POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Integrazione di Dati Geografici in un'Applicazione Java per Smartphone

Relatore: Prof. Marco Colombetti

Tesi di Laurea di:
Matteo Vergani, matricola 837098

Anno Accademico 2016-2017

Alla mia famiglia
e al motore del mio cuore,
Micol.

Indice

Lista delle figure

Ringraziamenti

Ringrazio innanzitutto il professor *Marco Colombetti*, del *Politecnico di Milano*, per avermi dato la possibilità di svolgere questa tesi e per la tempestiva assistenza fornitami durante la sua stesura.

Ringrazio i miei genitori, *Cesare* e *Tiziana*, e mia sorella *Alessia*, che mi hanno supportato (e sopportato) durante tutti questi anni di studio e non mi hanno fatto mancare mai nulla.

Ringrazio la mia ragazza *Micol*, che mi è sempre stata accanto, nei momenti belli e quelli di sconforto, sempre incoraggiandomi.

Ringrazio mia nonna *Angela*, che mi ha insegnato e fatto dono di preziosi consigli sin da quando ero piccolo.

Ringrazio mia zia *Anna*, che mi ha cresciuto e accudito, contribuendo anche lei alla mia formazione e tutto il resto della mia famiglia.

Infine ringrazio tutti i miei amici, in (non) rigoroso ordine alfabetico, *Chris*, *Fabio*, *Francesca*, *Gabriele*, *Lara*, *Magni*, *Marianna*, *Mattia*, *Piva*, *Putelli*, *Sigio*, *Varalta* e *Veronelli*, per i bellissimi momenti passati insieme.

Sommario

Nel mondo moderno, grazie all'immenso sviluppo delle tecnologie informatiche, l'acquisizione, la memorizzazione e la condivisione dei dati è sempre più rapida e semplice. In rete infatti, sono disponibili immense quantità di informazioni, memorizzate in formati totalmente differenti fra loro. Tra le tante rappresentazioni, per esempio, vi sono dati in formato tabellare (database) oppure dati sotto-forma di grafi (nodi collegati tra loro), chiamati *Linked Data*.

Lo scopo della tesi è quello di mostrare la fattibilità di integrare dati provenienti da *database* classici e dati in formato *Linked Data* in una applicazione per *smartphone*.

I dati in formato tabellare utilizzati provengono dai *dataset* che la *Regione Lombardia* pubblica in rete nell'ambito delle direttive europee e nazionali sulla informatizzazione e trasparenza del settore pubblico. I *Linked Data* invece sono ottenuti da *DBpedia*, un dataset *RDF* contenente informazioni estratte da *Wikipedia*, e *GeoNames*, un database di tipo geografico liberamente modificabile.

Il risultato è una "guida al territorio" della Lombardia, sotto-forma di applicazione *Android*, che consente agli utenti di trovare informazioni sui comuni e scoprire o approfondire la conoscenza del territorio dal punto di vista della cultura, dell'ambiente, del turismo e dei personaggi legati ad esso.

Abstract

In modern world, thanks to the immense developing of information technologies, acquiring, storing and sharing data has become faster and easier than ever. In fact, on the internet, there is a huge amount of information available, stored in totally different formats. Among the various representations, there exist, for instance, tabular data (called databases) or graph-structured data (interconnected nodes), called *Linked Data*.

The aim of the thesis is showing the feasibility of integrating classical databases with *Linked Data*, in a *mobile* application.

The tabular data are fetched from the *datasets* of *Regione Lombardia*. The region openly publishes them online, following European and national guidelines on digitalization and transparency of public administrations. Instead, *Linked Data* are fetched from *DBpedia*, an *RDF* dataset containing information extracted from *Wikipedia*, and *GeoNames*, an open geographical database.

The result is a "local guide" to the territory of Lombardy, in the form of an *Android* application. The app allows users to find information about comuni and discovering or deepening their knowledge about the territory in terms of culture, environment, tourism and people bound to it.

Capitolo 1

Introduzione

In questo capitolo introduttivo verranno spiegati gli obiettivi che la tesi si pone, le motivazioni che hanno spinto a realizzarla e una breve descrizione del risultato ottenuto.

1.1 Obiettivi e Motivazioni

Al giorno d'oggi internet mette a disposizione degli utenti una quantità di informazioni straordinaria, mai nemmeno immaginata nella storia del mondo. Tali informazioni spesso sono strutturate in modo da essere presentate o utilizzate da soli esseri umani, che sono in grado di comprenderne il significato ed sfruttarle a loro vantaggio.

Le macchine ancora non possiedono questa capacità, quindi necessitano di assistenza per interpretare i dati ed estrarne informazioni. Il *web semantico* si pone come obiettivo realizzare proprio questo, fornire ai dati un significato per poter eseguire *ragionamenti*.

RDF è uno degli standard più utilizzati in questo ambito, poiché è adottato per rappresentare i *Linked Data*, dati connessi tra loro tramite relazioni chiamate *predicati*. A tali predicati è possibile associare dei significati, creando le basi per poter eseguire dei ragionamenti.

L'obiettivo principale della tesi è mostrare la possibilità di integrare dei dati in formato *RDF*, con dati provenienti da altre fonti di diversa natura. Per dimostrarne la fattibilità si è deciso di realizzare un'applicazione concreta in grado di assolvere questo compito.

L'idea è stata quella di realizzare una "guida al territorio" della Lombardia, un'applicazione che possa risultare realmente utile nella vita di tutti i

giorni. Lo scopo è quello di promuovere la conoscenza delle peculiarità del territorio come l'ambiente, la cultura, le strutture alberghiere e le persone legate a tale territorio.

La scelta di svilupparla per *Android* è stata naturale conseguenza del tipo di applicazione che si è deciso di realizzare, poiché l'ambito si presta molto alla piattaforma mobile e consente inoltre di renderla fruibile da un numero maggiore di persone.

Una volta fissate le linee guida si è passati alla scelta delle fonti di informazioni. Quelle prese in considerazione sono:

- I dati della *Regione Lombardia*, pubblicati online in formato tabulare, accessibili tramite delle apposite *API* (maggiori informazioni nel prossimo capitolo). I dataset disponibili sono ufficiali, costantemente aggiornati e coprono svariate aree di competenza.

- *DBpedia*, probabilmente il più grande dataset mondiale di dati in formato *RDF*. Estrae informazioni strutturate da *Wikipedia*, quindi contiene dati riguardanti qualsiasi ambito.

- *GeoNames*, un database geografico open, in cui ogni utente registrato può contribuire. Contiene informazioni su entità geografiche e località in tutto il mondo, memorizzate in formato *RDF*.

Le motivazioni che hanno portato a scegliere queste fonti sono varie: innanzitutto, i dati della *Regione Lombardia* sono necessari in un'applicazione sul territorio Lombardo. *DBpedia* invece è un punto di riferimento nell'ambito dei dati in formato *RDF* ed appoggiandosi a *Wikipedia* è anche una certezza nell'ambito della conoscenza in generale. *GeoNames* offre un servizio differente alle precedenti fonti e consente di reperire dati che spesso non sono disponibili altrove.

L'ultima motivazione, e forse la più importante, è che tali fonti di dati si compensano. I dati della Regione sono di tipo istituzionale, quindi conferiscono affidabilità e autorità alle informazioni che forniscono. Per contro però, i dati che contengono sono di natura molto pratica: un indirizzo, un prezzo, un sito. Qui subentra *DBpedia*: essendo fondamentalmente un'enciclopedia in formato *RDF*, consente di aggiungere descrizioni, immagini, collegamenti ad altre risorse, arricchendo di informazioni la risorsa in questione. *GeoNames* completa il quadro fornendo entità che non sono presenti nelle altre fonti e un approccio facilitato alle query geografiche.

1.2 Struttura del Documento

La tesi è strutturata nel modo seguente:

- Nel **capitolo 2** vengono mostrati i requisiti che l'applicazione deve possedere, l'architettura generale scelta e le tecnologie utilizzate. Il tutto corredato da motivazioni e limitazioni che hanno portato a tali scelte.
- Nel **capitolo 3**, tramite spiegazioni e diagrammi, viene trattata in dettaglio l'architettura delle singole componenti del sistema ed eventualmente le motivazioni che hanno portato a scelte specifiche.
- Nel **capitolo 4** vengono mostrati degli esempi di utilizzo della applicazione realizzata, mostrando nel dettaglio il flusso di esecuzione e l'interazione tra le varie componenti.
- Nel **capitolo 5** vengono tratte le conclusioni sul lavoro svolto ed indicate le limitazioni affrontate e i possibili sviluppi futuri.

Capitolo 2

Architettura del Sistema e Scelte Tecnologiche

Nel presente capitolo verranno mostrati i requisiti generali, le scelte architetturali e le tecnologie adottate per la realizzazione dell'applicazione *DiscoverLombardia*. Verranno inoltre fornite le motivazioni che hanno portato a tali scelte ed eventuali pro e contro delle soluzioni implementate.

2.1 Requisiti

L'obiettivo della tesi, come già detto, è quello di mostrare concretamente la possibilità di integrare dati provenienti da fonti eterogenee (come *RDF* e *DBMS* classici) in un'applicazione *Android*.

L'applicazione deve quindi essere fruibile su smartphone, sottostando ai limiti architetturali imposti dalla piattaforma ed essere in grado di mascherare totalmente all'utente la diversa provenienza delle informazioni. Per ottenere questo risultato, i dati devono essere integrati all'interno della stessa pagina, senza presentare discontinuità grafiche o logiche tra una tipologia e l'altra.

L'applicazione realizzata si chiama *DiscoverLombardia* e raccoglie dati di vario genere su tutti i comuni della Lombardia: informazioni di carattere generale, punti di interesse di tipo ambientale, alberghi, musei e personaggi legati al comune. L'utente è quindi in grado di consultare tutte queste informazioni in un formato semplice e moderno come un'app per smartphone e scoprire di più sul territorio della Lombardia.

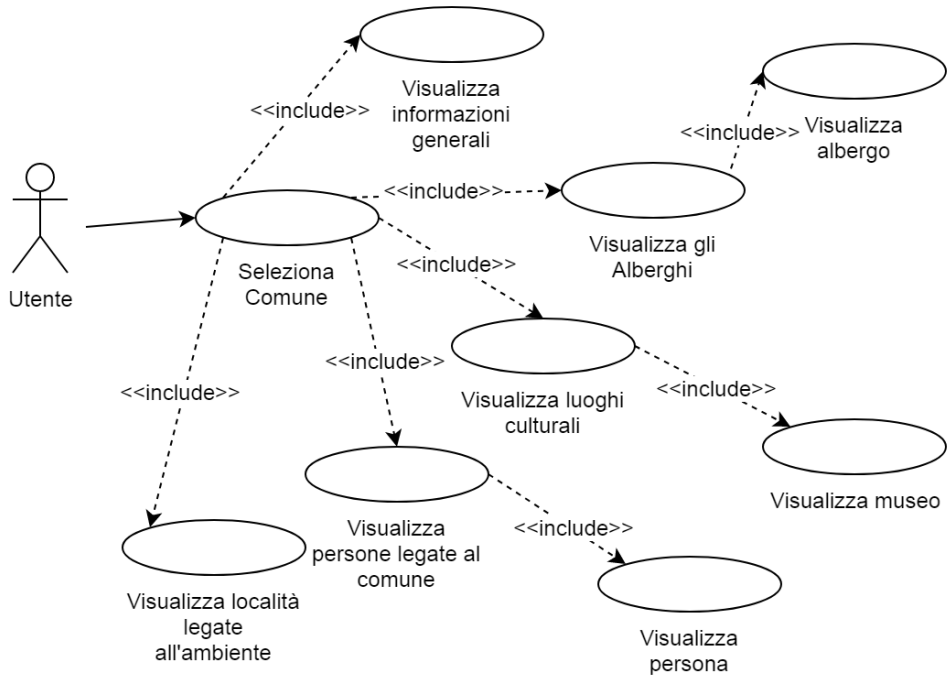


Figure 2.1: Diagramma use-case dell'applicazione DiscoverLombardia.

Figura 2.3 descrive, tramite un diagramma *use-case*, le funzionalità fornite dal sistema e come l'utente può interagire con l'applicazione *DiscoverLombardia*.

Com'è possibile notare, una volta selezionato un comune si può accedere a tutte le informazioni citate in precedenza: gli alberghi presenti, la lista delle persone legate al comune, i musei o le bellezze ambientali nelle vicinanze oltre che delle informazioni generali. Dalle sezioni dedicate ai musei, gli alberghi e le persone è inoltre possibile esplorare in dettaglio le singole entità.

L'applicazione consente anche all'utente di navigare al di fuori di essa:

- i link a pagine web vengono aperti con il browser predefinito del sistema
- gli indirizzi email consentono di aprire il client di posta predefinito del sistema e iniziare a scrivere una email all'indirizzo selezionato
- i numeri di telefono consentono di aprire il dialer dello smartphone ed effettuare la chiamata
- gli indirizzi postali vengono aperti con l'applicazione mappe predefinita del sistema

2.2 Architettura

L'architettura utilizzata si basa sul pattern architetturale del *Model-View-ViewModel*, abbreviato con l'acronimo *MVVM*. Consiste, come mostrato in figura 2.2, nell'identificazione di tre componenti principali: la *View*, il *Model* e il *ViewModel*.

La *View* consiste nell'interfaccia grafica dell'applicazione e delle operazioni ad esse collegate, mentre il *Model* è la componente che gestisce l'accesso ai dati e la *business logic*. Il *ViewModel* invece, è il mediatore tra queste due entità, che si fa carico unicamente di notificare le modifiche del *Model* e ricevere comandi dalla *View* [1].

Le motivazioni che hanno portato a tale scelta sono molteplici. Innanzitutto, sin dalla sua introduzione da parte di *Microsoft* in *WPF* [2, 1], *MVVM* si è consolidato come una delle architetture più utilizzate per la realizzazione di App per smartphone e tablet. I vantaggi principali di questo pattern sono la flessibilità e la coerenza, dati dalla separazione logica delle componenti, in particolare del *Model* (che contiene i dati dell'applicazione) e della *View* (l'interfaccia grafica che l'utente visualizza), che permette di renderle sviluppabili, modificabili e testabili in modo indipendente [1]. Inoltre, come verrà trattato in dettaglio successivamente, le nuove librerie architetturali introdotte da *Android* favoriscono l'utilizzo di tale approccio [3].

Nello specifico, il pattern architetturale scelto, applicato all'applicazione realizzata *DiscoverLombardia*, appare come in figura 2.3. Si possono individuare i seguenti componenti:

- La *View* è rappresentata, come in ogni applicazione *Android*, dalle varie *Activity* e dai *Fragments* contenuti all'interno di esse. Il loro compito è eseguire il *binding* tra l'interfaccia grafica e i dati esposti dal *ViewModel*

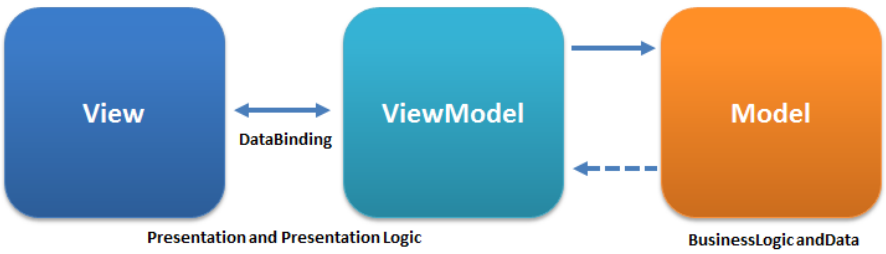


Figure 2.2: Rappresentazione grafica del Model-View-ViewModel

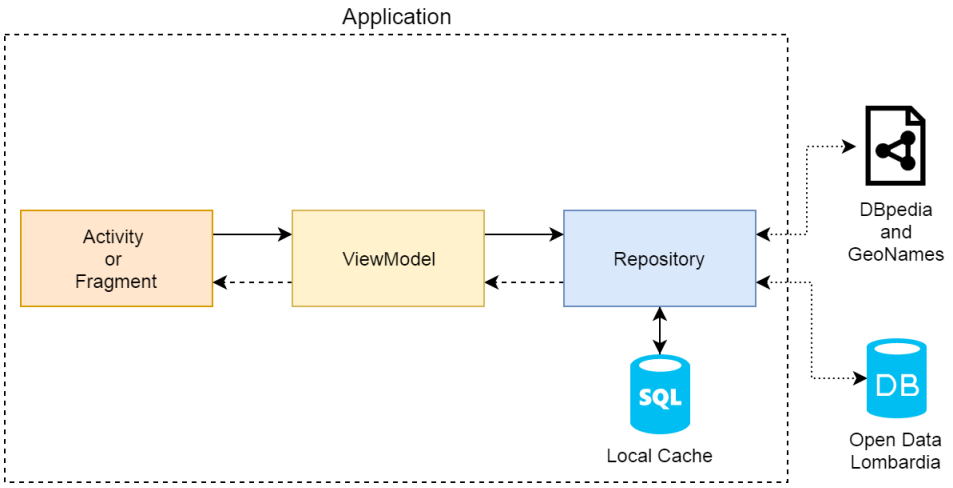


Figure 2.3: Architettura dell'applicazione DiscoverLombardia

ed incorporare logica di presentazione. Per ricevere notifiche all'atto di modifica o ricevimento dei dati le *Activity* osservano il *ViewModel* di loro competenza.

