UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

ShallWeGo: AI-assisted mobility crowdsourcing platform

RELATORE

Prof. **Fabio Palomba**

Università degli studi di Salerno

CANDIDATO

Hermann Senatore

Matricola: 0512105743

Anno Accademico 2020-2021

Il mondo è come un libro e chi non viaggia ne conosce una pagina soltanto.

(Sant'Agostino)

Sommario

La Tesi sviluppata si posiziona nell'ambito della creazione di strumenti informatici per assistere gli utenti del Trasporto Pubblico Locale nella loro esperienza quotidiana. L'obiettivo principale di questa Tesi consiste nello sviluppo di un'applicativo mobile che permetta l'accesso ad una piattaforma online chiamata ShallWeGo basata sulla collaborazione tra utenti che ha come scopo quello di consentire lo scambio di informazioni sull'organizzazione del Trasporto Pubblico Locale in una determinata zona. Il vero motore di ShallWeGo è quindi il singolo utente, che può mettere a disposizione la propria conoscenza sul topic in questione (organizzazione delle fermate sul territorio, aziende di trasporto e linee espletate da queste ultime) agli altri utenti, che quindi possono sopperire alle potenziali difficoltà di comunicazione da parte delle aziende di trasporto. Particolare attenzione deve essere dedicata tuttavia all'affidabilità di queste segnalazioni che devono essere controllate in qualche modo. È stato quindi previsto, tramite la definizione di un Agente Intelligente, un sistema di verifica delle segnalazioni anch'esso basato sulla partecipazione attiva degli utenti della piattaforma. Data una segnalazione sarà quindi possibile andare a determinare il gruppo di utenti che secondo determinate "metriche" risultino più adatti a verificarla e stabilire se quest'ultima possa essere integrata o meno sulla piattaforma.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Contesto applicativo e Motivazioni

1.1.1 Lo stato del Trasporto Pubblico Locale

Negli ultimi due anni, complici anche gli avvenimenti che stanno interessando il mondo ed i successivi provvedimenti, le abitudini dei cittadini in tema di mobilità sono drasticamente cambiate. Tra i numerosissimi settori che sono interessati da questa ondata di cambiamenti ce n'è uno in particolare che ne ha risentito particolarmente: quello del Trasporto Pubblico Locale (**TPL**) che, secondo un rapporto stilato da varie associazioni di rappresentanza di aziende di trasporto presentato al Parlamento Italiano all'inizio del 2021 (Asstra [2021]) e ripreso dal quotidiano "Il Sole 24 Ore" in data 25 Gennaio dello stesso anno De Girolamo [2021] ha visto un crollo dei ricavi dell'entità dei Miliardi di Euro. Nello specifico, dal crollo della fruizione dei mezzi pubblici si stima che nelle casse delle aziende siano entrati complessivamente 250 milioni di Euro in meno al mese, portando le perdite complessive a circa 2 Miliardi. Tutto ciò, come anticipato, è causato sostanzialmente da un crollo delle presenza da parte dei cittadini su autobus, treni e metropolitane in un primo momento dovuto alle restrizioni imposte, almeno nel caso dell'Italia, nella prima metà del 2020 e complice la percezione da parte del cittadino dei mezzi pubblici come "poco sicuri" dal punto di vista sanitario risultando, continua il rapporto, in un calo del 90% della domanda. Questi due fattori (crollo della domanda e conseguente diminuzione delle entrate) hanno generato in molti casi conseguenze pesanti sullo "stato di salute" delle aziende del settore, con successive

rimodulazioni orarie e una sostanziale diminuzione delle corse, specie nelle prime e nelle ultime ore della giornata. La diminuzione delle corse inevitabilmente apre la porta ad uno dei problemi principali che sta affliggendo il settore nell'ultimo periodo: l'affollamento delle corse stesse.

1.1.2 Tecnologia e TPL

Con la sempre più capillare diffusione della tecnologia nelle mani del cittadino da un decennio a questa parte, i principali attori che operano in questo campo hanno riconosciuto l'importanza di andare a creare un sistema di informazione puntuale sul funzionamento del TPL in una determinata zona. Piattaforme come Google Maps, ad esempio, già da tempo forniscono dettagli sulla presenza di fermate e dettagli delle corse delle aziende di trasporto che operano in quella zona. Nel caso particolare di Google Maps, questi dati, secondo la documentazione ufficiale del servizio "Google Transit" disponibile all'indirizzo `https://support.google.com/transitpartners/answer/1111471` sono ricavati principalmente dalle comunicazioni che giungono alla piattaforma da parte delle singole aziende che decidono di partecipare al programma, avendo poi l'opzione di condividere anche dati in tempo reale che permettano agli utenti di seguire in diretta l'andamento delle corse delle linee che gli interessano, tramite la piattaforma "Realtime Transit", che comunque è riservato alle aziende di trasporto.

1.1.3 Problemi delle attuali soluzioni

Come accennato in precedenza, i dati che sono disponibili sulle piattaforme più diffuse sono comunicati periodicamente dalle aziende che aderiscono a programmi come il sopracitato Google Transit su Maps. Tuttavia, questo approccio non è esente da problemi di natura pratica: la presenza dei dati disponibili sulle varie piattaforme risulta quindi, nella maggior parte dei casi, essere subordinata all'adesione delle varie aziende alle stesse. Se questo problema non si pone per le aziende di medie/grandi dimensioni che hanno a disposizione risorse economiche e tecnologiche, lo stesso non si può dire per le aziende più piccole a carattere locale, che potrebbero avere difficoltà di carattere logistico anche solo nel posizionare gli adeguati segnali di riconoscimento delle fermate che utilizzano. Ne consegue quindi che gli utenti occasionali di queste aziende di trasporto (si pensi ad esempio ad una persona che si trova per la prima volta in una determinata zona) siano costretti a cercare informazioni presso altri cittadini che "ne sanno più di loro". Una piattaforma che ha provato a porre rime-

dio a questo problema è Moovit che ha lanciato nel 2015 un servizio chiamato "Mooviters' Community" che permette agli utenti di andare ad aggiungere dettagli alla piattaforma stessa che non siano stati già comunicati alle aziende. Questo approccio verrà trattato nel capitolo successivo dedicato proprio alle piattaforme presenti ad oggi sul mercato.

1.2 Obiettivi della tesi

L'obiettivo di questo lavoro è quindi proporre un primo esempio di soluzione ai due problemi di cui si è trattato in precedenza: creare una piattaforma, chiamata **ShallWeGo**, che metta a disposizione dei suoi membri gli strumenti adatti a diffondere la propria conoscenza in merito di organizzazione di linee e fermate nella loro zona con gli altri utenti, formando quindi una community che vede nella reciproca collaborazione il principale mezzo per raggiungere una vera e propria utilità per il singolo cittadino.

1.3 Metodologie e risultati

1.3.1 Approccio allo sviluppo della piattaforma

Lo sviluppo della piattaforma è cominciato, così come accade in tutti i progetti che prevedano la presenza di un software, con la raccolta e l'analisi dei requisiti, ovvero su *cosa* dovesse fare la piattaforma. Una volta raccolti questi requisiti si è provveduto a categorizzarli in "prioritari" e "non prioritari". (Quelli individuati come prioritari sono stati elencati nella prima sezione del quarto capitolo, dedicata nello specifico ai **requisiti funzionali**.) Ciò ha permesso di rendere chiaro sin da subito cosa risultasse più urgente realizzare per ottenere una prima versione funzionante della piattaforma. Successivamente, si sono analizzate le varie strategie per lo sviluppo effettivo del software. Vista anche la distinzione tra requisiti "prioritari" e "non prioritari" appare palese come la strategia di sviluppo che più si adatta a questo contesto risulta quella **incrementale**. Lo sviluppo incrementale, nello specifico, consiste innanzitutto di implementare le funzionalità *core* della piattaforma e, successivamente in modo *incrementale* (e da qui la denominazione) tutte le varie funzionalità che vanno ad implementare i requisiti individuati in fase di analisi.

Le funzionalità *core* includono due componenti particolari che non sono direttamente collegate con il software ma che sono necessarie per il corretto funzionamento dello stesso. Queste due componenti sono:

- Un server esterno per il geocoding, basato sull'applicativo Nominatim (per permettere un accesso semplice e veloce a dati geografici)
- Un modulo di intelligenza artificiale che permette la verifica di dati ottenuti dagli utenti.

Per quanto riguarda la componente descritta al primo punti, trattandosi di un software già realizzato e reso disponibile al pubblico tramite licenza ***GNU Public License* 2.0**, non si può parlare di sviluppo ma di *configurazione* o di *tuning* per migliorare le prestazioni sulla macchina che lo ospita.

1.3.2 Perché l'Intelligenza Artificiale?

La componente di Intelligenza Artificiale descritta al secondo punto, invece, è stata quella verso la quale è stata posta la maggior parte dell'attenzione delle prime fasi di sviluppo. Ciò è giustificato dalla natura particolare della piattaforma, che permette agli utenti di essere la fonte di informazioni principale. Sorge quindi la necessità di prevedere un sistema molto veloce per la verifica (da parte di altri utenti, essendo la piattaforma *Community-driven*) della correttezza delle segnalazioni che pervengono alla piattaforma. Un primo approccio a questo problema potrebbe essere rappresentato dalla semplice ricerca esaustiva all'interno dell'insieme degli utenti. Quest'approccio risulta accettabile nel caso si consideri un dominio di utenti discretamente piccolo ma cessa di esserlo al crescere del numero di questi ultimi. È proprio in questo contesto che una tecnica di Intelligenza Artificiale si rivela essere adeguata: quella degli **Algoritmi Genetici**. L'idea che sta alla base degli Algoritmi Genetici ricalca a grandi linee (come d'altronde suggerisce il nome) quello dell'evoluzione di una popolazione di individui (un insieme di soluzioni ad un problema) descritta per la prima volta da Mendel nel 1859 nella sua opera "L'Origine della Specie", che si compone di tre "tappe" principali, ripetute moltissime volte:

- **Selezione**
- **Crossover** o **Accoppiamento** degli individui
- **Mutazione** degli individui

Questo processo permette di ottenere, con il susseguirsi delle *generazioni*, una popolazione composta da individui molto forti che si adattano ai vari cambiamenti nell'ambiente in cui si trovano.

Allo stesso modo, a partire da un insieme iniziale di soluzioni ad un problema, tramite un Algoritmo Genetico è possibile ottenere in maniera molto veloce (come risulta necessario nel caso della piattaforma in sviluppo) delle soluzioni ammissibili che risultino almeno "buone" per il problema che si sta affrontando. Non trattandosi di una ricerca esaustiva, tuttavia, non è garantito il raggiungimento della soluzione ottima.

Il funzionamento dell'Algoritmo Genetico sviluppato per ShallWeGo verrà descritto in dettaglio nel **Terzo Capitolo** che è dedicato interamente a questo *topic*.

1.3.3 Il problema delle API

L'applicazione oggetto di questo lavoro di Tesi fa largo uso di mappe e dati geografici in generale. Google Maps mette a disposizione le sue API in maniera gratuita fino ad esaurimento di un credito (che viene scalato man mano che si utilizzano i propri servizi) che si rinnova ogni mese. Superata questa soglia, è necessario sostenere un certo prezzo. Le tariffe, consultabili all'indirizzo `https://cloud.google.com/maps-platform/pricing`, prevedono infatti la possibilità di usufruire un credito gratuito dell'entità di 200$ mensili. In particolare, il servizio di Geocoding [1] costa 5$ ogni 1000 richieste. Il prezzo, quindi, aumenta all'aumentare delle richieste che vengono effettuate da chi utilizza l'applicazione. Sono tuttavia disponibili delle alternative di terze parti che permettono l'accesso gratuito a degli endpoint per effettuare le operazioni di Geocoding. Uno dei migliori servizi di questo genere è rappresentato da **Nominatim** che è utilizzato, tra l'altro, anche dal progetto **OpenStreetMap** per implementare la feature di ricerca nella loro piattaforma. Il funzionamento di Nominatim verrà descritto in dettaglio nel **Quarto Capitolo** della Tesi che tratta l'architettura ed il funzionamento di ShallWeGo.

