POLITECNICO DI TORINO

Corso di Laurea in Ingegneria del Software e dei Sistemi Digitali

Tesi di Laurea Magistrale

Sviluppo e progettazione di un'applicazione in ambiente iOS - Mobilitá Dinamica

Analisi dell'impronta di mobilitá del cittadino

Relatore:

prof. Giovanni Malnati

Candidato:

Filippo MAGRI'

ANNO ACCADEMICO 2017-2018

Indice delle Figure iii
Indice delle Fonti delle Figure v
Sommario vi
1. Capitolo 1 – Introduzione 1
1.1 La mobilità urbana 1
1.2 Il Progresso e l'impatto ambientale 3
1.3 Enti pubblici e la raccolta dei dati 4
1.4 Perché la raccolta dati è spesso incompleta 5
1.5 Fabbisogno di mobilità 6
1.6 Come è possibile intervenire 9
2. Capitolo 2 – Architettura Generale 11
2.1 I componenti principali 11
2.2 Applicazione Android 13
2.2.1 Localizzazione utente in ambito geografico 14
2.2.2 Riconoscimento del movimento e tracciatura del percorso 15
2.2.3 Geofencing 16
2.2.4 Funzionalità accessorie 16
2.3 Applicazione in iOS 20
3. Capitolo 3 – Funzionalità e Librerie 22
3.1 L'idea di base 22
3.2 La background mode di Apple 23
3.2.1 Eseguire task in background con un tempo limitato 24
3.2.2 Scaricare contenuti in background 24
3.2.3 Implementare task lunghi in background 26
3.2.3.1 Monitoraggio della posizione dell'utente 27
3.3 CoreLocation e MapKit 27
3.4 La mappa di calore 32
3.5 Rilevazione del movimento e co-processore M7 37
4. Capitolo 4 – Implementazione 40
4.1 Fase di avvio: TutorialVC e PolicyVC 40
4.2 Schermata Home: HomeVC 43
4.2.1 Modalità di Viaggio Manuale 47
4.3 Position Manager 51
4.4 Motion Recognition 67
5. Capitolo 5 – Test energetici 72
5.1 Algoritmi precedenti 72
5.2 Algoritmo finale 72
5.2.1 Confronto qualitativo: Android vs iOS 73
5.2.1.1 Utente leggermente dinamico 74
5.2.1.2 Utente mediamente dinamico 75
5.2.1.3 Utente dinamico 76
5.2.2 Confronto numerico: iOS vs Android 77
6. Capitolo 6 – Conclusione 80
6.1 Considerazioni finali e Sviluppi futuri 80
Bibliografia 81

Indice delle Figure

Figure 1 - Omnibus a due piani nella sua piena capacità 2
Figure 2 - Polveri sottili unione europea 3
Figure 3 – Paesi coinvolti nello studio 7
Figure 4 - Viaggi annuali per persona nei paesi più grandi (2015) 8
Figure 5 - Distribuzione media dei trasporti pubblici 8
Figure 6 - Trend di crescita dei grandi paesi (2000-2015) 9
Figure 7 - Architettura a microservizi 12
Figure 8 – Distribuzione sistemi operativi mobili in Francia (Luglio 2018 - Luglio 2018) 14
Figure 9 - Esempio di percorso di mobilità 15
Figure 10 - Esempio di un geofence in Android 16
Figure 11 - Schermata home e aggiunta di un nuovo viaggio 17
Figure 12 - Statistiche di viaggio 18
Figure 13 - Menu a tendina con schermate accessorie 19
Figure 14 - Mappa di calore e spostamenti cittadini 20
Figure 15 - Screenshots Mobilità Dinamica per iOS 21
Figure 16 - Sezione background-modes dentro la scheda Capabilities del progetto Xcode 26
Figure 17 - Esempio mappa di calore Android su Torino 32
Figure 18 - Estratto guida ufficiale per utilizzo della libreria DTMHeathmap 33
Figure 19 - Porzione di codice in ObjectiveC della libreria DTMHeatmap 34
Figure 20 - Porzione di codice scritta in Swift della funzione **loadHeatingPointsFromDataModel** dentro la classe DataManager 35
Figure 21 - Porzione di codice scritta in Swift dove viene richiamata la funzione **setUpHeatingPoints** 35
Figure 22 - Porzione di codice scritta in Swift della funzione **setUpHeatingPoints** in cui viene usata la libreria DTMHeatmap 35
Figure 23 - Porzione di codice scritta in Swift dove viene gestito il rendering per la mappa di calore 36
Figure 24 - Esempio mappa di calore su iOS 36
Figure 25 - Porzione di codice della libreria SOMotionDetector nel file SOMotionDetector.m 38
Figure 26 – Porzione di codice del metodo startDetection nel file SOMotionDetector.m 38
Figure 27 - Porzione di codice del metodo calculateMotionType dentro il file SOMotionDetector.m 38
Figure 28 - Porzione di codice di startDetection dentro i file SOMotionDetector.m 39
Figure 29 - Porzione di storyboard in cui viene gestita la fase iniziale 40
Figure 30 - Porzione di codice della classe PolicyVC 41
Figure 31 - Metodo subscribeAndLogin nella classe PolicyVC 41
Figure 32 – TabBarController con identificativo 'main' nella storyboard 42
Figure 33 – Visione riassuntiva delle relazioni a cui il TabBarController è direttamente collegato 42
Figure 34 - Home Screen iOS 43
Figure 35 - HomeVC IBOutlets 44
Figure 36 - HomeVC **viewDidLoad()** 45
Figure 37 - HomeVC **viewWillAppear()** 46