- Il *ViewModel*, rappresentato in *Android* dall'omonima classe [4], si occupa richiedere al *Repository* il caricamento dei dati, di ricevere aggiornamenti su di essi ed eventualmente propagarli alle *Activity*. Nell'applicazione ne sono presenti vari, ognuno dedicato ad un'entità logica del *Model* (es. Comune, Musei ecc.)

- Il Model, rappresentato nell'applicazione dal *Repository*, si occupa di gestire ogni aspetto dei dati: la ricezione di essi da internet, il salvataggio in cache, la lettura e la notifica dei cambiamenti. A causa di questa molteplicità di funzioni, il *Repository* è in realtà un'entità che comprende più oggetti, come è possibile osservare in figura 2.4. Il database rappresenta il *Model* vero e proprio, una copia locale dei dati ricevuti recentemente, mentre i due webservice si occupano di inviare le query ai relativi webserver e ricevere i dati richiesti.

La trattazione più dettagliata di tutte queste componenti verrà affrontata nel Capitolo 3.

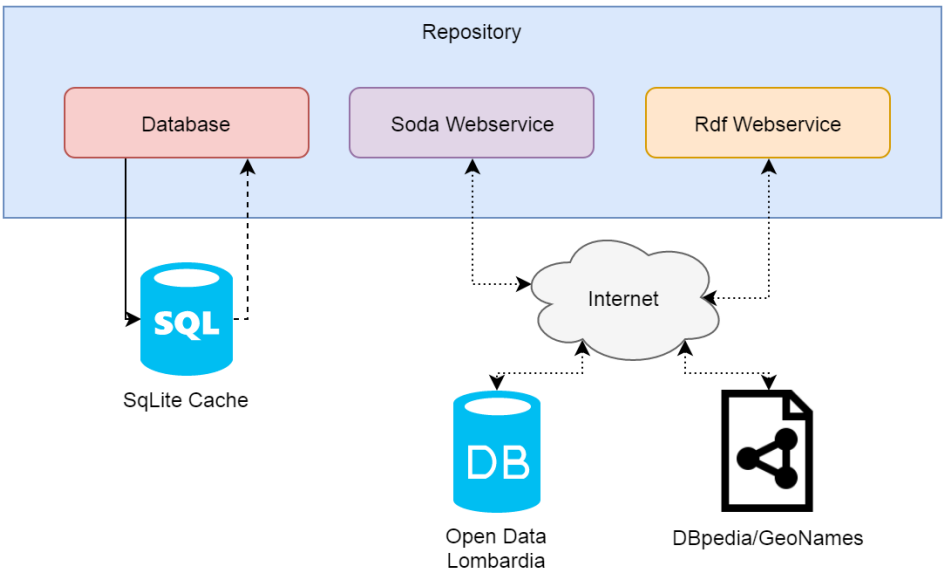


Figure 2.4: Dettaglio architetturale del Repository

2.3 Tecnologie e Risorse

L'applicazione *DiscoverLombardia* si basa sulla piattaforma *Java* ed è stata scritta nell'omonimo linguaggio col supporto dell'*Android Software Development Kit*. Essendo un'applicazione per smartphone *Android*, la scelta del framework era vincolata, mentre per quanto riguarda il linguaggio si è optato per *Java*, poiché è nativamente supportato [5] e attualmente il più utilizzato per lo sviluppo in questo ambiente [6]. In particolare, sono state utilizzate le seguenti librerie:

- le nuove librerie architetturali di *Android* [7], che forniscono utili strumenti a supporto dell'architettura scelta e sono fortemente consigliate dagli sviluppatori della piattaforma [3]. In particolare *LiveData* e *ViewModel* per quanto riguarda la notifica delle modifiche ai dati, mentre *Room* per la gestione della cache.
- *GeoJson*, che fornisce degli oggetti rappresentativi di entità geografiche (es. posizione, confini ecc.) [8].
- *Apache Commons Text* che fornisce alcune utilities per la manipolazione di stringhe di testo [9].
- *Glide*, un'efficace libreria per la gestione di risorse multimediali [10].

Per quanto riguarda le fonti da cui attingere ai dati, ne sono state utilizzate tre:

- *Open Data Lombardia*, che fornisce dati ufficiali e periodicamente aggiornati su varie risorse del territorio lombardo (es. Comuni) in classico formato di database relazionale (*DBMS*) [11]. Per l'accesso a tali dati, come suggerito dal sito della Regione Lombardia, sono state utilizzate le *Socrata Open Data API* (*SODA*) specifiche per la piattaforma *Android* [12].

- *DBpedia*, un database che contiene dati strutturati e collegati (*Linked Data*) di ogni genere estratti da *Wikipedia* e li rende disponibili sul *World Wide Web*. Utilizza internamente *RDF* come modello per rappresentare i dati e fa parte dell'universo del *Web Semantico* [13]. La richiesta di tali dati è effettuata tramite query *SPARQL* ai web server di *DBpedia* [14] e *DBpedia Italia* [15]. Per effettuare le query sono state utilizzate classi realizzate *ad hoc*. Non è stato infatti possibile utilizzare le librerie *ARQ* di *Jena* in ambito *Android* a causa di alcune dipendenze da librerie non supportate dalla macchina virtuale su cui *Android* si basa, *Dalvik*.

- *GeoNames*, un database che contiene informazioni relative ad entità geografiche (es. luoghi, divisioni amministrative ecc.) e le rende disponibili online, anche in formato *linked data*. Per accedere ai dati sono state utilizzate le API per linguaggio Java disponibili sul sito [16], che consentono un accesso più semplificato al webservice.

Capitolo 3

Implementazione

In questo capitolo verranno trattate in maggior dettaglio le varie componenti dell'applicazione, mostrandone i dettagli implementativi e motivandone, quando necessario, le scelte.

3.1 Activity e Fragment

In *Android* ogni applicazione non è un'entità atomica, ma si compone di diverse parti, spesso connesse tra loro, chiamate *Activity* o attività. Le *Activity* sono indipendenti e possono essere avviate dall'interno della stessa app oppure dall'esterno in modo totalmente trasparente e rappresentano una singola attività che l'utente può svolgere [17].

Tipicamente, ad ogni *Activity* è associata un'interfaccia grafica propria (modellata in un file *XML*) di cui gestiscono le interazioni con l'utente e implementano la logica di presentazione. Ogni applicazione *Android* contrassegna un'*Activity* come quella principale, chiamata *main Activity*. Le attività possono contenere uno o più *Fragment*.

I *Fragment* rappresentano una porzione di interfaccia grafica all'interno di un'*Activity*. Possono essere combinati tra loro oppure essere utilizzati in diverse attività e possiedono un ciclo di vita proprio. Intuitivamente si può considerarli come delle *"sotto-Activity"* [18].

Nell'applicazione *DiscoverLombardia*, ogni *Activity* deriva dalla classe *AppCompatActivity* per sfruttare le funzionalità dell'*ActionBar* contenuta nelle librerie di supporto di *Android*. Per quanto riguarda i *Fragment*, ognuno di essi deriva dalla canonica classe base *Fragment*.

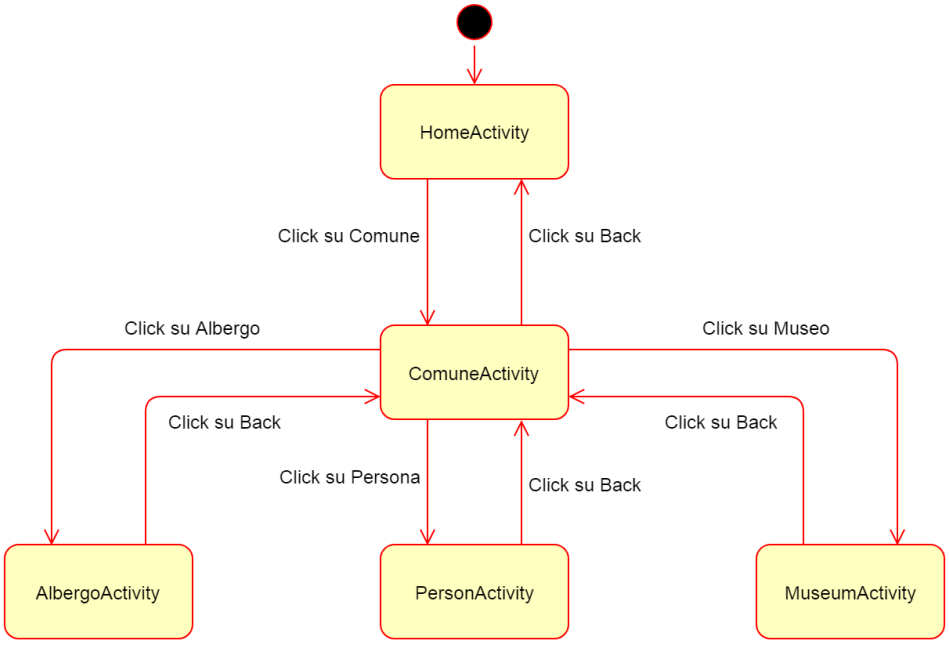


Figure 3.1: Diagramma a stati delle Activity

In figura 3.1 è rappresentata l'interazione ad alto livello tra le diverse activity all'interno dell'app.

3.1.1 HomeActivity

HomeActivity è la *main Activity* di *DiscoverLombardia*, ovvero il punto di ingresso standard dell'applicazione quando viene lanciata tramite *tap* sull'icona. In quanto tale, oltre alle funzionalità spiegate in seguito a questo capitoletto, l'attività ha il compito di lanciare l'inizializzazione delle componenti fondamentali al funzionamento di tutta l'applicazione.

L'interfaccia grafica presenta una *TextView* con autocompletamento in cui è possibile digitare il nome del comune che si vuole cercare. Mentre vengono inseriti i primi caratteri del comune, l'applicazione assiste l'utente mostrando in tempo reale le possibili scelte tra i comuni che contengono le lettere inseriti (Figura 3.2). In questo modo è possibile selezionare il comune senza doverlo scrivere interamente, favorendo l'usabilità dell'applicazione.

Cliccando su un elemento della lista, viene eseguita la funzione *onComuneClick*, che trasferisce la navigazione alla *ComuneActivity*, dove è possibile visualizzare le informazioni riguardanti il comune selezionato.

Figure 3.2: Screenshot della HomeActivity

Dal punto di vista implementativo, l'*Activity* è strutturata come mostrato in figura 3.3. Come si può notare, integra al suo interno un *ViewModel*, nello specifico quello riguardante i comuni (*ComuniViewModel*). Al momento della creazione, l'attività comanda a tale *ViewModel* di reperire i dati (la lista dei comuni) e inizia ad osservarlo per ricevere aggiornamenti. Nel momento in cui i nomi dei comuni diventano disponibili, vengono trasferiti al componente dell'interfaccia grafica di competenza, per essere mostrati all'utente al momento della digitazione. Questa operazione viene chiamata *binding*.

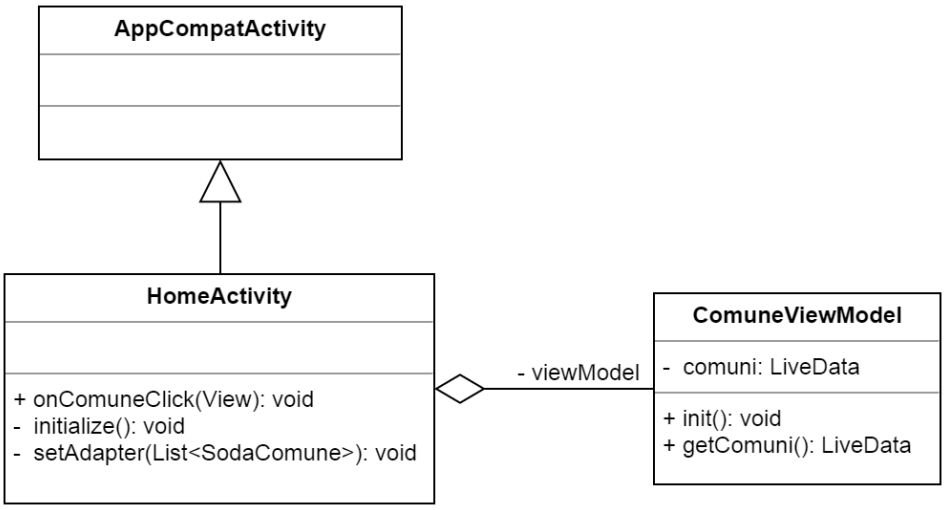


Figure 3.3: Diagramma UML della HomeActivity

3.1.2 ComuneActivity

ComuneActivity è l'attività fulcro di *DiscoverLombardia*, sia dal punto di vista dell'interazione con l'utente, sia dal punto di vista dell'integrazione dei dati. Da essa è infatti possibile accedere a tutte le altre *Activity* della applicazione e, al suo interno, ingloba informazioni provenienti da tutte le sorgenti di dati considerate.

È suddivisa in diversi *Fragment*, ognuno contenente diverse informazioni riguardanti il comune, come mostrato in figura 3.4. Il *Fragment* caricato di default quando viene lanciata l'attività è *ComuneOverview*, che contiene informazioni di carattere generale sul comune (l'argomento verrà trattato in dettaglio nell'apposita sezione). Per navigare da un *Fragment* all'altro l'*Activity* comprende un *drawer menu* (Figura 3.5). Quando si clicca su un elemento del menu, viene chiamata la funzione *onNavigationItemSelected()*, che identifica il tipo di elemento cliccato e carica il relativo *Fragment* (funzione *replaceFragment()*).

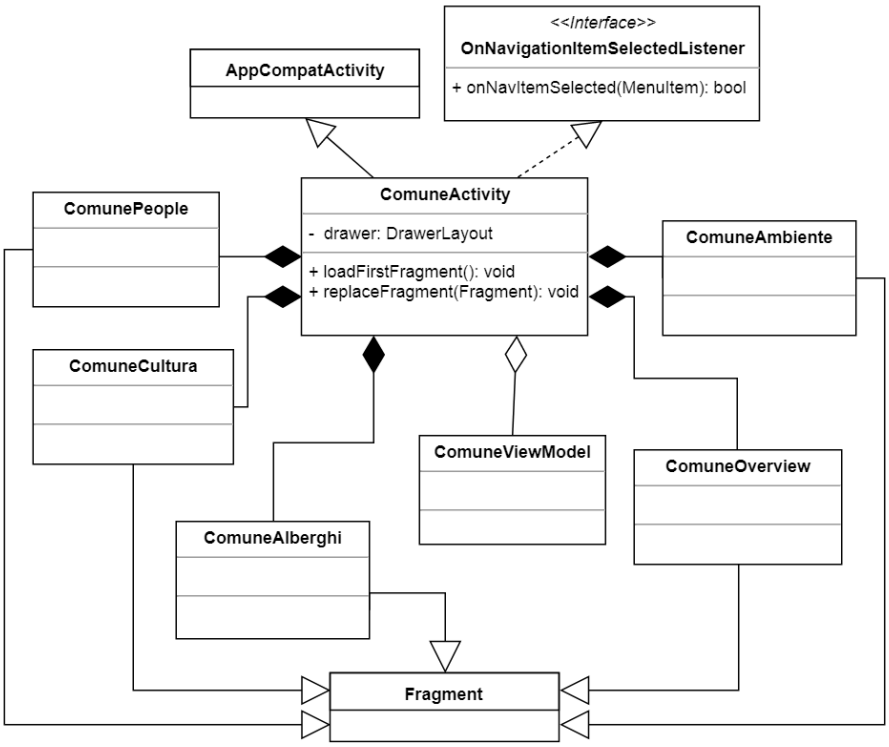


Figure 3.4: Diagramma UML della ComuneActivity

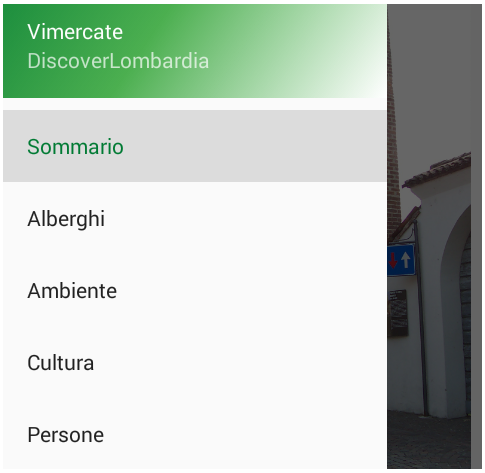


Figure 3.5: Dettaglio della schermata con il drawer menu aperto

L'*Activity* si occupa anche di lanciare l'inizializzazione del *ComuneViewModel*, sebbene non si occuperà di gestire i dati ricevuti, poiché delega quest'azione al *Fragment ComuneOverview*. Verranno ora mostrati in maggiore dettaglio i singoli *Fragment*.

ComuneOverview

ComuneOverview è il fragment principale di *ComuneActivity* e fornisce informazioni di carattere generale sul comune considerato, come per esempio una breve descrizione, i contatti o un'immagine. I dati provengono da entrambe le fonti utilizzate in questo progetto: i contatti, per esempio, vengono prelevati dai dataset della Regione Lombardia, mentre l'immagine e la descrizione vengono estratti da *DBpedia*. In figura 3.6 è possibile vedere la rappresentazione in *UML* del *Fragment*.

Appena caricato, il *Fragment* osserva entrambi i *LiveData* presenti nel *ComuneViewModel* per ricevere immediatamente notifiche quando i dati sono disponibili. Alla ricezione delle informazioni, viene eseguita la funzione *bindData()* che effettua il binding, ovvero passa i dati ai componenti dell'interfaccia grafica che li visualizzeranno sullo schermo.