1.3.4 Risultati ottenuti

Il risultato di questa esperienza di Tesi è stato quindi quello di aver sviluppato una prima versione di un applicativo mobile e della relativa componente server che implementasse l'idea di cui si è discusso. L'applicativo consiste quindi in un semplice file .apk per Android installabile e che, attualmente, previa registrazione ed ottenimento di uno username, permetta a chi vuole di cominciare ad effettuare segnalazioni riguardanti fermate, linee ed aziende di trasporti che conosce e che sa operare in una determinata zona. È, inoltre, in sviluppo

[1] Ovvero il processo che consente di ottenere il nome logico di un luogo associato ad una coppia di coordinate (Latitudine, Longitudine) e viceversa

una funzionalità che preveda la possibilità da parte di un utente di comunicare in tempo reale la propria posizione e, contemporaneamente, specificare su che linea si trova, così da permettere ad altri utenti interessati di farsi un'idea di quanto debbano aspettare per salire a bordo.

1.4 Struttura della tesi

La tesi è strutturata principalmente in cinque parti, che coprono gli aspetti principali del lavoro svolto, con un'attenzione particolare alla metodologia utilizzata per implementare in modo efficiente la valutazione delle segnalazioni che pervengono alla piattaforma da parte degli utenti.
In particolare:

- Il Primo Capitolo funge da introduzione al lavoro svolto per realizzare la piattaforma
- Il Secondo Capitolo descrive le principali soluzioni che sono attualmente utilizzabili dagli utenti per quanto riguarda il dominio del problema trattato dal lavoro di Tesi.
- Il Terzo Capitolo descrive accuratamente l'approccio usato per la valutazione delle segnalazioni degli utenti, che fa uso di una tecnica di **Intelligenza Artificiale** che permette di selezionare gli utenti più adatti a valutare una determinata segnalazione
- Il Quarto Capitolo, invece, illustra l'architettura su cui si basa l'applicazione e i dettagli implementatiti in termini di Framework, Linguaggi di Programmazione e Librerie utilizzate per lo sviluppo dell'applicativo Mobile.
- Infine, il Quinto Capitolo riassume tutti gli sviluppi futuri che permetterebbero all'applicazione di uscire dallo stato di "demo" ed essere quindi messa a disposizione del pubblico.

CAPITOLO 2

Stato dell'arte

Lo scopo di questo capitolo è di illustrare la situazione attuale dei vari applicativi che sono a disposizione degli utenti del trasporto pubblico locale (TPL) per permettergli di organizzare i loro spostamenti.

I principali attori sul panorama della diffusione di informazioni sul TPL sono senz'altro la piattaforma Maps di Google ed in particolare Moovit, che ha acquisito una maggiore popolarità nell'ultimo anno.

2.1 Google Maps

Transit è un servizio messo a disposizione da Google ed integrato nella piattaforma Maps che consente all'utente di visualizzare le informazioni *statiche* messe a disposizione dalle varie aziende di trasporto tramite il programma "Google Transit Partners", che quindi consistono nella programmazione giornaliera delle corse e dell'organizzazione delle fermate sul territorio in questione.
Oltre ai dati statici, le aziende di trasporto possono condividere con Google mediante la piattaforma **Realtime Transit** che quindi permette di visualizzare all'interno di Maps i dati in tempo reale sulla posizione dei mezzi pubblici in un determinato istante. Questa funzionalità, così come tutti i dati relativi al trasporto pubblico su Maps sono disponibili solo su iniziativa delle aziende che decidono di partecipare al programma. Un esempio di azienda che mette a disposizione i dati realtime è per esempio l'ATAC di Roma, che li offre a partire da Settembre 2019 (Adnkronos [2019])
I dati (siano essi statici o dinamici) sono comunicati a Google su base regolare dalle aziende (tipicamente settimanale, secondo quanto riportato nella sezione FAQ della pagina dedicata [3]) tramite il formato GTFS *(General Transit Feed Specification)* che permette di strutturare in maniera efficiente i dati riguardanti il trasporto pubblico.

Esistono due "varianti" del formato GTFS:

- **GTFS Statico**, che raccoglie i dati "statici" (quindi gli orari previsti delle corse e l'organizzazione delle fermate sul territorio)
- **GTFS Realtime**, che raccoglie i dati "dinamici" (come l'andamento delle corse in tempo reale)

Tuttavia, come accennato in precedenza, questi dati sono comunicati su richiesta delle aziende e solo un numero limitato di queste ultime (almeno in Italia) aderisce al programma, limitando così i dati a disposizione dell'utente.

[3] `https://developers.google.com/transit/gtfs/guides/faq`

2.2 Mobilità e crowdsourcing

La tecnica del crowdsourcing, applicabile nella maggior parte dei casi in cui ci sia bisogno di ottenere dati di interesse per un determinato dominio consiste nel ricavare direttamente questi ultimi da persone sul campo e che ritengano di possedere un'informazione che potrebbe tornare utile alla comunità. Anche nel dominio di interesse della piattaforma che si vuole sviluppare, è possibile andare ad isolare i dati di interesse (quelli utili cioè a fornire un sufficiente livello di informazione agli utenti del TPL). Possono essere riassunti in cinque punti:

- Posizione di una fermata in un determinato punto
- "Equipaggiamento" di una fermata (in termini di pensilina, segni identificativi e quadri orari disponibili al pubblico)
- Utilizzo di una fermata da parte di una determinata linea
- Destinazioni di una linea
- Eventi transitori (come presenza di traffico, avvisi su deviazioni o strade chiuse)

L'approccio del crowdsourcing è già utilizzato sia in Google Maps (di cui si è appena parlato) sia in un'altra piattaforma che risulta di particolare interesse, ovvero **Moovit**. Moovit sfrutta l'idea del crowdsourcing per ottenere dati altrimenti molto difficili da reperire.

2.2.1 La Community di Moovit: un caso di studio

Moovit è una piattaforma che al contrario di Google Maps si occupa esclusivamente di mobilità. Nasce nel 2012 e nel 2015 viene acquistata da Intel. Attualmente Moovit si sta impegnando per sviluppare soluzioni di Mobility as a Service. Il Mobility as a Service (MaaS) è un concetto relativamente nuovo, che permette all'utente di scegliere il modo che ritiene più adatto di spostarsi (tramite mezzi pubblici, car sharing e simili) usando una sola piattaforma che mette a disposizione tutti questi servizi, favorendo quindi l'interoperabilità tra i vari mezzi di trasporto.

Dal 2015, inoltre, Moovit mette a disposizione un servizio chiamato **Mooviter Community**, che permette al singolo utente della piattaforma di partecipare alla mappatura delle fermate e delle linee della sua zona tramite un Editor accessibile via web. I cambiamenti

dovranno quindi essere approvati dagli amministratori della piattaforma o da utenti esperti.

In particolare, la piattaforma di Moovit organizza gli utenti in "livelli", che potranno essere "scalati" man mano che si matura esperienza in termini di segnalazioni. Nello specifico, se un utente x di livello n effettua una segnalazione, chi potrà verificare questa segnalazione sarà solamente un certo utente z il cui livello risulta almeno $n + 1$ o che sia membro del Team. Il problema di questo approccio è che se un utente con molta esperienza si unisce alla piattaforma in un certo momento, egli non potrà immediatamente cominciare a verificare segnalazioni poiché inizialmente il suo livello sarà troppo basso.
Attualmente, l'Editor della Community di Moovit è disponibile in tutte le sue funzionalità esclusivamente via web tramite un browser che riporta uno *user-agent* desktop.

Oltre all'editor delle fermate e delle linee, nell'ultimo periodo Moovit ha messo a disposizione direttamente dall'app mobile uno strumento che permette di segnalare anche l'affollamento di una determinata corsa, come riportato sul sito web della piattaforma.[4]

[4] `https://moovit.com/press-releases/moovit-crowding-report/`

2.3 Il focus di ShallWeGo

La piattaforma ShallWeGo si basa principalmente sul modello delineato da Moovit con la sua Community rendendo tuttavia i dati ottenuti direttamente dagli utenti la fonte principale di informazioni, fornendo gli strumenti adatti per rendere disponibili i dettagli sulle aziende, sulle corse e sulle fermate in una determinata zona tramite un'applicazione mobile che ne semplifichi l'accessibilità anche all'esterno in quanto, come accennato in precedenza, la piattaforma di Moovit risulta principalmente web-based, separata dall'applicazione principale che è più conosciuta al grande pubblico.
Inoltre, a differenza della Community di Moovit, in ShallWeGo le segnalazioni non sono limitate solamente a dettagli statici ma si estendono anche ad eventi "dinamici" (come strade chiuse, deviazioni, traffico o incidenti), ai dettagli di una singola fermata (come l'affollamento, la presenza di pensiline, di paline di riconoscimento delle aziende che la utilizzano o dei quadri orari) ed all'andamento delle corse, sfruttando il dispositivo dell'utente, permettendo quindi ad altri membri della community di verificare la presenza e la posizione di una corsa nell'ambito di una linea, in modo simile a quanto avviene con Transit Realtime di Google, ma con la differenza che non risulta necessario che le aziende condividano i dati. La feature presente in ShallWeGo permette anche di condividere informazioni sulla corsa stessa quali l'affollamento, la temperatura o le condizioni del mezzo in quel momento (come ad esempio lo stato di funzionamento delle obliteratrici o dell'aria condizionata)
Infine, dal punto di vista della verifica delle segnalazioni, ShallWeGo non possiede un team che si dedica a valutare le segnalazioni, ma al contrario i verificatori più adatti vengono scelti mediante una componente di Intelligenza Artificiale creata proprio a questo scopo ed integrata nell'infrastruttura dell'applicazione. In particolare, l'Algoritmo per la scelta degli utenti si basa, oltre che sull'esperienza accumulata sulla piattaforma da parte di un singolo (in termini di segnalazioni effettuate e valutate) anche sulla distanza geografica dell'area in cui egli opera dal luogo della segnalazione. In questo modo, anche i nuovi iscritti alla piattaforma avranno la possibilità di valutare le segnalazioni inviate alla piattaforma. I dettagli sul funzionamento della componente di Intelligenza Artificiale saranno discussi nel capitolo successivo.

CAPITOLO 3

Background sull'Intelligenza Artificiale utilizzata

Lo scopo di questo capitolo è di illustrare in dettaglio la componente di Intelligenza Artificiale che sta alla base del funzionamento della piattaforma. In particolare, si descrive l'idea generale che sta alla base degli Algoritmi Genetici e viene proposta una panoramica delle operazioni tipiche di questo paradigma.

3.1 Descrizione del problema da affrontare

Come descritto in precedenza, la fonte dei dati disponibili in ShallWeGo risiede nella sua community di utenti, che mettono a disposizione la loro conoscenza del *topic* con gli altri. Il concetto base su cui si basa la piattaforma è quindi la **segnalazione** da parte del singolo. Di questi dati, non essendo forniti direttamente dalle aziende di trasporto, non è però garantita la precisione o addirittura la correttezza. Sorge quindi il bisogno di effettuare una qualche tipo di **validazione**.
Le persone più adatte a validare i dati di una certa segnalazione sono quelle che possiedono principalmente i seguenti requisiti:

- Risiedano (oppure operino abitualmente) in una zona vicina al luogo oggetto della segnalazione.
- Abbiano una certa "reputazione" all'interno della piattaforma. (ShallWeGo infatti tiene traccia del livello di attività in termini di segnalazioni e di verifica delle stesse. Questo livello, similmente a quanto avviene sul social network Reddit, viene chiamato "*karma*" e cresce all'aumentare dell'attività del singolo)

I "verificatori" sono assegnati all'atto dell'invio di una segnalazione da parte dell'utente. Se ci si pone nello scenario in cui la piattaforma cresce in termini di numero di utenti, una ricerca esaustiva all'interno dell'insieme degli iscritti per assegnare quelli più adatti risulta essere problematica in termini di complessità. Si è quindi deciso di sfruttare una tecnica particolare che permetta di evitarla: quella degli **Algoritmi Genetici**, particolarmente utili alla ricerca in domini molto grandi.