Figure 38 - HomeVC **viewDidAppear()** 46
Figure 39 - HomeVC **getDashboardData()** 47
Figure 40 - HomeVC **addTripButtonPressed()** 48
Figure 41 - Screens Viaggio Manuale 49
Figure 42 - HomeVC **currentTripChanged()** 50
Figure 43 - HomeVC **setConfigurationForTrip()** 51
Figure 44 - PositionManager: static property 52
Figure 45 - PositionManager **init()** 52
Figure 46 - PositionManager **customWillTerminate()** 52
Figure 47 - PositionManager **detectShaking()** 53
Figure 48 - PositionManager **locationManager(_manager, didUpdateLocations)** 54
Figure 49 - PositionManager **locationManager(_manager, didUpdateLocations)** [Folded Version] 56
Figure 50 - PositionManager **locationManager(...)**, primo flusso di esecuzione 57
Figure 51 - PositionManager **locationManager(...)**, secondo flusso di esecuzione 59
Figure 52 - PositionManager: **startMonitoringUser(...)** 62
Figure 53 - PositionManager: **timedCode()** 62
Figure 54 - PositionManager: frammento di codice **didUpdateLocations(...)** 63
Figure 55 - PositionManager: frammento di codice **didUpdateLocations(...)** 64
Figure 56 - Signature: **startMonitoring(for region)** 64
Figure 57 - PositionManager: **evaluateGeofenceActivationAndInsertLocationIntoDB(...)** 65
Figure 58 - PositionManager: closure di **evaluateGeofenceActivationAndInsertLocationIntoDB(…)** contenuta in un frammento di **didUpdateLocations(…)**. 66
Figure 59 - PositionManager: **handleOnExitEvent(…)** 67
Figure 60 - PositionManager: **calculateMotionType()** dentro **didUpdateLocations(...)**. 68
Figure 61 - PositionManager: **checkAuthorizationForMotionActivity()** 68
Figure 62 - PositionManager: **CalculateMotionType()** 70
Figure 63 - Scenario di utilizzo 1: Android - iOS 74
Figure 64 - Scenario utilizzo 2: Android – iOS 75
Figure 65 - Scenario di utilizzo 3: Android – iOS 76
Figure 66 - Istogramma energetico medio: iOS vs Android 78

Indice delle Fonti delle Figure

Fonte 1 - https://www.flickr.com/photos/londonmetropolitanarchives/6197734146/in/photostream/ 2
Fonte 2 - Air quality in Europe, 2017 3
Fonte 3 - Urban Public Transport in the 21st Century (UITP), Ottobre 2017 7
Fonte 4 - Urban Public Transport in the 21st Century (UITP), Ottobre 2017 8
Fonte 5 - Urban Public Transport in the 21st Century (UITP), Ottobre 2017 8
Fonte 6 - Urban Public Transport in the 21st Century (UITP), Ottobre 2017 9
Fonte 7 - https://smartbear.com/SmartBear/files/d9/d916bdd6-b349-495a-8d7d-53ce604e4e58.png (11) 12
Fonte 8 - http://gs.statcounter.com/os-market-share/mobile/france/#monthly-201707-201807 14
Fonte 9 - https://developer.android.com/training/location/geofencing#CreateAdd 16

Sommario

Lo scopo del progetto è stato quello di sviluppare un'applicazione *mobile* a supporto della mobilità degli utenti, volto a fornire una visione di insieme alla pubblica amministrazione della città francese di Compiégne che ha commissionato il software.

L'applicazione è in grado di capire il tipo di movimento dell'utente che sia esso in bici, in macchina, a piedi oppure di corsa. É inoltre in grado di comprendere la posizione dell'utente e di geo-localizzarlo con una buona precisione all'interno della città.

Tutto questo è stato possibile anche grazie all'intero sistema sviluppato dal team di TonicMinds s.r.l. Esso è composto da varie parti tra le quali: un server, un database, due applicazioni *mobile* una lato Android e l'altra lato iOS ed infine una applicazione web che viene adoperata dalla pubblica amministrazione di Compiégne per l'analisi dettagliata dell'impronta di mobilità dei suoi cittadini.

Il target fondamentale della cittadina francese sarà fornire un servizio di trasporto pubblico gratuito a tutti i suoi abitanti, dunque per farlo in modo coerente usufruirà in modo pragmatico della totalità dei dati raccolti dal sistema sia attraverso l'applicazione in versione Android sia attraverso la versione iOS.

In questa tesi mi sono occupato dello sviluppo dell'applicazione in ambiente iOS, e riassumerò le fasi seguite durante il progetto.

In una prima parte introduttiva si parlerà della mobilità urbana come concetto generale, indicando gli *step* che si sono susseguiti nell'evoluzione di essa fino ai giorni nostri. Inoltre verranno citati degli studi di approfondimento inerenti a due ambiti di interesse: l'impatto che l'industrializzazione e la diffusione dei mezzi a motore ha avuto sull'ambiente, e l'evoluzione del fabbisogno di mobilità delle cittadine moderne che sono in continua crescita e alla ricerca di soluzioni sempre più *smart*.

In una seconda parte verrà descritta l'architettura generale dell'intero sistema, con una visione ad ampio spettro delle interazioni fra i vari elementi, delle tecnologie utilizzate, e delle funzionalità già esistenti nell'applicazione Android.

Nella terza parte si partirà dall'idea di base che ha portato alla realizzazione di questo progetto, continuando con un approfondimento sulle librerie adoperate e sui motivi che erano alla base di determinate scelte di progettazione.

Nella parte centrale descriverò la fase di sviluppo, dunque si scenderà nel dettaglio delle singole implementazioni, attenzionando le porzioni più cruciali che hanno permesso il raggiungimento di certi obiettivi.

Infine descriverò i test effettuati e i possibili sviluppi futuri per aumentare eventualmente l'usabilità e l'attrattività del prodotto.

1. Capitolo 1 – Introduzione

1.1 La mobilità urbana

Camminare è stata la prima forma di mobilità urbana e allo stesso tempo la forma di spostamento più sostenibile in assoluto con un impatto ambientale pari a zero.

Intorno al 2000 A.C. l'uomo ha imparato ad addomesticare i cavalli e dunque ha iniziato ad usarli nel trasporto merci.
Negli anni a venire ci sono state altre evoluzioni tecnologiche che hanno influito direttamente sul modo di spostarsi delle persone: il carro da traino (181 – 234 D.C.) usato per trasportare ingenti quantità di merci, l'evoluzione delle infrastrutture di collegamento come ad esempio le strade romane in lastricato (100 – 200 D.C.) che permettevano un passaggio più veloce dei mezzi, gli zoccoli per i cavalli (770 D.C.) usufruiti per far percorrere agli animali distanze sempre più lunghe.

Ma esattamente quand'è che l'uomo si è iniziato a interessare allo spostamento efficiente all'interno della città, o ancor meglio al trasporto pubblico?
Possiamo attribuire la paternità di questa idea al famosissimo matematico e fisico Blaise Pascal, noto ai più per la famosa unità di misura che prende appunto il suo nome, ma che nel 1662 coniò la prima idea di trasporto pubblico.