LiveData è uno dei nuovi componenti architetturali introdotti da *Android*, funge da mediatore tra il *Model* e la *View* ed è *lifecycle-aware*, ovvero è in grado di gestire autonomamente le notifiche in base allo stato in cui sono i suoi osservatori. Se questi sono attivi lancia la notifica, altrimenti mantiene l'evento in coda fino a quando l'*Activity* non sarà riesumata [19].

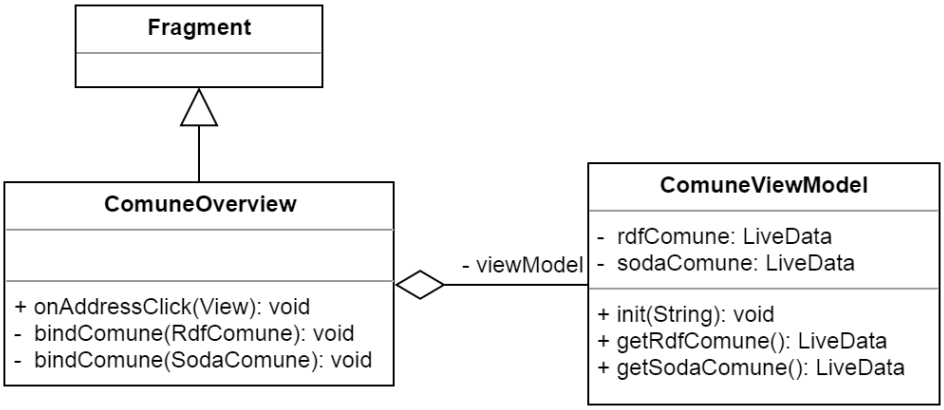


Figure 3.6: Diagramma UML del Fragment ComuneOverview

Questo approccio consente di sgravare il programmatore di tale gestione ed evitare quindi comuni errori.

Com'è possibile notare in figura 3.7, all'utente i dati appaiono perfettamente integrati all'interno della stessa schermata, mentre a livello implementativo sono contenuti in due oggetti separati racchiusi nello stesso *ViewModel*. Questa scelta è stata adottata per rendere l'operazione di raccolta dei dati il più possibile asincrona, favorendo la velocità dell'applicazione e la disponibilità dei dati anche se una fonte risulta irraggiungibile.

L'interfaccia grafica del *Fragment* è per buona parte interattiva: è infatti

Figure 3.7: Screenshot del Fragment ComuneOverview: si possono notare parte dell'immagine, la descrizione e alcuni contatti.

possibile cliccare su elementi come il sito web o l'indirizzo del comune e venire reindirizzati rispettivamente alla pagina internet, oppure alla data posizione sulla mappa.

ComuneAlberghi, ComuneCultura e ComunePeople

I *Fragment ComuneAlberghi*, *ComuneCultura*e *ComunePeople* verranno trattati in parallelo poiché presentano un comportamento e un'interfaccia simile tra loro. Com'è possibile notare in figura 3.8, contengono tutti un riferimento a *ComuneViewModel*, che fornisce i dati generali del comune utili per l'inizializzazione dei relativi *ViewModel*.

Appena un *Fragment* viene caricato, controlla che i dati siano già disponibili nel *ComuneViewModel*: se così non fosse osserva il *ViewModel* per ricevere una notifica non appena lo diventano. Una volta ricevuti, procede all'inizializzazione del proprio *ViewModel* e, come in precedenza, lo osserva, in attesa di aggiornamenti. Al momento della ricezione delle informazioni, viene eseguito il binding e quindi visualizzata a schermo la lista degli elementi richiesti: gli alberghi per *ComuneAlberghi*, i musei per *ComuneCultura* e le persone per *ComunePeople*.

Dal punto di vista grafico, i *Fragment* appaiono come in figura 3.9: una

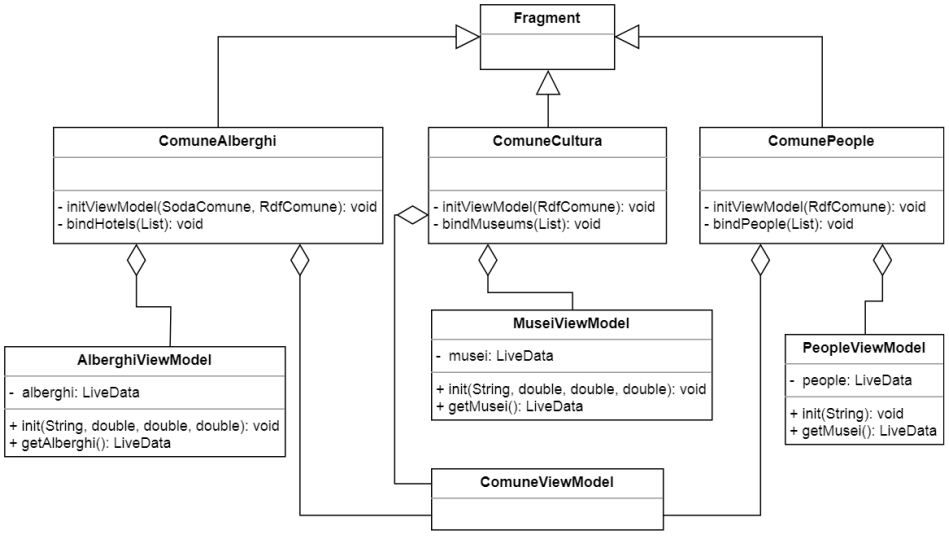


Figure 3.8: Diagramma UML dei Fragment ComuneAlberghi, ComuneCultura e ComunePeople.

Figure 3.9: Screenshot del Fragment ComuneCultura.

lista di elementi su cui è possibile cliccare e, nel caso di alberghi e musei, la distanza dal centro geografico del comune. Quando si esegue il click, la navigazione viene indirizzata alla relativa *Activity* (*AlbergoActivity*, *MuseumActivity* o *PersonActivity*), che mostra in maggior dettaglio le informazioni riguardanti l'elemento su cui si è cliccato.

Le sorgenti di dati sfruttate variano da *Fragment* a *Fragment*:

- *ComuneAlberghi* ottiene la lista degli alberghi dagli *Open Data Lombardia*, utilizzando per la *query* sia il nome del comune prelevato dalla stessa fonte, che le coordinate geografiche ottenute da *DBpedia*.

- *ComuneCultura* ottiene la lista dei musei dagli *Open Data Lombardia* utilizzando per la *query* solo dati di *DBpedia*.

- *ComunePeople* ottiene la lista di persone da *DBpedia* utilizzando dati del comune estrapolati da *DBpedia* stessa.

ComuneAmbiente

Il *Fragment ComuneAmbiente* contiene informazioni relative all'ambiente (es. parchi, fiumi, montagne ecc.)e al paesaggio (es. statue, fontane, rovine ecc.) che circonda il comune selezionato. All'utente appare come una lista di località, con relativa distanza dal centro geografico del comune (figura 3.10).

Cliccando su uno degli elementi della lista, a differenza dei *Fragment* precedentemente trattati, viene lanciata l'applicazione mappa presente sul cellulare con un marker in corrispondenza della della località.

Figure 3.10: Screenshot del Fragment ComuneAmbiente.

Non implementare un'*Activity* apposita con le informazioni su tale località è stata una scelta forzata dalla scarsità di dati a disposizione, come verrà trattato in dettaglio nel capitolo finale.

Le informazioni relative alle features geografiche sono state tutte ricavate da *GeoNames*, che offre un ottimo servizio di ricerca nelle vicinanze di una data posizione, suddiviso per categorie e sottocategorie (Es. Categoria: idrografia Sottocategoria: fiume).

Dal punto di vista strutturale, *ComuneAmbiente* è pressoché identico agli altri *Fragment* dell'attività: implementa una funzione per eseguire il binding dei dati (*bindPlaces()*) e una per inizializzare l'*AmbienteViewModel* associato (*initViewModel()*). Contiene sia il dedicato *AmbienteViewModel*, che il *ComuneViewModel*, poiché il primo è inizializzato con le informazioni provenienti dal secondo.

3.1.3 AlbergoActivity, MuseumActivity, PersonActivity

Le *Activity AlbergoActivity*, *MuseumActivity* e *PersonActivity* veranno trattate parallelamente poiché presentano vari aspetti in comune. Lo scopo di tali *Activity* è infatti quello di presentare all'utente maggiori informazioni riguardo all'elemento che contengono: un albergo, un museo oppure una persona.

Com'è possibile apprezzare dalla figura 3.10, dal punto di vista implementativo ogni *Activity* contiene un riferimento al *ViewModel* di propria competenza. La prima azione che viene eseguita al lancio è l'inizializzazione del *ViewModel* e la successiva osservazione degli oggetti contenuti all'interno

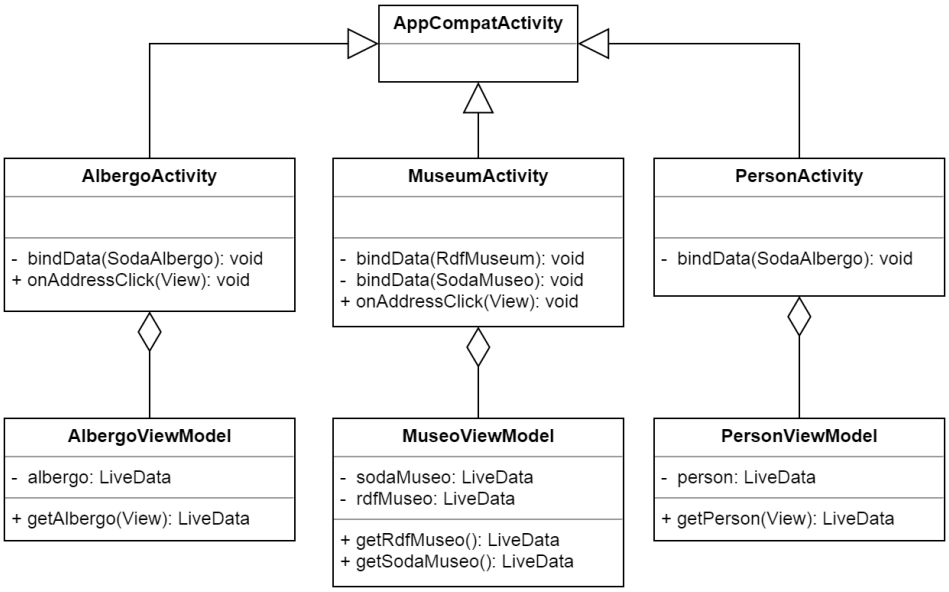


Figure 3.11: Diagramma UML rappresentante AlbergoActivity, MuseumActivity e PersonActivity

di esso. Al completamento dell'operazione di ottenimento dei dati, l'*Activity* viene notificata e si procede con il binding.

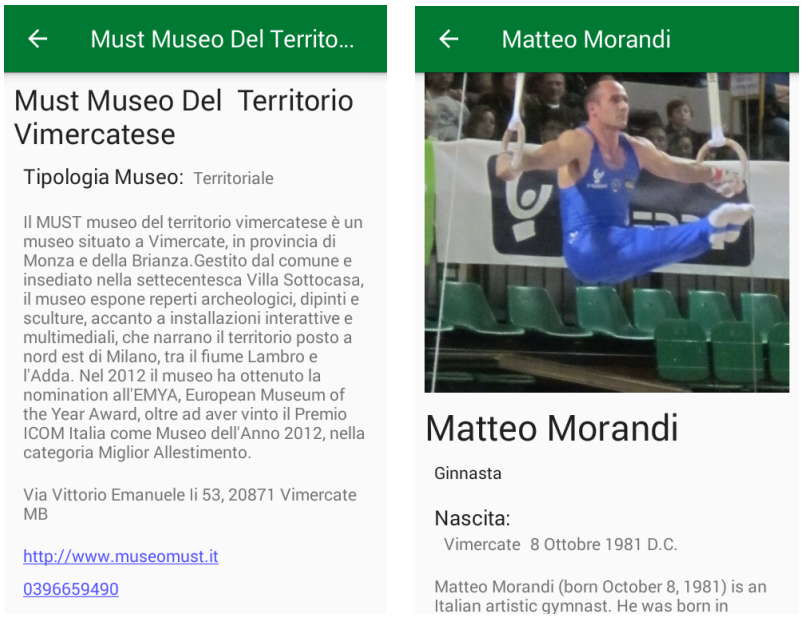
Per quanto riguarda la grafica, l'interfaccia si presenta come in figura 3.12. Alcuni elementi sono interattivi, come numeri di telefono, siti internet, email o indirizzi. Se tali informazioni non presentano errori di formattazione, cliccando su di essi apre l'applicazione installata più adatta per gestire tale informazione, come pe esempio il client di posta per l'indirizzo email oppure la mappa per gli indirizzi postali.

Le fonti di informazioni utilizzate nelle tre *Activity* sono variegate, nello specifico:

- *AlbergoActivity* utilizza solo dati provenienti dalla *Regione Lombardia* poiché non è stato possibile trovare maggiori informazioni sugli alberghi nell'ambito dei *linked data*.

- *MuseumActivity* utilizza dati provenienti sia dalla *Regione Lombardia* che da *DBpedia* (es. descrizione e immagine).

- *PersonActivity* utilizza solo dati provenienti da *DBpedia* poiché la *Regione Lombardia* non tratta informazioni riguardanti le persone.



(a) *MuseumActivity* (b) *PersonActivity*

Figure 3.12: Screenshot di due delle Activity in questione

3.2 ViewModel

Il livello architetturale subito inferiore a quello della *View* è rappresentato dai *ViewModel*. In *Android*, *ViewModel* è una vera e propria classe astratta, introdotta con le nuove librerie architetturali [7], che è possibile derivare (come realizzato nel caso di *DiscoverLombardia*) per implementare *ViewModel* personalizzati.

Un *ViewModel* è un oggetto adibito alla memorizzazione e alla gestione di dati legati all'interfaccia grafica, ovvero dati che verranno visualizzati a schermo o comunque utilizzati dalla *View*. Consente di sgravare l'interfaccia grafica dal gestire il reperimento dei dati, favorendo il disaccoppiamento.

Il framework di *Android* gestisce autonomamente la vita del *ViewModel* in base al ciclo di vita dell'*Activity* o del *Fragment* a cui il *ViewModel* è associato, persistendo l'oggetto tra successive distruzioni e creazioni di tale componente [3]. Per accedere al *ViewModel* relativo ad un certo contesto (*Activity* o *Fragment* nel caso di *DiscoverLombardia*) bisogna richiederli, come mostra la figura 3.13, al *ViewModelProvider*. Tale approccio consente anche di gestire facilmente la condivisione di un *ViewModel*, per esempio,

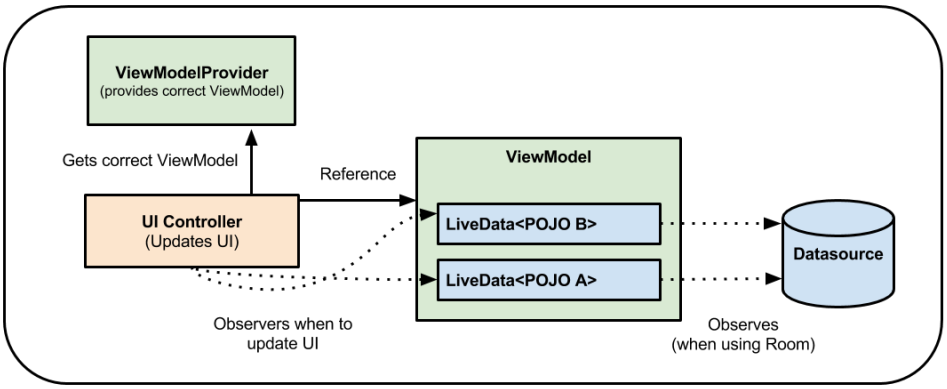


Figure 3.13: Il ViewModel nell'architettura di un'app Android.

tra *Fragment*.

All'interno di *DiscoverLombardia*, questa possibilità viene sfruttata nella *ComuneActivity* e relativi *Fragment*. Il *ComuneViewModel* è associato al contesto dell'attività per farlo persistere durante tutta la sua durata ed ogni *Fragment* ci accede richiedendo al *ViewModelProvider* dell'*Activity* il riferimento a tale *ViewModel*.

Verrà ora preso come esempio il *ComuneViewModel* per spiegare nel dettaglio la struttura ed il funzionamento di tutti i *ViewModel* dell'applicazione. Una rappresentazione dell'architettura è data in figura 3.14.

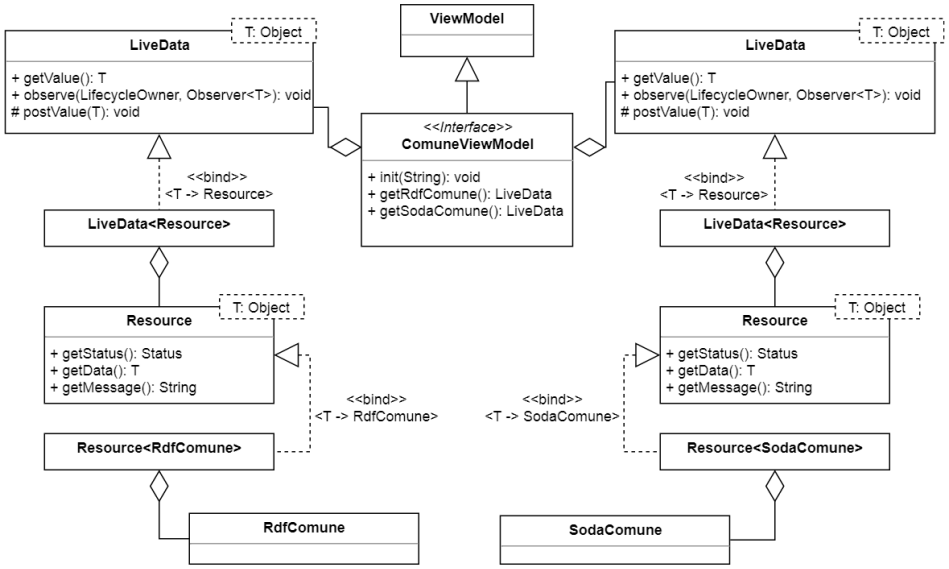


Figure 3.14: Diagramma UML dettagliato del ComuneViewModel.