3.2 Algoritmi genetici

Un algoritmo genetico non consiste tanto in una tipologia vera e propria di algoritmi quanto in un paradigma (una **metaeuristica**) che consente di implementare un algoritmo di **ricerca** per risolvere problemi di ottimizzazione. In quanto metaeuristica, il paradigma che sta alla base degli algoritmi generici non garantisce di trovare la soluzione ottima nello spazio di ricerca che si sta considerano ma permette di trovare in maniera molto veloce (e in questo risiede uno dei suoi vantaggi che lo fanno preferire ad altri approcci) delle soluzioni che vi si avvicinano. Un altro vantaggio che rende molto utile l'impiego di algoritmi che si basano su questa metaeuristica, consiste nel fatto che essi siano sempre applicabili, a prescindere dalla

struttura del problema.
In particolare, ad un algoritmo che sfrutta questa metaeuristica viene presentato un insieme iniziale di soluzioni ammissibili al problema che si sta affrontano e, secondo determinate metriche che forniscono una stima di quanto *"buona"* sia una certa soluzione a quel determinato problema, stabilire la migliore (o le migliori), andando a crearne di nuove se necessario.

3.3 Struttura e funzionamento degli algoritmi genetici

In generale, il funzionamento degli algoritmi genetici (e da qui la denominazione del paradigma) ricalca a grandi linee quella dell'evoluzione delle specie descritte da Charles Darwin nel 1854 nella sua opera *"L'origine della specie"* per cui l'evoluzione delle specie presenti in natura segue un cammino ben preciso, che consiste in tre fasi ben precise, a partire da una popolazione iniziale di individui:

- Selezione naturale
- Accoppiamento tra individui (detto anche, nel contesto che si sta trattando, *crossover*)
- Mutazione di un gene di un individuo

Un individuo, secondo quanto descritto da Darwin, può essere visto come un insieme di caratteristiche (o *geni*) che ne definiscono l'identità. Mediante l'accoppiamento i geni di due individui si mescolano, risultando nella creazione di un altro individuo che possiederà quindi una combinazione di quelli dei suoi genitori.
Con una certa probabilità, inoltre, avviene il fenomeno della mutazione, descritto poc'anzi.

Infine, sulla base di quanto buone siano le caratteristiche di un individuo, esso potrà "sopravvivere" ed eventualmente riprodursi per formare nuovi individui o "morire", non propagando oltre i suoi geni, che evidentemente non erano abbastanza adatti all'ambiente in cui si trovava. Il processo quindi risulta *iterativo*.

La strategia utilizzata da un algoritmo che implementa questo paradigma è particolarmente simile a quella descritta da Darwin per il suo campo di studi.

Portando avanti la similitudine con quanto avviene in natura, quando si vuole affrontare un dato problema (che chiameremo x) mediante la tecnica degli algoritmi genetici innanzitutto vengono generate in maniera più o meno casuale una serie di soluzioni ammissibili per x, composte da differenti caratteristiche (i *geni*). Questo insieme di soluzioni al problema rappresenta la cosiddetta **popolazione iniziale**. A partire da quest'ultima, viene solitamente

calcolato l'indice di "bontà" di ogni singola soluzione (che in gergo viene chiamata **fitness**), tramite una funzione apposita, chiamata appunto ***funzione di Fitness***. Una volta valutata questa quantità è necessario stabilire quanti e quali individui possano essere ammessi a riprodursi.
Per questa ragione, a questo stadio si affronta solitamente la fase di **selezione**, che consiste nel far sopravvivere solamente gli individui migliori in termini di fitness. Gli individui che sopravvivono alla selezione sono ammessi al cosiddetto **mating pool**, ovvero saranno membri dell'insieme degli individui che si possono riprodurre.
Una volta effettuato l'accoppiamento, la popolazione risultante è rappresentata da un nuovo insieme di individui che, come accennato in precedenza, posseggono una parte dei geni del primo genitore ed una parte dei geni del secondo, mutuandone quindi le loro caratteristiche.
Si passa poi all'ultimo stadio dell'evoluzione, in cui avviene (con una certa probabilità) la **mutazione** di uno o più geni che compongono un individuo all'interno della popolazione.
A questo punto l'algoritmo si trova davanti ad una scelta:

- Ricominciare dall'operazione di selezione usando come popolazione quella risultante dall'operazione di mutazione
- Terminare il processo

Per prendere una decisione, l'algoritmo fa riferimento alla cosiddetta **condizione di terminazione**, che rappresenta le condizioni che si devono verificare per mettere fine al processo (come ad esempio il numero di iterazioni effettuate, il tempo di esecuzione o il peggioramento in termini di fitness media della popolazione dopo un numero n fissato di iterazioni). Se la condizione di terminazione viene soddisfatta, allora il processo giunge al termine e solitamente viene restituito l'individuo migliore in termini di fitness dell'ultima popolazione, al netto di accorgimenti particolari e di operazioni di post-processing.

Le operazioni di selezione, crossover e mutazione vengono implementate tramite procedure chiamate **operatori genetici**, la cui definizione rappresenta uno degli step più importanti dell'implementazione di un algoritmo di ricerca che sfrutta questa metaeuristica, assieme a stabilire come debba essere calcolata la fitness e come e quando debba terminare il processo.

3.4 L'algoritmo utilizzato da ShallWeGo

In questa sezione verrà descritto accuratamente l'algoritmo di ricerca utilizzato dalla piattaforma che segue il paradigma degli algoritmi genetici e che va a risolvere il problema di

ottimizzazione consistente nel selezionare i migliori potenziali verificatori di una determinata segnalazione.

3.4.1 Codifica dell'Individuo

Nell'algoritmo presente in ShallWeGo, si definisce individuo un insieme di 5 utenti (nella popolazione iniziale essi sono presi in maniera casuale da quelli che operano nella provincia della segnalazione. In questo modo, sfruttando la vicinanza territoriale, viene aumentata la possibilità che questi utenti siano a conoscenza dell'oggetto della segnalazione per la quale sono stati scelti. La distanza tra il luogo della segnalazione e quello in cui opera l'utente viene calcolata effettuando delle operazioni di geocoding tramite il server Nominatim messo a disposizione per l'occasione. Questo aspetto verrà trattato in dettaglio nel capitolo dedicato all'architettura della piattaforma.

3.4.2 Funzione di Fitness

All'inizio di questo capitolo, è stato delineato l'*identikit* del verificatore ideale per una certa segnalazione. Le metriche utilizzate, riassumendo, sono:

- La distanza in chilometri tra il luogo di una segnalazione e l'area in cui opera un utente.
- La "reputazione" (o karma) che un utente si è costruito durante la sua permanenza sulla piattaforma

Dal punto di vista matematico, la funzione di fitness può essere formalizzata in questo modo, se x rappresenta il valore (naturalmente $>= 0$) in chilometri della distanza ed y il valore di karma dell'utente:

$$f(x,y) = \begin{cases} \frac{4(250+\frac{100}{x+0,3})+2y}{2} & \text{se } x < 15 \\ \frac{4(\frac{30}{x})^2+2y}{2} & \text{altrimenti} \end{cases}$$

La struttura dell'equazione conferma quanto accennato poc'anzi sul dare la priorità alla distanza geografica piuttosto che (almeno in un primo momento della vita della piattaforma, con una distribuzione più uniforme degli utenti) al karma. Infatti, un utente che opera in un'area che dista meno di **15km** dal luogo della segnalazione ha intuitivamente più probabilità di essere a conoscenza della struttura della rete di trasporti in quella zona (e quindi risulta più adatto al ruolo di verificatore).
Ed infatti, visto l'andamento della funzione $f(x) = \frac{100}{x}, x \neq 0$ (il cui andamento è mostrato

nella figura successiva e che a valori piccoli di x associa valori grandi di $f(x)$) e vista la presenza della costante additiva (250), un utente che opera in quel range di distanza dalla segnalazione risulta molto forte agli occhi dell'algoritmo.

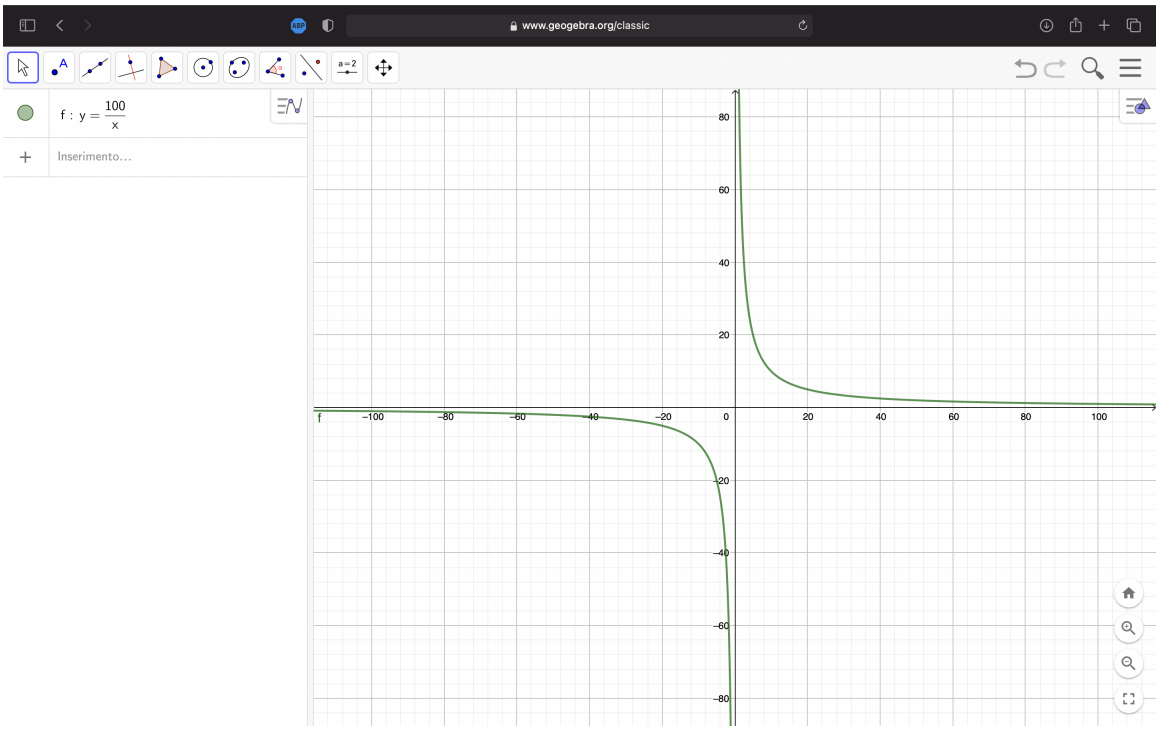

Figura 3.1: L'andamento della funzione $100/x$

3.5 Operatori genetici

Verranno ora illustrate le scelte riguardo all'implementazione dei vari operatori genetici che prendono parte al processo di evoluzione tipico degli algoritmi genetici.

3.5.1 L'operatore genetico di Selezione

Il primo operatore che viene applicato in una singola iterazione dell'algoritmo è quello di **Selezione**, che, riprendendo quanto descritto in precedenza, permette di stabilire a quali individui possano essere ammessi a riprodursi ed eventualmente a mutare (sulla base della loro fitness). In letteratura, esistono diversi approcci a quest'operatore. Quelli principali sono stati illustrati durante la sesta lezione del corso di *Fondamenti di Intelligenza Artificiale* tenuta dal Prof. Fabio Palomba e dal Dott. Emanuele Iannone.
Tra questi vanno menzionati in particolare:

- L'approccio basato su **roulette wheel** in cui gli individui con la fitness più alta hanno più possibilità di sopravvivere a questa fase, proprio come succede ad un elemento posto su una roulette che ha una porzione di area maggiore rispetto agli altri. In questo caso, la "porzione" di roulette dedicata ad un individuo è direttamente proporzionale al valore della sua fitness (normalizzata in $[0, 1]$)

- L'approccio basato su **rank**, nel quale ogni individuo della popolazione è ordinato in modo decrescente rispetto alla sua fitness e ad ogni posizione in questa lista viene associato un numero, il cosiddetto *rango*: la probabilità di selezione del singolo risulta inversamente proporzionale al suo rango.

- L'approccio basato su **truncation**, in qualche modo simile al precedente (ne mutua l'ordinamento degli individui) che consiste nel fissare un numero $n < |P|$ (se P rappresenta la popolazione) e di selezionare i primi n individui nella popolazione che hanno la fitness più alta

Per l'algoritmo utilizzato in ShallWeGo si è scelto di utilizzare l'approccio basato su **roulette wheel** in quanto risulta quello di più immediata comprensione ed implementazione (essendo molto più fedele a quanto avviene in natura). Inoltre, non essendoci alcun caso in cui la fitness possa scendere sotto il valore 0 quest'approccio è stato applicabile senza problemi.