La prima linea di autobus pubblica infatti fu lanciata a Parigi in Francia, dove erano stati creati dei percorsi di carrozze trainate da cavalli con una capacità di 7-8 passeggeri e ciascuna carrozza era programmata per attraversare le strade parigine per tutto il giorno.
Evidentemente Pascal era un uomo molto avanti ai suoi tempi, ma purtroppo le persone non erano ancora pronte per questo tipo di approccio innovativo infatti l'idea perse prontamente popolarità nel corso di circa dieci anni.
La causa principale fu il costo che era troppo elevato e dunque le corse venivano usate solo dalla nobiltà. Inoltre ai comuni cittadini all'epoca non era mai permesso cavalcare dunque anche questo elegante concetto di trasporto pubblico non prese piede, i tempi non erano ancora maturi. Nel 1675, la novità si era esaurita e la popolazione aveva abbandonato l'idea del trasporto pubblico. (1)

Velocemente in avanti di circa 150 anni, e l'idea del transito pubblico ha finalmente iniziato a prendere piede. Precisamente nel 1826 sempre in Francia nasce Omnibus che era sostanzialmente un vagone trainato da uno o più cavalli a seconda della dimensione. Alcuni avevano addirittura la parte superiore aperta. E i prezzi erano scesi significativamente, tanto da interessare non soltanto la parte benestante della città ma soprattutto la classe media.

Fonte 1 - https://www.flickr.com/photos/londonmetropolitanarchives/6197734146/in/photostream/

Figure 1 - Omnibus a due piani nella sua piena capacità

La Francia dunque fu di nuovo la prima a testare il concetto di spostamento urbano pubblico e senza andare troppo in là nel tempo, già nel 1828, l'idea si diramò anche in altre città europee arrivando persino in America fino a New York City.

Pochi anni più tardi nell'intorno fra il 1826 e il 1873 si iniziano ad utilizzare degli approcci un po' più moderni, sia in Europa sia oltre oceano. Un esempio è la famosa funicolare di San Francisco, che prese piede in quegli anni proprio per eliminare la necessità di gestione dei cavalli che dovevano ancora tirare pesanti carrozze lungo le colline della vallata. Questi nuovi tram si spostavano grazie a un morsetto che era attaccato nella parte bassa della cabina a un cavo di metallo, e quest'ultimo si muoveva grazie a un motore fuori bordo che era posizionato a valle o a monte dell'intero impianto tramviario. (1)

Nel 1860 intanto, sempre in Francia, nasce proprio il termine bicicletta inteso più o meno come lo conosciamo noi oggi cioè azionato attraverso i pedali e con a bordo anche un sistema frenante. Proprio in quel periodo la bicicletta cominciò ad essere sempre più utilizzata come mezzo di trasporto alternativo. (2)

Nel 1881 iniziano ad essere impiegati i tram per come li conosciamo oggi, ossia azionati attraverso un cavo ad alta tensione posizionati al di sopra delle vetture seguendo parallelamente le strade. Le ruote di metallo a contatto con i binari agivano come messa a terra per il circuito evitando dunque che i passeggeri venissero folgorati con una scarica elettrica. Questa invenzione portò a un ribasso del prezzo per la singola corsa, coinvolgendo dunque sempre più pedoni a farne buon uso.

Nel 1886 il famoso ingegnere tedesco Karl Friedrich Benz, considerato l'inventore dell'autovettura, costruì il primo veicolo azionato attraverso un motore a combustione interna. Esso era capace di raggiungere la velocità massima di 16 Km/h e questo ovviamente fu un grande traguardo tecnologico. (3) Ma all'epoca ovviamente ancora era presto per considerarlo

il principale mezzo di trasporto delle persone comuni, infatti nei paesi occidentali le uniche scelte di trasporto pubblico disponibili alla maggior parte della popolazione rimanevano il trasporto via tram, e la bici.

Gli anni del dopoguerra hanno portato con sé un incremento del benessere e della domanda di mobilità sia di persone sia di merci. Tra il 1950 e il 1979 in Gran Bretagna il numero di strade per i veicoli è quintuplicato, e un andamento simile si è avuto anche in altri paesi occidentali. Si è iniziato a investire maggiormente nella progettazione di strade sempre più grandi e più belle. Quei periodi furono talmente prosperi per le aziende automobilistiche che in alcuni paesi come Stati Uniti e Gran Bretagna addirittura ci fu una diminuzione dell'investimento pubblico sullo spostamento a piedi e su quello in bici. (4)
Durante gli anni ottanta/novanta il prezzo del petrolio rimase basso e pressoché stabile e questo causò l'incremento significativo delle automobili fino agli anni duemila.

1.2 Il Progresso e l'impatto ambientale

L'avvento dell'automobile, inteso come mezzo di trasporto personale, ha facilitato e incoraggiato l'intensiva decentralizzazione e urbanizzazione delle aree suburbane. Le persone hanno iniziato a trovare più gradevole vivere in zone periferiche della città in modo da essere più vicini alla natura. L'automobile è stata sicuramente un segno distintivo di progresso tecnologico ma allo stesso tempo al giorno d'oggi è stata criticata su molti aspetti. Ad esempio l'utilizzo eccessivo di essa ha portato ad un quantitativo di emissioni di polveri sottili sempre più alto, persino maggiore a quello prodotto dai treni. (5)

L'EEA (European Environment Agency) è un'agenzia europea che si occupa di monitoraggio e qualità ambientale a livello europeo e con cadenza biennale pubblica dei report molto dettagliati.
In uno dei loro ultimi report "Air quality in Europe" dell'anno 2017 viene fatto uno studio aggiornato sulla qualità dell'aria in un arco di tempo pari a 15 anni a partire dagli anni 2000, usufruendo dei dati provenienti da stazioni di monitoraggio presenti in tutta Europa.

