiveData**
- Gli step 6 e 14 mostrano l'interazione tra ViewModel e Entity tramite il **pattern Observer**
- Gli step 10 e 13 mostrano l'interazione tra Entity e Repository tramite **callback**

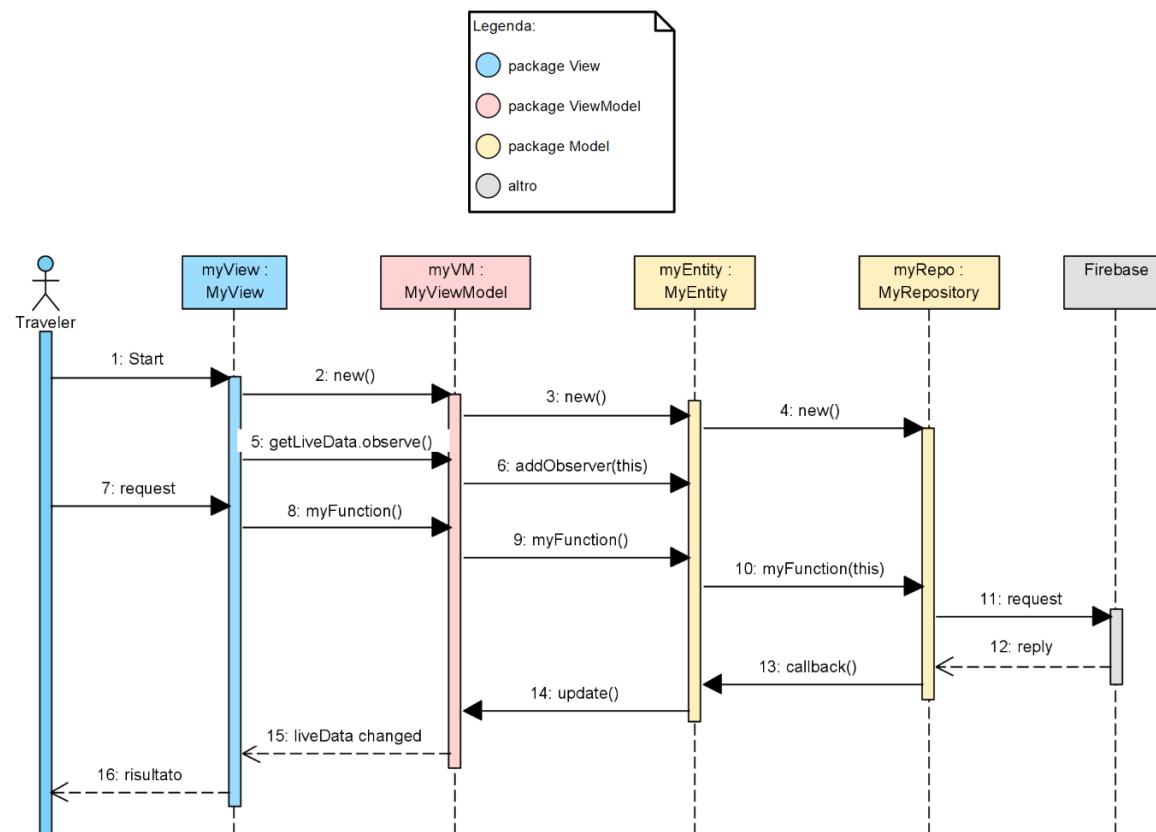


Figura 5.18: Dinamica di una richiesta

L'adozione del meccanismo di callback è stata necessaria per via della natura **asincrona** ed **Event-driven** di Firebase, che non rende possibile il passaggio del risultato di una query come semplice valore di ritorno della funzione dello step 10.

La reply dello step 12, infatti, è un evento asincrono che deve essere catturato da un opportuno listener, che si occuperà poi di avvisare l'Entity tramite la callback.

La dinamica di ogni interazione sarà descritta in maniera molto più dettagliata nei diagrammi successivi. Ulteriori dettagli sulla loro implementazione in Java è invece oggetto del capitolo 6.

Saranno ora mostrati diagrammi che descrivono la dinamica dei principali casi d'uso sviluppati:

- Pubblica Memory
- Visualizza Memory
- Sign up

Sebbene sia stato implementato anche il caso d'uso Log in, la sua dinamica non è stata modellata perché pressoché identica a quella del caso d'uso Sign up, il quale è stato preferito perché mostra un'interazione in più con l'utente e con Firebase.

5.8.1 Pubblica Memory

Il seguente Activity Diagram di basso livello (figura 5.19) descrive il caso d'uso Pubblica Memory con riferimento anche all'interfaccia utente. L'utente ha infatti una duplice scelta, poiché può avviare il vero e proprio caso d'uso o da un pulsante presente sulla schermata della mappa, o cliccando su uno specifico stato (quest'ultimo descritto anche dai sequence diagram successivo).

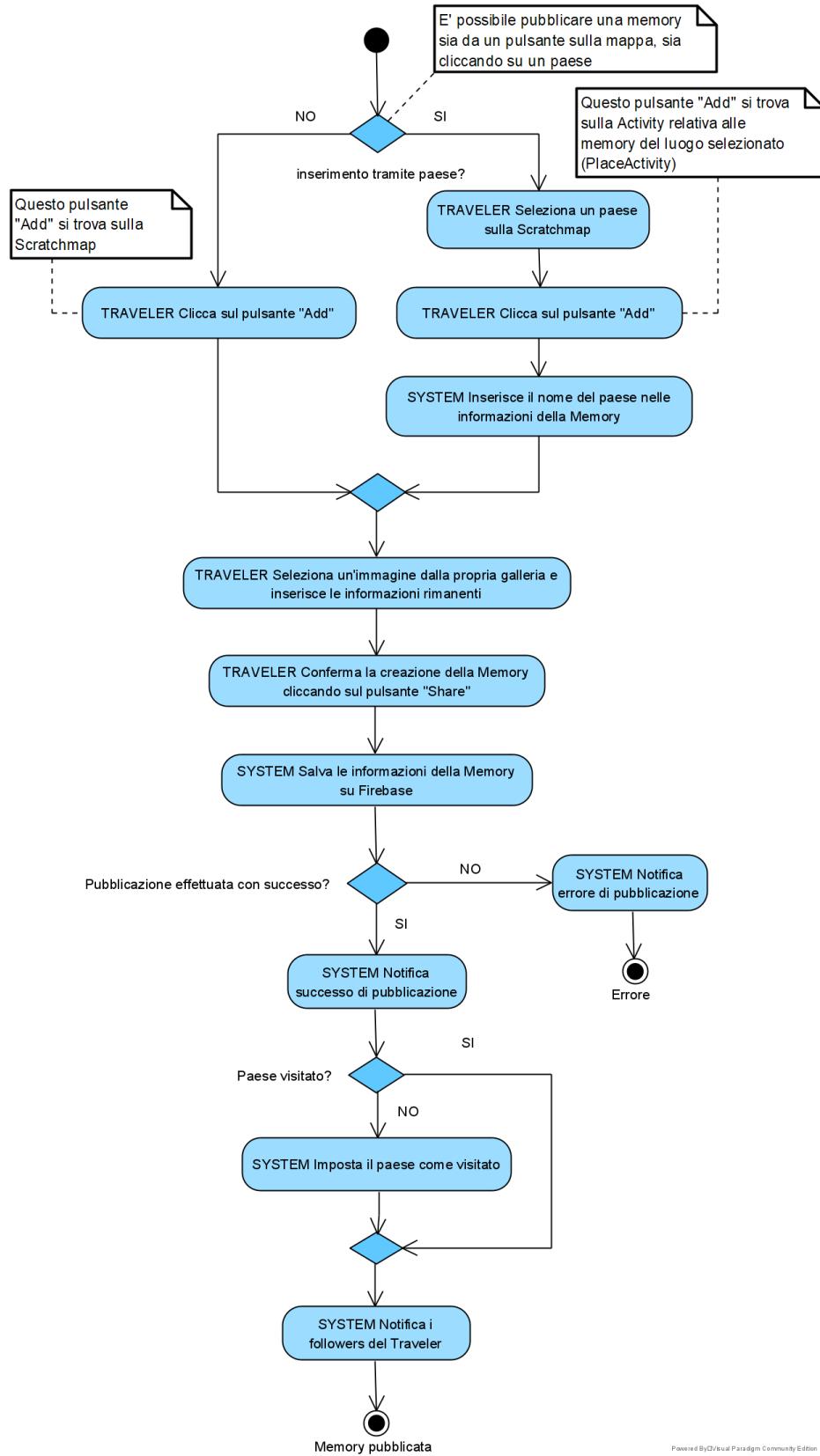


Figura 5.19: Activity Diagram di Pubblica Memory

Per descrivere in maniera più dettagliata e di basso livello il caso d'uso, sono stati realizzati due Sequence Diagram:

- Uno di basso livello, il più possibile fedele al codice, in cui sono mostrati in maniera dettagliata anche il funzionamento dei LiveData e dei Listener
 - Uno di livello leggermente più alto, in cui, per semplificare la lettura, sono stati semplificate le interazioni con LiveData e Listener

In figura 5.20 è mostrato il Sequence Diagram di dettaglio di Pubblica Memory nella versione in cui l'utente clicca su uno specifico paese.

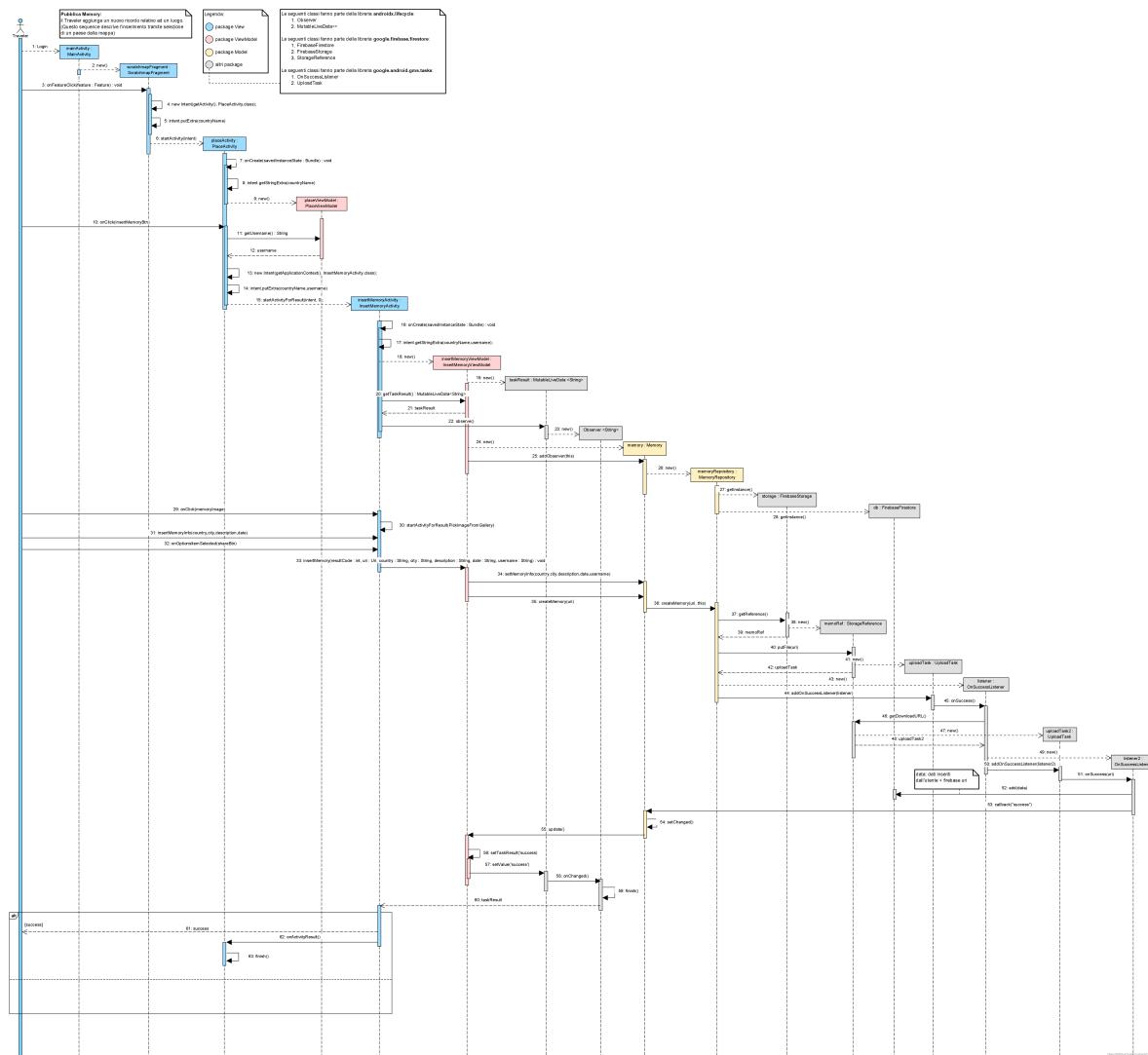


Figura 5.20: Sequence Diagram dettagliato di Pubblica Memory

In figura 5.21 è mostrata la notazione usata per semplificare la porzione di sequenze relativa alla comunicazione tramite LiveData. Per rendere questa semplificazione generale e applicabile anche ai successivi sequenze diagram non

si è fatto riferimento alle classi specifiche di Pubblica Memory, ma piuttosto a generiche classi di View e ViewModel.