3.5.2 L'operatore genetico di Crossover

Dopo l'applicazione dell'operatore di Selezione (e quindi con al definizione di quelli che sono i componenti del cosiddetto **mating pool**), si procede con la fase di **crossover**, che prevede l'accoppiamento (con una determinata probabilità) di coppie di individui all'interno del mating pool. Esistono diverse tecniche che permettono di implementare questa operazione. Tra queste si menzionano:

- La tecnica **Single Point** che dati due individui x = $(x_0, x_1, ..., x_n)$ ed y = $(y_0, y_1, ..., y_m)$, prevede la creazione di due nuovi individui j e k che siano il risultato di un incrocio tra i geni dei loro genitori, dopo aver selezionato il cosiddetto **punto di taglio**, ovvero quel punto z tale che:

  - $j = x_0, x_1, ..., x_z, y_{z+1}, ..., y_m$ e
  - $k = y_0, y_1, ..., y_z, x_{z+1}, ..., x_n$

- La tecnica ***k*-points**, che rappresenta una generalizzazione della tecnica Single Point, in cui un individuo viene diviso in k porzioni prima di essere incrociato con un altro individuo

La tecnica che viene usata da ShallWeGo è quella del **Single Point**. C'è da precisare che nel caso del problema che si sta affrontando la posizione di un gene all'interno di un individuo risulta irrilevante e di conseguenza un approccio k-points non avrebbe portato ad un qualche tipo di miglioramento.

3.5.3 L'operatore genetico di Mutazione

L'ultimo operatore genetico che viene applicato durante un'iterazione è quello di **Mutazione** che ricalca, come accennato precedentemente, il processo di mutazione spontanea che avviene in natura. Una mutazione consiste nel cambiamento casuale di un gene all'interno di un individuo. Anche in questo caso, esistono diversi approcci che in generale possono essere seguiti.
Tra questi si menzionano:

- La tecnica del **Bit Flip** che, come suggerisce il nome, è applicabile solo nel caso di individui codificati in modo *binario* e consiste nel scegliere in maniera casuale un gene all'interno dell'individuo e "capovolgere" il suo valore (quindi da 0 si passa ad 1 e vice-versa).

- La tecnica del **Random Resetting** che prevede il cambiamento di un gene preso casualmente all'interno dell'individuo con un altro valore ammissibile per quel gene preso altrettanto casualmente.

- La tecnica dello **Swap** che prevede lo scambio di due geni all'interno dell'individuo

- La tecnica dello **Scramble** che prevede una permutazione dei geni all'interno di un individuo

La maggior parte di questi approcci risultano piuttosto rilevanti in un contesto in cui la posizione di un gene cambia la sua rilevanza del suo valore in un individuo, come nel caso di codifiche binarie (il bit più a sinistra se è posto ad 1 aumenta di gran lunga il valore del numero rappresentato in binario).
Data la particolare natura del problema che si sta affrontando, non è possibile andare ad utilizzare la tecnica del Bit Flip e, nello stesso modo, le tecniche di Swap e Scramble non

sono rilevanti in quanto la codifica dell'individuo non risente della posizione di un gene in particolare.
Di conseguenza, la strategia che viene usata per questo algoritmo è quella del **Random Resetting**. In particolare, scelto un utente che è presente all'interno di un individuo, lo si sostituisce con un nuovo utente preso dal pool dei potenziali candidati ad essere verificatori di una segnalazione. Nel caso il nuovo utente che ha sostituito il precedente fosse già presente all'interno dell'individuo, il processo si ripete fino a non avere utenti uguali.

3.6 Termine del processo ed accorgimenti adottati

3.6.1 Condizione di terminazione

L'algoritmo termina quando si verifica una delle seguenti tre condizioni:

- Vengono superate le 25 iterazioni del processo
- Viene superato il limite di tempo stabilito (nel caso dell'algoritmo della piattaforma, quest'ultimo è fissato a 5 minuti)
- Per tre iterazioni consecutive la fitness della popolazione è minore rispetto a quella della miglior popolazione creata fino a quel momento

3.6.2 La strategia dell'archivio

Come ulteriore accorgimento nello sviluppo dell'algoritmo per la piattaforma, è stata implementata la cosiddetta **Strategia dell'Archivio** che consiste nel conservare una popolazione separata che non evolve composta da individui particolarmente forti, che si va a costruire man mano che le iterazioni vanno avanti. Nel caso specifico di ShallWeGo si è provveduto, per ogni iterazione, a tenere traccia del miglior individuo in termini di fitness della popolazione risultante dall'applicazione dei tre operatori genetici. Il migliore tra questi tre verrà quindi aggiunto all'archivio. Essendo costituito da individui "forti", l'archivio può essere usato come alternativa alla popolazione che viene restituita dopo il termine della computazione dell'algoritmo. Questo, infatti, è proprio l'approccio scelto per la piattaforma: se al termine delle iterazioni la popolazione risultante avrà una fitness media minore o uguale a quella dell'archivio, la popolazione restituita sarà sostituita dall'archivio.
La popolazione restituita sarà quindi soggetta a delle operazioni di postprocessing.

3.6.3 Postprocessing

Dopo l'esecuzione dell'algoritmo il risultato è quindi una popolazione di individui, a loro volta composta da utenti. In precedenza, è stato specificato come la fitness di un individuo sia calcolata tramite una media aritmetica della fitness calcolata sui singoli geni. Si sfrutta quindi questa possibilità per effettuare un lavoro di post-processing sulla popolazione risultato. In particolare, si è scelto di effettuare le seguenti operazioni:

- Si ottiene, a partire dal singolo individuo, la lista degli utenti che esso contiene, ordinati in maniera decrescente rispetto al loro valore di fitness;
- Per ognuna di queste liste, viene preso il primo elemento. Il risultato di questa operazione sarà di fatto un nuovo individuo formato dagli utenti più "forti" di quelli presenti tra gli individui della popolazione risultante dall'algoritmo.

Sebbene le operazioni di post processing (trattandosi prettamente di ordinamento) assumano una complessità di almeno $\mathcal{O}(n \log n)$, esse vengono effettuate su un dominio particolarmente più piccolo rispetto a quello dei candidati (che mettendosi nel contesto di una piattaforma ben avviata potrebbero essere in numero molto elevato o comunque tale da rendere la ricerca esaustiva poco efficiente) e che allo stesso tempo permettono di ottenere un risultato ancora migliore rispetto a quello ottenuto con la semplice evoluzione, riuscendo cioè a creare un gruppo di utenti che risultino "forti tra i forti" in termini di fitness assoluta. Ciò giustifica la presenza sia della procedura di evoluzione degli individui sia il postprocessing. Le operazioni di ordinamento non richiedono un ulteriore calcolo della fitness in quanto in fase di implementazione è stato effettuato il caching di quel valore per quella determinata istanza dell'algoritmo e sicuramente questo valore alla fine sarà già stato calcolato e conservato, riducendo il calcolo finale della fitness ad un semplice accesso ad una variabile che non richiede operazioni particolari.

3.6.4 Risultati dell'algoritmo con i parametri correnti

Nella sua fase di testing, l'algoritmo è stato eseguito su un insieme molto grande di utenti generato in maniera casuale (un utente per ogni comune e con fitness presa casualmente tra 0 e 65). Nella tabella che segue sono riportati i risultati dell'algoritmo su diverse istanze. In particolare,

- Nella **prima colonna** è riportato il comune della segnalazione (e le sue coordinate);

- Nella **seconda colonna** sono riportati comune di residenza e livello di karma degli utenti selezionati dall'algoritmo per verificare quella determinata segnalazione, dopo l'applicazione del post-processing;

Fisciano	Cava de' Tirreni (karma: 48.6) Siano (karma: 12.5) Baronissi (karma: 17.8) Roccapiemonte (karma: 47.8) Trentinara (karma: 53.6)
Nocera Inferiore	Nocera Inferiore (karma: 29.2) Sant'Egidio del Monte Albino (karma: 39.3) San Marzano sul Sarno (karma: 53.4) Roccapiemonte (karma: 47.8) Nocera Superiore (karma: 24.2)
Salerno	Giffoni Sei Casali (karma: 45.3) Pontecagnano Faiano (karma: 44.2) Baronissi (karma: 17.8) Laviano (karma: 46.8) Maiori (karma: 46.6)

Tabella 3.1: Risultati dell'algoritmo su diversi input.

Si noti come nel caso dell'istanza dell'algoritmo per la città di Nocera Inferiore sia presente un utente che opera proprio lì: egli rappresenta quindi parte della soluzione ottima. Nel caso della città di Salerno, invece, è stato incluso un utente la cui distanza dal luogo della segnalazione si aggira attorno ai 47km in linea d'aria. Quest'utente è stato incluso nella soluzione in virtù del proprio valore di *karma* che risulta discretamente alto.

CAPITOLO 4

ShallWeGo

Lo scopo di questo capitolo è di descrivere in termini di implementazione il funzionamento e l'architettura della piattaforma. In particolare, si focalizza l'attenzione sulle tecnologie utilizzate per lo sviluppo del server e del client della piattaforma, oltre che ad altre tecnologie impiegate per il corretto funzionamento delle componenti.

4.1 Requisiti funzionali

Allo scopo di questo lavoro di tesi, le seguenti funzionalità visibili all'utente sono state implementate:

- L'utente può visualizzare tramite una mappa le fermate del trasporto pubblico presenti nin una determinata zona ed ottenerne i dettagli;
- Allo stesso modo, l'utente può visualizzare gli eventi temporanei in una determinata zona ed ottenerne i dettagli;
- L'utente può aggiungere delle fermate ad un elenco di fermate preferite;
- L'utente può visualizzare i dettagli e le destinazioni di una linea di trasporto pubblico;
- L'utente può seguire in diretta su una mappa l'andamento di una corsa e visualizzarne i dettagli;
- L'utente può segnalare l'affollamento di una fermata;
- L'utente può segnalare l'affollamento di una corsa;
- L'utente può segnalare la presenza di una fermata in un punto preciso sulla mappa o rapidamente presso la propria posizione;
- L'utente può segnalare la presenza di un'azienda di trasporto;
- L'utente può aggiungere una linea di trasporto pubblico ad un'azienda;
- L'utente può verificare, se gli sono state assegnate, tutti i tipi di segnalazioni appena descritti.
- L'utente può effettuare una segnalazione di un evento temporaneo in un punto preciso della mappa o rapidamente presso la propria posizione.
- L'utente può fornire aggiornamenti sulla propria posizione nell'ambito di una corsa espletata da una linea di trasporto pubblico e condividerne i dettagli.

4.2 Architettura del sistema

Il sistema ShallWeGo si configura come una classica architettura **client-server**.

- Il server consiste *Java Enterprise application*, che accetta connessioni dal client sotto forma di richieste ad API REST.
- Il client è stato realizzato tramite un'applicazione utilizzabile sulla piattaforma Android dalla versione 10 in poi, a causa di scelte implementative riguardo alcune librerie incluse nel progetto. Queste scelte saranno documentate più avanti nel capitolo.

Per quanto riguarda la parte server, essa risulta composta da due macrocomponenti:

- La Java Application già accennata
- La componente che ospita la logica ed il database utilizzato da Nominatim, che è dislocato su un'altra macchina che viene utilizzata solo per questo scopo.

4.3 ShallWeGo: Il Server

Il server, come precedentemente menzionato, è realizzato usando le specifiche Java Enterprise Edition (da qui in poi *Java EE*), usando a questo scopo il framework ***Spring Boot***, che permette una più semplice realizzazione dei task necessari a sviluppare un'applicazione che sfrutta il paradigma Client-Server, come ad esempio la gestione delle API accessibili dall'esterno i cosiddetti *endpoint* oppure la gestione dei dati persistenti.

4.3.1 Class-Diagram

Di seguito è mostrato il diagramma delle classi del progetto Server. Sono state omesse le classi di servizio usate da Spring e da JPA per il corretto funzionamento del Server stesso. Sono inoltre state omesse le classi che compongono l'Algoritmo Genetico.