Fonte 2 - Air quality in Europe, 2017

Pollutant	EU reference value (a)	Exposure estimate (%)	WHO AQG (a)	Exposure estimate (%)
$PM_{2.5}$	Year (25)	7-8	Year (10)	82-85
PM_{10}	Day (50)	16-20	Year (20)	50-62
O_3	8-hour (120)	7-30	8-hour (100)	95-98
NO_2	Year (40)	7-9	Year (40)	7-9
BaP	Year (1)	20-25	Year (0.12) RL	85-91
SO_2	Day (125)	< 1	Day (20)	20-38

Key	< 5 %	5-50 %	50-75 %	> 75 %

Figure 2 - Polveri sottili unione europea

La tabella mostrata sopra, prelevata dal medesimo report, si può vedere come tra il 2013 e il 2015 più dell'80% della popolazione urbana della unione europea ha inalato polveri sottili PM2.5 in zone cittadine con livelli di particolato superiori alle norme di leggi stabilite dall'OMS (Organizzazione mondiale della sanità). Questo pulviscolo atmosferico dannoso per la salute ha origine sia naturale sia artificiale, come ad esempio le emissioni dovute al riscaldamento domestico o alla combustione della benzina nei mezzi a motore.
La speranza dell'ente che ha effettuato questo studio è quella di abbassare entro il 2050 i valori di inquinamento atmosferico in tutta l'unione europea, migliorando significativamente la qualità dell'aria e mantenendo il tasso di inquinamento sotto certe soglie prestabilite.

Ovviamente per il raggiungimento comune di questo auspicabile obiettivo bisogna adottare delle strategie di cooperazione partendo dall'ambito locale fino ad espandersi in modo globale. Inoltre sarà importante ideare soluzioni trasversali che coinvolgano diversi settori economici spaziando dall'ambito agricolo, ma anche tecnologico e del trasporto pubblico. (6)

1.3 Enti pubblici e la raccolta dei dati

Negli ultimi decenni il modo di spostarsi delle persone sta cambiando e le città cercano di stare a passo con i tempi.
In giro per l'Europa prendono piede sempre nuovi *concept* di movimento urbano, un esempio sono i vari servizi di car sharing, car pooling, bike sharing, o il peer-to-peer ride sharing rappresentato per antonomasia dall'ormai famoso Uber che cerca di affermarsi nei vari Paesi nonostante alcuni contrasti con le società di trasporto locali. (7)

Quello che si sta cercando di raggiungere è una mobilità di merci e di persone che risulti sostenibile sia dal punto di vista economico che ambientale e sociale.

Secondo il quattordicesimo rapporto ISFORT sulla mobilità in Italia, basato su un campione di italiani di 12.200 individui (di età compresa fra i 14 e gli 80 anni) si è scoperto che in media gli spostamenti effettuati da gran parte degli italiani non supera i 12 chilometri globalmente, mentre se si guarda strettamente la tratta urbana (ossia gli spostamenti che hanno origine e fine nel perimetro comunale) la media non supera i 5 chilometri.

Le strategie adottate in Europa fanno notare quanto sia essenziale concentrarsi su modi di trasporto che non inficino negativamente sull'ambiente. A parte le modalità più classiche e ormai ben conosciute quali il trasporto in bici o quello a piedi, ce ne sono delle altre che ultimamente sono valorizzate sempre di più. Un esempio è la mobilità condivisa (dunque il car pooling, car sharing etc.), oppure quella basata sull'utilizzo di trasporti inter-modali e ovviamente ultimi ma non meno importanti quelle basate sull'utilizzo di motori puliti.
Per fare tutto questo bisogna avere a disposizione svariati tipi di informazione. Parliamo di qualcosa che sia in sintonia con le tecnologie moderne, ad esempio informazioni geo-referenziate e real-time che possano essere utilizzate su due fronti: il primo per fornire ai cittadini servizi più dettagliati possibili, e il secondo per aiutare gli amministratori delle città nella pianificazione delle infrastrutture e delle richieste annesse per eventuali finanziamenti pubblici. (7)

In generale le amministrazioni pubbliche hanno però un accesso alle informazioni limitato e poco intuitivo. Addirittura a volte i dati sono troppo vecchi, di non semplice comprensione, o formattati con un tipo difficile da maneggiare o trasformare.
Spesso c'è un grande numero di enti pubblici e privati che si occupano entrambi di trasporto e dunque questo causa confusione nel reperire le informazioni sulle necessità reali di spostamento degli abitanti. La difficoltà di accesso ai dati è anche estesa a quelle infrastrutture che dovrebbero fornirle in modo semplice e diretto, ma spesso non è così. Parliamo in questo caso di record relativi ai parcheggi, alle zone a traffico limitato. Uno dei problemi principali è che spesso i gestori appartengono a settori diversi o antagonisti addirittura. Dunque spesso accade che o non hanno le informazioni fruibili in modalità digitale oppure semplicemente decidono di non condividerle.
Questa purtroppo è la situazione nel panorama italiano, ed essa non è molto lontana da quella a livello europeo. Per sopperire a queste problematiche l'UE ha deciso infatti di investire nella ricerca, impostando come obiettivo principale il concetto di raccolta e condivisione di dati tra le svariate piattaforme che si occupano di trasporto cittadino.
Esistono progetti finanziati a livello europeo per sostenere le città stesse nella scoperta di nuovi approcci di mobilità, favoreggiando soprattutto quelle alternative o le inter-modalità di trasporto. Un esempio importante in cui figura anche il Politecnico di Torino è il progetto

Opticities. Il polito insieme all'ente CSI del Piemonte rappresentano la nazione Italia in mezzo a un totale di otto stati membri dell'UE. (7)

Questo progetto ha portato alla sperimentazione di diversi strumenti innovativi tra cui la Mappa Multimodale dei trasporti, il Carpooling Hub e l'applicazione per smartphone TUeTO. Tutti insieme hanno come obiettivo comune quello di fornire un supporto decisionale migliore al cittadino.
La Mappa Multimodale ad esempio, anche se in fase beta, fornisce una visione globale del trasporto multimodale in ambito urbano, è dunque destinata soprattutto agli operatori nel settore pubblico che si occupano della pianificazione dei trasporti.
Il Carpooling hub è un aggregatore di dati provenienti da più siti web di carpooling che offre attraverso una unica interfaccia web tutte le possibili opzioni di passaggio in auto fornito da privati; in questo modo l'utente potrà fare una scelta più oculata e veloce.
L'applicazione android TUeTO, progettata da una società francese, utilizza i dati in tempo reale della zona urbana di Torino attraverso i dati offerti dai provider locali. Essa è in grado di suggerire all'utente il percorso multimodale migliore ma considerando diversi fattori variabili come ad esempio: i cicli dei semafori, i ritardi dovuti a ingorghi, percorsi bloccati per via di lavori in corso etc.