Per quanto il diagramma UML risulti complicato, la struttura del *ViewModel* è estremamente semplice: contiene due oggetti di tipo *LiveData*, uno per contenere i dati ricevuti dagli *Open Data Lombardia*, l'altro per contenere quelli ricevuti da *DBpedia*.

I dati vengono richiesti tramite la funzione *init()*, a cui vengono passati i parametri necessari a reperirli. All'interno della funzione viene ottenuto il *Repository* di competenza (in questo caso *ComuniRepository*) tramite un *Service Locator*. A questo punto vengono chiamate le funzioni *getRdfComune()* e *getSodaComune()*, che richiederanno i dati *RDF* e *SODA* al *Model* e restituiranno dei *LiveData*.

Il *LiveData* funge da mediatore, infatti implementa tutti i metodi necessari a tale compito: *observe()* per consentire agli osservatori di sottoscriversi alle modifiche di tale oggetto e *getValue()* per ottenerne il contenuto. Al proprio interno però il *LiveData* non contiene direttamente l'oggetto *RDF* o *SODA*, bensì una *Resource*.

La *Resource* è una classe che incapsula i dati in un altro livello di astrazione e consente di specificare se le informazioni sono presenti, se sono in fase di caricamento o se si è verificato un errore durante il recupero.

Se i dati sono disponibili, *getStatus()* restituirà *SUCCESS* e *getData()* l'oggetto richiesto. Se la richiesta è ancora pendente, *getStatus()* restituirà *LOADING*. Infine, se si è verificato un errore, *getData()* restituirà *ERROR* e *getMessage()* il relativo messaggio.

3.3 Repository

Il *Repository* rappresenta l'anello di congiunzione tra il *Model* e i *ViewModel*. La funzione principale è quella di fornire un'interfaccia semplice per richiedere i dati e mascherare totalmente la complessità degli strati sottostanti. Per l'implementazione del *Repository* sono state seguite le direttive date dagli sviluppatori della piattaforma *Android* [3].

All'interno di *DiscoverLombardia*, com'è possibile vedere in figura 3.15, sono presenti più *Repository*, uno per ogni area di competenza. Per coerenza si è poi deciso di raggrupparli in un'interfaccia chiamata *Repository*.

Ogni *Repository* offre delle semplici funzioni che permettono di reperire un particolare tipo di dato: un museo, degli alberghi ecc. L'oggetto che viene ritornato è sempre un *LiveData* contenente una *Resource*, in modo che sia

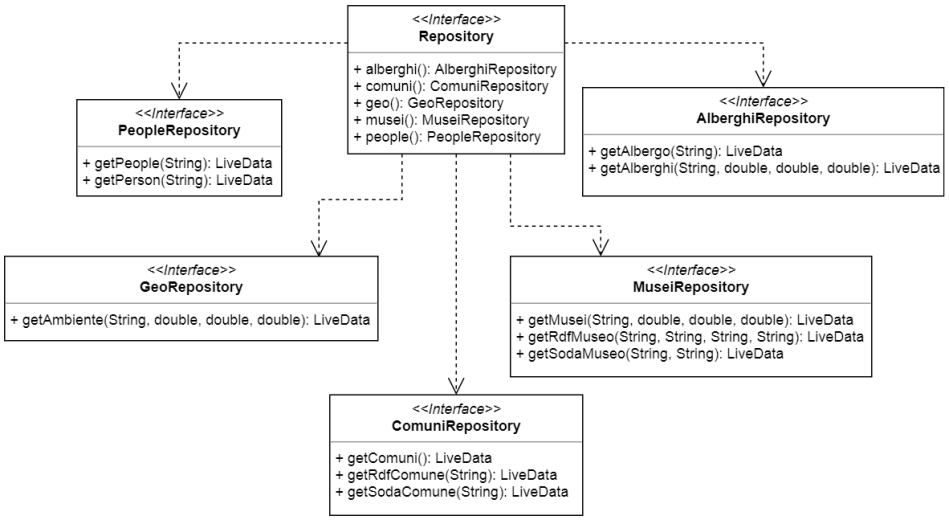


Figure 3.15: Diagramma UML dei Repository.

possibile ricevere notifiche sullo stato della richiesta.

Ad ogni interfaccia presente in figura 3.15, corrisponde poi una implementazione vera e propria che definisce i metodi esposti da tale interfaccia. Poiché le implementazioni sono molto simili tra loro, verrà preso come esempio solamente *ComuneRepositoryImpl* e spiegato in dettaglio.

L'architettura di *ComuneRepositoryImpl* è mostrata in figura 3.16. Il *Repository* al suo interno contiene i riferimenti ai tre servizi utili alla raccolta dei dati all'interno dell'applicazione: *SodaWebService*, *RdfWebService* (trattati nella sezione 3.5) e il *ComuneDao* trattato nella sezione 3.4). Le prime due interfacce servono a reperire i dati dalle fonti in rete, mentre il *DAO* serve a gestire la copia locale dei dati.

Ruolo importantissimo all'interno del *Repository* lo ricopre la *NetworkBoundResource*. Per spiegare maggiormente nel dettaglio di cosa si tratta è opportuno, in primo luogo, spiegare il tipo di procedimento che si è voluto ottenere per l'acquisizione dei dati e le componenti coinvolte.

L'applicazione possiede una cache locale realizzata tramite un database, che consente di tenere una copia dei dati recenti per evitare di ricaricarli ogni volta dalla rete (maggiori dettagli nella sezione 3.4). All'interno di *DiscoverLombardia* tale database è la *single source of truth*.

La *single source of truth* (*SSoT*) è una fonte di dati che si considera come autentica, pertinente e referenziabile all'interno di un'applicazione ed

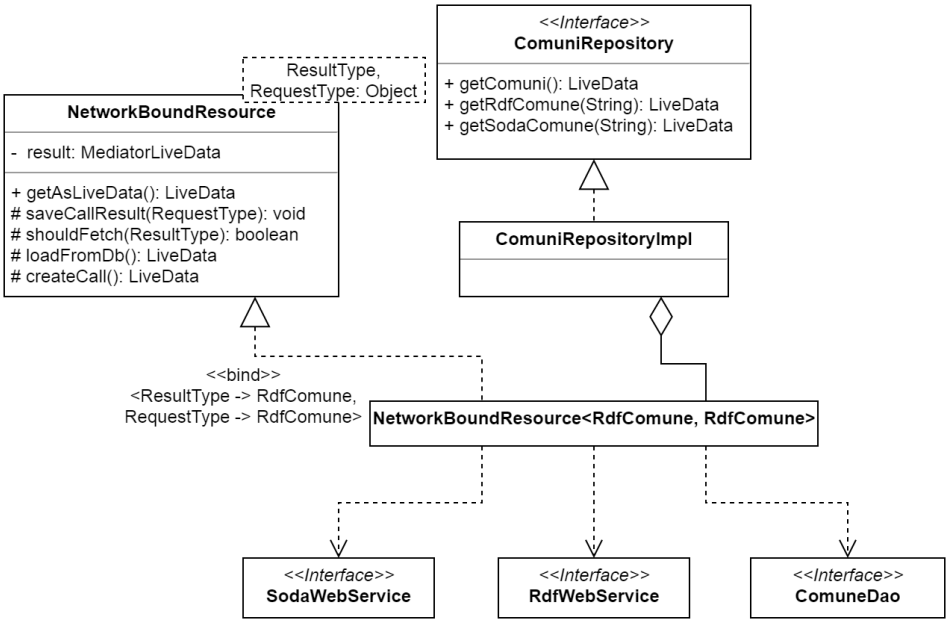


Figure 3.16: Diagramma UML del ComuniRepository.

è di fondamentale importanza. Tutti gli aggiornamenti e le modifiche dei dati passeranno per tale fonte, che poi propagherà i risultati a tutto il resto del sistema. In questo modo è possibile evitare problemi derivanti per esempio da duplicati o inconsistenze [20].

Per rispettare la *SSoT*, i dati, quando ricevuti, vengono dapprima salvati nel database e in seguito prelevati nuovamente da esso, per assicurare che non ci siano discrepanze tra ciò che viene mostrato all'utente sullo schermo. Questo complesso procedimento è schematizzato in figura 3.17.

Com'è possibile notare, nel diagramma è presente una funzione *shouldFetch()*, che nell'implementazione si occupa di controllare se i dati contenuti nella cache siano aggiornati. Se non lo sono, vengono nuovamente richiesti tramite la rete.

La *NetworkBoundResource*, citata in precedenza, è una classe di supporto, che si occupa esattamente di eseguire questo procedimento ogni volta che un dato viene richiesto. Il *Repository*, all'interno di ogni metodo che restituisce dei dati, crea una nuova istanza di tale classe ed esegue l'*override* di tali metodi:

- *saveCallResult()*: in cui viene inserito il codice per salvare nel database interno l'oggetto appena ricevuto dal servizio web (nel caso di *Comu-*

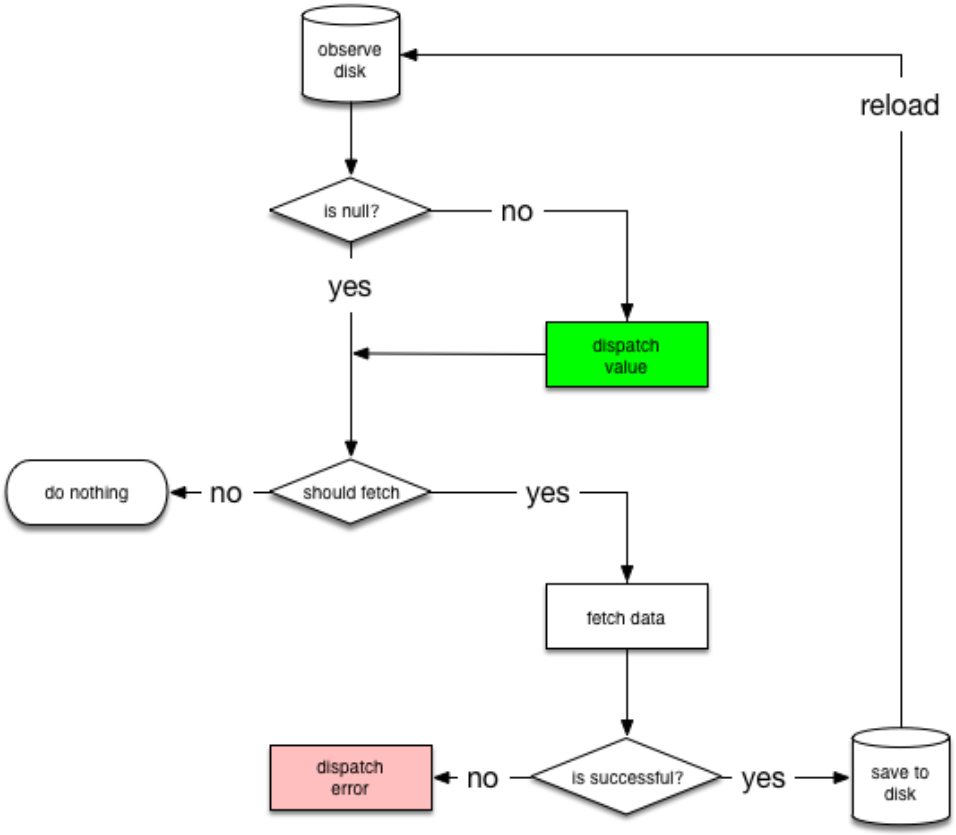


Figure 3.17: Diagramma di flusso dei dati all'interno dell'applicazione [3].

niRepository viene chiamata l'apposita funzione del *ComuneDao*).

- *shouldFetch()*: in cui vengono specificate le condizioni per cui effettuare nuovamente la richiesta dei dati al web server (nel caso in esame se l'oggetto non è presente nella cache oppure è più vecchio di un giorno).
- *loadFromDb()*: in cui si inserisce il codice per prelevare i dati dal database (nel caso in esame viene chiamata l'apposita funzione del *ComuneDao*).
- *createCall()*: in cui viene inserito il codice per reperire i dati dal servizio web (nel caso in esame viene chiamata l'apposita funzione di *SodaWebService* oppure di *RdfWebService*).

Dopo aver creato la *NetworkBoundResource* ed eseguito l'*override* dei suoi metodi, la funzione del *Repository* chiama un altro dei suoi metodi, *getAsLiveData()*, e restituisce il suo contenuto.

3.4 Data Layer

Il *Data Layer* si occupa di gestire il database interno dell'applicazione, che nel caso di *DiscoverLombardia* ha una pura funzione di cache. Come consigliato dalle linee guida per lo sviluppo in *Android* [3], il database è stato realizzato con la libreria *Room* [21]. Room consente di creare un livello di astrazione attorno ad un database *SqLite*, in modo da sfruttarne la potenza ma mantenendo una semplice interfaccia [22].

Dal punto di vista implementativo, *Room* richiede alla classe che rappresenterà il database nell'applicazione tre cose: di essere astratta, estendere *RoomDatabase* ed essere annotata con *@Database*. In *DiscoverLombardia* questa funzione è svolta da *AppDatabase*, la cui architettura è mostrata in figura 3.18. L'implementazione effettiva di *AppDatabase* non è necessaria da parte del programmatore, poiché verrà costruita direttamente dalla libreria.

Alla classe adibita a database vanno successivamente aggiunti i *DAO*. Nel contesto di *Room* i *DAO* (*Data Access Object*), sono semplici interfacce *Java* che definiscono delle operazioni sul database (inserimento, cancellazione ecc.) e sono contrassegnate con l'annotazione *@Dao*.

I metodi dichiarati nell'interfaccia, a seconda della funzione, vengono annotati con annotazioni diverse: per esempio con *@Query* è possibile specificare la query *SQL* che quella funzione andrà a eseguire, *@Insert* invece

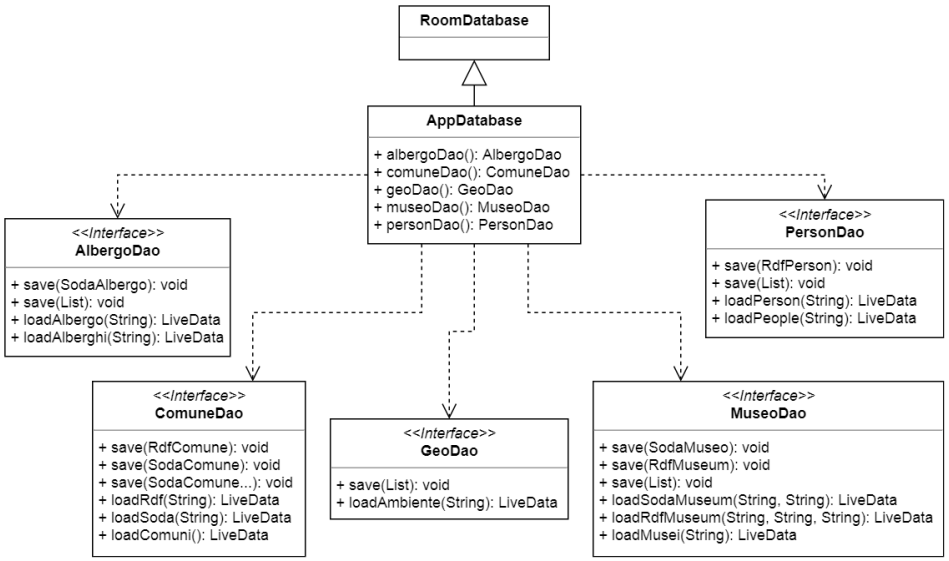


Figure 3.18: Diagramma UML dell'AppDatabase e dei relativi DAO.

informa che quel metodo sarà utilizzato per inserire tuple nel database. Come per *AppDatabase*, anche per i *DAO* non è richiesta l'implementazione dell'interfaccia, poiché verrà fornita da *Room*.

L'ultimo aspetto da tenere in considerazione è rappresentato dagli oggetti che contengono i dati (*Data Objects*). Per essere gestite da *Room*, le classi rappresentanti *Data Objects* devono essere contrassegnate con l'annotazione *@Entity*. L'applicazione *DiscoverLombardia* utilizza una decina di tali oggetti tra *RDF* e *SODA*, per cui verranno mostrati solo due esempi (*RdfComune* e *SodaComune*) che risultano comunque esplicativi per tutti gli altri casi. In figura 3.19 è possibile osservare una rappresentazione della gerarchia di oggetti all'interno dell'app.

Tutti i *Data Objects* in *DiscoverLombardia* ereditano, direttamente o indirettamente, da *CacheableObject* poiché, quando utilizzati, verranno memorizzati nella cache.