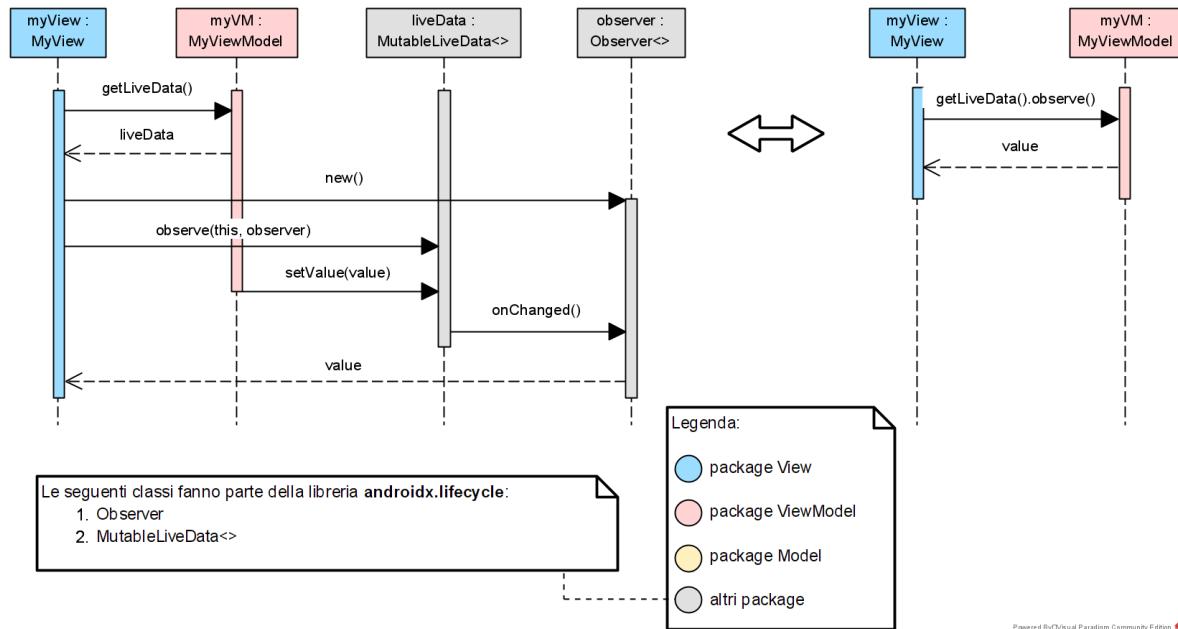


Figura 5.21: Semplificazione dell'interazione con LiveData

In figura 5.22 è mostrata la semplificazione, invece, della porzione di sequenze relativa al funzionamento di un Listener per il servizio di Cloud Storage di Firebase.

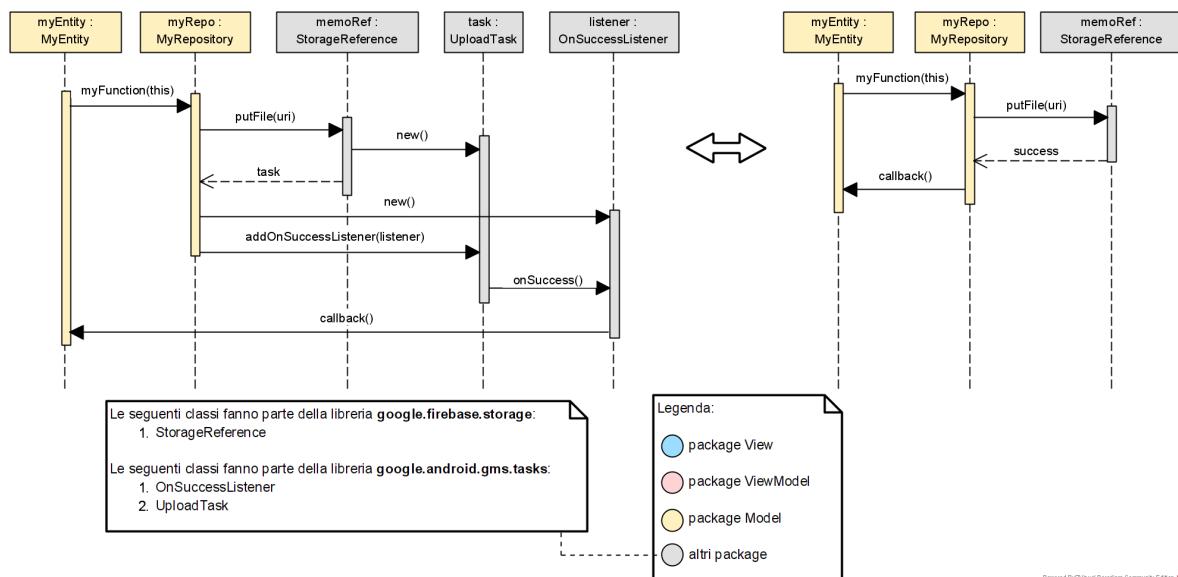


Figura 5.22: Semplificazione della gestione di StorageReference con Listener

Sfruttando queste semplificazioni, è stato generato il Sequence Diagram in figura 5.23, molto più compatto e leggibile di quello appena visto. Da qui in

poi, tutti i Sequence Diagram useranno questa stessa notazione semplificata, dal momento in cui la dinamica di questi elementi rimane invariata.

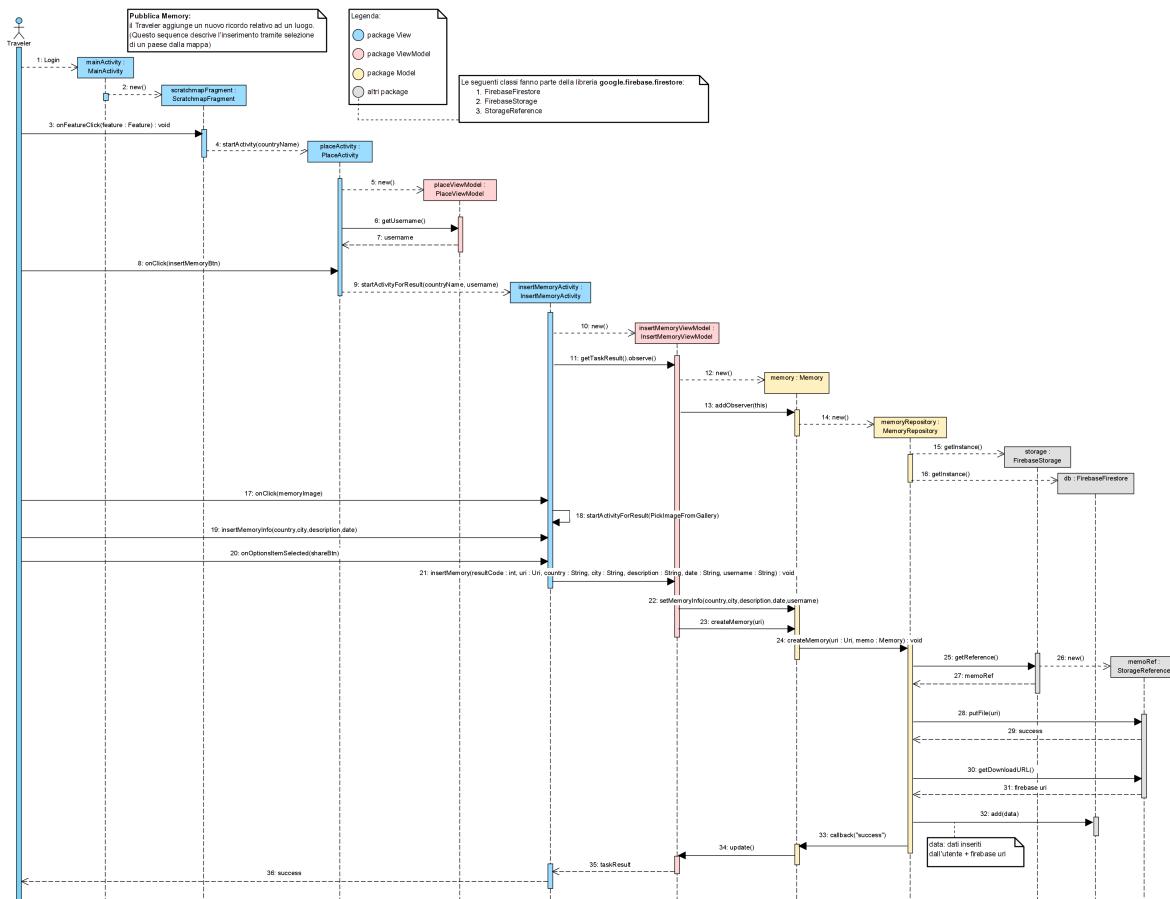


Figura 5.23: Sequence Diagram di Pubblica Memory

5.8.2 Visualizza Memory

Prima di mostrare il Sequence Diagram del caso d’uso Visualizza Memories, mostriamo una notazione semplificata analoga a quella vista sopra per il Listener, ma applicato ad una query su Firestore (figura 5.24).

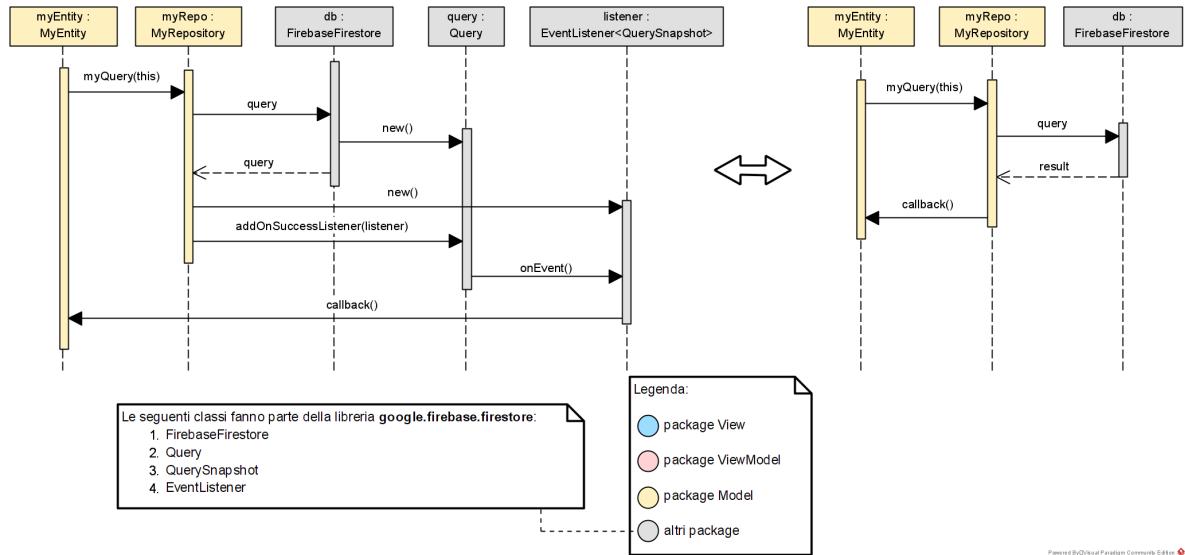


Figura 5.24: Semplificazione della gestione della query con Listener

Il Sequence Diagram in figura 5.25 descrive il caso d'uso Visualizza Memory, in particolare lo scenario 2.c, in cui il sistema mostra l'elenco di Memories relative ad un luogo specifico e pubblicate dai Traveler seguiti dall'utente.

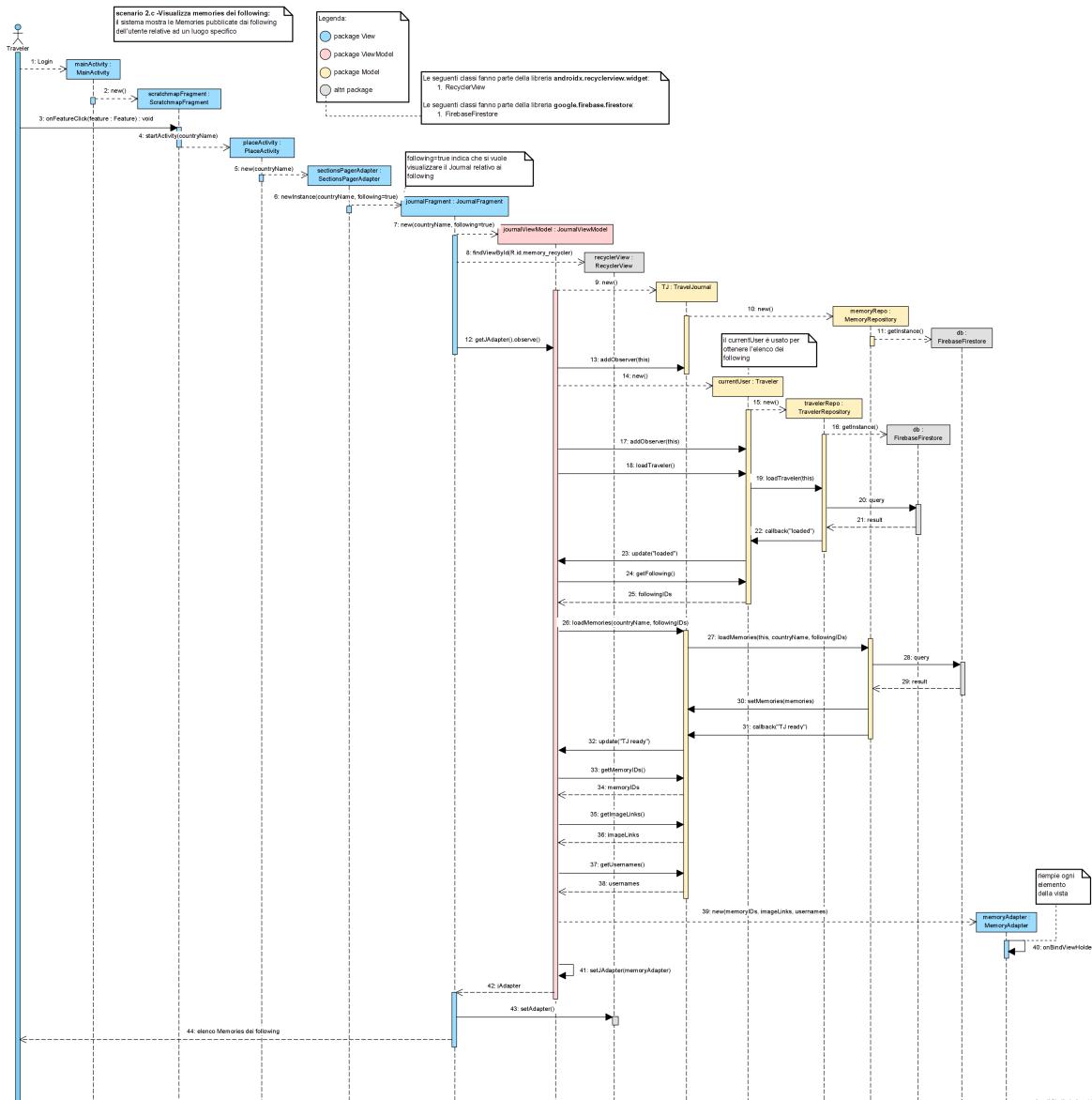


Figura 5.25: Sequence Diagram di Visualizza Memories

5.8.3 Sign up

Prima di mostrare il Sequence Diagram del caso d'uso Sign Up, mostriamo una semplificazione di notazione analoga a quella vista sopra per il Listener, ma mostrando l'utilizzo del servizio di autenticazione di Firebase al posto del servizio di Cloud Storage (figura 5.26).