Questa è comunque una novità importante perché differisce dal servizio offerto da Google Maps che invece non fornisce alcun tipo di informazione inerente percorsi che integrino più modi di trasporto in real-time. (7)

Tutti e tre questi strumenti sono progetti all'avanguardia, essi integrano l'idea innovativa di trasporto multimodale e la mettono al servizio del cittadino. Si è evidenziato però che per rendere questi servizi quanto più efficienti possibili è necessaria una qualità dei dati significativa e una quantità di dati elevata.

Inoltre è importante che questi dati digitali vengano memorizzati con dei formati standard e che, una volta raccolti, si utilizzi anche una architettura software che garantisca un flusso continuo e organizzato di *record*. Più ci sarà collaborazione fra enti pubblici e privati sulla condivisione dei dati digitali di mobilità urbana e più sarà il valore di queste informazioni.

1.4 Perché la raccolta dati è spesso incompleta

Approssimativamente negli ultimi cento anni si è cercato di comprendere al meglio il tipo di mobilità dei cittadini ma il metodo che è andato per la maggiore è stato quello classico del sondaggio. Quest'ultimo è basato su informazioni statiche dove al massimo vengono indicati i punti di partenza e arrivo, con un determinato mezzo di trasporto, ed eventualmente una stimata durata di percorrenza. Ovviamente, dato che questo tipo di indagine è stata per tanto tempo la risorsa principale di informazioni di mobilità, tutti coloro che erano incaricati della pianificazione delle infrastrutture, pubbliche e non, si sono basati su di esse.
Il problema principale di questa fonte di raccolta dati è che ha delle carenze macroscopiche, ad esempio non può rilevare cambiamenti nel nostro modo di viaggiare giorno dopo giorno. Quindi in definitiva questo tipo di approccio, anche se è stato adottato in larga scala, non è mai riuscito a dare una panoramica completa di come le persone effettivamente si spostano all'interno del contesto urbano. (8)

Con l'avanzare della tecnologia si è iniziato ad utilizzare metodi sempre più sofisticati:

- i contatori automatici di mezzi di trasporto sulle autostrade.
- le fotocamere speciali installate nelle zone più affollate della città che riescono a riconoscere anche il numero di bici e di pedoni.

- le tessere elettroniche di abbonamento al trasporto pubblico che una volta lette dal dispositivo a bordo del bus riescono a dare una idea al gestore dei trasporti su dove l'utente ha iniziato il suo viaggio. A Torino ad esempio da metà 2017 la società dei trasporti pubblici G.t.t. ha cercato di indurre più abbonati possibile, prima penalizzando con sanzioni chi non lo facesse e poi introducendo invece un sistema a premi per chi collaborava.
- record anonimi rilasciati dalle compagnie telefoniche per capire in base alle celle utilizzate dove gli utenti si spostassero effettivamente.

Tutti questi esempi elencati sono passi da gigante rispetto all'ormai obsoleto sondaggio, tuttavia anche questi metodi presentano dei difetti. (8)

Prendiamo in esame il semplice uso della tessera di abbonamento elettronico della G.t.t., essa è significativa per segnalare che l'utente si sta spostando a partire da una determinata fermata dell'autobus ma poi la tessera non viene ripassata di nuovo sotto il lettore quando il passeggero scende dal bus; dunque non abbiamo alcuna informazione sul dove il passeggero effettivamente finisca il suo viaggio. Questo è un tipico esempio di informazione incompleta.

Per quanto riguarda i dati rilasciati anonimamente dalle compagnie telefoniche, essi sono anche incompleti perché non basta sapere più o meno l'area in cui l'utente si è spostato ma ci piacerebbe conoscere anche il modo in cui l'utente si è spostato.
Fortunatamente però da quando c'è stato l'avvento degli smartphone si è iniziata a sentire una forte aria di cambiamento. Essi sono in grado di geo-localizzare, di fornire informazioni *real-time*, sono in grado di interpretare il tipo di movimento. Da quando sono diventati onnipresenti hanno dunque accresciuto tantissimo il valore dei dati che possono raccogliere, infatti ipotizzando di avere un bacino di utilizzatori abbastanza ampio è possibile intuire, in maniera molto più accurata del passato, il tipo di mobilità dell'utente.
Sicuramente anche l'utilizzo degli smartphone ha delle criticità da gestire, un esempio sono i dati che inizialmente potrebbero non essere ben organizzati o magari poco accurati. Inoltre un altro aspetto importante, da tenere sempre a mente, è la salvaguardia della privacy di chi effettivamente sta condividendo i suoi dati di mobilità. Bisogna trovare un modo per proteggere al meglio l'utente facendolo sentire tranquillo in modo che possa continuare collaborare.
Questo tipo di raccolta dati attraverso dispositivi intelligenti di ultima generazione è sicuramente molto innovativo e all'avanguardia ma non bisogna dimenticare che, come in tutte le cose, può comunque presentare delle debolezze o dei punti ciechi. Basti pensare a quelle persone avverse all'utilizzo della tecnologia oppure che magari non possono permettersi uno smartphone. Quest'ultime magari vivendo in zone periferiche della città sono quelle che hanno più bisogno di trasporti pubblici, perché è l'unica alternativa a loro possibile, però se ci basassimo solo sui dati raccolti dagli smartphone saremmo portati a pensare, erroneamente, che in quella zona della città non c'è bisogno di connessioni.
Dunque per prendere le decisioni più accurate possibile servirà sicuramente l'utilizzo di un po' tutte queste nuove tecnologie, ma con la consapevolezza di dover calibrare propriamente i dati alle zone urbane in questione magari cercando di unire sorgenti di dati vecchie con sorgenti di dati nuove e non dimenticando di applicare dei fattori di correzione specifici per il tipo di tecnologia che si sta usando. (8)

1.5 Fabbisogno di mobilità

La UITP, associazione internazionale per i trasporti pubblici, ha come obiettivo il miglioramento della qualità di vita e del benessere economico delle persone e lo fa promuovendo il trasporto sostenibile in tutte le aree urbane in giro per il mondo.
In questa associazione troviamo ad oggi 96 paesi e un *network* di circa 18.000 contatti.