All'interno di una cache, deve essere possibile determinare se gli oggetti sono aggiornati oppure no. *CacheableObject* fornisce esattamente questa fun-

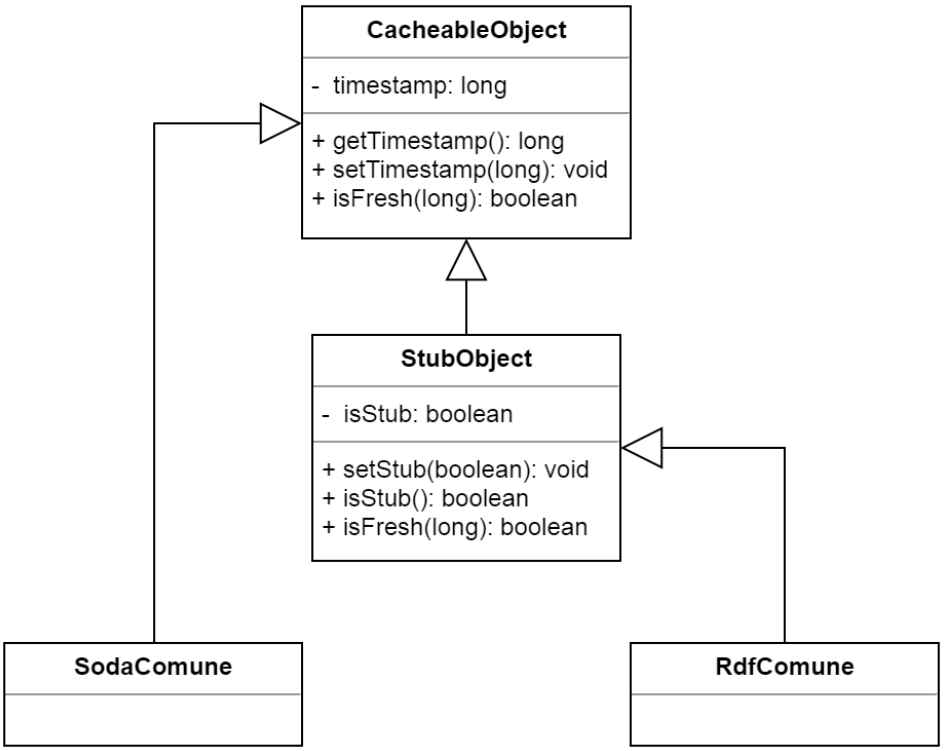


Figure 3.19: Diagramma UML della gerarchia di oggetti in DiscoverLombardia.

zionalità: contiene un campo *timestamp* che indica l'istante in cui l'oggetto viene memorizzato nel database ed una funzione *isFresh()* per stabilire se il dato è da considerarsi aggiornato rispetto all'intervallo di tempo passato come parametro.

Può capitare che per alcuni *Data Object* non sia possibile reperire tutte le informazioni necessarie, a causa, per esempio, di un problema di connessione. È stata quindi definita la classe *StubObject* (sottoclasse di *CacheableObject*) che rappresenta oggetti reperiti parzialmente. Contiene un campo *isStub* che indica se l'oggetto è incompleto ed esegue l'*overriding* di *isFresh()* per includere l'incompletezza nella condizione che considera l'oggetto non aggiornato. Tipicamente gli oggetti che derivano da *StubObject* sono di tipo *RDF*, visto che spesso sono ottenuti da più fonti ed è quindi possibile che alcuni dati vengano reperiti e altri no.

3.5 Servizi Web

I web service utilizzati in *DiscoverLombardia*, come già discusso, sono tre: *Open Data Lombardia*, *DBpedia* e *Geonames*. In seguito verranno descritti come sono stati sfruttati e integrati all'interno dell'applicazione.

3.5.1 Dati Lombardia

Per accedere ai dati della *Regione Lombardia* sono state utilizzate le *API* Socrata (*SODA*). La libreria *Java* consente di effettuare query fornendo l'*endpoint* e il nome del database. Le query è possibile costruirle da codice tramite le classi *Query* ed *Expression*, astraendo al programmatore il linguaggio vero e proprio utilizzato.

In figura 3.20 è mostrata l'architettura adottata in *DiscoverLombardia*. L'interfaccia *SodaWebService* definisce i metodi che verranno utilizzati per ottenere i dati, mentre *SodaWebServiceImpl* si occupa di implementarli.

Per eseguire le query viene utilizzata la classe *Consumer*, fornita dalle librerie, a cui si passa un oggetto di tipo *Query* e un'interfaccia di tipo *Callback*. Entrambi le classi sono fornite da *SODA*. *Query*, come dice il nome stesso, contiene la query e il nome del dataset su cui eseguirla, mentre *Callback* è un'interfaccia che definisce un metodo *onResults()* che verrà chiamato al ricevimento della risposta dal web service.

Per gestire la risposta del server e notificare la disponibilità dei dati, sono

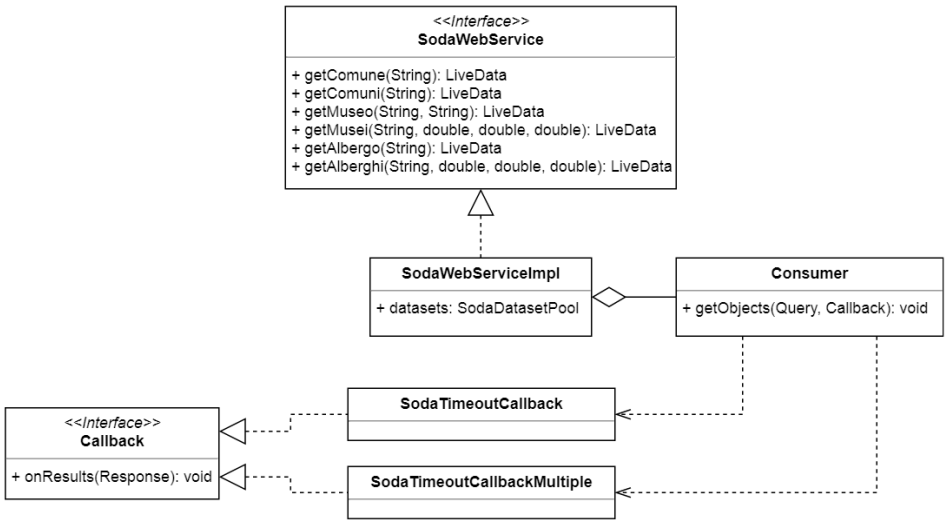


Figure 3.20: Diagramma UML del SodaWebService.

state create due classi che implementano *Callback*, chiamate *SodaTimeoutCallback* e *SodaTimeoutCallbackMultiple*: la prima per gestire risposte che devono restituire un solo elemento, la seconda per ricevere risultati multipli.

3.5.2 DBpedia

Per accedere ai dati di *DBpedia*, l'applicazione utilizza query *SPARQL* ai server di *DBpedia* e *DBpedia Italia*. Le query vengono effettuate tramite classi realizzate *ad hoc*, poiché non è stato possibile includere le librerie *ARQ* di *Jena* in ambiente *Android*.

In figura 3.20 è mostrata l'architettura ad alto livello, successivamente verranno trattate in maggior dettaglio tutte le componenti.

RdfWebService definisce l'interfaccia con cui interagire per richiedere i *Data Object* relativi alle varie entità trattate dal sistema, mentre *RdfWebServiceImpl* ne fornisce l'implementazione.

I metodi implementati da *RdfWebServiceImpl* devono essere eseguiti in modo asincrono, per evitare il bloccaggio dell'applicazione mentre si attende la ricezione dei dati. Per poter fare questo, vengono utilizzati degli oggetti di tipo *Crawler* che creano dei *thread* indipendenti in cui eseguono le query ai web service. Il codice vero e proprio dei metodi è scritto all'interno di questi oggetti, così come il *LiveData* che conterrà il risultato della query.

RdfWebServiceImpl, come è possibile notare dal diagramma UML, uti-

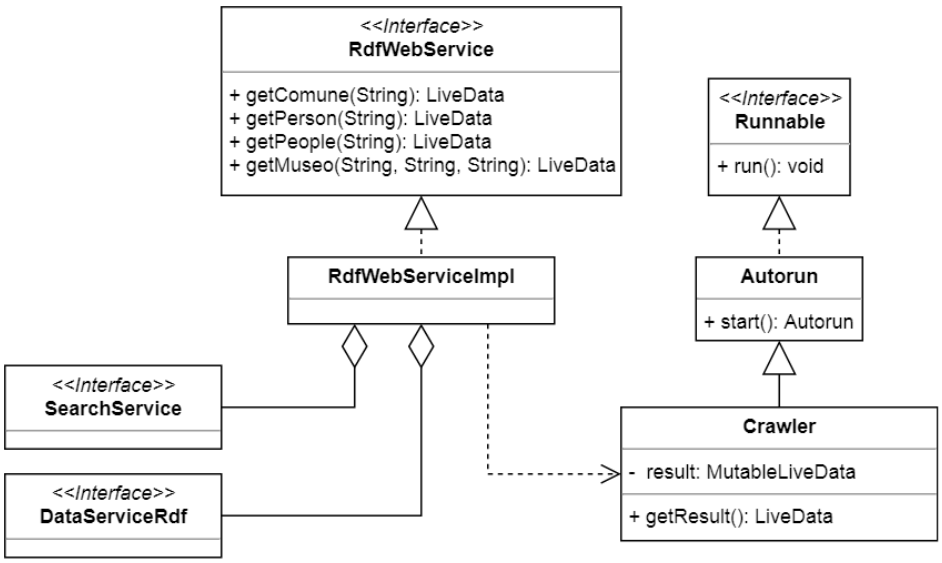


Figure 3.21: Diagramma UML ad alto livello del RdfWebService.

lizza anche due interfacce: *SearchService* e *DataServiceRdf*. La prima fornisce un servizio di ricerca dei comuni, ovvero permette di ottenerne l'*URI* della risorsa *RDF* a partire dal nome. La seconda definisce le funzioni che il servizio di richiesta dei *Data Object* deve fornire. L'implementazione di tali interfacce verrà trattata in maggior dettaglio nelle sezioni che seguono.

SearchService

Il servizio di ricerca permette, come già detto, di ottenere l'*URI* della risorsa *RDF*, a partire dal nome del comune. L'interfaccia è generica, mentre l'implementazione realizzata si appoggia al servizio *Lookup* di *DBpedia* [23]. Tali *API* consentono di effettuare una richiesta filtrando per classe (es. museo, città ecc.) e ottenere una lista di risorse plausibili ordinate per ranking. In figura 3.22 è mostrata l'architettura che consente di effettuare la ricerca.

DBpediaSearch implementa l'interfaccia *SearchService*, fornendo corpo alla funzione *getComuneUri()*. Il metodo si appoggia ad un'altra classe chiamata *DBpediaSparqlEndpoint*, che gestisce la query a più basso livello, effettuando la richiesta *HTTP* ed eseguendo il parsing della risposta in un *SearchResult*. La funzione utilizzerà poi uno dei metodi forniti da *SearchResult* per restituire il risultato più rilevante.

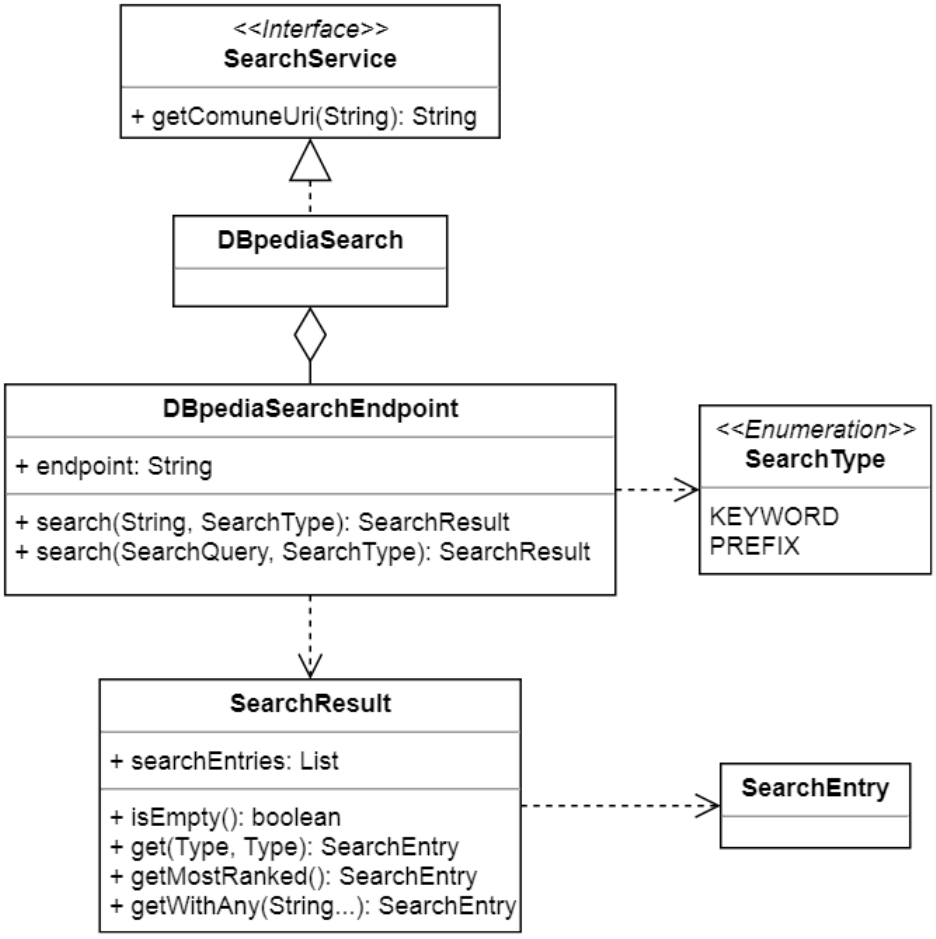


Figure 3.22: Diagramma UML del SearchService.

In particolare, è stata implementata una funzione chiamata *getWithAny()*, che consente di specificare delle parole chiave da cercare all'interno del risultato della ricerca. In questo modo si è riuscito a distinguere, per esempio, tra due comuni omonimi, specificando come chiavi di ricerca "Lombardy" e "Lombardia".

DataServiceRdf

Il *DataServiceRdf* è l'interfaccia che definisce le funzioni per ottenere tutti i *Data Object* riguardanti dati *RDF*. *DBpediaSparql* è una sua implementazione, che consente di ottenere gli oggetti recuperandoli da *DBpedia* tramite query in *SPARQL*. La figura 3.23 ne è mostra l'architettura.

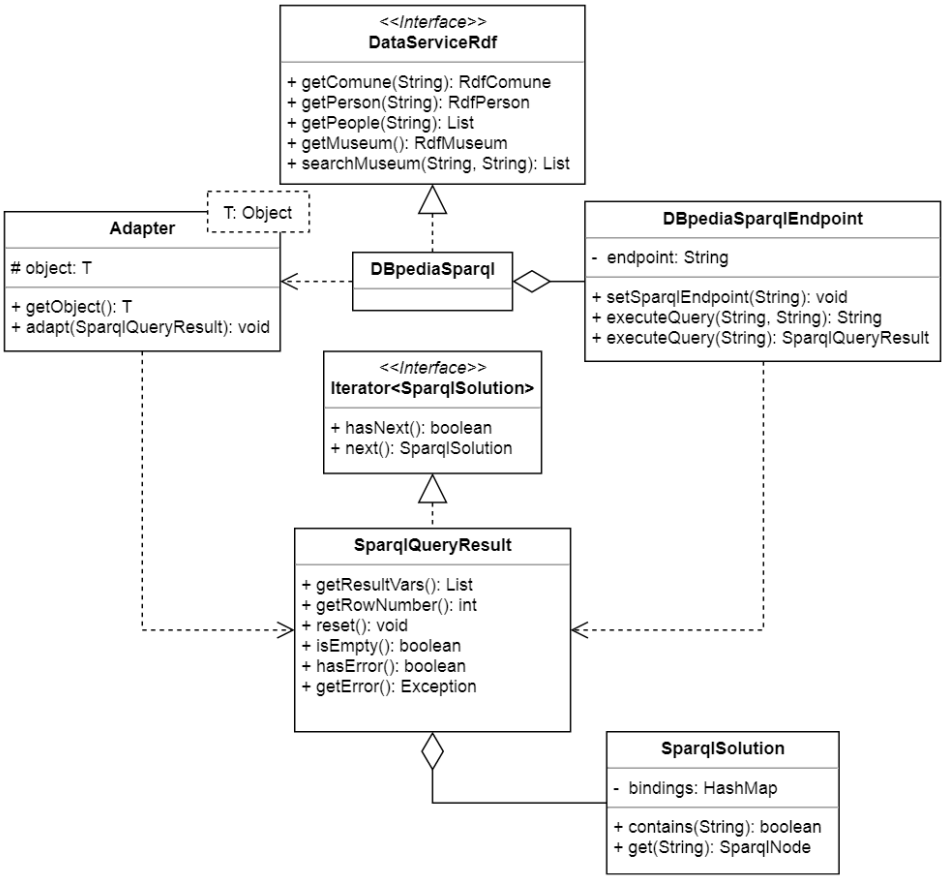


Figure 3.23: Diagramma UML del DataServiceRdf.

Come è possibile notare, *DBpediaSparql* internamente possiede un oggetto di tipo *DBpediaSparqlEndpoint*. Tale classe si occupa di eseguire qualsiasi query a basso livello: effettua la richiesta *HTTP* al server di *DBpedia* ed esegue il parsing della risposta ritornando un oggetto di tipo *DBpediaSparqlResponse*.