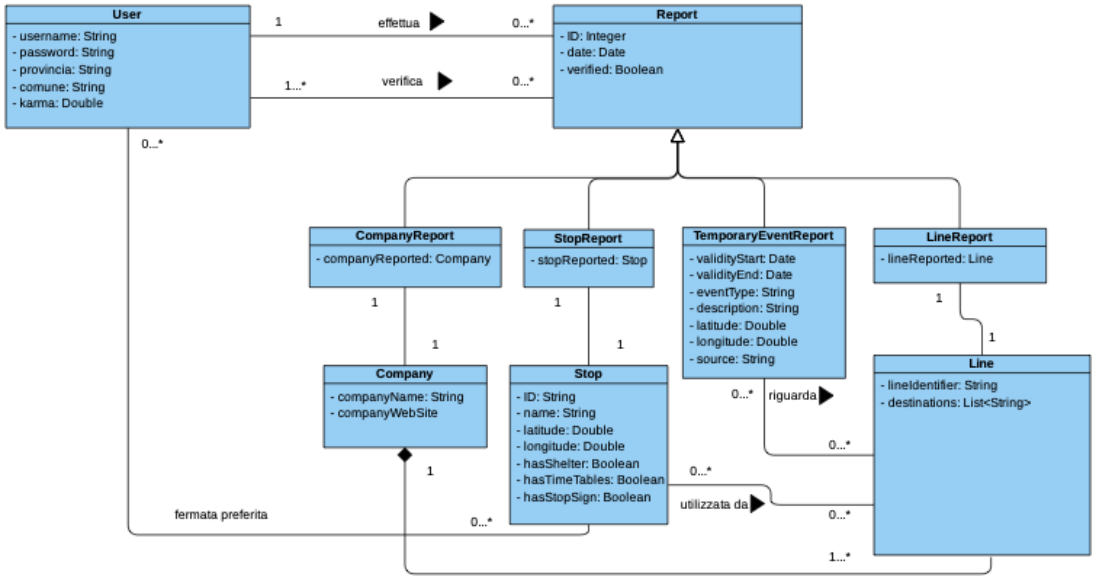

Figura 4.1: Il Class Diagram del Server

4.3.2 Accessibilità delle API dall'esterno

Spring mette a disposizione il meccanismo dei **Controller**, che normalmente è inquadrato nel contesto di un'architettura MVC, anche se nel caso di ShallWeGo non vi è necessità di andare a creare un'applicazione di web che renda necessaria l'implementazione di tale pattern architetturale. Di conseguenza, essendo la parte che riguarda l'interfaccia utente gestita tramite un'applicazione Android, non è stata implementata nessuna parte che faccia le veci della "View" lato server. La comunicazione tra server e client avviene tramite JSON sfruttando il meccanismo delle API REST. La classe responsabile per la comunicazione con l'esterno è chiamata **Controller**, collocata nel pacchetto omonimo e decorati con l'annotazione **@RestController**. Questa annotazione permette al framework di interpretare quella classe come un Controller che accetta richieste sulla porta 8080 e all'indirizzo `/`. I vari metodi di quella classe che corrispondono alle diverse API esposte all'esterno sono decorati con una delle seguenti annotazioni:

- **@GetMapping("/url")** esegue il metodo annotato ogni qual volta all'indirizzo `/url` viene generata una richiesta di tipo **GET**;
- **@PostMapping("/url")** esegue il metodo annotato ogni qual volta all'indirizzo `/url` viene generata una richiesta di tipo **POST**;

- **@PutMapping("/url")** esegue il metodo annotato ogni qual volta all'indirizzo `/url` viene generata una richiesta di tipo **PUT**;
- **@DeleteMapping("/url")** esegue il metodo annotato ogni qual volta all'indirizzo `/url` viene generata una richiesta di tipo **DELETE**;

4.3.3 Gestione dei dati persistenti: breve introduzione a JPA

Data la necessità di conservare una notevole quantità di dati (utenti, fermate, linee e così via) un approccio che prevede l'utilizzo di file è stato escluso a priori perché quasi impossibile da gestire data la complessità. Si è scelto quindi di utilizzare un database di tipo relazionale (RDBMS). Tuttavia, prevedendo il dominio applicativo numerose associazioni tra le entità, un approccio "classico" tramite **JDBC** avrebbe comunque portato ad un livello di complessità molto elevato (si pensi a dover gestire manualmente tutte le query con i vari join tra le tabelle). A questo scopo, torna molto utile sfruttare le caratteristiche di **ORM** (*Object-Relational Mapping*) e, successivamente, di **JPA**, (*Java Persistence API*).

ORM è definito come "una tecnica di programmazione che permette l'integrazione di sistemi software che aderiscono al paradigma della programmazione orientata agli oggetti (**OOP**) con sistemi **RDBMS**".

Uno dei principali vantaggi di ORM risiede nel contrasto della complessità di gestione della persistenza derivata dalla mancanza di compatibilità tra dati salvati all'interno di un database e oggetti in un linguaggio di programmazione, oltre che ad una sostanziale indipendenza dal *vendor* che fornisce il DBMS utilizzato. Wikipedia [2021]

Il framework Spring, nello specifico, mette a disposizione una serie di API che implementano le specifiche di **JPA**. JPA è definito come "un insieme di specifiche che mette a disposizione degli sviluppatori un insieme di servizi e di strutture dati che permettono l'ORM e quindi la gestione dei dati persistenti all'interno di applicazioni Java." (The Java EE 7 Authors [2014]). Essendo solamente una specifica, JPA necessita di un'implementazione. Quella di riferimento è **EclipseLink**, sviluppata dalla Eclipse Foundation a partire dal 2015.

Il framework Spring, tuttavia, utilizza come implementazione delle specifiche JPA la libreria open source **Hibernate 5.5** sviluppata in partnership con l'azienda *Red Hat*.

Per effettuare il mapping tra oggetti istanze di una classe e righe presenti in una tabella in un database, JPA si serve del concetto di **Entity**, che rappresenta un'istanza di una classe (in gergo, **POJO**, ovvero *Plain Old Java Object*) che "dichiara" di poter essere salvata all'interno di un database.

Le specifiche JPA introducono il concetto di **Configuration By Excpetion**. In questo modo, se non specificato altrimenti, le impostazioni riguardo persistenza e mapping di oggetti risulteranno essere quelle di default. Per fare un esempio concreto, in un progetto dove è previsto l'utilizzo di JPA tutte le classi Java vengono viste come tali fino a che non viene utilizzata l'annotazione Entity su una di queste. Ad esempio:

```
@Entity
public class POJO implements Serializable {
    @Id
    private Integer id;

    public POJO() {}
}
```

Listing 1: File: POJO.java

Tramite questo setup è possibile rendere la classe contenuta in *POJO.java* un'entity di cui è possibile effettuare la persistenza su un database.

Per poter rappresentare un'Entity, un POJO deve rispettare i seguenti requisiti: (Goncalves [2013])

- Essere annotata con `@javax.persistence.Entity`;
- Avere un attributo annotato con `@javax.persistence.Id` che ne denota la chiave primaria (nel caso di chiave primaria semplice: è possibile anche avere chiavi composte, come descritto più avanti)
- Deve possedere un costruttore vuoto che abbia come modificatore di accesso **`public`** o **`protected`**. Sono ammessi ulteriori costruttori.
- Non deve essere una classe *final*
- Non deve essere una classe interna ad un'altra classe
- Deve implementare l'interfaccia `Serializable`.

Le Entity presenti nel dominio applicativo sono le seguenti:

- **Stop**, che modella una fermata dei mezzi pubblici,

- **Line**, che modella una linea di trasporto pubblico,
- **Company**, che modella un'azienda di trasporto pubblico,
- **Report**, che modella il concetto di segnalazione,
- **LineReport**, che modella il concetto di segnalazione di una linea,
- **CompanyReport**, che modella il concetto di segnalazione di un'azienda,
- **StopReport**, che modella il concetto di segnalazione di una fermata,
- **TemporaryEventReport**, che modella il concetto di segnalazione di un evento temporaneo,
- **User**, che modella un utente registrato alla piattaforma.

oltre che ad altre classi "di servizio" create per mappare relazioni di tipo Many-to-Many.

Si ponga attenzione sulla modellazione del concetto di segnalazione. Qui di seguito è mostrato lo scheletro delle classi che modellano i vari tipi di segnalazione:

```
@Entity
public abstract class Report {
    @Id
    private Integer Id;
}
```

Listing 2: File: Report.java

```
@Entity
public class StopReport extends Report {
    private Stop stopReported;
}
```

Listing 3: File: StopReport.java

```
@Entity
public class LineReport extends Report {
    private Line lineReported;
}
```

Listing 4: File: LineReport.java

```
@Entity
public class CompanyReport extends Report {
    private Company companyReported;
}
```

Listing 5: File: CompanyReport.java

```
@Entity
public class TemporaryEventReport extends Report {
    private Date validityStart;
    private Date validityEnd;
    private String eventType;
    private String description;
    private String latitude;
    private String longitude;
    private String source;

    private List<Line> affectedLines;
}
```

Listing 6: File: TemporaryEventReport.java

Si noti l'utilizzo della keyword **extends** che segnala l'utilizzo dell'ereditarietà, un concetto tipico dei linguaggi Object-Oriented come Java e che non è modellato in nessun DBMS

relazionale. Per ovviare a questo problema, JPA mette a disposizione la seguente annotazione:

```
public @interface Inheritance {
    InheritanceType strategy() default SINGLE_TABLE;
}
```

Quest'annotazione, attraverso il campo *strategy* permette di stabilire con che strategia effettuare il mapping di una gerarchia.

I valori che *strategy* può assumere sono i seguenti: (Goncalves [2013])

- SINGLE_TABLE, che crea una sola tabella all'interno del database che rappresenta l'intera gerarchia e che per disambiguare tra i vari tipi delle sottoclassi usa un attributo chiamato "DTYPE" che assume come valore il nome della sottoclasse di un determinato elemento della tabella. Questa strategia, in virtù della configuration by exception è quella utilizzata di default in assenza dell'annotazione.
- TABLE_PER_CLASS, che crea una tabella all'interno del database per ogni sottoclasse in una strategia.
- JOINED, che colloca gli attributi che compaiono esclusivamente nelle sottoclassi della gerarchia in altre tabelle, una per membro della gerarchia.

In ShallWeGo, la strategia utilizzata è quella SINGLE_TABLE poiché si è preferito non aumentare la complessità dello schema.

Per permettere l'esecuzione delle query di inserimento, retrival ed eliminazione delle informazioni, Spring mette a disposizione il meccanismo delle **Repository**. L'obiettivo delle Repository Spring è "quello di ridurre la quantità di codice *boilerplate* necessario per implementare il layer di accesso ai dati per diversi tipi di entità persistenti.".

La creazione delle Repositories è demandata al programmatore che può crearne una creando un'**interfaccia** che estende la classe `CrudRepository<T, ID>` oppure `JpaRepository<T, ID>`.