CAPITOLO 5. ARCHITETTURA E PROGETTAZIONE

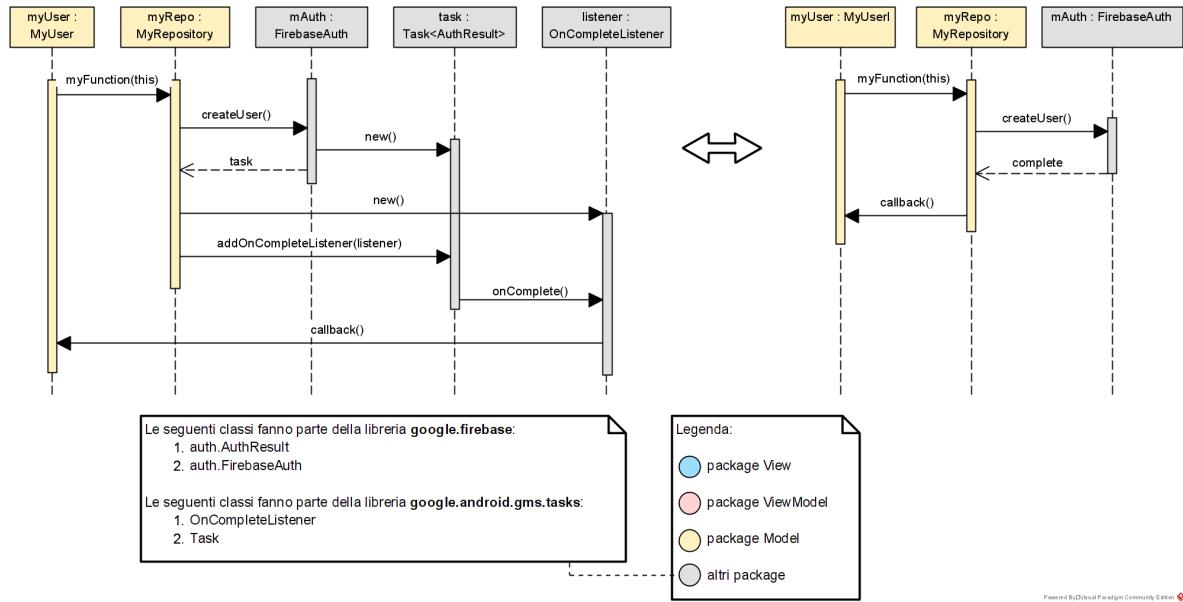


Figura 5.26: Semplificazione della gestione dell'autenticazione con Listener

In figura 5.27 il Sequence Diagram completo di Sign Up.

CAPITOLO 5. ARCHITETTURA E PROGETTAZIONE

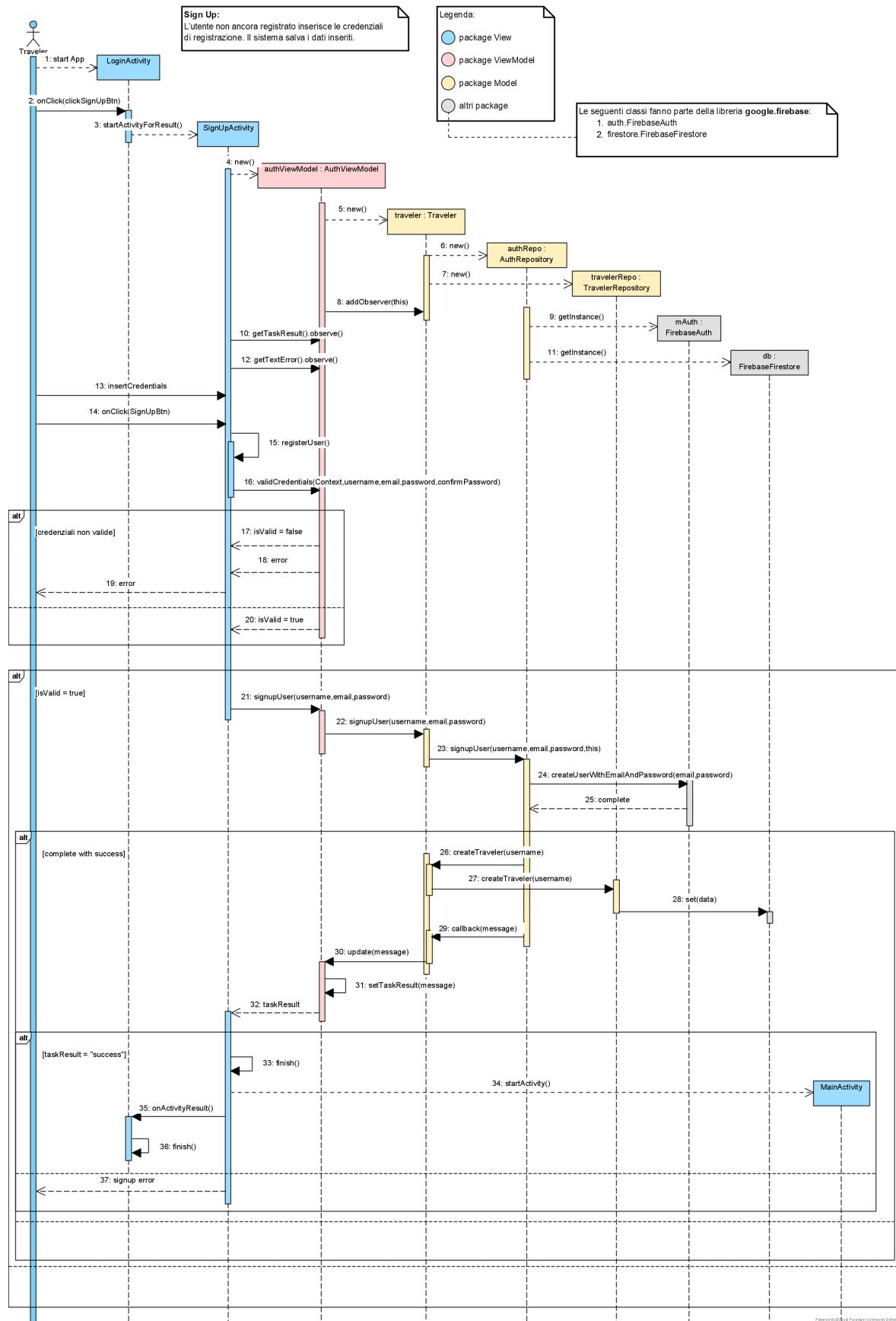


Figura 5.27: Sequence Diagram di Sign Up

5.9 Deployment

Come passaggio finale della fase di progettazione, è utile definire quali sono gli eseguibili prodotti dal team di sviluppo e distribuiti agli utenti finali del sistema, indicando i nodi fisici su cui essi andranno eseguiti nonché quelli su cui sono in esecuzione i servizi esterni di cui il sistema si serve. Tali informazioni sono state schematizzate nel diagramma di Deployment riportato in figura 5.28.

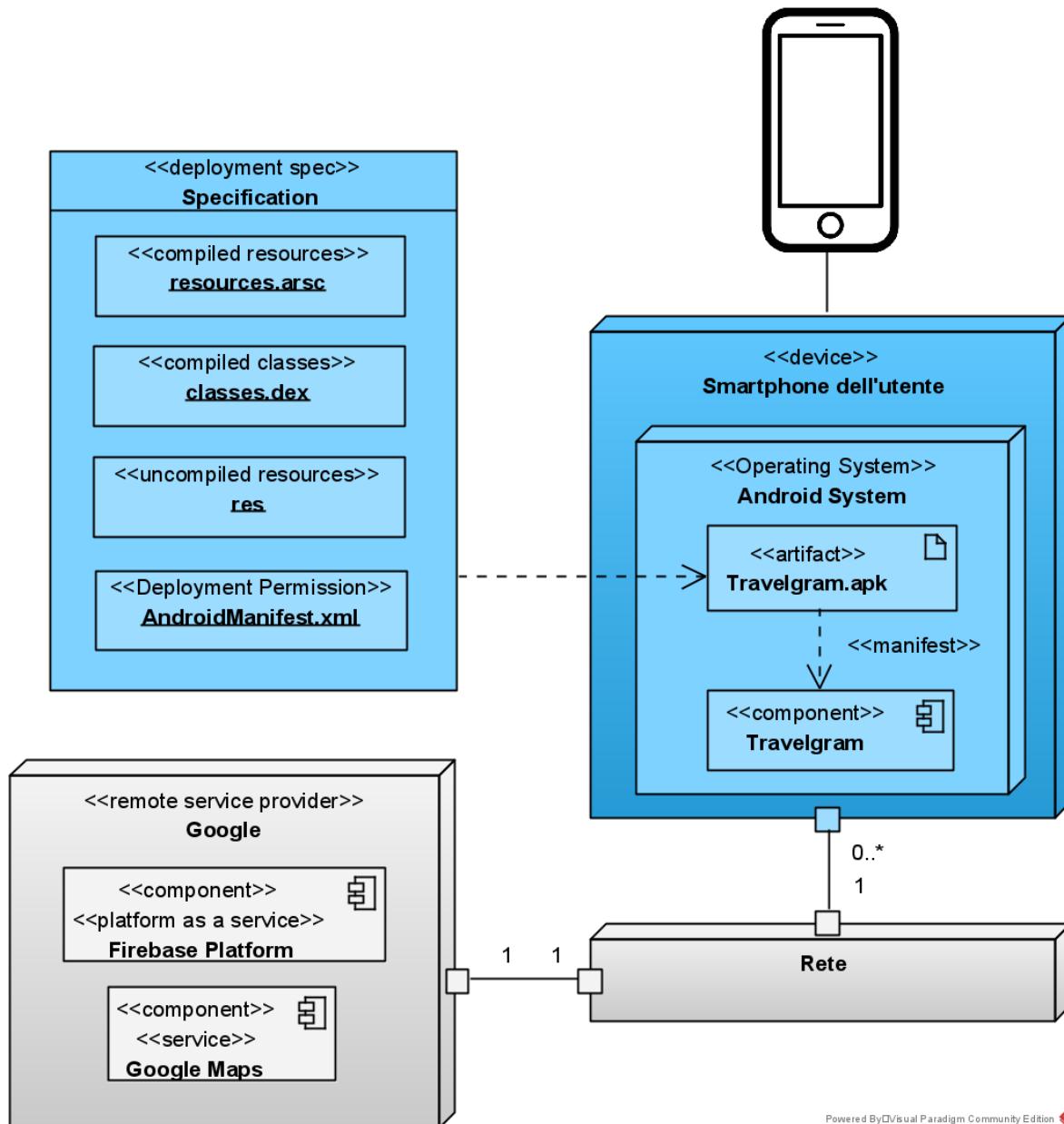


Figura 5.28: Deployment Diagram

Sono stati individuati due tipi di nodi:

- *Device*: rappresenta la tipologia di nodo su cui è eseguita l'applicazione

- *Remote Service Provider*: fa riferimento alla tipologia di nodi che ospitano servizi esterni

In più, il diagramma mostra anche la *rete* come sistema di comunicazione dei nodi.

I servizi esterni utilizzati sono quelli di Google:

- *Google Maps*, stereotipato come <<service>>
- *Firebase Platform*, stereotipato come <<platform as a service>>, cioè un insieme di servizi

Il prodotto software distribuito dal team di sviluppo all'utente finale è l'*Android Package Travlegram.apk*, etichettato con lo stereotipo <<artifact>> che indica proprio gli artefatti software. Esso è in esecuzione nel Sistema Operativo *Android System*, ospitato a sua volta dal nodo *Smartphone dell'utente* di tipo *device*, e manifesta il componente *Travelgram*. Il nodo *device* che ospita l'applicazione utente è collegato alla rete mediante un'associazione *1 a 0..** in quanto il sistema può essere collegato a una e una sola rete, e alla rete possono essere collegati zero o più *device* (perché potrebbero essere tutti disconnessi).