La classe *DBpediaSparqlResponse* racchiude al suo interno tutte le informazioni riguardanti la query: i risultati (chiamati soluzioni), le variabili selezionate, informazioni su un eventuale errore, ecc. Per accedere alle soluzioni contenute, implementa l'interfaccia *Iterator*<*SparqlSolution*>, rendendone facile l'utilizzo.

DBpediaSparql non si occupa direttamente di leggere le soluzioni e popolare i campi del *Data Object* richiesto, ma delega l'operazione ad un *Adapter*. Una *Factory class* apposita (*AdapterFactory*) fornisce la corretta implemen-

tazione dell'*Adapter* in base all'oggetto passato come parametro. Viene quindi chiamata la funzione *adapt()* che inserirà in ogni apposito campo la giusta informazione estratta da *DBpediaSparqlResponse* e l'oggetto è pronto per essere restituito.

3.5.3 GeoNames

Per accedere ai dati di *GeoNames* sono state utilizzate le librerie *Java* apposite, disponibili sul sito ufficiale [16]. Forniscono la possibilità di eseguire query di vario genere al loro database, ma per quanto riguarda *Discover-Lombardia* è stata utilizzata la funzione *findNearby*.

Il metodo consente di ottenere tutti i punti di interesse entro un certo raggio da una posizione data, filtrando i risultati in base al tipo (piazza, statua, giardino, fiume ecc.). Figura 3.23 mostra l'architettura costruita attorno a tale servizio.

Si è fornita un'interfaccia *GeoService* che descrive il comportamento desiderato ad alto livello e una sua implementazione: *GeoNamesWebService*. Tale classe, come spiegato per *RdfWebServiceImpl*, utilizza un oggetto esterno (*GeoNamesNatureCrawler*) per eseguire il metodo *findNearbyNature* in modo asincrono. L'implementazione vera e propria della funzione è quindi contenuta in *GeoNamesNatureCrawler*, che utilizza la classe *WebService*, for-

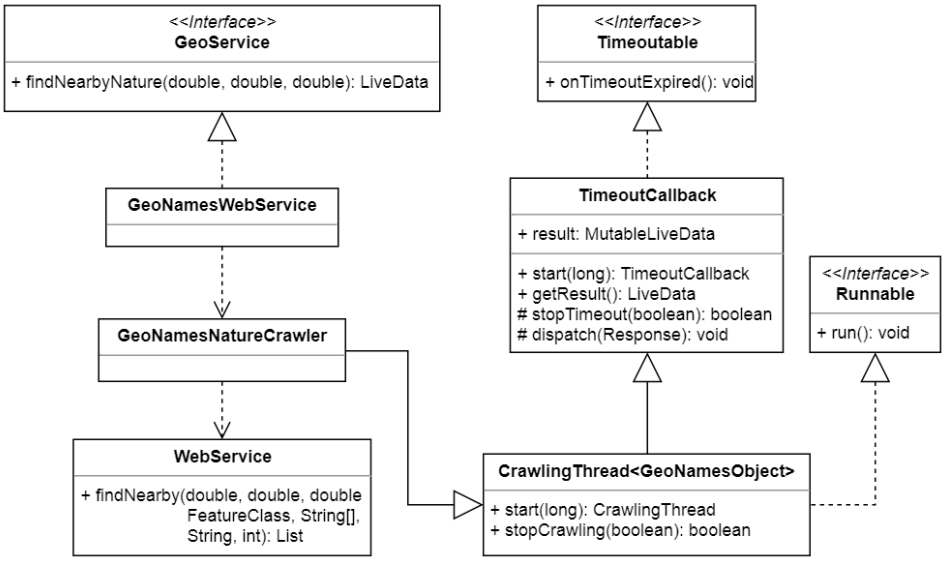


Figure 3.24: Diagramma UML del GeoService.

nita dalle librerie di *GeoNames*, per eseguire le richieste al server, ricevere i risultati e notificare dell'avvenuta ricezione.

Capitolo 4

Esempi d'Utilizzo

In questa sezione verranno presi in considerazione degli esempi concreti di utilizzo dell'applicazione, spiegando nel dettaglio il funzionamento delle componenti definite nel capitolo precedente.

4.1 Dettagli di un Comune

Lo scenario trattato in questa sezione, come schematizzato in figura 4.1, è il seguente:

> **Premesse**: l'utente si trova nella attività principale dell'applicazione (*HomeActivity*) e non ha recentemente visitato il comune che inserisce.
>
> **Azioni**: l'utente inizia a scrivere nella casella di testo e, una volta comparso il nome del comune desiderato nel menù a tendina, ci clicca sopra.
>
> **Risultato**: l'utente visualizza la pagina con le informazioni generali del comune selezionato.

La schermata iniziale, in cui l'utente si trova, è mostrata in figura 4.2(a). Inizia quindi a digitare il nome del comune, che in questo esempio sarà "Milano". Una volta comparsa l'opzione nel menù a tendina, l'utente ci clicca sopra (figura 4.2(b)).

Al momento del click, viene eseguita la funzione *onComuneClick()* presente in *HomeActivity*, cui viene passata la *View* sul quale si è cliccato. Il metodo provvede a creare un nuovo *Intent* di tipo *ComuneActivity*, a cui

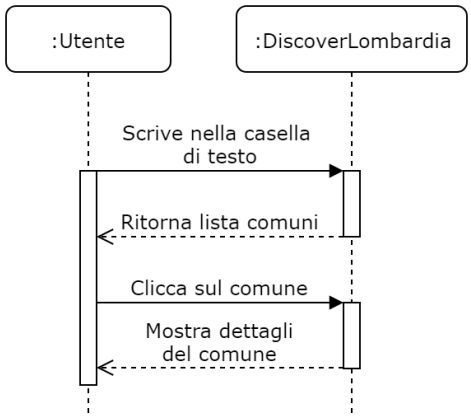
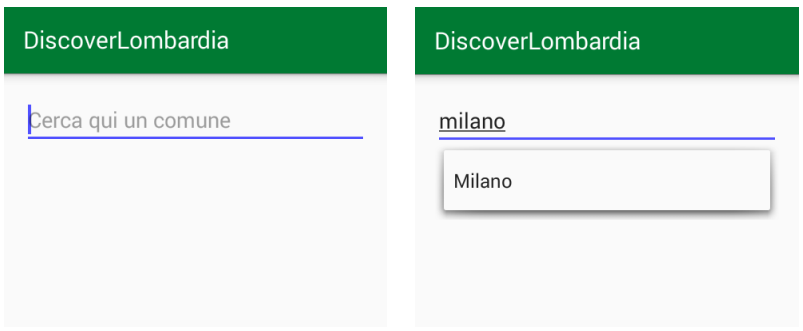


Figure 4.1: Scenario 1: l'utente visita la pagina di un comune.

passa "Milano" (il nome del comune) e poi chiama *startActivity()* per avviare l'attività.

A questo puntola navigazione passa dalla *HomeActivity* alla *ComuneActivity*. La prima funzione che viene eseguita nella neo-lanciata attività è *onCreate()*, che si occupa di inizializzare i componenti grafici dell'interfaccia (come la *Action Bar* e il *Drawer Menù*) ed eseguire le seguenti operazioni:

1. reperire il *ComuneViewModel* tramite il *ViewModelProvider*.
2. inizializzare tale *ViewModel* chiamando la funzione init() e passandogli la stringa "Milano", ricevuta dalla precedente *Activity*.
3. caricare il *Fragment ComuneOverview*, che permetterà di visualizzare le informazioni sul comune contenute nel *ComuneViewModel* una volta



(a) *La HomeActivity* (b) *L'utente scrive "milano"*

Figure 4.2: Screenshot della HomeActivity prima e dopo aver scritto il nome del comune.

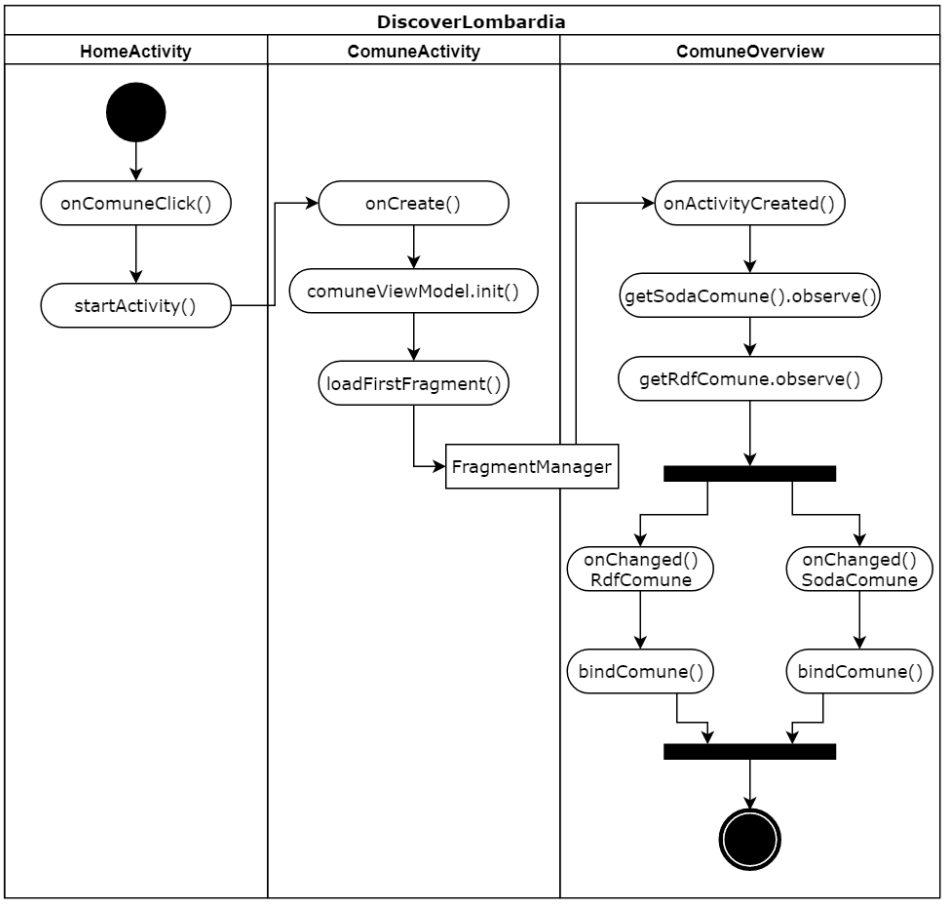


Figure 4.3: Activity Diagram ad alto livello dello scenario 1.

che verranno ricevuti i dati.

Quando viene chiamata la funzione *init()*, come già detto, le viene passato come parametro la stringa "Milano". Internamente il *ComuneViewModel* controlla se i *LiveData* che contiene (rdfComune e sodaComune) sono già stati inizializzati. Se invece sono uguali a *null*, viene richiesto al *Locator* il *ComuneRepository*, a cui viene ordinato di ottenere il *LiveData*, chiamando la relativa funzione *getComune()* e passandogli "Milano".

All'interno del *Repository*, la prima operazione effettuata è reperire dalla cache i *Data Object* tramite il *ComuneDao* (funzione *loadRdf()/loadSoda()*). Siccome abbiamo supposto che il comune non è stato visitato recentemente, l'oggetto ritornato dal *DAO* sarà *null*. Viene quindi chiamata la funzione *shouldFetch()* che, notando nullo l'oggetto, ritorna *true*, indicando la neces-

sità di eseguire una richiesta al web service.

Il *ComuneRepository* decide quindi di chiedere i dati ai due servizi:

- chiamando la funzione *RdfWebService.getComune("Milano")* viene creato un nuovo oggetto di tipo *ComuneCrawler*, lanciato su un nuovo *thread*, che si occuperà di reperire i dati e notificare la loro ricezione. Nel frattempo viene restituito l'oggetto di tipo *LiveData* in cui verrà salvato il risultato della richiesta.

- chiamando *SodaWebService.getComune("Milano")* vengono immediatamente costruiti gli oggetti *Query* (che conterrà la query da eseguire e il dataset su cui eseguirla) e *SodaTimeoutCallback*. Viene poi chiamata la funzione *getObjects()* a cui vengono passati i due parametri: la query da eseguire e l'oggetto *Callback* che riceverà la risposta dal web server. Nel frattempo viene restituito l'oggetto *LiveData* contenuto in *SodaTimeoutCallback*, che conterrà il risultato della query.

Ricevuti i due *LiveData*, *ComuneViewModel* li assegna alle variabili membro *rdfComune* e *sodaComune* e termina l'esecuzione.

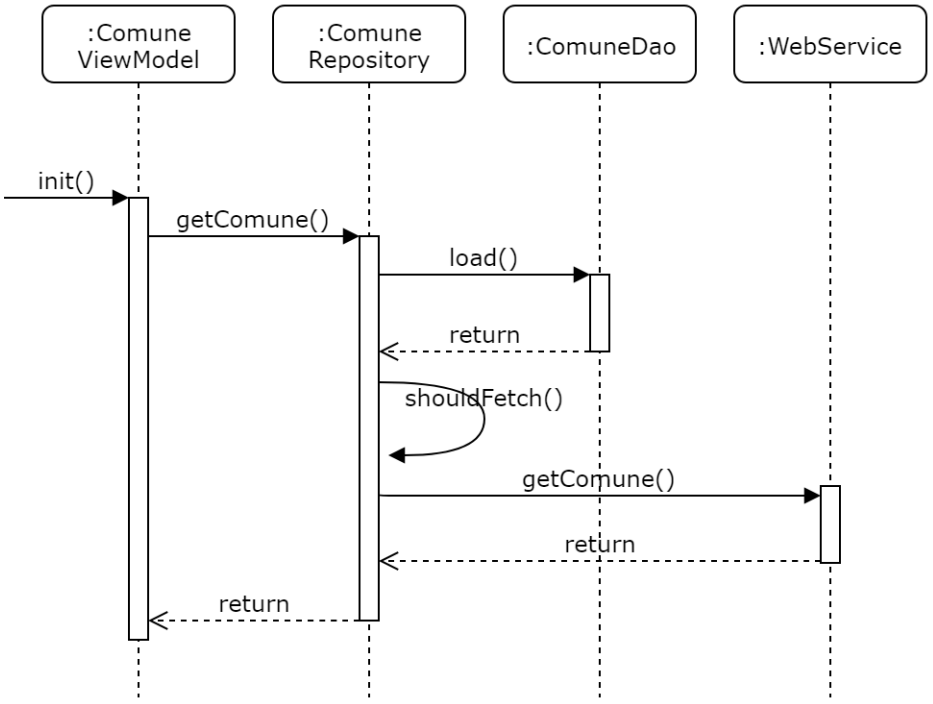


Figure 4.4: Sequence diagram dettagliato della funzione init().

Figura 4.4 rappresenta con un sequence diagram tutte le operazioni eseguite all'interno della funzione *init()*.

A questo punto *ComuneActivity* esegue la funzione *loadFirstFragment()*. Per caricare il *Fragment*, l'*Activity* non se ne occupa direttamente, ma fa uso del proprio *FragmentManager*. Crea una nuova transazione, gli passa una nuova istanza di *ComuneOverview* (a cui passa il nome del comune) ed esegue il *commit*.

Quando il *Fragment* viene caricato, il framework *Android* chiama la funzione *onActivityCreated()*, in cui vengono eseguite le seguenti operazioni:

1. viene richiesto al *ViewModelProvider* dell'*Activity* un riferimento al *ComuneViewModel*.

2. viene chiamata la funzione *init()* sul *ViewModel* passandogli la stringa "Milano", per essere sicuri che venga inizializzato (tipicamente tale funzione non eseguirà nulla poiché l'inizializzazione è già lanciata da *ComuneActivity*).

3. il *Fragment* osserva i due oggetti contenuti nel *ViewModel*, rdfComune e sodaComune, per ricevere aggiornamenti quando i dati saranno disponibili. È stato possibile osservare i due *LiveData* poichè sono stati inizializzati tramite la funzione *init()* trattata precedentemente.

In questo frangente il *Fragment ComuneOverview* rimane in attesa di ricevere aggiornamenti riguardanti i dati da parte dei due web service.

Nel frattempo il *thread ComuneCrawler* si occupa del reperimento dei dati (figura 4.5). Per prima cosa chiede al *SearchService* di ottenere l'*URI* della risorsa "Milano" chiamando la funzione *getComuneUri()*.

Internamente *DBpediaSearch* (che è il *SearchService* utilizzato) si occupa di eseguire la query al web server tramite l'oggetto *DBpediaSearchEndpoint* contenuto al suo interno. La query specifica come parola chiave "Milano" e come classe di appartenenza utilizzata per il filtraggio dei risultati "place", ovvero località.