- **T** indica la classe dell'Entity a cui la repository si riferisce;
- ID indica la classe della **chiave primaria** di T

La particolarità delle Repositories di Spring è sostanzialmente quella di poter andare a creare delle query semplicemente andando ad aggiungere un metodo all'interfaccia appena

creata. Ad esempio, se nel caso di un'Entity si ha necessità di effettuare query basandosi su due attributi (X ed Y), allora sarà sufficiente aggiungere questo metodo all'interfaccia associata all'Entity:

```
public interface EntityRepository extends
 JpaRepository<Entity, Integer> {
    public List<Entity> findByXAndY(String X, String Y);
}
```

Listing 7: Esempio di Repository con custom query

In ShallWeGo, le seguenti Repository sono state utilizzate:

- **CompanyRepository** che permette di effettuare query riguardanti l'Entity **Company**
- **LineRepository** che permette di effettuare query riguardanti l'Entity **Line**
- **ReportRepository** che permette di effettuare query riguardanti l'Entity **Report**
- **StopRepository** che permette di effettuare query riguardanti l'Entity **Stop**
- **TemporaryEventReportRepository** che permette di effettuare query riguardanti l'Entity **TemporaryEventReport**
- **UserRepository** che permette di effettuare query riguardanti l'Entity **User**

Nel caso particolare di **UserRepository**, si può notare come sia stata introdotta una query per recuperare dal database tutti gli utenti che operano in una determinata provincia. Questo si è reso necessario per il funzionamento dell'algoritmo di assegnazione dei verificatori di una segnalazione:

```
public interface UserRepository extends JpaRepository<User, String> {
    List<User> findByProvincia(String provincia);
}
```

Listing 8: File: UserRepository.java

4.3.4 Mappe e Dati Geografici: OpenStreetMap e Nominatim

OpenStreetMap è un **WebGIS** lanciato nel 2004 che si propone come alternativa open source ai servizi commerciali che forniscono mappe o dati geografici in generale (si pensi

ad esempio ai nomi delle strade oppure delle attività commerciali). Adotta un approccio **community-driven**: di conseguenza, i dati sono aggiornati periodicamente da parte di utenti che aderiscono al progetto. Questi dati sono consultabili tramite l'interfaccia principale del servizio, disponibile all'indirizzo `https://www.openstreetmap.org/`. All'interno di questa pagina è presente, come nella maggior parte dei servizi di questo tipo, una casella di ricerca che permette di cercare indirizzi o punti di interesse. La funzionalità di ricerca è implementata sfruttando il servizio **Nominatim**, già accennato in precedenza. Nominatim consiste in un sistema **geocoding** open source scritto in PHP e sviluppato con il patrocinio dello stesso progetto OpenStreetMap. Esso permette di effettuare ricerche di dati geografici presenti all'interno di un database PostgreSQL a partire da un insieme di parole chiave e, allo stesso modo, di ottenere suddetti dati a partire da una coppia di coordinate (Latitudine, Longitudine). Il progetto Nominatim viene inoltre usato anche dalla stessa piattaforma OpenStreetMap per fornire la funzionalità di ricerca all'interno della propria interfaccia web. Il server Nominatim, nel caso specifico della piattaforma, è stato utilizzato principalmente per ottenere due tipologie di informazioni:

- A partire dal nome di un comune e dalla sua provincia, una coppia di coordinate (Latitudine, Longitudine) che ne rappresenti la posizione approssimativa al fine di computare la fitness di un individuo nel contesto dell'algoritmo genetico
- Per fornire, oltre alla posizione in termini di coordinate, anche un indirizzo approssimativo (e quindi un nome mnemonico) per una segnalazione indicata come *evento temporaneo*. In questo caso, si parla di *reverse geocoding*

Uno dei principali vantaggi di questa piattaforma consiste nella possibilità da parte di un utente di installare la propria istanza del server, importando all'interno di un database una serie di dati provenienti da diverse fonti. La strada del self-hosting è quella che preferita nel caso di ShallWeGo, poiché nonostante sia disponibile un endpoint pubblico a cui possono essere indirizzate delle richieste, disponibile alla pagina `https://nominatim.openstreetmap.org/` esso impone una restrizione di esattamente una interrogazione al secondo, pena il *ban* dell'indirizzo IP sorgente dall'utilizzo del servizio. Nel caso particolare di ShallWeGo è stata utilizzata la versione 3.7.2 del software, installata su una macchina che esegue il Sistema Operativo Arch Linux e che utilizza *Apache httpd* come server web per esporre il servizio in rete.

I servizi di Nominatim sono accessibili sostanzialmente in due modi:

- Tramite un'interfaccia utente di debug che include sia una casella di ricerca, sia una mappa dove visualizzare i risultati
- Tramite API REST, che permettono l'output testuale dei risultati in diversi formati (tipicamente JSON e XML).

Avendo necessità di utilizzare il risultato all'interno di un'altra componente, è stata reputata non essenziale la configurazione della GUI di Nominatim: in ShallWeGo viene usata solamente la sua interfaccia che prevede output testuale.

In particolare, di seguito sono descritti i servizi di Nominatim utilizzati dalla piattaforma per i suoi scopi:

- **/search**: il servizio principale e quello più usato. Permette di ottenere risultati in termini di dati OSM (e quindi anche le coordinate geografiche corrispondenti) di un luogo a partire da un insieme di parole chiave. Queste ultime possono essere specificate tramite il parametro **q**. Ad esempio, la query associata a `http://nominatim.openstreetmap.org/search/?q=UniversitaDegliStudidiSalernoFisciano&format=json` permette di ottenere informazioni sul campus di Fisciano dell'Università degli Studi di Salerno in formato JSON (Si veda la figura 4.1);
- **/reverse**: permette di ottenere approssimativamente l'indirizzo associato a delle coordinate geografiche specificate tramite i parametri **lat** e **lon**, che rappresentano rispettivamente la latitudine e la longitudine del luogo. Ad esempio, la query associata a `http://nominatim.openstreetmap.org/reverse/?lat=40.774283&lon=14.790868&format=json` permette di ottenere le informazioni associate alle coordinate (40.774283, 14.790868) in formato JSON. (Si veda la figura 4.2)

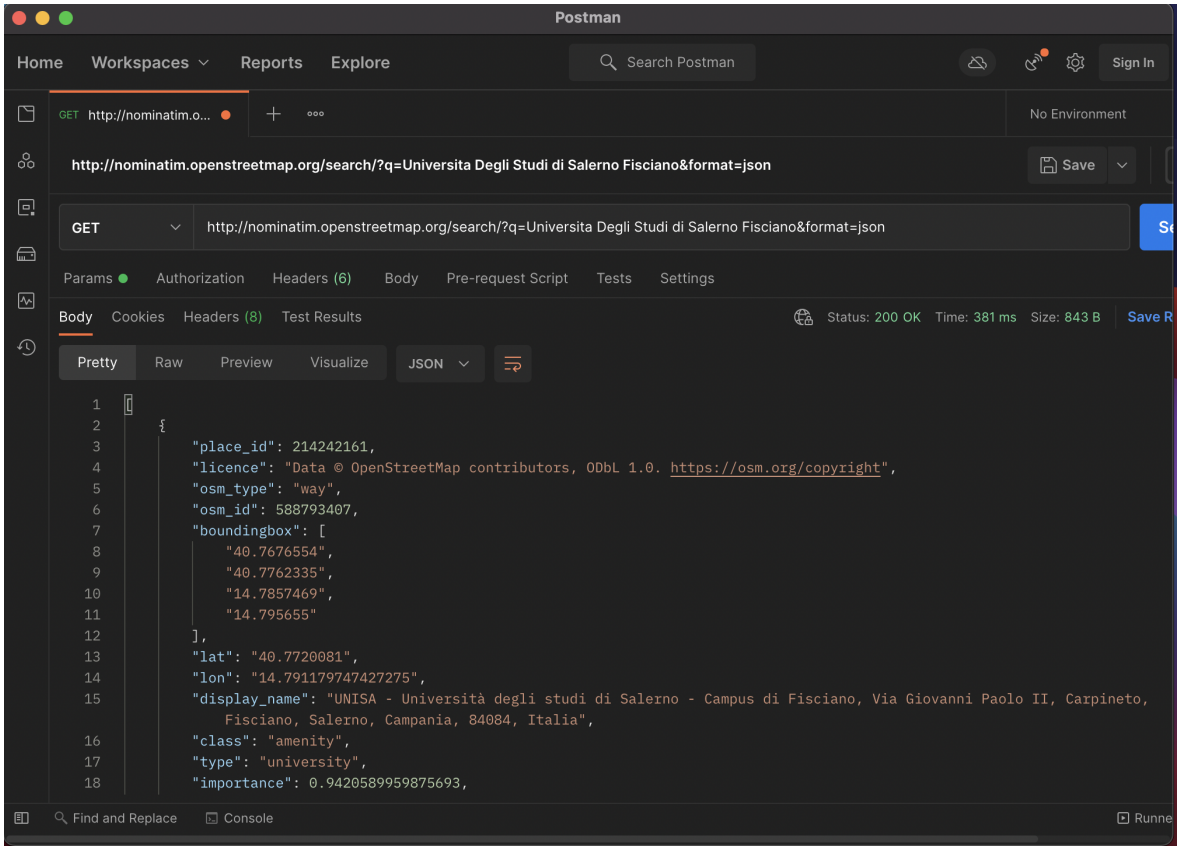

Figura 4.2: Search

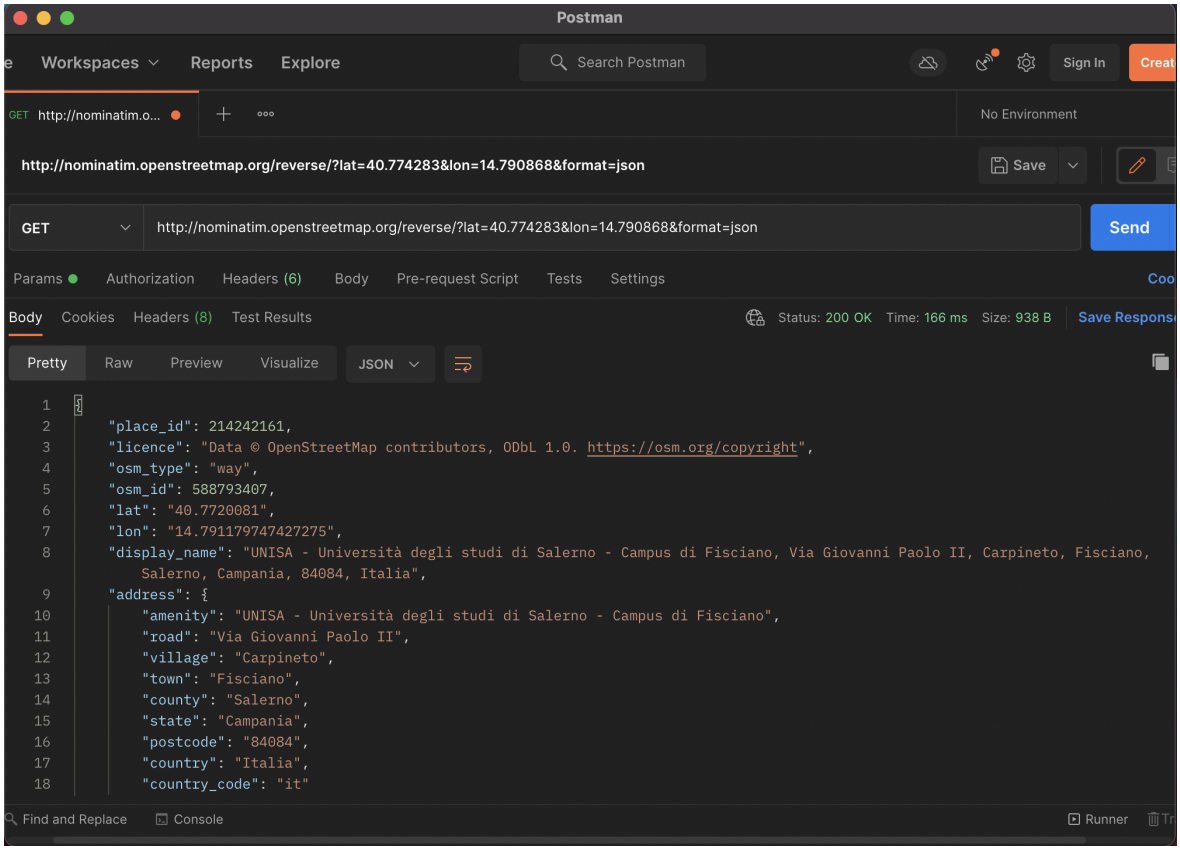

Figura 4.3: Reverse

Fonte dei dati utilizzati

Nominatim di per sé risulta essere solamente un servizio che permette di effettuare geocoding a partire da un dataset ma non ne fornisce nessuno out of the box. È stato quindi necessraio ottenere questi dati da entità terze. Un progetto che mette a disposizione pubblicamente questo tipo dei dati è chiamato Geofabrik, che ospita sui suoi server pacchetti organizzati per Stato o per Continente. Il formato di questi pacchetti risulta essere $.osm.pbf$, che è possibile importare all'interno di Nominatim attraverso un applicativo chiamato **osm2pgsql**. Quest'applicativo permette di importare dati organizzati secondo lo standard di OpenStreetMap all'interno di un database PostgreSQL/PostGIS.

4.4 ShallWeGo: Il client

Dopo aver analizzato in dettaglio la Java Application che contiene la logica applicativa di ShallWeGo ed aver descritto il funzionamento del Server di Geocoding si passa ora all'analisi del client della piattaforma. Il client messo a disposizione dalla piattaforma consiste in un'applicazione Android sviluppata usando il linguaggio Java.