Gli elementi che sono inclusi nel package *Travelgram.apk* sono indicati nel riquadro *Specification*:

- *Classes.dex*, ossia le classi compilate (<<compiled classes>>)
- *Resources.arsc*, ossia le risorse compilate utili all'applicazione (<<compiled resources>>)
- *Res*, ossia risorse utili al sistema non compilate (<<uncompiled resources>>)
- *AndroidManifest.xml*, ossia il file che contiene le informazioni necessarie al sistema per essere eseguito (<<deployment permission>>)

Capitolo 6

Documentazione dell'implementazione e manuale d'uso

Nel seguente capitolo verranno descritte le varie scelte di implementazione. In particolare, il codice prodotto si riferisce ai seguenti Casi d'Uso:

1. Registrazione di un utente
2. Login di un Traveler
3. Following di un utente
4. Pubblicazione di una Memory
5. Visualizzazione delle Memories
6. Segnatura di un Luogo

Questi rappresentano il nucleo dell'applicazione e permettono quindi di avere un'esperienza che seppur non ancora completa, rappresenta le funzionalità principali del sistema. Per cui, seguendo le scelte viste nei capitoli precedenti, abbiamo svolto un lavoro di implementazione che le segue in maniera precisa.

6.1 Organizzazione del Codice

Come già visto, il pattern architettonale utilizzato è stato l'MVVM, con la scelta di separare il Model in due sottopackage denominati entity e repository, mentre il linguaggio di programmazione è stato Java. Le classi java si presentano nei vari package come in figura 6.1.

CAPITOLO 6. DOCUMENTAZIONE DELL'IMPLEMENTAZIONE E MANUALE D'USO

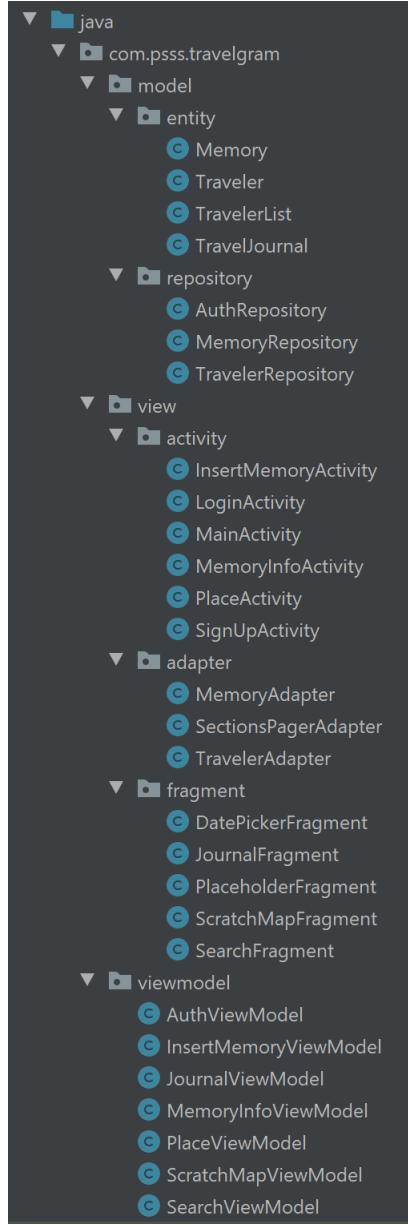


Figura 6.1: Classi java

Si descrivono quindi brevemente le caratteristiche di ogni package:

- *model.entity*: contiene le classi che implementano le entità del nostro dominio.
- *model.repository*: sono le classi che instanziano gli oggetti necessari alla comunicazione con il Server per il prelievo e l'inserimento dei dati dal/nel Database.
- *view.activity*: contiene le classi per l'interfaccia utente e per la gestione degli input ed output dall'esterno.

- *view.fragment*: classi che rappresentano porzioni di interfaccia utente in una activity.
- *view.adapter*: classi che permettono di gestire più elementi dello stesso tipo in un formato compatto (ViewHolder) all'interno di una activity o fragment in modo da realizzare delle liste visibili all'utente.
- *viewmodel*: classi che fanno da intermediari tra View e Model fornendo dati provenienti dal Model alla UI (con eventuali trasformazioni) e convertendo gli input provenienti dalla View in azioni sul Model.

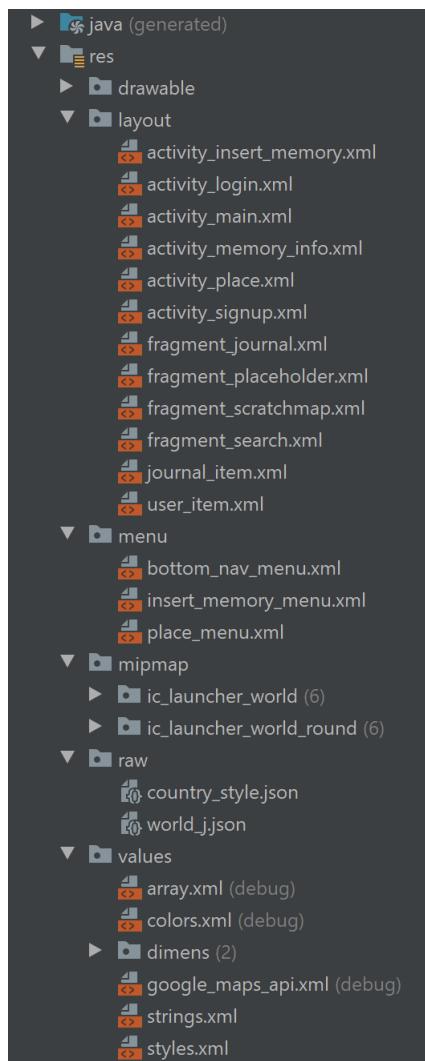


Figura 6.2: Risorse

In figura 6.2 sono presentate le risorse xml e json utilizzate:

- La cartella *drawable* contiene le icone ed altri oggetti grafici
- La cartella *layout* contiene i file xml relativi alle viste

- La cartella *menu* contiene i file xml relativi agli elementi inclusi in varie toolbar o navigation bar
- La cartella *mipmap* contiene l'icona principale dell'applicazione, in diverse risoluzioni
- La cartella *raw* contiene altri elementi diversi da xml, in questo caso solo dei json
- La cartella *values* contiene xml che contengono informazioni usate dall'applicazione

6.2 Descrizione dei file realizzati

Nella tabella seguente si riporta una spiegazione di tutte le classi realizzate:

Classe	Package	Descrizione
Memory, Traveler	Model.Entity	Sono le classi che rispecchiano le entità del dominio di interesse.
TravelerList, TravelJournal	Model.Entity	Sono le classi Information Expert delle classi di dominio che permettono di gestire una collezione di Travelers o Memories.
AuthRepository	Model.Repository	Repository relativa ai dati utili per la registrazione ed il login. Istanzia gli oggetti per la comunicazione con il Database ed in particolare con il servizio Authentication di Firebase.
MemoryRepository, TravelerRepository	Model.Repository	Sono le repository responsabili per la comunicazione con il Database e per il prelievo/inserimento dei dati desiderati.
MainActivity	View.Activity	Si tratta dell'attività principale di inizio dal quale poi vengono presentati i vari Fragment relativi alla ScratchMap, al TravelJournal e alla Search.

CAPITOLO 6. DOCUMENTAZIONE DELL'IMPLEMENTAZIONE E MANUALE D'USO

Classe	Package	Descrizione
LoginActivity, SignUpActivity, In- sertMemoryActivity, MemoryInfoActivity, PlaceActivity,	View.Activity	Le prime due sono le classi necessarie per le operazioni di autenticazione. La terza rappresenta la schermata di inserimento di una nuova memory. La quarta si riferisce alla schermata che si mostra quando si vogliono visualizzare i dettagli di una Memory. L'ultima rappresenta la pagina relativa ad un Luogo.
DatePickerFragment, JournalFragment, ScratchMapFrag- ment, SearchFragment, PlaceholderFrag- ment	View.Fragment	Il primo apre un Dialog pop-up per inserire la data durante la schermata di pubblicazione di una nuova Memory. Il JournalFragment è necessario per le schermate dove si mostra l'elenco di tutte le Memories. Il terzo rappresenta la ScratchMap che viene visualizzata una volta autenticati. Il quarto è la parte di interfaccia utente per ricercare nuovi Travelers da seguire. L'ultimo è un frammento placeholder per sezioni dell'applicazione non ancora implementate.
MemoryAdapter, TravelerAdapter	View.Adapter	Permettono di realizzare le RecyclerView, infatti al loro interno, sono realizzati i ViewHolder, contenitori dei dati da rappresentare in ogni cella. Le RecyclerView permettono di visualizzare un elenco di oggetti, renderizzando solo gli oggetti dell'elenco che devono essere visibili in quel momento.
SectionsPagerAdapter	View.Adapter	Permette di gestire un insieme di Fragment sottoforma di pagine.

Classe	Package	Descrizione
AuthViewModel, InsertMemoryView- Model, JournalViewModel, MemoryInfoView- Model, PlaceViewModel, ScratchMapView- Model, SearchViewModel	ViewModel	Sono i ViewModel che fanno da ponte tra Activity/Fragment e il Model. Gestiscono i dati delle rispettive View e permettono il passaggio di essi da e verso l'entity e la repository. Consentono di astrarre le View dalla responsabilità di gestione dei dati.

Si pone l'attenzione anche su alcuni file che sono ritenuti fondamentali all'interno dell'applicazione:

Risorsa	Descrizione
AndroidManifest.xml	Describe le informazioni essenziali dell'applicazione come il nome, i componenti (attività e servizi) e i permessi richiesti
raw/world_j.json	Contiene tutti i paesi del mondo e relative informazioni (nome, coordinate) necessarie per il funzionamento della mappa.
values/strings.xml	Contiene tutte le stringhe costanti utilizzate all'interno delle Activity dell'applicazione che si riferiscono a particolari elementi nella UI.
values/styles.xml	Contiene gli stili creati da noi per migliorare l'UI dell'applicazione.
Values/google_maps_api.xml	Contiene la key per utilizzare le API di Google Maps.
build.gradle	Contiene le informazioni necessarie per il build dell'applicazione come l'ID dell'applicazione, la versione di Android, le dipendenze da altre librerie o servizi esterni.

6.3 Design Pattern

I principali design pattern utilizzati sono stati:

- Pattern GRASP

- Pattern Repository
- Pattern Observer

6.3.1 Pattern GRASP

I pattern GRASP sono molto utili perché permettono di gestire al meglio le responsabilità di ogni classe. Si basano sul concetto di *Responsibility Driven Design* (RDD), ovvero un modo di progettare ad oggetti facendosi guidare dalle responsabilità. Questo tipo di pattern è essenziale per scrivere un codice più pulito, comprensibile, organizzato, manutenibile e modificabile.

Si è fatto uso dei seguenti pattern GRASP:

- Information expert
- Session controller
- Basso accoppiamento
- Alta coesione

Information expert: Ogni oggetto dovrebbe avere le responsabilità per cui possiede le informazioni necessarie a soddisfarle. Nel caso in cui occorre gestire una lista di oggetti di una certa classe, quindi, non è opportuno assegnare tale responsabilità a questa classe, poiché questa conosce solo le informazioni di un unico oggetto.

È stato quindi creata una classe apposita in due occasioni:

- *TravelJournal*, usata quando si vuole gestire un insieme di più *Memory*
- *TravelerList*, usata quando si vuole gestire un insieme di più *Traveler*

Session controller: Le architetture a strati prevedono una separazione tra UI e logica applicativa. La UI delega le richieste al controller, che coordina vari oggetti del dominio per soddisfare la richiesta.

Nel caso dell'applicazione Travelgram, che adotta il pattern MVVM, non esiste un vero e proprio controller. L'obiettivo delle classi ViewModel, infatti, è quello di presentare i dati provenienti dal Model alla View, attraverso opportune conversioni di formato. Il ViewModel incapsula quindi la logica di preparazione dei dati, più che quella di controllo. Tuttavia, nel caso in cui occorre conoscere informazioni legate a più entità, ha senso affidare al ViewModel il compito di coordinare le varie entità del model per ottenere tali informazioni.

Le classi ViewModel, sebbene non siano dei veri e propri controller, possono essere viste come implementazioni del pattern GRASP **Session Controller** (o Controller di Caso d'Uso), ovvero un oggetto che gestisce gli eventi di uno specifico caso d'uso, senza corrispondere a nessuno specifico oggetto del dominio. Ogni

View, infatti, inoltra gli input ad un oggetto ViewModel ad essa associata, che si occupa poi di gestire la richiesta. Poiché su applicazioni mobile, in genere, una View gestisce un singolo caso d'uso, anche il suo ViewModel gestirà unicamente quello stesso caso d'uso.

Low coupling: L'obiettivo è quello di cercare di creare meno dipendenze possibili tra le classi. Troppe dipendenze, infatti, rendono il codice più difficile da modificare e manutenere, dal momento che anche una piccola modifica può impattare un gran numero di classi.

Alta coesione: Consiste nel mantenere gli oggetti focalizzati e comprensibili. L'uso di questo pattern è dimostrato dall'uso di oggetti di entità, ognuno dei quali raggruppa tutte le funzionalità relative solo a se stessa. Altre classi pensate per supportare l'alta coesione sono le classi Repository, ognuna delle quali contiene tutte le interrogazioni relative ad un'unica collezione dati, senza suddividere tali richieste in più classi.

6.3.2 Pattern Repository

Come già accennato, il package Repository è visto come un'astrazione del database. Nella nostra implementazione, ogni oggetto Entity ha la responsabilità di usare uno specifico oggetto Repository per ottenere dati dal database o inserire dati sul database. Per via della natura asincrona di Firebase, un oggetto Repository comunica il successo di un'operazione all'Entity tramite callback.

6.3.3 Pattern Observer

Il pattern Observer permette a uno o più oggetti (*observers*) di osservare i cambiamenti di stato di un altro oggetto (*subject*). Nell'applicazione Travelgram si è fatto largo uso di questo pattern in diverse varianti:

- Data binding: realizzato con i metodi della classe nativa di Android "MutableLiveData"
- Pattern Observer: classica implementazione del pattern attraverso le classi di java.util
- Callback: può essere vista come una variante semplificata del pattern

La dinamica di queste comunicazioni è stata già trattata nella sezione 5.8 (dinamica del sistema), ma in questa sezione si mostra in dettaglio l'implementazione.