Tra le altre cose questa organizzazione si occupa anche di raccogliere e analizzare un grande quantitativo di dati inerenti la mobilità urbana e il trasporto pubblico.
Grazie a tutte le risorse che ha a disposizione riesce a pubblicare periodicamente statistiche e report che coprono un vasto *range* di settori.
Uno degli ultimi report è il "*Urban Public Transport in the 21st Century*" pubblicato ad Ottobre 2017. Esso offre una panoramica del fabbisogno di mobilità basandosi su dati a livello globale, coinvolgendo di fatto nello studio ben 39 paesi, di cui 27 europei.

Fonte 3 - Urban Public Transport in the 21st Century (UITP), Ottobre 2017

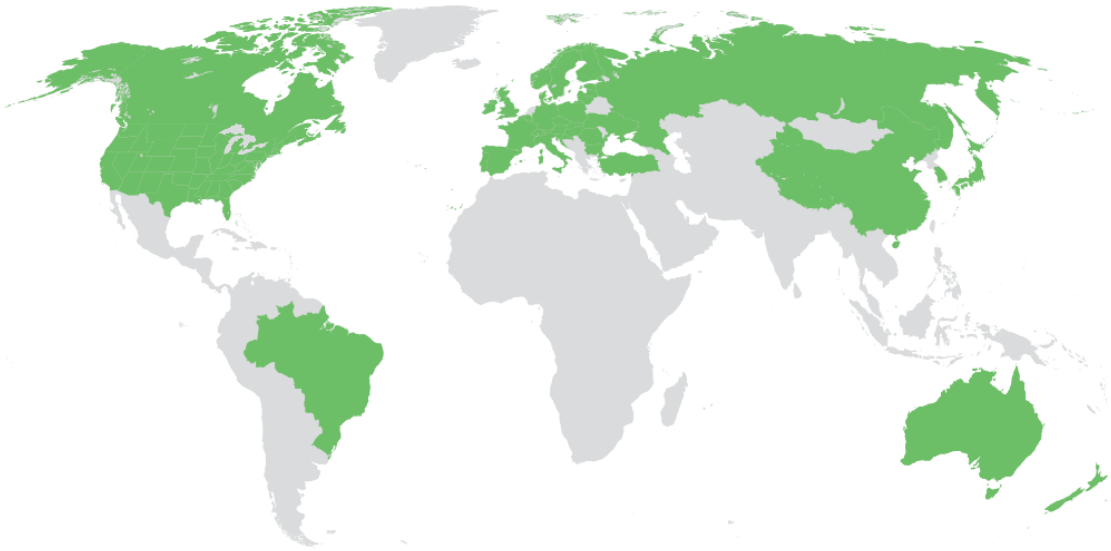


Figure 3 – Paesi coinvolti nello studio

Per dare una idea dei paesi coinvolti quella sopra è la mappa che li evidenzia con il colore verde. I dati raccolti inerenti i trasporti pubblici arrivano in stragrande maggioranza dalla Cina, mentre al secondo posto abbiamo l'Europa, presa come entità a sé stante, e al terzo il Giappone.

Fonte 4 - Urban Public Transport in the 21st Century (UITP), Ottobre 2017

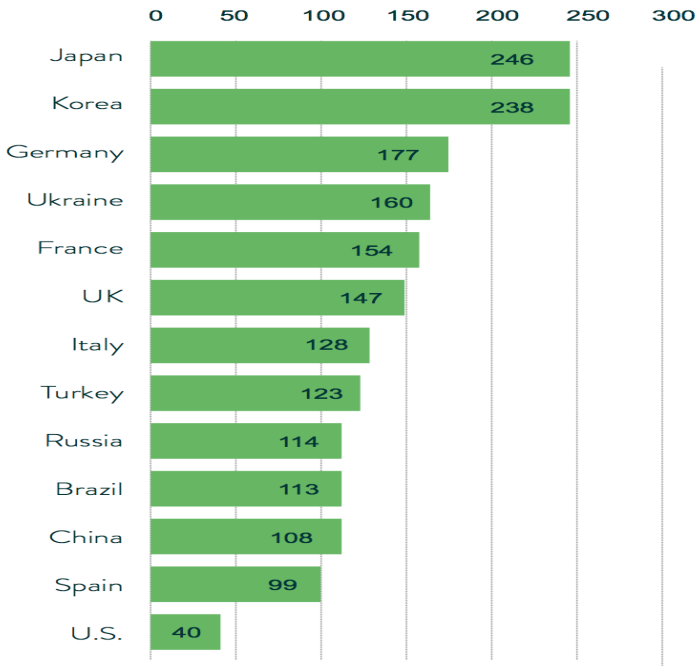


Figure 4 - Viaggi annuali per persona nei paesi più grandi (2015)

La panoramica mostrata sopra ci da una idea di quanti viaggi vengono fatti annualmente per persona, e lo stato europeo con più corse annuali è rappresentato dalla Germania, a seguire Francia, Regno Unito, Italia. (9)

Fonte 5 - Urban Public Transport in the 21st Century (UITP), Ottobre 2017

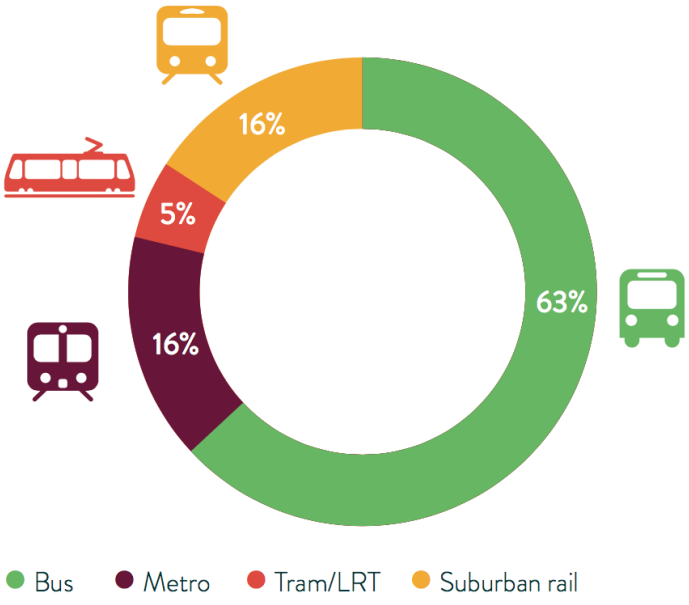


Figure 5 - Distribuzione media dei trasporti pubblici

Questo grafico mostra in media la distribuzione dei trasporti pubblici prendendo in esame tutti i paesi menzionati a inizio paragrafo esclusa però la Turchia per via di una insufficienza di dati. Si evince dallo studio che il bus resta comunque il mezzo predominante nello

spostamento cittadino e a seguire abbiamo la metro e le ferrovie suburbane.
Questa comunque resta una indicazione di massima perché se andiamo nel dettaglio scopriamo che in alcuni paesi, presi singolarmente, la distribuzione media è differente; ad esempio in Giappone il mezzo predominante sono le ferrovie suburbane.