4.4.1 Breve introduzione allo sviluppo Android

La piattaforma Android è storicamente molto aperta allo sviluppo di applicativi di terze parti. Già a partire dalla versione 1.0 (rilasciata nel 2009), Google si è impegnata a fornire un'API completa e semplice da usare per facilitare sia lo sviluppo di app sia il funzionamento di nuove versioni del sistema operativo stesso su un determinato hardware (tramite il progetto **AOSP**, ovvero *Android Open Source Project*)

Un'applicazione Android, astraendosi dal funzionamento a basso livello del sistema operativo (il cui *kernel*) si basa su Linux, consiste soprattutto in **Activity**.

Un'Activity, secondo la documentazione ufficiale, è un "task singolo e specifico che l'utente può effettuare" Google [2021a]. Quasi tutte le Activity interagiscono con l'utente e, tramite operazioni normalmente trasparenti allo sviluppatore, si occupano della composizione delle finestre, che saranno poi utilizzate per mostrare la User Interface creata da chi sviluppa l'applicazione (che si specifica usando un file XML che è strettamente legato ad un'Activity).

Fino al 2019, Java ha rappresentato il linguaggio di programmazione principale per lo sviluppo di app Android. Alla conferenza Google I/O di quello stesso anno è stato annunciato l'impiego di **Kotlin** come linguaggio primario. Uno dei vantaggi di Kotlin è rappresentato dalla sua sintassi più concisa rispetto a quella di Java mantenendo comunque l'**interoperabilità** con quest'ultimo.

4.4.2 Activity presenti nell'applicazione

Come accennato in precedenza, le app Android sono costituite perlopiù da Activity, che permettono l'interazione con l'utente.

In ShallWeGo ne sono presenti 18. Ognuna di queste rappresenta una feature specifica della piattaforma.

Nello specifico:

- **IntroSlideShow**: fornisce una presentazione ed una panoramica sullo scopo e sul funzionamento dell'applicazione. È utilizzata inoltre per concedere i permessi per l'utilizzo dei servizi di geolocalizzazione;
- **RegisterActivity**: permette la registrazione di un'utente alla piattaforma;
- **LoginActivity**: permette il login di un utente registrato alla piattaforma;
- **MainActivity**: l'Activity principale di qualsiasi app Android e la prima ad essere avviata quando l'utente apre l'app. Contiene una mappa dove sono presenti, sotto forma di marker, gli oggetti di interesse della piattaforma (fermate, corse ed eventi temporanei);
- **AlertsNearby**: permette all'utente di visualizzare gli eventi temporanei più vicini alla propria zona;
- **FavoriteStops**: permette all'utente di accedere velocemente alle fermate che ha aggiunto all'elenco delle sue preferite;
- **StopDetails**: permette all'utente di accedere alla pagina dei dettagli di una fermata specifica;
- **EventDetails**: permette all'utente di consultare i dettagli di un evento temporaneo;
- **CompanyReportActivity**: permette all'utente di effettuare la segnalazione di un'azienda di trasporto;
- **LineReportActivity**: permette all'utente di effettuare la segnalazione di una linea;
- **StopReportActivity**: permette all'utente di effettuare la segnalazione di una fermata e di specificare le linee che la utilizzano;
- **TemporaryEventReportActivity**: permette all'utente di effettuare la segnalazione di un evento temporaneo ed eventualmente di specificare le linee coinvolte;
- **MyReportsActivity**: permette all'utente di visualizzare tutte le segnalazioni che ha effettuato nel tempo;
- **NewRideActivity**: permette all'utente di comunicare la propria posizione nell'ambito di una corsa espletata da una determinata linea. Permette di comunicarne anche l'affollamento e lo stato del mezzo in termini di presenza di aria condizionata e di macchina validatrice dei titoli di viaggio.

- **VerifyReports**: permette all'utente di visualizzare tutte le segnalazioni non ancora approvate a cui è stato assegnato in qualità di *verificatore*;
- **VerifyCompanyReport**: permette al *verificatore* di fornire il proprio voto sulla segnalazione di un'azienda di trasporto;
- **VerifyLineReport**: permette al *verificatore* di fornire il proprio voto sulla segnalazione di una linea espletata da un'azienda di trasporto;
- **VerifyStopReport**: permette al *verificatore* di fornire il proprio voto sulla segnalazione di una fermata;

4.4.3 Librerie di terze parti utilizzate nell'applicazione

Dal 2013, i progetti Android utilizzano il toolkit **Gradle**. Gradle consiste in una serie di utility eseguite sulla JVM che permettono la semplificazione e l'automazione del processo di *build* delle applicazioni. Uno dei compiti principali di un build manager come Gradle consiste nella gestione delle dipendenze e dei moduli di terze parti che possono essere inclusi in un'applicazione. In un progetto Android, solitamente, le dipendenze vanno specificate nel file *build.gradle* dell'applicazione.

Durante lo sviluppo di ShallWeGo, per fornire tutte le funzionalità indicate in precedenza, si è fatto uso di diverse librerie di terze parti, il cui funzionamento ed impiego nella piattaforma verrà descritto di seguito.

OsmDroid: un'alternativa *free* a Google Maps

Come accennato nell'introduzione, l'applicazione fa largo uso di mappe e dati geografici in generale. Si è parlato di come applicativi e librerie libere ed open-source potessero andare a risolvere il problema della dipendenza da servizi come le API di Google Maps, che prevedono un costo superata una certa soglia di utilizzo periodica. Anche l'integrazione di una mappa all'interno di un'Activity (che accade molto spesso nel contesto di ShallWeGo), se si opta per il servizio messo a disposizione da Google Maps, richiede il possesso di un'**API key** che permette di identificare univocamente la fonte delle richieste e tenere sotto controllo le soglie di utilizzo, eventualmente procedendo ad addebiti nel caso di sforamento. Per far fronte a questa problematica (che si fa più urgente man mano che l'utilizzo della piattaforma aumenta) è stata impiegata la libreria **OsmDroid**, che si propone come un'alternativa completa e gratuita alle API di Google Maps per l'integrazione di mappe all'interno di applicazioni Android. I dati

della mappa fornita da questa libreria provengono dal progetto OpenStreetMap, di cui si è discusso precedentemente. Il codice sorgente e la documentazione della libreria sono disponibili interamente su GitHub all'indirizzo `https://github.com/osmdroid/osmdroid`.

AppIntro

AppIntro è una libreria creata dallo sviluppatore **Paolo Rotolo** la cui documentazione è disponibile su GitHub all'indirizzo `https://github.com/AppIntro/AppIntro`. Fornisce il supporto alla creazione, sotto forma di *slides* di un'introduzione all'applicazione. Questo modulo semplifica notevolmente il processo di richiesta dei permessi, diventato più complesso dall'introduzione di Android 5.0. La libreria è stata impiegata per realizzare l'activity **IntroSlideShow**.

Volley

Volley è una libreria sviluppata da **Google** per fornire delle API da usare per svolgere task che richiedono l'interazione con la rete. L'impiego di questa libreria è stato preferito poiché la mole delle richieste da effettuare non è tale da giustificare l'utilizzo di altre librerie che Google stessa consiglia per trasferimenti di grandi moli di informazioni attraverso la rete come il **Download Manager** accessibile dalle API di Android. Il funzionamento di Volley si basa sulla presenza di una *coda* di richieste (realizzata tramite la classe `RequestQueue`) alla quale possono essere aggiunte delle richieste e all'evenienza regolarne la priorità in caso di richieste multiple. Dopo aver aggiunto una richiesta alla coda, Volley provvederà a spedirla in maniera **asincrona** e ad eseguire il metodo di callback `onResponse()` dal quale è possibile provvedere all'aggiornamento della UI. In ShallWeGo, la libreia è stata utilizzata in quasi tutte le Activity per realizzare l'interazione con la Java Application tramite file JSON dai quali sono estratte le informazioni necessarie per aggiornare l'interfaccia in modo corretto.

Gson

Gson è una libreria anch'essa sviluppata da **Google** che permette la serializzazione e la deserializzazione di informazioni da ed in stringhe JSON. La libreria mette a disposizione le classi `JsonObject` e `JsonArray` che modellano rispettivamente un oggetto ed un array di oggetti JSON e che permettono di accedere (tramite il metodo `get()`) in maniera strutturata ai vari campi della stringa di cui viene effettuato il parsing. Allo stesso modo, le classi `JsonObject` e `JsonArray` permettono di organizzare dei dati che possano essere convertiti

correttamente in una stringa JSON. Insieme a Volley, questa è una delle librerie fondamentali che permettono il corretto funzionamento della comunicazione con le API esposte dalla Java Application. Dunque, questa libreria è stata usata ogni qual volta è risultato necessario comunicare col server.

Material Components

Material Components è una libreria sviluppata da Google che fornisce l'implementazione (sia dal punto di vista di classi Java sia dal punto di vista di elementi XML pronti da inserire nei file di layout) di una serie di componenti grafiche tipiche del **Material Design**. Il Material Design è un sistema di design introdotto da Google stessa nel 2014 con la versione 5.0 di Android ed in continua evoluzione anche oggi. Il layout delle Activity di ShallWeGo è realizzato usando quasi esclusivamente queste componenti.

SpeedDial

SpeedDial è una libreria creata dallo sviluppatore **Roberto Leinardi** la cui documentazione è disponibile all'indirizzo `https://github.com/leinardi/FloatingActionButtonSpeedDial`. Fornisce un'implementazione concreta di un **Floating Action Button** (o **FAB**) seguendo le specifiche descritte alla pagina `https://material.io/components/buttons-floating-action-button#types-of-transitions`. Un Floating Action Button è solitamente un pulsante posizionato in vari punti del layout di un'Activity che permette di "richiamare" l'attenzione dell'utente, indicandogli il punto in cui è possibile compiere le azioni principali che quella determinata Activity mette a disposizione, che verranno mostrate alla pressione del FAB (*Speed Dial*). La libreria sopracitata permette proprio questo comportamento. In ShallWeGo, questa libreria è utilizzata in accoppiata col FAB presente nella MainActivity che, una volta premuto, offre l'opportunità di eseguire i seguenti task:

- Riposizionare la mappa in corrispondenza della posizione attuale dell'utente;
- Effettuare una segnalazione nella posizione attuale dell'utente;
- Cominciare il tracciamento in diretta di una corsa. (Il funzionamento di questo aspetto viene trattato in dettaglio nella sezione successiva).

Le figure 4.4 e 4.5 mostrano rispettivamente il FAB presente nella MainActivity e lo Speed Dial ad esso associato.

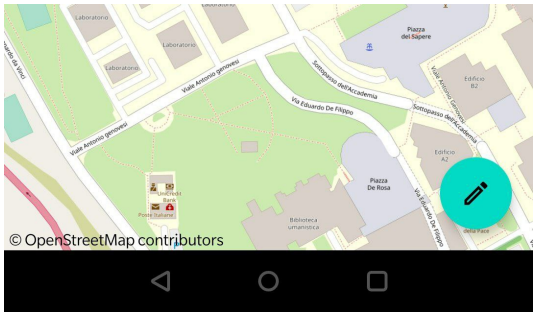

Figura 4.4: FAB nella Main Activity

Figura 4.5: Speed Dial associato al FAB

4.4.4 Il tracciamento in diretta delle corse: overview sui servizi di Android

Una delle funzionalità di ShallWeGo permette all'utente collegato alla piattaforma di mettere a disposizione degli altri utenti la propria posizione all'interno di una corsa espletata da una linea di trasporto pubblico. Realizzare una funzionalità del genere implica dover recuperare ad intervalli regolari la posizione esatta del dispositivo che l'utente sta utilizzando ed inviare una richiesta alla Java Application che si occuperà di aggiornare i dati in suo possesso.

Services

Per task di questo genere, Android mette a disposizione dello sviluppatore una serie di API che ne permettono un'implementazione facile e veloce: i **Servizi**.

Secondo la documentazione ufficiale, un Servizio è "una componente applicativa utile ad eseguire un task dalla durata prolungata, che può continuare anche se l'utente passa ad un'altra applicazione" (Google [2021d]).