Data Binding: L'oggetto ViewModel contiene un oggetto di tipo MutableLiveData della libreria androidx.lifecycle. È stata quindi usata tale implementazione del pattern observer per la comunicazione tra View e ViewModel, usando i metodi:

- *getValue()* per ottenere un riferimento al dato
- *observe()* per tenere traccia dei cambiamenti

Si noti come il viewModel non conosce affatto l'oggetto View. In figura 6.3 è mostrato un esempio.

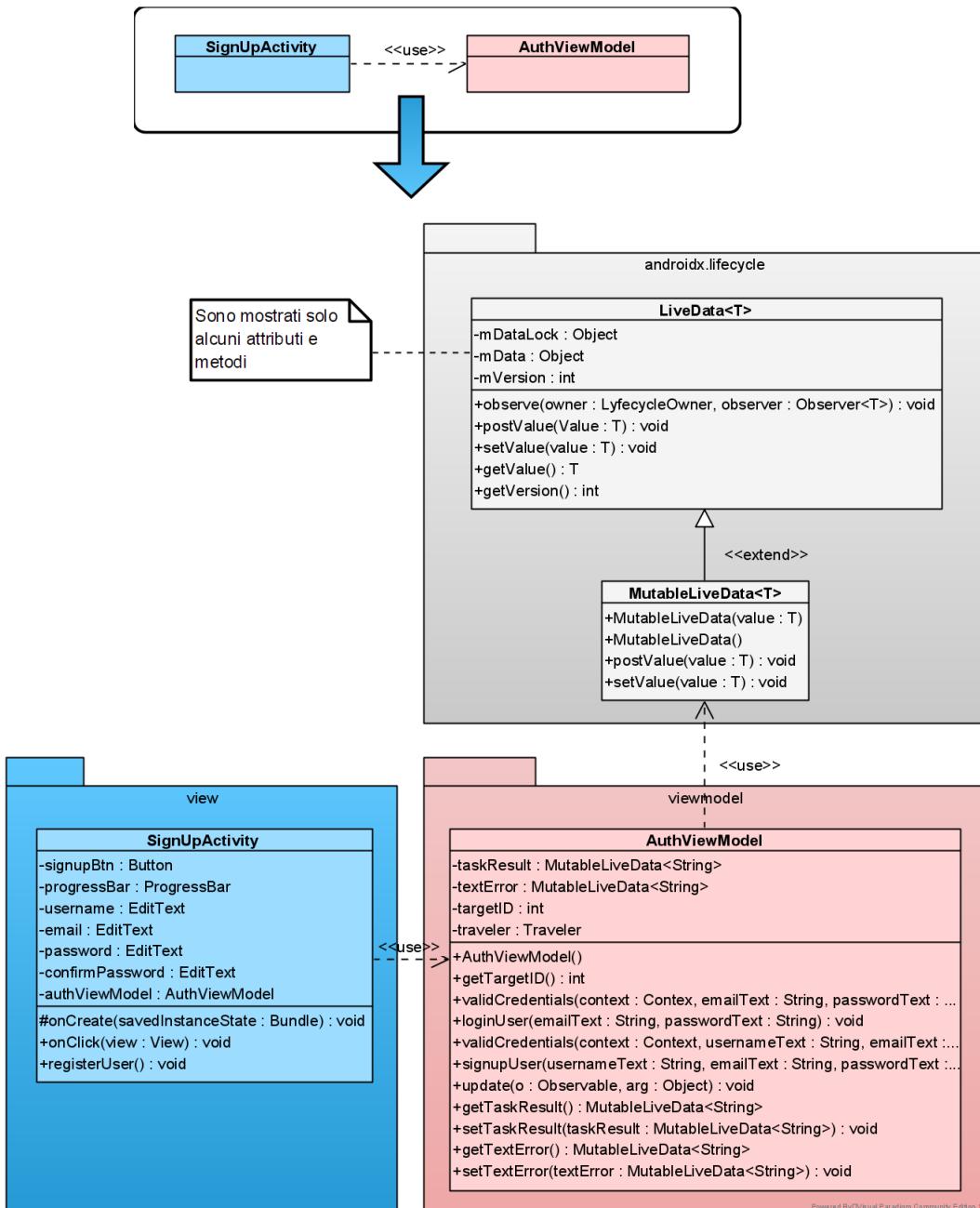


Figura 6.3: Data binding con LiveData

Pattern Observer: La versione classica del pattern è stata usata per la comunicazione tra ViewModel e Model.Entity ed implementata sfruttando le classi Observer e Observable della libreria java.utils.

Sono stati usati i metodi:

- *addObserver(this)* per osservare il subject
- *setChanged()* per indicare un cambiamento di stato
- *notifyObservers()* per notificare gli observer del cambiamento
- *update()* per gestire la notifica del cambiamento

In figura 6.4 è mostrato un esempio.

CAPITOLO 6. DOCUMENTAZIONE DELL'IMPLEMENTAZIONE E MANUALE D'USO

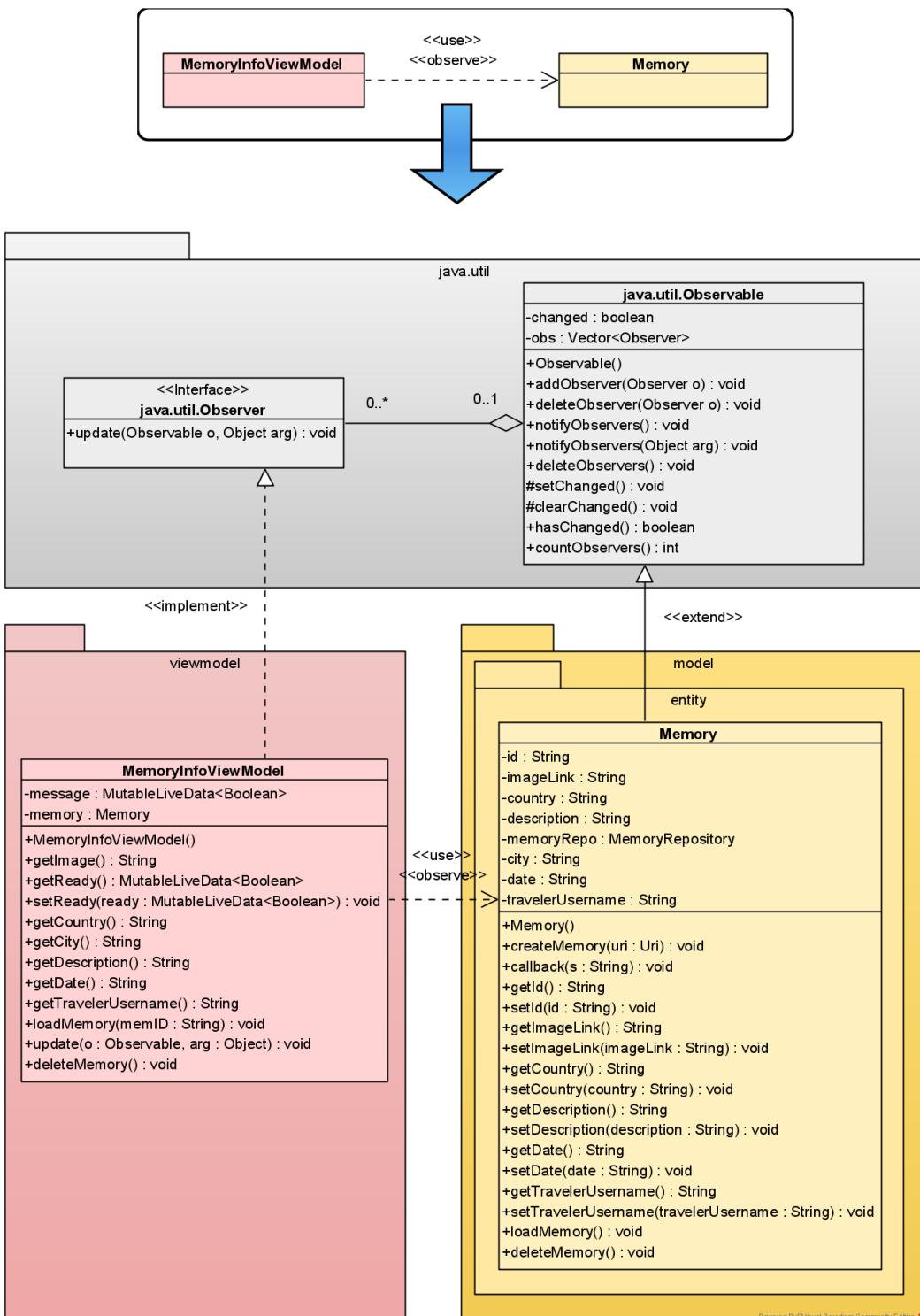


Figura 6.4: Pattern Observer

Callback: Per la comunicazione tra Entity e Repository, è stato scelto di usare una versione semplificata del pattern Observer, dal momento che in questi casi vi è sempre solo un unico osservatore. In particolare, l'oggetto Entity passa il suo riferimento all'oggetto Repository, così che quest'ultimo possa avvisarlo e

passargli il risultato una volta che l'operazione ha termine attraverso il metodo `callback()`.

6.4 Firestore Query

Le query per l'interrogazione del database Firestore di Firebase possono rac cogliere uno specifico documento o un insieme di documenti all'interno di una collezione.

Per l'applicazione Travelgram, si è fatto uso anche di particolari query dette **document snapshot**, che permettono di ottenere degli aggiornamenti in tempo reale dal database. Ogni volta che il contenuto del documento osservato cambia, la query è rieseguita automaticamente.

Le query sono molto performanti grazie all'utilizzo di opportuni **indici**. Ogni volta che viene inserito un nuovo documento sul database, viene creato automaticamente un indice per ogni campo del documento. Ciò permette di ottimizzare le operazioni di lettura, che avvengono quindi in tempi brevissimi, penalizzando però i tempi di scrittura di documenti con molti campi. Questo problema è però trascurabile, dal momento che in media il numero di scritture è inferiore del numero di letture per molti ordini di grandezza. L'indicizzazione rende il processo di interrogazione scalabile, poiché fa sì che le performance non siano proporzionali all'intero dataset, ma solo al result set.

Filtraggio delle query: Piuttosto che eseguire delle query generali ed effettuare successivamente un'operazione di filtraggio in java, si è preferito eseguire il filtraggio direttamente nelle query. Il motivo è che nel primo caso, se ad esempio avessimo voluto recuperare tutte le Memory relative a un dato luogo, avremmo dovuto prima ottenere tutte le Memory presenti sul database (che in un social potrebbero essere milioni), operazione che avrebbe provocato un eccessivo ed ingiustificato consumo di risorse (memoria, dati, batteria, tempo).

Query con più risultati: Come anticipato nella sezione 5.6 (class diagram), nei casi in cui è stato necessario recuperare informazioni relative a più oggetti dello stesso tipo (ad esempio un insieme di Memory o una lista di Traveler), si è preferito ottenere tutti i risultati attraverso un'unica query. Questo metodo, infatti, è molto più efficiente rispetto al caso in cui si effettuino, in un ciclo for, molteplici query, ognuna delle quali recupera un singolo elemento. Ciò è vero per vari motivi:

- Si riduce l'**overhead**: l'overhead di un'unica grande query è sicuramente minore di quello di molte query piccole
- È più **facile da gestire**: essendo le query asincrone, e dovendo gestire ognuna di esse con un Listener dedicato, è più difficile gestire un grande numero di query alla volta

- È più **economico**: il costo del servizio Firestore è proporzionale al numero di letture effettuate. Ogni query conta infatti come un'unica lettura, a prescindere dal numero di documenti restituiti. In particolare, il prezzo è di \$ 0.06 ogni 100 mila letture^[19].
- È più **scalabile**: all'aumentare del numero di documenti da recuperare, la query da effettuare è sempre e solo una

6.5 Manuale di Utilizzo

Giunti a questo punto, presentiamo un sintetico Manuale di utilizzo del software. Si consiglia caldamente tuttavia, di prendere visione del video di anteprima^[11], poiché risulta essere sicuramente più esplicativo e chiarificatore rispetto agli screenshot che verranno riportati in questa sezione, che non mettono purtroppo in risalto le scelte di Design pronunciate in precedenza.

6.5.1 Configurazione ed installazione

Il nostro software non prevede complessi step di configurazione, se non l'unica necessità di poter installare applicazioni di terze parti. Per l'installazione del software, è stato generato un file .apk compatibile con i sistemi Android dalla versione 5.0 (Lollipop) in poi^[21]. Il file va caricato sul dispositivo Android che si ha a disposizione ed installato attraverso un qualunque File Manager. Nel caso venisse chiesto, quindi, è importante abilitare la possibilità di installare applicazioni derivanti da fonti sconosciute.

6.5.2 Flow di navigazione ed utilizzo

Di seguito vengono riportati quindi alcuni screenshot dell'applicazione con una descrizione di cosa rappresentano.