Fonte 6 - Urban Public Transport in the 21st Century (UITP), Ottobre 2017

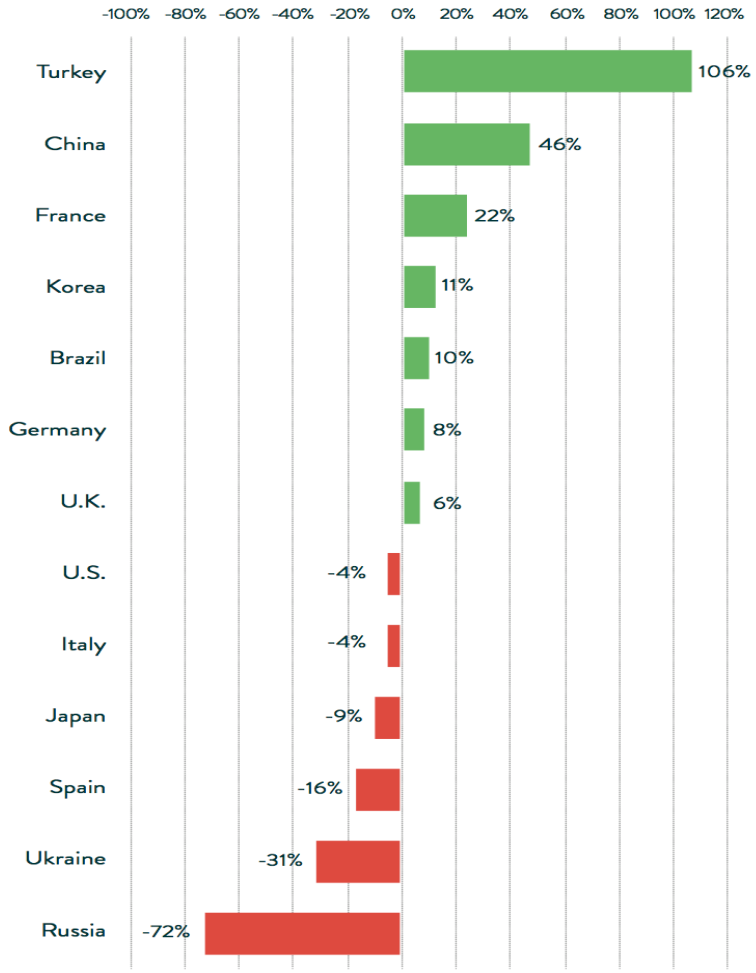


Figure 6 - Trend di crescita dei grandi paesi (2000-2015)

Nell'istogramma soprastante si vede il trend di crescita dei paesi più vasti considerando come finestra temporale quella che va dal 2000 al 2015. In alto abbiamo la Turchia e la Cina che hanno rispettivamente un tasso di crescita del 106% e del 46%, questo dato sicuramente evidenzia come i paesi in questione si stiano concentrando sul miglioramento effettivo del trasporto pubblico con risultati sostanziali. La Francia con il suo 22% non è tra le prime in classifica però se la contestualizziamo in un panorama Europeo va automaticamente in testa. (9)

In sintesi quello che si è denotato attraverso questa analisi durata 15 anni, effettuata a livello globale, è che in generale la maggior parte dei paesi sotto esame ha vissuto un incremento nell'utilizzo del trasporto pubblico o almeno ha mantenuto un utilizzo costante.

1.6 Come è possibile intervenire

Uno degli ostacoli più grandi da superare è molto frequentemente la poca cooperazione che vige fra il settore pubblico e quello privato. Quello che accade è che spesso tra di loro non condividono informazioni cruciali quali ad esempio la qualità o la quantità delle corse. I

motivi sono i più disparati, ma primo fra tutti è sicuramente la rivalità che spinge entrambe le fazioni ad essere testardi ed egoisti nei confronti dell'altro. Potrebbero addirittura fare indagini simili, mettersi d'accordo e condividere i costi ma invece di essere innovativi preferiscono essere eccessivamente riservati, con il risultato che alla fine magari raccolgono dati simili spendendo in totale il doppio del capitale. La situazione, purtroppo, è ancora più triste quando si scopre che le indagini svolte dai due concorrenti sono complementari, dunque in realtà se tra di loro collaborassero alla fine potrebbero guadagnare entrambi in termini di qualità ed esaustività dei dati. Dunque in definitiva se abbassassero un po' la guardia potrebbero auspicabilmente aiutarsi a vicenda al fine di raggiungere obiettivi ben più ampi mirati al benessere della intera maglia di mobilità urbana. (8)

Abbiamo diversi esempi in giro per il mondo che rappresentano un esempio di collaborazione da seguire per la condivisione di dati sulla mobilità. Basti prendere il caso di città come Helsinki dove le società che si occupano di trasposto pubblico se usufruiscono dei fondi statali devono obbligatoriamente tenere i loro dati aperti e accessibili a tutti. Oppure il caso di Uber a Londra che ha deciso di aprire l'accesso ai suoi dati sulle corse londinesi, ma in forma anonima. (10)