In Android esistono due tipi di Servizi:

- **Foreground**;
- **Background**

Un **Background Service** è un servizio che permette di svolgere task, come suggerisce la denominazione, in background (ovvero anche quando l'Activity che l'ha fatto partire non è visibile a schermo). L'utente solitamente non è tenuto a sapere se e quando questo tipo di servizi sia in esecuzione. Quest'ultimo aspetto potrebbe portare a problemi di privacy o sicurezza in caso di impiego da parte di applicazioni malevole o anche di prestazioni nel caso in cui il task da svolgere sia pesante in termini di utilizzo di risorse. Per questa ragione, dalla versione 26 delle API (ovvero a partire dalla versione 8.0 di Android), sono state introdotte delle limitazioni ai servizi in background.
Se un'applicazione non è più visibile a schermo (nella maggior parte die casi, quando l'utente torna alla *home* sul proprio device), essa è considerata essere in background. Quando un'applicazione è in background, a partire da Android 8.0 il sistema le concede un intervallo di tempo prestabilito, scaduto il quale il sistema **termina tutti servizi in background dell'applicazione**. Data la natura della feature che deve essere realizzata, questo comportamento risulta non accettabile (e dunque non è stato possibile implementarla tramite un background service).

Un **Foreground Service**, invece è "un servizio che permette di svolgere task di cui l'utente dovrebbe essere informato" (Google [2021b]). Per funzionare, un Foreground Service deve obbligatoriamente mostrare una notifica all'utente nella barra di stato, appunto per permettere all'utente di rendersi conto della presenza di quel task in esecuzione. Applicazioni che fanno uso di Foreground Service sono per esempio i player musicali, che mostrano come notifica la canzone attualmente in riproduzione. Contrariamente a quanto ci si aspetterebbe dalla denominazione, i Foreground Service **possono essere in esecuzione anche in background**, a patto che siano stati avviati dall'utente o da un'Activity in esecuzione in quel momento. Il vantaggio principale di utilizzare un Foreground Service sta nel fatto che, mentre quest'ultimo risulta in esecuzione, l'applicazione che lo ospita non viene considerata in background ed il Sistema non la termina per recuperare risorse. Inoltre, da Android 10 (API 29) in su è stata rimossa la possibilità di concedere il permesso alla geolocalizzazione anche in background, demandando all'utente il compito di concederlo direttamente dalle impostazioni di sistema. Alla luce di quanto si è detto, l'utilizzo di un foreground service è un buon modo per evitare la gestione di questa limitazione. In ShallWeGo, infatti, si è deciso di utilizzare quest'approccio per la gestione degli aggiornamenti sulla posizione.

Al di là della tipologia di servizio, ognuno di questi deve essere dichiarato all'interno del *Manifest* dell'Applicazione, che può essere visto come un descrittore di ciò che l'app contiene. Nel Manifest vanno specificate tutte le activity e tutti i servizi presenti.

LiveTracking

LiveTracking è il Foreground Service che si occupa degli aggiornamenti sulla posizione di una corsa tracciata da un utente.

```
<service
    android:name=".LiveTracking"
    android:foregroundServiceType="location"
    android:enabled="true"
    android:exported="true"
    android:stopWithTask="true">
</service>
```

Listing 9: Snipped di AndroidManifest.xml che contiene la dichiarazione del Servizio

Si noti l'attributo `android:foregroundServiceType`. È un attributo specifico dei Foreground Services che ne determina il tipo. Se non lo si specifica, quando il Servizio sta per essere avviato, il Sistema Operativo solleva un'eccezione.

L'attributo `android:stopWithTask` permette di specificare se il servizio debba essere arrestato quando l'utente chiude dal multitasking o termina l'applicazione (se posto a **`true`**) o meno (se posto a **`false`**). Si è scelto di utilizzare quest'opzione per dare all'utente un modo veloce per terminare la propria corsa in modo veloce senza passare per l'applicazione (o in caso di possibili malfunzionamenti).

Avvio del Servizio

Quando un utente decide di condividere la propria posizione allo scopo di eseguire il tracciamento in diretta di una corsa, è invitato, nell'Activity **NewRide** a compilare una serie di campi che indicano lo stato della corsa e la linea che la espleta. A questo punto, la **MainActivity** provvede ad inviare i dati alla Java Application tramite una chiamata all'API `initRide`, che conterrà tutti i dettagli sulla corsa. Il server risponderà con l'**ID** della corsa appena comunicata. Alla ricezione di questa risposta, la MainActivity provvede ad avviare il servizio tramite il metodo `startForegroundService(Intent intent)`. Il parametro `intent` del metodo rappresenta "una descrizione astratta di un'operazione da svolgere" (Google [2021c]). In generale, gli Intent vengono utilizzati per scambiare dati di piccola entità tra le varie Activity all'atto del loro avvio, ma anche per comunicare ad un Sevizio che si intende avviarlo o comunque comunicargli qualcosa, come nel caso di LiveTracking, mediante il metodo `setAction()`, che va a specificare l'azione che si intende far intraprendere al servizio, che sarà interpretata da quest'ultimo tramite la sovrascrittura del metodo `onStartCommand()`. Per l'avvio del Servizio, l'Action dell'intent inviato tramite `startForegroundService(Intent intent)` risulta essere *START_SERVICE*.

Aggiornamenti periodici: FusedLocation e LocationUpdates

Per ottenere informazioni sulla geolocalizzazione, il servizio fa uso dell'API di Google denominata **FusedLocation**, presente nella libreria dei **Google Play Services** dedicata proprio alla gestione della posizione. FusedLocation è "un'API di localizzazione che combina intelligentemente diverse fonti (GPS o Rete Mobile) per mettere a disposizione le informazioni di geolocalizzazione alle app". (Google [2021e]).

La particolarità dell'API FusedLocation consiste nel supporto ai **Location Updates**, tramite i quali è possibile ottenere aggiornamenti continui sulla posizione del dispositivo specificandone la precisione e l'intervallo col quale ottenere un nuova posizione. Nello specifico, per poter iniziare a ricevere aggiornamenti continui sulla posizione, è stato necessario ottenere un'istanza del `FusedLocationProviderClient` tramite il metodo `getFusedLocationProviderClient`. Una volta ottenuto quest'oggetto è necessario andare a creare una nuova **LocationRequest**, tramite il metodo statico `LocationRequest.create()`. Questo oggetto serve a determinare i vari parametri della richiesta, come la priorità che le verrà riservata e l'intervallo (in millisecondi) tra una richiesta e l'altra. Come ultimo step, è necessario definire un **metodo di callback**, ovvero un metodo che verrà eseguito in corrispondenza del risultato di una singola richiesta. Il metodo di callback è definito tramite l'implementazione di una **classe anonima** chiamata appunto `LocationCallback`.
Il metodo di callback implementato nel servizio LiveTracking si occupa, una volta disponibile una posizione, della comunicazione di quest'ultima col server. La Java Application espone un'API chiamata `updateRideLocation`, che permette proprio di portare a termine questo task. Il processo avviene iterativamente grazie proprio al meccanismo dei Location Updates.

Termine del Servizio

Il processo di comunicazione della posizione in diretta al server termina nello stesso esatto momento in cui termina il Servizio. Il Servizio può terminare in due modi:

- L'utente preme il **pulsante di terminazione** sulla notifica nella barra delle notifiche che comunica l'esecuzione del processo.
- L'utente chiude l'applicazione e la rimuove dalle app recenti.

Quando una delle due condizioni si verifica, al servizio è passato un **Intent** che contiene l'azione *STOP_SERVICE*. In questo caso, il client provvede a contattare l'API `terminateRide`, che consente al server di cancellare la corsa dall'elenco di quelle in esecuzione.

Comunicazione tra Servizio ed Activity

LiveTracking si occupa nello specifico di aggiornare i dati sulla geolocalizzazione e del loro invio al server ma non si occupa (e non può occuparsi) di rendere coerente l'Interfaccia Utente dei cambiamenti che raccoglie. Sulla mappa presente nella MainActivity è presente, quando il tracciamento in diretta è attivo, un Marker che rappresenta la posizione in un

determinato momento di quella specifica corsa. All'aggiornarsi della posizione, anche quella del Marker deve essere aggiornata: sorge quindi la necessità di mettere in comunicazione in qualche modo il Servizio e la MainActivity. Questo problema può essere risolto mediante il meccanismo dei **Broadcast**. Un Broadcast è un tipo specifico di messaggio che viene inviato da un'app o dal Sistema Operativo quando avviene un evento considerato di interesse. Questo meccanismo funziona anche tra più componenti della stessa applicazione (ed in questo caso non è presente nemmeno l'overhead introdotto dalla *Inter-Process Communication*). Quest'ultimo apsetto, unito all'immediatezza con la quale avviene questa comunicazione, ha fatto sì che la scelta per l'implementazione tra LiveTracking e la MainActivity ricadesse proprio su questo meccanismo. Nello specifico:

- All'interno del servizio, viene sfruttato il metodo `sendBroadcast(Intent intent)`, che permette di inviare un broadcast che ha come *action* quella specificata da `intent`.
- All'interno della MainActivity è presente un `LocalBroadcastReceiver` che, tramite il metodo `onRecevive()` permette di intercettare messaggi Broadcast. All'interno di questo metodo è implementata la logica per aggiornare la UI sia quando è presente un nuovo aggiornamento della posizione da parte del provider FusedLocation sia quando il Servizio termina e il Marker che simboleggia la corsa che l'utente stava tracciando deve essere rimosso dalla mappa.

Per permettere il funzionamento dei receivers è necessario **registrare** il ricevitore, tramite il metodo `registerReceiver(BroadcastReceviver receiver, IntentFilter filter)`. Il secondo parametro del metodo, filter, è necessario per specificare quali tipi di messaggi broadcast dovrebbero far attivare il ricevitore. È possibile specificarli tramite il metodo `setAction(String action)` della classe `IntentFilter`, dove *action* corrisponde al tipo di messaggio che si vuole aggiungere al filtro.

CAPITOLO 5

Sviluppi futuri

CAPITOLO 6

Conclusioni

BREVE SPIEGAZIONE CONTENUTO CAPITOLO

Bibliografia

[Adnkronos 2019] ADNKRONOS: Raggi: Orari in tempo reale su GMaps. (2019). – URL https://www.adnkronos.com/raggi-orari-bus-in-tempo-reale-su-gmaps_3KnBSCR6JLTAX1ZwI5oLhS (Citato a pagina 9)

[Asstra 2021] ASSTRA, Agens: Trasporto pubblico locale – Interventi prioritari. (2021), S. 12 (Citato a pagina 1)

[De Girolamo 2021] DE GIROLAMO, Alfredo: TPL in ginocchio: due miliardi in meno di ricavi. (2021). – URL https://www.ilsole24ore.com/art/tpl-ginocchio-due-miliardi-meno-ricavi-AD4mLaFB (Citato a pagina 1)

[Goncalves 2013] GONCALVES, Antonio: *Beginning Java EE 7*. 1st. USA : Apress, 2013. – ISBN 143024626X (Citato alle pagine 29 e 32)

[Google 2021a] GOOGLE: *Android API Documentation: Activity*. 2021. – URL https://developer.android.com/reference/android/app/Activity. – [Controllato il 30-08-2021] (Citato a pagina 38)

[Google 2021b] GOOGLE: *Android API Documentation: Foreground Services*. 2021. – URL https://developer.android.com/guide/components/foreground-services. – [Controllato il 30-08-2021] (Citato a pagina 45)

[Google 2021c] GOOGLE: *Android API Documentation: Intent*. 2021. – URL https://developer.android.com/reference/android/content/Intent. – [Controllato il 01-09-2021] (Citato a pagina 46)

[Google 2021d] GOOGLE: *Android API Documentation: Services*. 2021. – URL `https://developer.android.com/guide/components/services`. – [Controllato il 30-08-2021] (Citato a pagina 44)

[Google 2021e] GOOGLE: *Google Play Services API Documentation: FusedLocationProvider*. 2021. – URL `https://developers.google.com/location-context/fused-location-provider`. – [Controllato il 31-08-2021] (Citato a pagina 46)

[The Java EE 7 Authors 2014] THE JAVA EE 7 AUTHORS: *The Java EE 7 Tutorial*. 7.0. https://docs.oracle.com/javaee/7/tutorial/persistence-intro.htm: , 2014 (Citato a pagina 28)

[Wikipedia 2021] WIKIPEDIA: *Object-Relational Mapping — Wikipedia, L'enciclopedia libera*. 2021. – URL `https://it.wikipedia.org/wiki/Object-relational_mapping`. – [Online; controllata il 19-agosto-2021] (Citato a pagina 28)

Siti Web consultati

- OpenStreetMap Nominatim – `https://nominatim.org`

Ringraziamenti

INSERIRE RINGRAZIAMENTI QUI