CAPITOLO 6. DOCUMENTAZIONE DELL'IMPLEMENTAZIONE E MANUALE D'USO

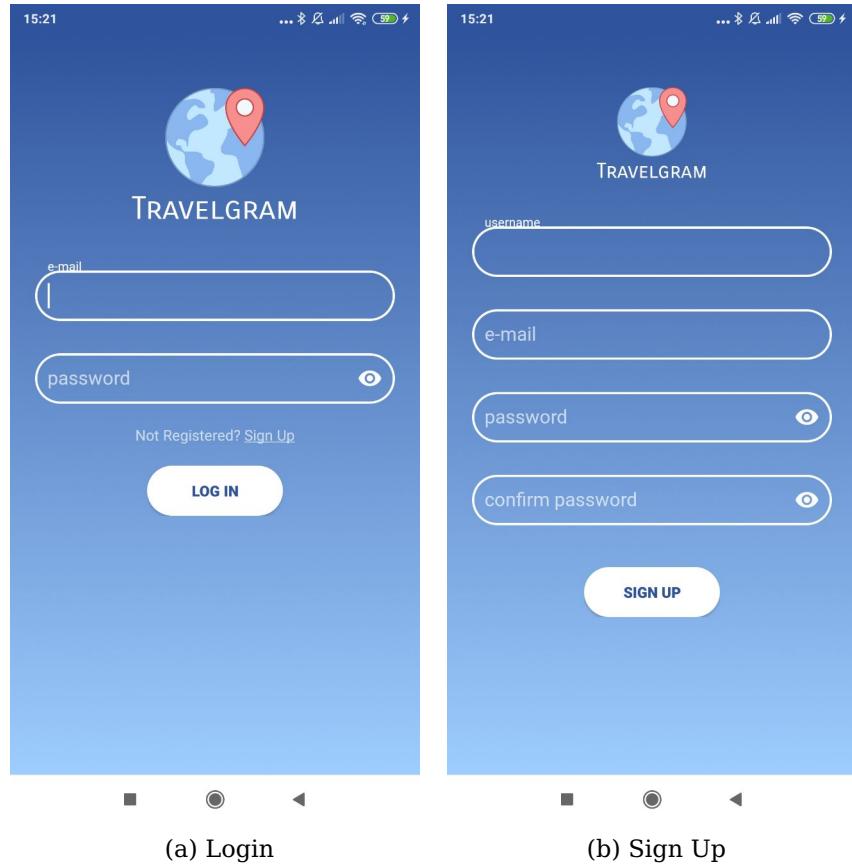


Figura 6.5: Autenticazione

Autenticazione: All’apertura dell’applicazione si verrà portati nella schermata in figura 6.5a per effettuare il Login. Nel caso in cui non si disponga ancora di un account, cliccando sull’apposita scritta si verrà invece portati sulla schermata di Sign Up (figura 6.5b) per la creazione di un nuovo Traveler.

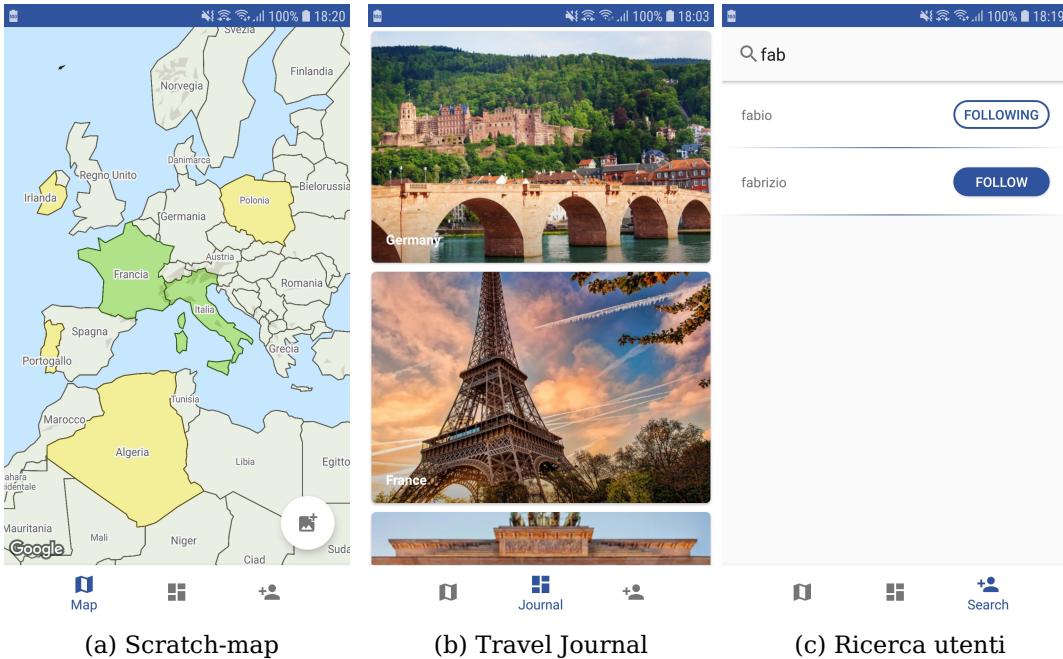
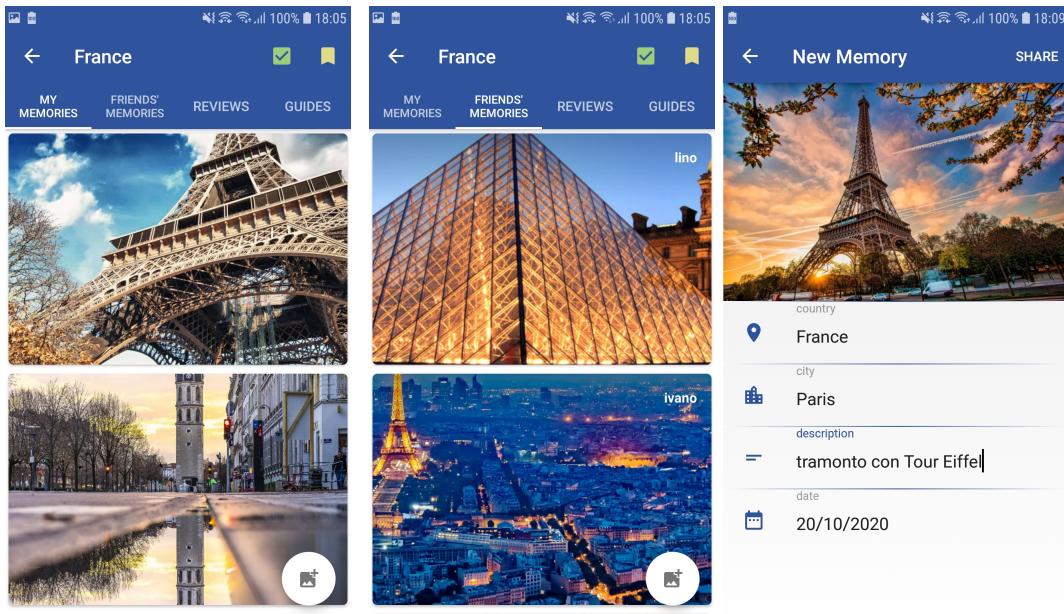


Figura 6.6: Schermata principale

Schermata Principale: Una volta effettuato l'accesso quindi apparirà la schermata principale dell'applicazione in cui è possibile effettuare diverse azioni:

1. Di default viene mostrata la *scratchmap* personale dell'utente (figura 6.6a) con tutti i luoghi visitati o da visitare. Cliccando sul bottone flottante, si può passare alla schermata di pubblicazione di una memory (descritta successivamente)
2. Si può accedere al *Travel Journal* (figura 6.6b), visualizzando quindi tutte le memories pubblicate dall'utente loggato.
3. Si può accedere alla schermata dove ricercare ed effettuare il *follow/unfollow* di altri Travelers (figura 6.6c).



- (a) Visualizzazione delle pro-
prie Memories relative al luogo
go
- (b) Visualizzazione delle Me-
mories degli utenti seguiti
relative al luogo
- (c) Inserimento di una nuova
Memory

Figura 6.7: Schermata di un luogo

Schermata di un luogo: Una volta acceduto un Luogo interagendo con la mappa:

1. Di default vengono mostrate tutte le memories dell'utente relative a quel Luogo (figura 6.7a). È inoltre possibile tramite i Toggle Button in alto a destra impostare il Luogo come visitato o da visitare.
2. Cliccando sul tab Friends' Memories vengono mostrati i ricordi relativi ai following dell'utente (figura 6.7b).
3. Cliccando sui pulsanti in alto a destra è possibile segnare il luogo come "visited" o come "wish"
4. Cliccando sul pulsante flottante si accede alla schermata di Pubblicazione di una Nuova Memory (figura 6.7c) in cui è possibile inserire l'immagine, il paese, la città, una descrizione e la data del nuovo ricordo.
5. Alla fine della pubblicazione, si verrà riportati alla ScratchMap e questa sarà opportunamente aggiornata se necessario.

Si fa notare inoltre che se ci si trova nel tab "Friends' Memories" e uno dei followings in contemporanea pubblica una nuova Memory di quel Luogo; la schermata verrà aggiornata automaticamente senza bisogno di nessun refresh.

Infine, in qualunque momento, cliccando su una Memory è possibile vederne le informazioni nel dettaglio, così come mostrato in figura 6.8.

CAPITOLO 6. DOCUMENTAZIONE DELL'IMPLEMENTAZIONE E MANUALE D'USO

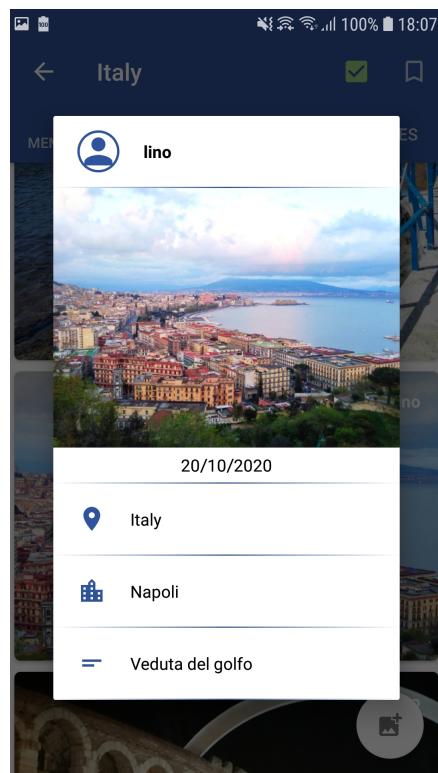


Figura 6.8: Visualizzazione dei dettagli di una Memory

Capitolo 7

Testing

Un sistema non testato è un sistema che non funziona, pertanto in questo elaborato è dedicata un'ampia parte alla descrizione delle tecniche di testing adottate. Dato che i requisiti cambiano di continuo, il design e il codice subiscono numerose modifiche, quindi il testing è stata un'attività eseguita di continuo durante il ciclo di sviluppo, a tutti i livelli: testing di unità, testing di integrazione e testing di accettazione.

Questo capitolo pone particolare attenzione ad una tecnica di *testing automation* tipica per i sistemi Android nota come *Capture and Replay*, utilizzata dal team di sviluppo per la generazione automatica di casi di test per alcune funzionalità di interfaccia.

7.1 Necessità del testing semi-automatico

Il testing è il processo di esecuzione di un codice con determinati parametri. Il fine è quello di scovare dei malfunzionamenti ed in seguito risolverli con il debugging; ma in realtà permette anche di dimostrare indirettamente che il software rispetti specifiche e requisiti, e fornisce una stima della sua affidabilità.

È possibile eseguire un programma con parametri particolari tramite i *Test Cases*. Il problema è che, al crescere della complessità e della grandezza del sistema, il numero dei casi di test aumenta e la loro derivazione manuale può diventare lunga, ripetitiva e onerosa. Per ovviare a questo problema ci sono varie alternative, in questo elaborato approfondiremo una tecnica chiamata "*User Session Based Testing*" in quanto l'ambiente di sviluppo Android Studio mette a disposizione il tool *Record Espresso Test*.

Un testing basato su sessioni utente prevede l'interazione con l'applicazione in esecuzione in maniera sistematica o casuale (in quest'ultimo caso si parla di "*monkey testing*"). Questa tecnica è di tipo black box, si basa sull'analisi di input immessi e output ottenuti interagendo con l'interfaccia dell'applicazione, senza conoscere cosa ci sia effettivamente dietro la user interface.

Le interazioni sono effettuate da utenti che possono approcciare in maniera diversa in base all'esperienza e al loro ruolo: tester interni all'azienda hanno il com-

pito di condurre un'analisi sistematica dell'applicazione per coprire più casi d'uso possibili, mentre utenti esterni potrebbero comportarsi in maniera naturale e utilizzare solo una parte delle funzionalità dell'applicazione, non preoccupandosi delle altre più specifiche che non rientrano nei loro interessi.

Record Espresso Test opera in due fasi:

- **Fase di Capture:** genera semi-automaticamente un caso di test memorizzando tutte le interazioni che l'utente esegue sull'interfaccia dell'applicazione, in seguito crea automaticamente codice eseguibile fedele alle interazioni registrare;
- **Fase di replay:** esegue il caso di test generato nella fase precedente, e riesegue le azioni effettuate in fase di record nello stesso ordine con cui sono state registrate (ma non con le stesse tempistiche). Questa fase serve essenzialmente a verificare che il caso di test corrisponda effettivamente alle azioni memorizzate.

7.2 Fase di Capture

Nella fase di Capture l'applicazione viene eseguita utilizzando *Espresso* (e non il solito "run" di Android Studio), il tool avvierà l'applicazione sull'emulatore (o sul dispositivo reale collegato) e contemporaneamente avvierà anche il sistema di *Capture*. Quest'ultimo è un ambiente controllato che registra in un log le interazioni fatte sull'interfaccia dall'utente, e ciò è visibile durante l'utilizzo dell'applicazione tramite una schermata del tipo di figura 7.1.