Il risultato viene ricevuto sottoforma di *SearchResult*, che contiene fino a cinque *SearchEntry* ordinate in base al ranking. Per discriminare tra le varie entry, si sceglie il risultato con ranking più alto che contiene all'interno di almeno uno dei campi della risposta (*URI*, nome, descrizione ecc.) la parola "Lombardy" o "Lombardia". Questa misura di sicurezza è stata introdotta per eliminare il rischio dovuto a comuni omonimi in diverse regioni. A questo

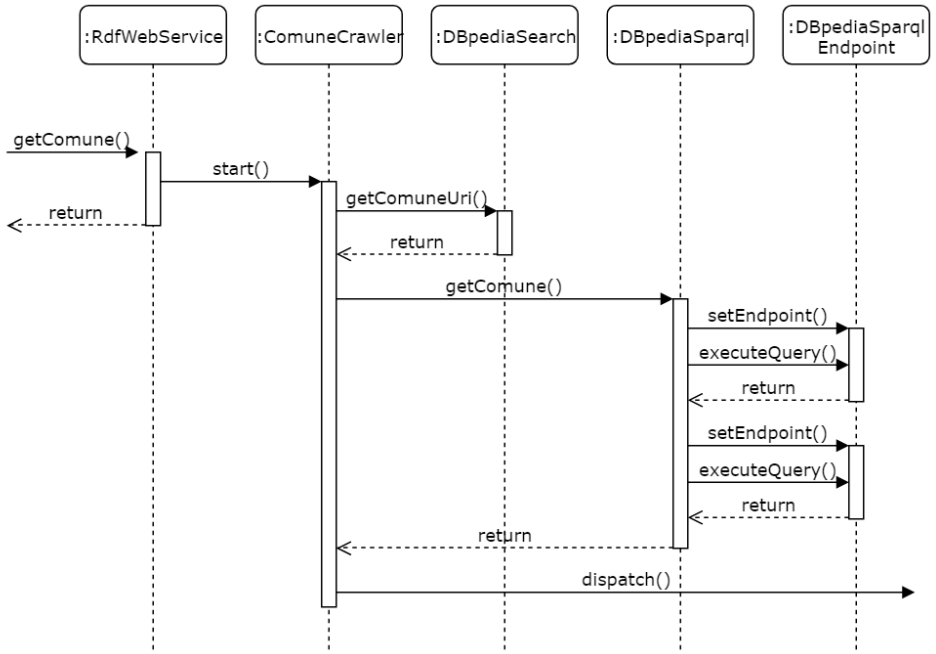


Figure 4.5: Sequence diagram dettagliato del RdfWebService.

punto la *SearchEntry* prescelta è stata identificata e ne verrà quindi restituito l'*URI*.

Una volta che il *ComuneCrawler* è in possesso dell'*URI*, è in grado di eseguire la query per ottenere *RdfComune*. Chiama quindi la funzione *getComune()* del *DataServiceRdf* in suo possesso (ovvero *DBpediaSparql*), passandogli l'*URI* di Milano.

DBpediaSparql costruisce la query, setta in *DBpediaSparqlEndpoint* come indirizzo del server "dbpedia.org" e infine chiama la funzione *DBpediaSparqlEndpoint.executeQuery()*, che ritorna un oggetto di tipo *SparqlQueryResult*. Per trasferire le informazioni contenute nel risultato all'interno di un *RdfComune* esegue le seguenti istruzioni:

1. crea un oggetto di tipo *RdfComune* vuoto.

2. richiede un oggetto di tipo *Adapter* dalla *AdaperFactory* passandogli il *Data Object* appena creato. La factory restituirà quindi l'istanza corretta ad adattare il tipo di dato *RdfComune*.

3. chiama la funzione *adapt()* dell'*Adapter* passandole lo *SparqlQueryResult*.

4. l'oggetto *RdfComune* ora contiene le informazioni estratte dalla risposta.

Siccome alcune informazioni riguardo al comune vengono reperite da DBpedia Italia, dallo *SparqlQueryResult* viene estratto anche l'*URI* della risorsa su tale sito. Il crawler procede quindi a settare l'indirizzo del server a "it.dbpedia.org" e a ripetere le istruzioni appena menzionate. Una volta finito il procedimento, l'oggetto *RdfComune* contiene tutte le informazioni necessarie e viene restituito.

Da notare: se per caso la seconda query non dovesse andare a buon fine, il *Data Object* verrà contrassegnato come "stub" chiamando la funzione *setStub()* e passando *true* (maggiori informazioni sugli *StubObject* nella sezione 3.4 sul *DataLayer*).

Quando il *ComuneCrawler* riceve l'oggetto, controlla la presenza di errori ed esegue il *dispatching*, che verrà trattato a breve, subito dopo l'excursus sui dati della *Regione Lombardia*.

Per quanto riguarda i dati richiesti tramite *SODA*, tutta la procedura, dalla richiesta al ricevimento del *SodaComune*, è effettuata dalle librerie. Quando l'oggetto è disponibile, viene chiamata la funzione *onResults()* del

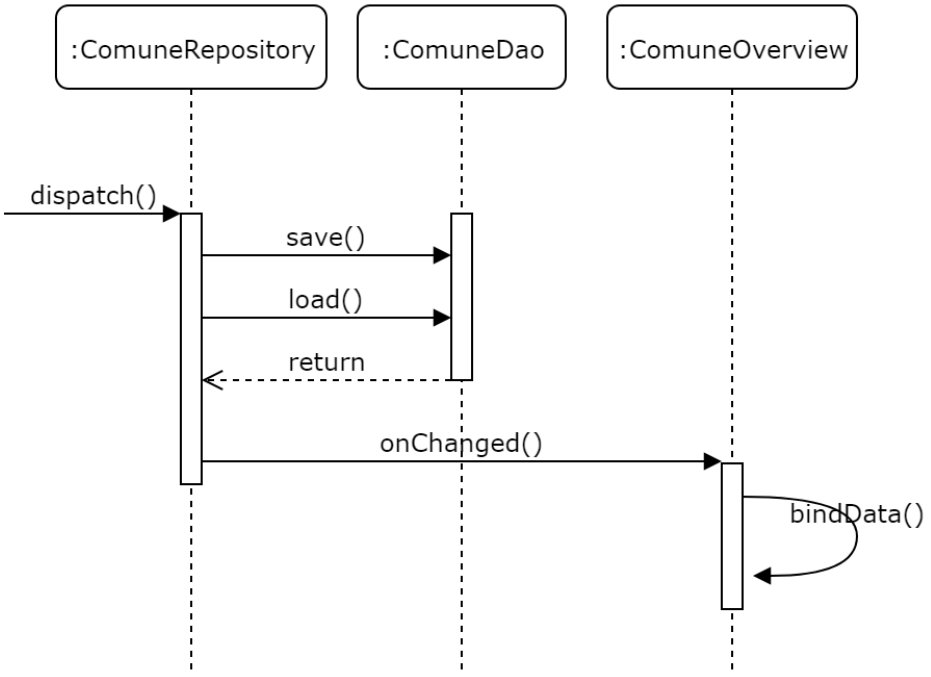
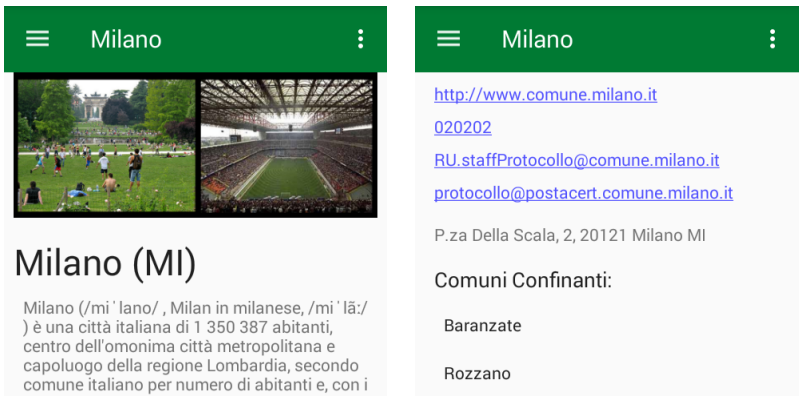


Figure 4.6: Sequence diagram riguardante la notifica del ricevimento dei dati.



(a) La foto, il nome e la descrizione. (b) I contatti, l'indirizzo e i comuni limitrofi.

Figure 4.7: Screenshot della pagina dei dettagli di Milano

Callback che si è passato al momento della richiesta. In tale metodo viene controllata la presenza di errori nella risposta e poi viene effettuato il *dispatching*.

A questo punto l'esecuzione procede parallelamente, poiché il flusso di esecuzione, benché non contemporaneo, risulta il medesimo. Quando viene eseguita la funzione *dispatch()* (nel *ComuneCrawler* o nel *SodaTimeoutCallback()*) accade ciò che è rappresentato in figura 4.6.

Il *ComuneRepository* viene notificato dell'arrivo dei dati, quindi procede al salvataggio nel database e al loro successivo reperimento tramite query (seguendo il principio della *Single Source of Truth*). Quando i dati saranno nuovamente disponibili, il *Repository* notifica tutti gli osservatori del *LiveData*, restituito in precedenza (figura 4.4) e ora contenuto nel *ComuneViewModel*, della presenza di tali informazioni. Nel caso in esame l'unico observer è il *Fragment ComuneOverview*, che riceve l'oggetto, controlla la presenza di eventuali errori ed infine esegue il *binding*. L'utente è ora in grado di visualizzare le informazioni appena ricevute a schermo (figura 4.7).

4.2 Dettagli di un Museo

Lo scenario trattato in questa sezione, come schematizzato in figura 4.8, è il seguente:

Premesse: l'utente si trova nella *ComuneActivity* dell'applicazione,

in particolare sulla pagina delle informazioni generali sul comune e supponiamo che abbia visitato recentemente la sezione dei musei.

Azioni: l'utente clicca sul tasto che mostra il *drawer menu*, seleziona "Cultura" e appare la lista dei musei. Poi sceglie un museo che gli interessa e ci clicca sopra. Si suppone che l'utente non abbia visualizzato i dettagli di tale museo recentemente.

Risultato: l'utente visualizza la pagina con le informazioni del museo selezionato.

L'utente si trova nella pagina delle informazioni sul comune, che supponiamo essere Milano, e clicca in alto a sinistra sul tasto contrassegnato da tre barre orizzontali per aprire il *drawer menu* (figura 4.9(b)). Una volta apparso, clicca sull'elemento "Cultura".

Viene invocata la funzione *onNavigationItemSelected* della *ComuneActivity*, che identifica l'elemento cliccato e carica il rispettivo *Fragment* (*ComuneCultura*) tramite la funzione *replaceFragment()*.

Per rimpiazzare *ComuneOverview*, *ComuneActivity* sfrutta il proprio *FragmentManager*: crea una nuova transazione, gli passa il nuovo *Fragment ComuneCultura* ed esegue il commit.

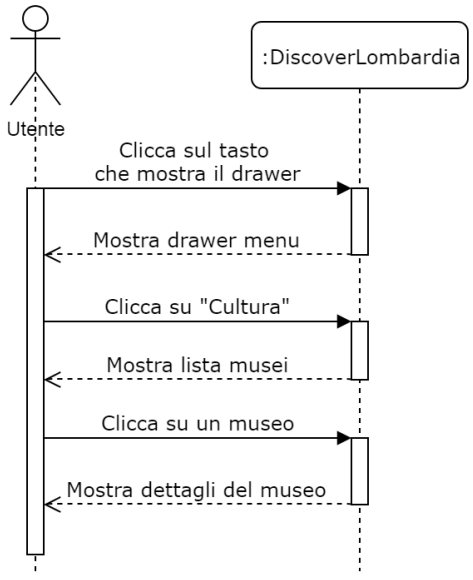
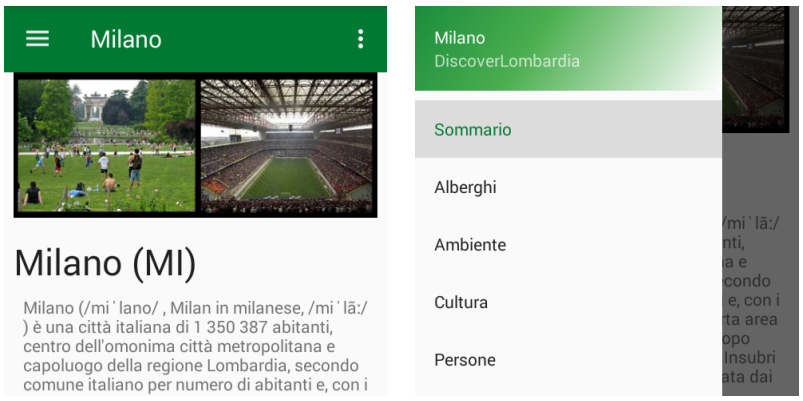


Figure 4.8: Scenario 2: l'utente guarda i musei del comune e vede i dettagli di uno di essi.



(a) La pagina del comune.

(b) *Drawer menu* aperto.

Figure 4.9: Screenshot prima e dopo l'apertura del drawer menu

Quando il *Fragment* è caricato, la prima funzione che viene eseguita è *onActivityCreated()*. All'interno di tale metodo vengono svolte le seguente operazioni:

1. *ComuneCultura* richiede al *ViewModelProvider* della *ComuneActivity* il *ComuneViewModel*.

2. da tale *ViewModel* viene reperito l'oggetto *RdfComune*, che viene passato alla funzione *initViewModel()* come parametro. Il metodo verrà trattato in dettaglio a breve.

3. viene inizializzato il componente *RecyclerView* che conterrà la lista dei musei.

4. viene inizializzato l'*Adapter* dei musei, che si occuperà di eseguire il binding dei musei all'interno del *RecyclerView*.

La sopracitata funzione *initViewModel()* si occupa, come suggerito dal nome, di inizializzare il *MuseiViewModel*. Come di consueto, il *ViewModel* viene reperito tramite il *ViewModelProvider*. Successivamente viene chiamato il metodo *init()* a cui vengono passati il nome del comune ("Milano"), la latitudine e la longitudine (prelevati dall'oggetto *RdfComune*) e il raggio massimo (fissato a 10km) entro cui cercare i musei.

La funzione *init()* funziona esattamente come spiegato nello scenario 1 per quanto riguardava i comuni. Una volta eseguita, all'interno del *MuseiViewModel* sarà presente un *LiveData* contenente una lista di oggetti di

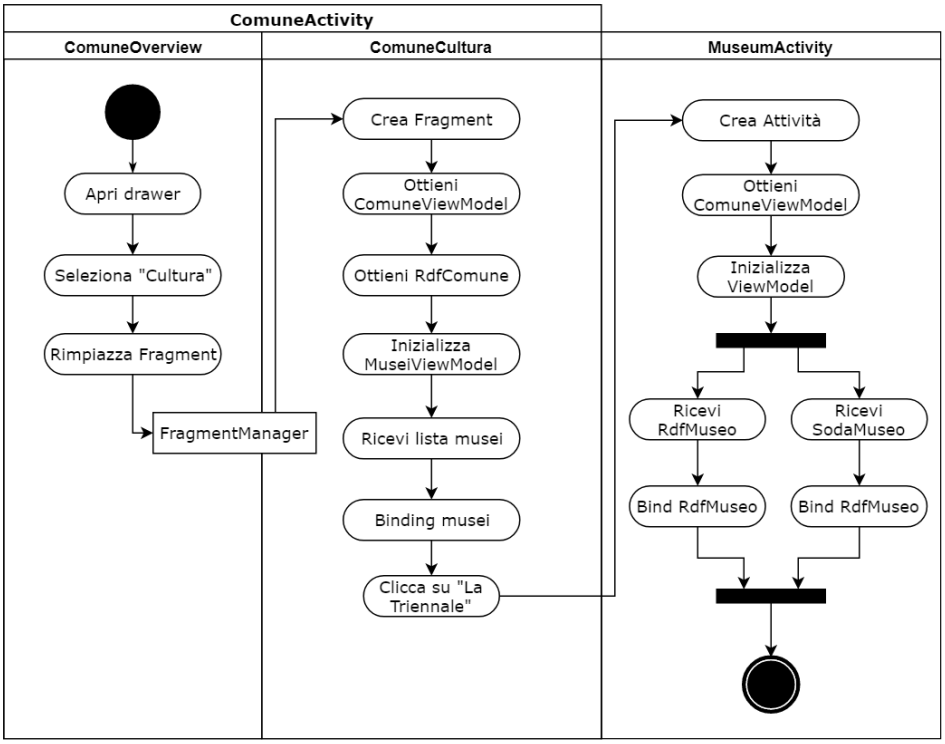


Figure 4.10: Activity diagram dello scenario 2 ad alto livello.

tipo *SodaMuseoComune*, che rappresentano le tuple della tabella ausiliaria *museo_ comune*, introdotta per sostenere la relazione *N a N* tra i musei e i comuni. *ComuneCultura* osserva quindi il *LiveData* in cui verranno depositati i dati per essere aggiornato non appena si rendono disponibili.

Poiché si è supposto che l'utente abbia recentemente visitato la pagina dedicata ai musei di Milano, la lista di essi è già presente nella cache. I dati sono quindi recuperati rapidamente: il *MuseiRepository* esegue la funzione *loadMusei()* del *MuseoDao*, che restituisce la lista e notifica il *Fragment*.

ComuneCultura riceve i dati come parametro nel metodo *onChanged* dell'*Observer* e procede subito al binding con la funzione *bindData()*. La lista di musei viene quindi ordinata per distanza dal centro geografico di Milano e passata all'*Adapter*. Per notificare l'aggiornamento dei dati al *RecyclerView*, viene chiamato il metodo *notifiyDataSetChanged()* del medesimo *Adapter*.

A questo punto l'utente è in grado di visualizzare la lista dei musei a schermo (figura 4.11) e scegliere di quale vuole conoscere maggiori infor-

Figure 4.11: Parte dela lista dei musei di Milano.

mazioni. Una volta scelto il museo desiderato (supponiamo "La Triennale di Milano"), ci clicca sopra.

Al momento del click, viene eseguita la funzione *onMuseoClick*, a cui viene passato l'oggetto *SodaMuseoComune* dedicato alla Triennale. I dati contenuti nell'oggetto, insieme all'*URI* di Milano (ottenuto dal *ComuneViewModel*), vengono passati all'*Intent* creato per lanciare la *MuseumActivity*. Infine tale *Activity* viene eseguita chiamando *startActivity()*.