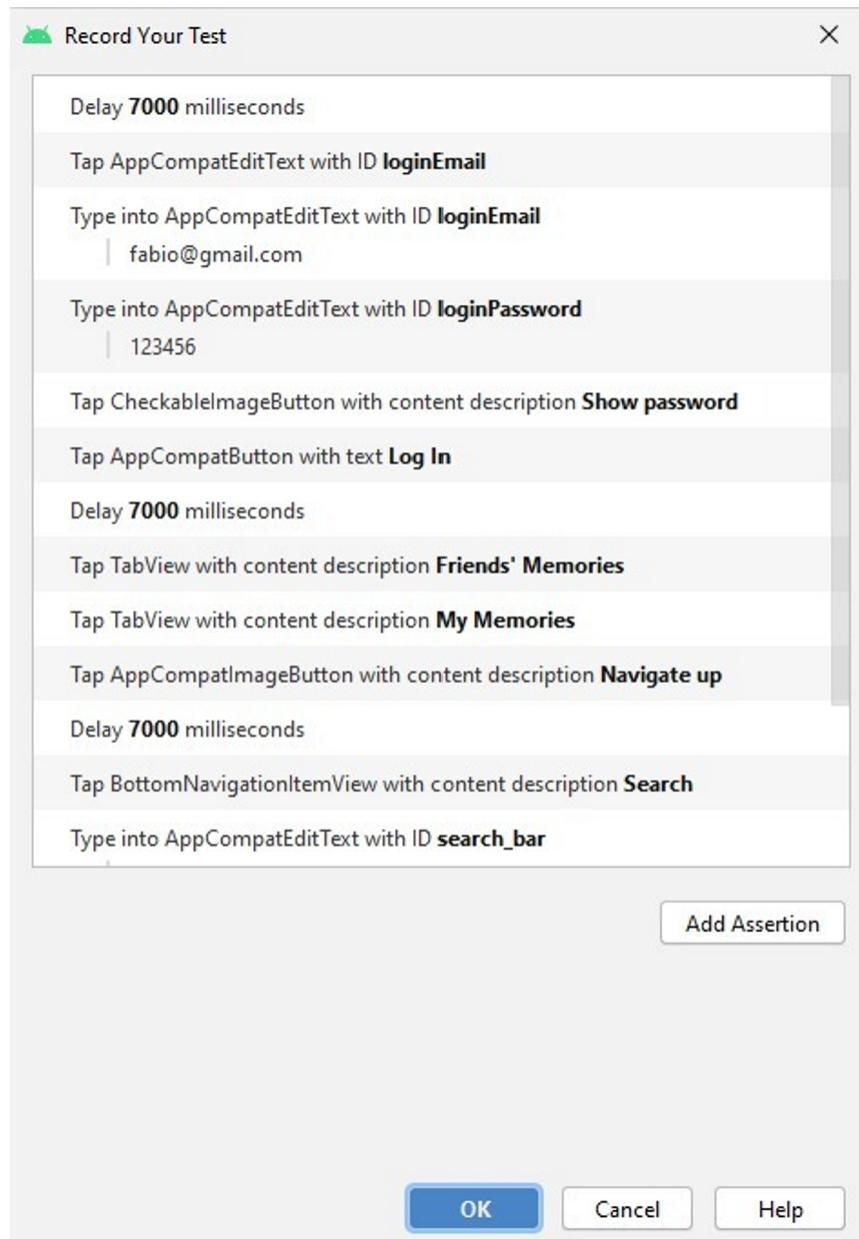


Figura 7.1: Schermata di Capture

Inizialmente questa schermata presenterà il messaggio “*No events recorded yet*”, successivamente questo verrà sostituito dall’elenco delle azioni registrate nel momento in cui esse sono eseguite.

“Add assertions” permette di verificare l’esistenza di determinati elementi dell’interfaccia o il contenuto di essi attraverso delle asserzioni, in cui Espresso crea una vista dell’interfaccia dell’applicazione ed aiuta il tester a riconoscere i vari elementi evidenziandoli con dei riquadri rossi; è possibile selezionarli cliccandoli sulla vista oppure scegliendoli dalla lista del menù *Edit Assertion*.

Il tipo di asserzione varia in base all’elemento selezionato (ad esempio un pulsante può esistere o meno, e quindi l’asserzione sarà *Assert Button with ID*

X exists/does not exist; per le sezioni caratterizzate da testo è inoltre possibile asserire *Text is* e controllare cosa vi sia scritto.

La fase di capture si conclude con la generazione del codice. Qui gli oggetti della GUI coinvolti nel record sono identificati da un ID, spesso anche dal testo con cui vengono mostrati sull'interfaccia, o anche da quali classi derivano (*ChildOfPosition*).

Per riferirsi ad un oggetto di una determinata vista dell'interfaccia si utilizza il metodo *onView()*, questo ritorna un oggetto *ViewInteraction* che permette di interagire con la vista. Per indicare da che tipo di attributo esso sia identificato si utilizza il metodo *with*()*, dove * sta per il tipo di attributo; ad esempio per Id e testo si utilizzano rispettivamente *withId()* e *withText()*.

Inoltre è possibile specificare se un oggetto sia caratterizzato da più di un attributo, e considerarli tutti con il metodo *allOf()*.

Il metodo *perform()* simula le interazioni utente sulla UI, gli argomenti necessari della funzione sono uno o più oggetti *ViewAction*. La classe *ViewAction* presenta una lista di metodi relativi alle azioni più comuni, come il click del mouse con *click()*, scrivere testo in aree apposite con *typeText()*, cancellare il testo con *clearText()*, scorrere la vista con *scrollTo()*. Infine i metodi della classe *ViewAssertions* sono usati per controllare che l'interfaccia rispetti lo stato desiderato in seguito alle azioni effettuate.

7.3 Generazione dei Test Cases

Sono state effettuate determinate interazioni con il fine di coprire più funzionalità possibili dell'applicazione, e quindi simulare al meglio un normale comportamento di un generico utente. Quindi i seguenti casi di test saranno relativi a fase di registrazione, login, following di utenti.

Per quanto riguarda l'interazione con la ScratchMap sono sorti dei problemi: a differenza dei bottoni e della navbar (widget tipici delle applicazioni), gli stati della mappa interattiva non vengono riconosciuti dal sistema di capture seppure sono cliccabili. Infatti Espresso recepisce l'interazione con i pulsanti dell'interfaccia, ma non l'interazione con la Scratchmap, di conseguenza quest'azione non verrà registrata e non sarà presente nel caso di test auto-generato.

Quindi questa azione non verrà replicata nella fase di replay, e tutte le interazioni successive porteranno ad una terminazione negativa del caso di test poiché si aspettano l'interfaccia relativa allo stato specifico (mentre sarà ancora presente l'interfaccia iniziale)

7.3.1 Test Case 1 - Signup

Il primo caso di test generato riguarda la registrazione dell'utente. Sono stati riempiti i campi dell'interfaccia di signup con i valori: espresso, espres-

so@gmail.com, 123456. I valori inseriti sono coerenti sia con la formattazione dell'e-mail e sia con i vincoli imposti per la password.

E' stato testato il solo caso positivo in quanto lo scopo principale di questo caso di test voleva essere la verifica della post-condizione del caso d'uso Signup. Inoltre tutti i casi di test che violano i vincoli di email e password sono stati testati manualmente in fase di implementazione.

Nella **fase di Capture**, Espresso ha correttamente recepito tutte le interazioni dell'utente, e le ha memorizzate come in figura 7.2.

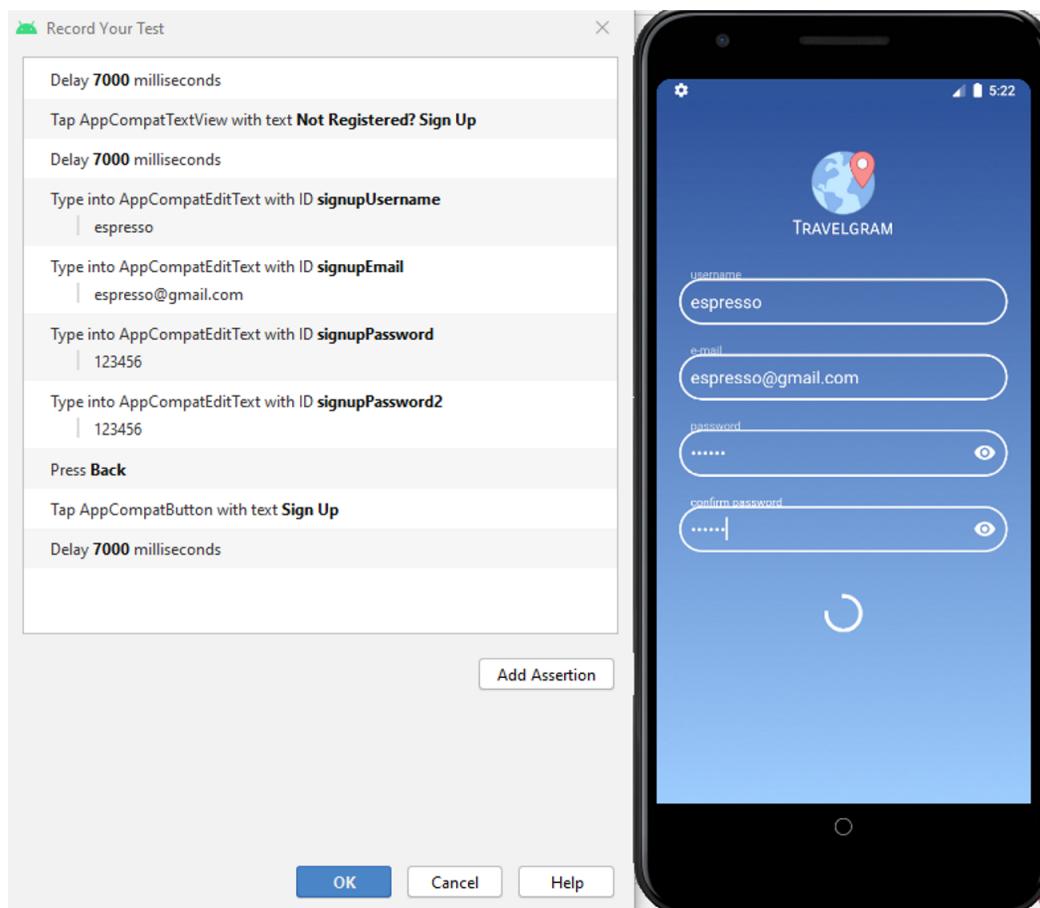


Figura 7.2: Test Case 1 - fase di capture

Il codice del caso di test risultante dalla memorizzazione dei passi appena effettuati è mostrato in figura 7.3.

```

36  ➤ public class Test1_Signup {
37
38      @Rule
39      public ActivityTestRule<LoginActivity> mActivityTestRule = new ActivityTestRule<>(LoginActivity.class);
40
41      @Test
42      public void test1_Signup() {
43
44          try { Thread.sleep( millis: 7000 ); } catch (InterruptedException e) { e.printStackTrace(); }
45
46          ViewInteraction appCompatTextView = onView(withId(R.id.clickSignup));
47          appCompatTextView.perform(scrollTo(), click());
48
49          try { Thread.sleep( millis: 2000 ); } catch (InterruptedException e) { e.printStackTrace(); }
50
51          ViewInteraction appCompatEditText = onView(withId(R.id.signupUsername));
52          appCompatEditText.perform(scrollTo(), replaceText( stringToBeSet: "espresso"), closeSoftKeyboard());
53
54          ViewInteraction appCompatEditText2 = onView(withId(R.id.signupEmail));
55          appCompatEditText2.perform(scrollTo(), replaceText( stringToBeSet: "espresso@gmail.com"), closeSoftKeyboard());
56
57          ViewInteraction appCompatEditText3 = onView(withId(R.id.signupPassword));
58          appCompatEditText3.perform(scrollTo(), replaceText( stringToBeSet: "123456"), closeSoftKeyboard());
59
60          ViewInteraction appCompatEditText4 = onView(withId(R.id.signupPassword2));
61          appCompatEditText4.perform(scrollTo(), replaceText( stringToBeSet: "123456"), closeSoftKeyboard());
62
63          // pressBack();
64
65          ViewInteraction appCompatButton = onView(withId(R.id.signupBtn));
66          appCompatButton.perform(scrollTo(), click());
67
68          try { Thread.sleep( millis: 7000 ); } catch (InterruptedException e) { e.printStackTrace(); }
69      }

```

Figura 7.3: Test Case 1 - codice

Per motivi di leggibilità e formattazione si è provveduto a ritoccare manualmente il codice. Un esempio di come fosse il codice originale e di come questo appaia attualmente è riportato nella figura 7.4.

```

83  ➤ /*
84      ViewInteraction appCompatEditText2 = onView(
85          allOf(withId(R.id.signupEmail),
86              childAtPosition(
87                  childAtPosition(
88                      withId(R.id.signupEmailLayout),
89                      0),
90                  0)));
91      appCompatEditText2.perform(scrollTo(), replaceText("espresso@gmail.com"), closeSoftKeyboard());
92  */
93
94
95      ViewInteraction appCompatEditText2 = onView(withId(R.id.signupEmail));
96      appCompatEditText2.perform(scrollTo(), replaceText( stringToBeSet: "espresso@gmail.com"), closeSoftKeyboard());

```

Figura 7.4: Test Case 1 - semplificazione codice

Le istruzioni eliminate erano ridondanti in quanto contribuivano ad identificare dei widget che già erano ben identificati dal solo ID univoco. Anzi, queste si sono rivelate controproducenti in alcuni casi: l'esecuzione del caso di test ha

portato al crash dell'applicazione, in quanto Espresso ha erroneamente assegnato alle CompatTextView dei valori di "childAtPosition" errati. Ad esempio la seconda CompatTextView era contrassegnata dal valore 0. Successivamente si è testato il caso di test correggendo manualmente il valore 0 in 1, in questo modo il caso di test ha avuto esito positivo. Tuttavia si è scelto di non considerare le istruzioni di "childAtPosition" per il motivo spiegato sopra.

Un'ulteriore correzione manuale è stata quella di eliminare il "press back" generato automaticamente. Questo riportava l'applicazione alla LoginActivity, comportandone il crash. In realtà questa interazione è stata registrata da Espresso dopo aver premuto il pulsante "indietro" (proprio dell'emulatore e non dell'applicazione) poiché la tastiera copriva il pulsante di "Sign up". Correzioni minori prevedono modifiche dei delay per una maggiore fluidità nell'esecuzione dei casi di test.

In **fase di Replay** sono state effettuate correttamente le stesse azioni, e si è arrivati alla post-condizione prevista: il replay ha inserito nuovamente i dati di un utente già registrato in fase di capture, e pertanto un utente con quella determinata e-mail esisteva già. Il caso d'uso è terminato con un errore da parte del sistema visualizzabile dall'applicazione: *"The email address is already in use by another account"* (figura 7.5).

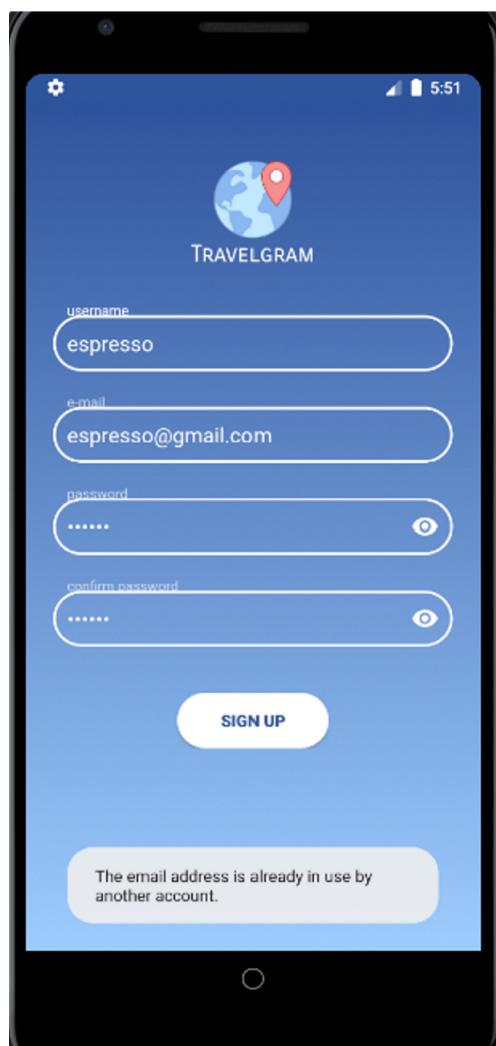


Figura 7.5: Test Case 1 - messaggio di errore

7.3.2 Test Case 2 - Login

In questo secondo caso di test si vuole verificare il corretto funzionamento del login. Ad effettuare l'autenticazione sarà l'utente appena creato, quindi i parametri inseriti sono "espresso@gmail.com" e "123456".

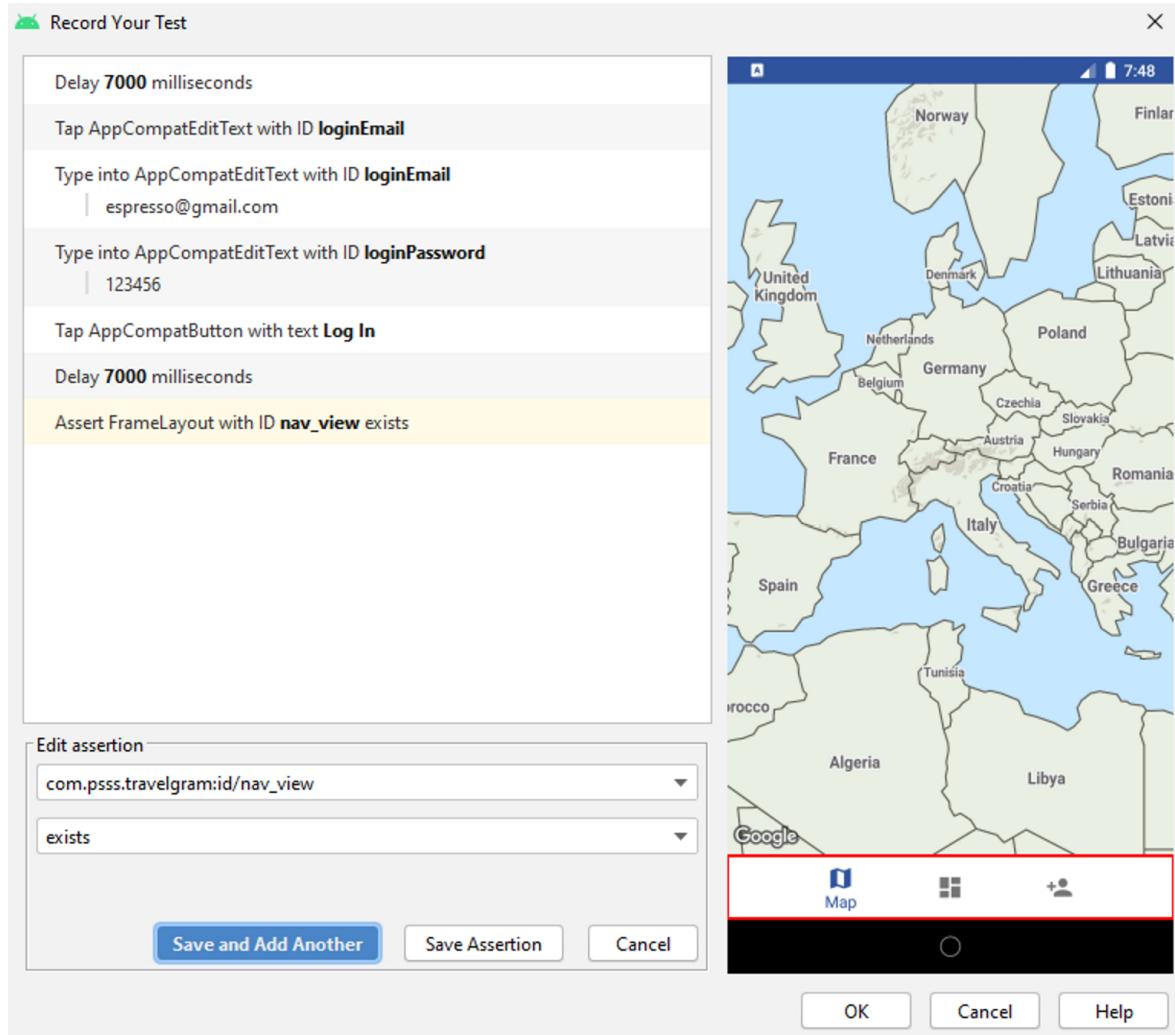


Figura 7.6: Test Case 2 - fase di capture

Come si può notare dalla lista di interazioni all'interno del log (figura 7.6), si è voluto testare il funzionamento delle asserzioni: se la navbar "nav_view" esiste, allora il login è andato a buon fine, poiché si può accedere a questa schermata contenente la navbar solo se si è un Traveler già registrato ed autenticato.

Dalla figura 7.7 si può notare che la fase di replay ha avuto successo, quindi è sicuramente risultato "true" il check sulla navbar indicata nell'asserzione (elemento riquadrato in rosso dallo stesso tool Espresso).

```

app ✘ Test2_Login ✘
Tests passed: 1 of 1 test – 42 s 224 ms

10/16 21:59:13: Launching 'Test2_Login' on Pixel 3a API 27.
Running tests

$ adb shell am instrument -w -r -e debug false -e class 'com.psss.travelgram.view.activity.Test2_Login' com.psss.travelgram.test/androidx.test.runner.AndroidJUnitRunner
Connected to process 9758 on device 'emulator-5554'.

Started running tests
Tests ran to completion.

```

Figura 7.7: Test Case 2 - fase di replay

Nel caso in cui non fosse stata trovata la navbar, il test avrebbe riportato esito negativo. Ciò è successo in un primo tentativo di replay, in quanto Espresso ha erroneamente impostato il valore di *childAtPosition*, motivo per cui anche questa volta si è intervenuti manualmente con la correzione della porzione di codice. L'esito negativo del test è riportato in figura 7.8, con esplicito riferimento al non aver trovato la navbar.

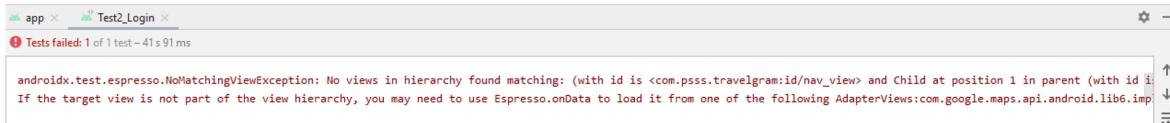


Figura 7.8: Test Case 2 - errore

7.3.3 Test Case 3 - Following

Per poter eseguire questo caso d'uso, la pre-condizione è quella di essere autenticati, motivo per cui la prima parte dell'attuale caso di test richiama parzialmente le istruzioni del caso di test precedente. La seconda parte invece prevede l'interazione con la navbar dell'interfaccia principale, la ricerca di un determinato utente esistente, il click sul bottone "Follow". La fase di capture è mostrata in figura 7.9.

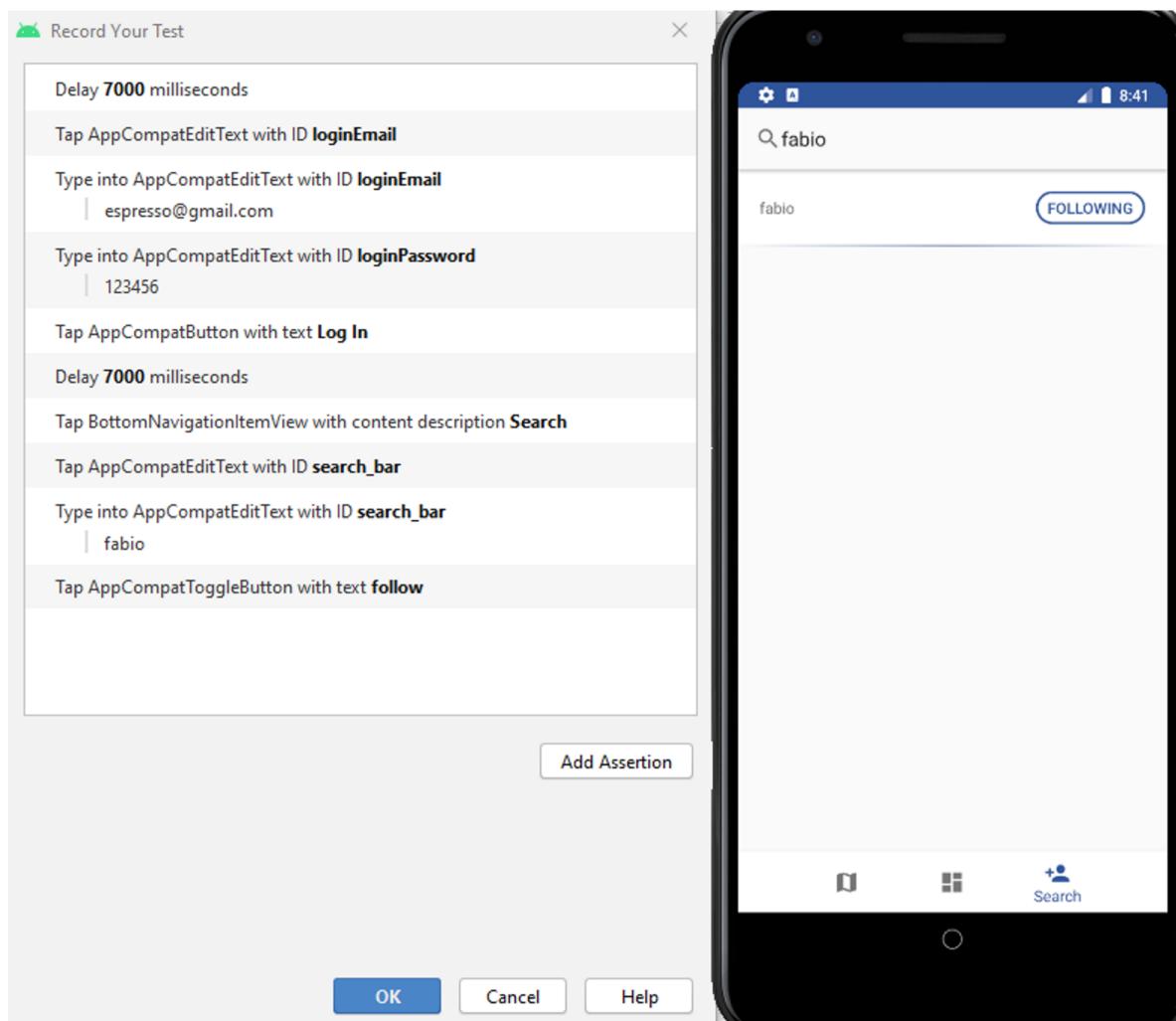


Figura 7.9: Test Case 3 - fase di capture

Bibliografia

- [1] **Travelgram** **Preview:** <https://www.youtube.com/watch?v=QBkgVoCjopg>
- [2] **GitHub Repository:** https://github.com/fabiom95/ProgettoPSSS_Travelgram
- [3] **Documentazione Android:** <https://developer.android.com/guide>
- [4] **Documentazione Firebase:** <https://firebase.google.com/docs/android/setup>
- [5] **Documentazione Maps SDK:** <https://developers.google.com/maps/documentation/android-sdk/map>
- [6] **Firebase Patterns:** <https://firebase.googleblog.com/2013/03/where-does-firebase-fit-in-your-app.html>
- [7] **Architettura MVVM:** <https://it.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel>
- [8] **Architettura raccomandata da Android Jetpack:** <https://developer.android.com/jetpack/guide>
- [9] **Firestore pricing:** <https://firebase.google.com/docs/firestore/pricing>