Il framework di *Android* crea quindi *MuseumActivity* e chiama la funazione *onActivityCreated()*, che esegue le seguenti azioni:

1. chiede al *ViewModelProvider* un *MuseoViewModel*.
2. inizializza il *ViewModel* chiamando la funzione *init()* e passandole come parametri tutte le informazioni che l'*Activity* estrae dall'*Intent* (in seguito verrà trattato più nel dettaglio).
3. osserva i *LiveData* relativi a *RdfMuseum* e *SodaMuseo* per ricevere aggiornamenti su quando saranno disponibili.

La procedura di inizializzazione del *ViewModel* è identica a quella trattata nello scenario 1. Alla funzione *init()* vengono passati gli id del museo e della sede del museo per ottenere i dati dalla Regione Lombardia, mentre per quanto riguarda i dati *RDF*, vengono passati il nome del museo e il nome della sede (che nel caso in questione coincidono e sono "La Triennale di Milano"), il comune effettivo in cui il museo è ubicato (Milano) e l'*URI* di Milano.

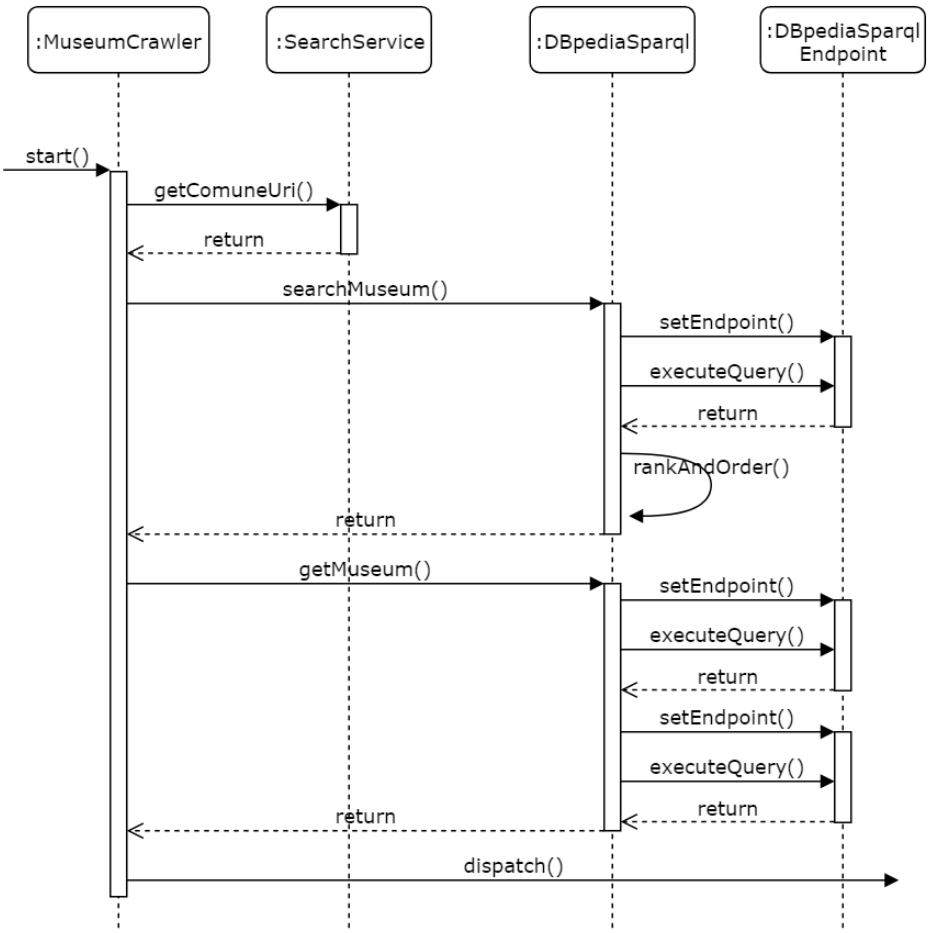


Figure 4.12: Sequence diagram del MuseumCrawler.

Come già detto, la procedura non si discosta da quella già trattata nello scenario 1: *RdfMuseum* e *SodaMuseo* vengono richiesti al *MuseiRepository*, che controlla se sono contenuti nella cache. Visto che non sono presenti, li richiede entrambi ai rispettivi *WebService*.

Per quanto riguarda *SodaComune*, viene richiesto tramite la funzione *getMuseo()*, cui vengono passati gli id del museo e della sede. All'interno del metodo viene costruita la *Query*, poi passata alla funzione *getObjects()* insieme all'oggetto *Callback* e si restituisce il *LiveData*.

Per quanto riguarda *RdfComune* invece, la chiamata al *RdfWebService* differisce rispetto allo scenario 1, come è possibile notare nel sequence diagram in figura 4.12.

La funzione dedicata al reperimento dei musei è *getMuseum()*. Quando

invocata, viene creato un oggetto di tipo *MuseumCrawler* e lanciato su un nuovo *thread*, in modo che non blocchi l'interfaccia grafica. Come solito il *Crawler* racchiude il *LiveData* in cui verrà salvato il risultato, che viene immediatamente prelevato tramite *getResult()* e ritornato.

ComuneCrawler inizia quindi l'esecuzione richiedendo l'*URI* del comune al *SearchService* tramite la funzione *getComuneUri*. L'operazione è identica a quella trattata nello scenario 1 ed è necessaria poiché il comune in cui è ubicato il museo non è necessariamente lo stesso di cui si è vista la lista musei, infatti in tale lista potrebbero apparire dei musei situati nei comuni vicini.

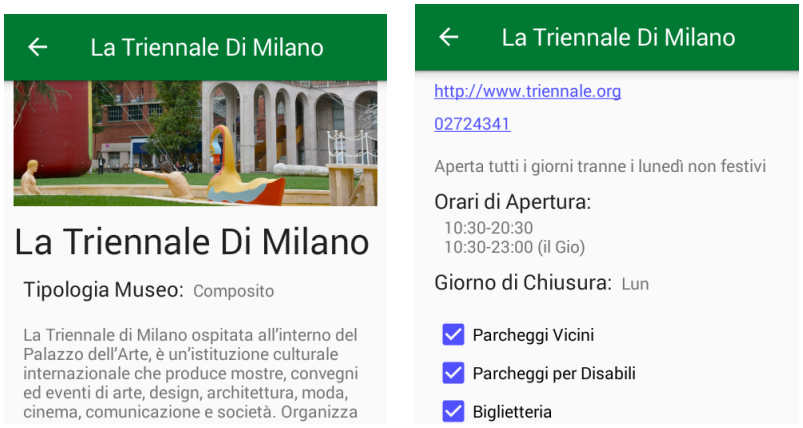
Dopo questo passaggio, è necessario ottenere l'*URI* della Triennale su *DBpedia*. Si procede quindi chiamando il metodo *searchMuseum()* del *DataServiceRdf* e passandogli il nome della sede del museo ("La Triennale di Milano") e l'*URI* appena ottenuto (che sarà quello di Milano).

La funzione *searchMuseum()* (implementata in *DBpediaSparql*) internamente assegna all'indirizzo del *DBpediaSparqlEndpoint* "dbpedia.org", costruisce la query e la esegue chiamando *executeQuery()* sull'endpoint. Il risultato della query è una lista di tutti i musei di Milano presenti su DBpedia con i relativi *URI*.

A questo punto, tramite il metodo interno *rankAndOrder*, ad ogni elemento della lista viene assegnato un punteggio di verosimiglianza, calcolato confrontando il nome del museo e "La Triennale di Milano". Il punteggio è calcolato con la *cosine similarity* [24], utilizzando *tf-idf* [25] per ottenere i vettori associati ai nomi dei musei. La lista viene poi ordinata in ordine decrescente di ranking e restituita.

MuseumCrawler riceve il risultato di *searchMuseum()* e controlla eventuali errori. Se il risultato non dovesse contenere elementi, si prova chiamare nuovamente la funzione *searchMuseum()* passando il nome del museo invece che la sede.

Una volta ottenuta la lista ordinata e non vuota, viene prelevato il primo elemento e viene confrontato il ranking di tale risultato con la soglia minima di verosimiglianza (impostata a 0.75). Se il punteggio è maggiore di tale soglia, significa che il risultato molto probabilmente è il museo che stiamo cercando, quindi viene preso l'*URI* di tale museo e passato alla funzione *DataServiceRdf.getMuseum()* che si occupa di reperire le informazioni su di esso.



(a) La foto, il nome, la tipologia e la descrizione.

(b) I contatti, gli orari e altre informazioni.

Figure 4.13: Screenshot della MuseumActivity che mostrano i dettagli della Triennale di Milano.

Il metodo *getMuseum()* funzione analogamente a *getComune()*, trattata nello scenario 1: cerca informazioni sulla Triennale sia su "dbpedia.org" che su "it.dbpedia.org", utilizza un *Adapter* per inserire i dati all'interno di un oggetto di tipo *RdfMuseum* ed infine lo ritorna.

Il *Crawler*, ricevuto l'oggetto, esegue il *dispatching*, anch'esso già trattato nello scenario 1 sia per *RdfMuseum* che per *SodaMuseo*.

MuseumActivity riceve quindi la notifica della presenza dei dati quando viene chiamata la funzione *onChanged()*. Controlla che non siano presenti errori ed esegue il binding dell'oggetto tramite il metodo *bindData()*, cui passa l'oggetto ricevuto. Tale funzione si occupa poi di eseguire il binding di tutti i dati contenuti nell'oggetto con il relativo componente dell'interfaccia grafica.

L'utente può ora visualizzare tutte le informazioni sulla Triennale, come mostrato in figura 4.13.

Capitolo 5

Conclusioni

La tesi ha avuto come scopo quello di dimostrare la fattibilità di integrare dati strutturalmente diversi come *Linked Data* e database relazionali. Nonostante alcuni limiti dovuti alla piattaforma scelta (es. mancanza di librerie dedicate ad *RDF*) l'obiettivo è stato raggiunto e il risultato può considerarsi soddisfacente: l'integrazione è riuscita e l'utente che utilizza l'applicazione non è in grado di notare la differenza di provenienza delle informazioni.

DiscoverLombardia è un esempio di come tale integrazione possa essere utile non solo dal punto di vista teorico o tecnologico: può essere sfruttata nella vita di tutti i giorni e portare benefici a chiunque. L'applicazione infatti promuove la cultura e la conoscenza del territorio ed è sviluppata su una piattaforma che al giorno d'oggi è accessibile a tutti.

5.1 Limitazioni

Durante l'impostazione e lo svolgimento del progetto, sono emerse varie limitazioni che ne hanno ostacolato lo sviluppo:

- **La mancanza di dati**: questo problema riguarda principalmente i dati in formato *RDF*. Purtroppo le fonti che forniscono informazioni in tale formato sono ancora fin troppo poche, oppure, anche se presenti, sono inattive o non aggiornate. A questo problema si aggiunge il fatto che, come nel caso di *DBpedia*, anche se le informazioni sono presenti in gran numero, non sono ancora abbastanza, soprattutto per quanto riguarda il territorio italiano. Basti pensare che la maggior parte dei musei considerati in questo progetto non ha una pagina su *Wikipedia* (e

quindi non è presente su *DBpedia*) e non esiste una fonte dedicata che contenga informazioni in formato *RDF* sui musei. Per quanto riguarda gli *Open Data* delle pubbliche amministrazioni italiane, non ne sono ancora stati trovati molti disponibili online e spesso sono frammentati fino ad arrivare a livello comunale.

- **La casualità dei dati**: anche questo problema riguarda principalmente i dati *RDF*, ma si è riscontrato anche con quelli della Regione Lombardia. Troppo spesso le informazioni sono incomplete, duplicate, errate o malformate in modo imprevedibile. Soprattutto per quanto riguarda *DBpedia*: non ci sono dati garantiti, presenti in ogni risorsa e con un certo formato. Un esempio lampante è il nome di una risorsa: a volte è ottenibile tramite rdfs:label, altre volte con foaf:name, oppure addirittura con entrambi o nessuno dei due. Può darsi che i nomi siano uguali oppure no. A volte un singolo *predicate* è legato a più di un nome (es. Milano e Comune di Milano). Spesso esistono *predicate* identici ma con *URI* differenti (es. dbo:deathPlace vs. dbp:deathPlace), oppure non è prevedibile il tipo dell'oggetto che verrà ottenuto: un *literal* oppure un *URI*. Infine può capitare che le informazioni ci siano, ma non essendo collegate tra loro tramite opportuni predicati si perde l'essenza dei *Linked Data*.

- **La disomogeneità dei dati**: benché i *Linked Data* soffrano dei problemi indicati ai punti precedenti, hanno il grande vantaggio di essere fruibili in modo universale tramite il linguaggio *SPARQL*. Ciò non si può dire per i dati della Regione Lombardia, che risultano spezzettati in diversi dataset, spesso anche se riguardanti lo stesso argomento (es. Musei Lombardi e Musei in Provincia di Monza e Brianza). Per integrarli occorre quindi trovare soluzioni ad hoc che considerino i singoli casi. Anche i nomi delle colonne non sono standardizzati, rendendo difficile eseguire eventuali *join* tra tabelle generiche. Se poi si volesse estendere l'ambito dell'app da regionale a nazionale, le differenze di formati e convenzioni aumenterebbe ulteriormente.

5.2 Sviluppi Futuri

L'applicazione potrebbe essere ulteriormente sviluppata, aggiungendo funzionalità o migliorando quelle già presenti. Ad esempio si potrebbero inte-

grare altri dataset dal sito della Regione Lombardia, come mercati e sagre paesane, agriturismi, chiese ed edifici, impianti di risalita ecc. tutte informazioni che arricchirebbero la *user experience* e la conoscenza del territorio, ma che non è stato possibile introdurre sia per motivi di tempo, sia perché dal punto di vista dell'integrazione non avrebbero aggiunto nulla di rilevante. Lo stesso discorso vale per *DBpedia*, in quanto contiene molte altre informazioni da poter estrarre.

Potrebbe anche essere interessante integrare nuove fonti di dati come i social network: notizie, opinioni, foto sui luoghi e sui comuni presenti nell'applicazione.

Sarebbe anche molto utile inserire strumenti di intelligenza artificiale all'interno dell'applicazione: per esempio eseguendo del semplice *text mining* per estrarre informazioni o descrizioni da siti web generici (es. ottenere i punti di interesse di un comune dalla pagina di *Wikipedia*), oppure riconoscere che due risorse siano la stessa entità e collegarle tra loro "semanticamente" senza l'ausilio di un essere umano (es. riconoscere che la pagina *Wikipedia* di un museo e la relativa pagina *Facebook* si riferiscono allo stesso museo e integrare entrambe le informazioni nell'app).

Bibliografia

[1] (2018, Mar.) Model–view–viewmodel. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Model-view-viewmodel

[2] Microsoft, Ed., *Patterns - WPF Apps With The Model-View-ViewModel Design Pattern*, ser. MSDN Magazine Blog, vol. 24, no. 02, Microsoft, Feb. 2009, written by Josh Smith. [Online]. Available: https://msdn.microsoft.com/en-us/magazine/dd419663.aspx

[3] Guide to app architecture. Google. [Online]. Available: https://developer.android.com/topic/libraries/architecture/guide.html

[4] Viewmodel. Google. [Online]. Available: https://developer.android.com/topic/libraries/architecture/viewmodel.html

[5] (2018, Mar.) Android software development. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Android_software_development

[6] (2018, Mar.) Android (operating system). Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Android_(operating_system)

[7] Android architecture components. Google. [Online]. Available: https://developer.android.com/topic/libraries/architecture/index.html

[8] (2017, Jul.) Geojson pojos for jackson. OpenDataLab. [Online]. Available: https://github.com/opendatalab-de/geojson-jackson

[9] (2017, Dec.) Commons text. Apache. [Online]. Available: https://commons.apache.org/proper/commons-text/

[10] (2018, Mar.) Glide. Bumptech. [Online]. Available: https://github.com/bumptech/glide

[11] (2018, Mar.) Open data lombardia. Regione Lombardia. [Online]. Available: https://dati.lombardia.it/

[12] (2015, Mar.) Soda android sdk. Socrata. [Online]. Available: https://github.com/socrata/soda-android-sdk

[13] (2017, Nov.) Dbpedia. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/DBpedia

[14] Virtuoso sparql query editor. DBpedia. [Online]. Available: https://dbpedia.org/sparql

[15] Dbpedia italia. DBpedia Italia. [Online]. Available: http://it.dbpedia.org/sparql

[16] (2015, Oct.) Java client for geonames webservices. GeoNames. [Online]. Available: http://www.geonames.org/source-code/

[17] Introduction to activities. Google. [Online]. Available: https://developer.android.com/guide/components/activities/intro-activities.html

[18] Fragments. Google. [Online]. Available: https://developer.android.com/guide/components/fragments.html

[19] Livedata. Google. [Online]. Available: https://developer.android.com/topic/libraries/architecture/livedata.html

[20] Single source of truth. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Single_source_of_truth

[21] Room persistence library. Google. [Online]. Available: https://developer.android.com/topic/libraries/architecture/room.html

[22] Save data in a local database using room. Google. [Online]. Available: https://developer.android.com/training/data-storage/room/index.html

[23] Dbpedia lookup. DBpedia. [Online]. Available: http://wiki.dbpedia.org/projects/dbpedia-lookup

[24] Cosine similarity. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Cosine_similarity

[25] tf-idf. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Tf-idf