Un altro problema non meno importante è relativo al continuo aumento dei prezzi delle case. Prendiamo ad esempio le zone centrali della città, li è più probabile che una volta aumentati i servizi e le connessioni, aumentino anche i costi degli immobili. Come diretta conseguenza si ha un incremento della massa di persone che decide di spostarsi in aree leggermente meno care, magari nelle periferie dei sobborghi più vicini. Li solitamente le corse sono poco frequenti. Quello che accade però è che per tratte semplici che di solito potrebbero richiedere con la macchina dai trenta ai quaranta minuti, con i mezzi pubblici i tempi si raddoppiano o addirittura triplicano. Ecco che diventa difficile decidere quale è la frequenza giusta del servizio da offrire. Esistono anche persone che hanno dei turni lavorativi notturni, spesso la loro categoria è poco rappresentata nei questionari di mobilità e questo rappresenta un altro buco nelle informazioni atte a prendere una decisione più accurata possibile. (8)

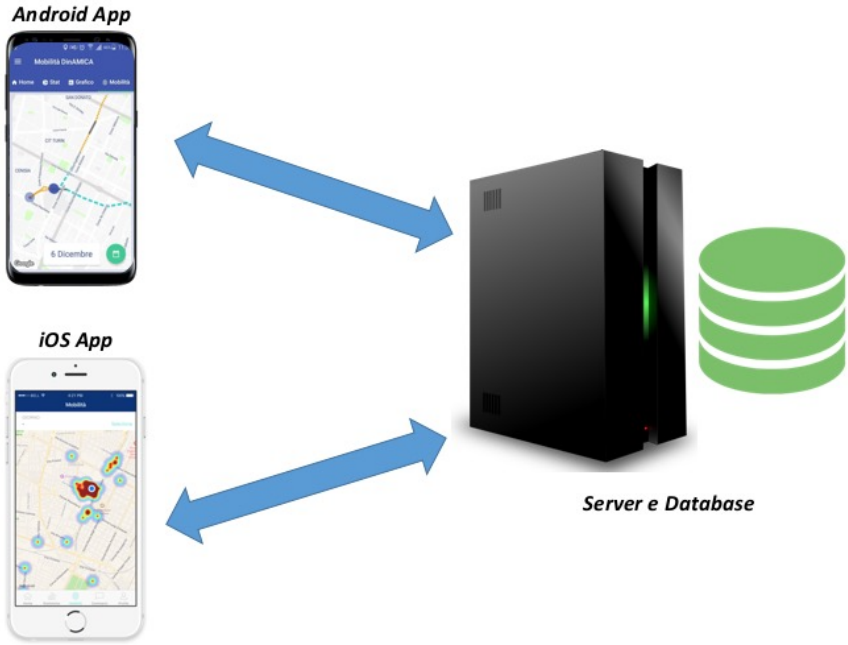
Una soluzione potrebbe essere l'utilizzo di applicazioni *real-time*, software installati nello smartphone capaci di monitorare e fornire dati sullo spostamento effettivo del cittadino. Risolverebbe tanti problemi legati alle carenze di informazione che si hanno con i normali metodi di indagine statistica.
Risolverebbe anche il problema inerente a come l'utente dal *bus-stop* percorre gli ultimi chilometri per raggiungere la destinazione finale. Questo permetterebbe di intuire meglio se c'è bisogno di creare un nuovo stallo per il bike sharing oppure se ad esempio c'è bisogno di aggiungere una nuova tratta di pista ciclabile.
Il cellulare, essendo ormai diventato quasi una estensione di noi stessi, è quasi sempre con noi dunque ha un potenziale notevole che può essere sfruttato.

La tesi in oggetto tratta proprio la realizzazione di una applicazione iOS atta alla raccolta più accurata possibile di informazioni inerenti il fabbisogno di mobilità dell'utente, all'interno di un contesto urbano.
Tante sfide sono state affrontate durante la realizzazione di questo prodotto, una fra le tante è stata come limitare il consumo di batteria che è spesso inevitabile quando si usufruisce del sensore gps o di quello di movimento.
In che modo queste sfide sono state affrontate o superate verrà trattato approfonditamente nei prossimi capitoli.

2. Capitolo 2 – Architettura Generale

2.1 I componenti principali

L'architettura dell'intera struttura messa in piedi è composta da diversi elementi:



- Il server.
- Tipologia uno di client rappresentato dall'applicazione sviluppata in Android.
- Tipologia due di client rappresentato dall'applicazione sviluppata in iOS.
- Il database contenuto dentro il server.

La parte client è rappresentata dall'applicazione eseguita sui dispositivi mobili che siano essi Android o iOS. Essa è in grado di comprendere in che modo l'utente si muove però il modo in cui lo comprende e la logica che utilizza è leggermente diversa se si parla di ambiente Android o di ambiente iOS. Entrambe saranno in grado di distinguere se l'utente si sta spostando in bici, in auto o a piedi.
Inoltre entrambe le parti client sono in grado di attivare il riconoscimento di un *geofence* ossia una sorta di confine geografico virtuale in cui l'utente staziona per un certo lasso di tempo. Questo *geofence* è importante per capire meglio in quali punti durante un certo tragitto giornaliero l'utente si ferma, che siano esse brevi o lunghe soste, memorizzando data e tempo di ingresso e data e tempo di uscita.

La parte server è costituita da un server a microservizi. Il concetto di microservizi come architettura software è nato nel 2011 a Venezia durante un *workshop*.
Essa è sostanzialmente una tecnica di sviluppo software che tende a trasformare una singola applicazione monolitica in tanti piccoli sotto problemi, gestiti singolarmente da dei sotto servizi bassamente accoppiati fra di loro.
Il beneficio di usare questa tecnica è proprio di ottenere una modularità maggiore, avendo più parti di una singola applicazione divisi in moduli più piccoli ogni microservizio avrà un suo database e potrà accedere solo e unicamente a quello. Qualora si avrà bisogno di reperire altre informazioni da altri microservizi lo si farà avviando una comunicazione http fra di loro.
Ogni elemento è disaccoppiato l'uno dall'altro dunque è possibile usare delle tecnologie diverse per ogni servizio, e l'interscambio di informazioni fra le differenti istanze di servizio

sarà fatto sempre e solo attraverso delle interfacce che sfruttano il protocollo http.
Questa tecnologia è stata scelta perché ha anche il grande vantaggio che in caso di aggiornamento di un componente, esso può essere rinnovato con estrema facilità e velocità senza impattare sugli altri microservizi dell'intera architettura software.
Un ultimo, ma non meno importante, vantaggio è che questa tipo di sistema riesce a scalare abbastanza velocemente. Questo significa che se un domani c'è bisogno di potenziare un servizio lo si potrà fare senza grandi difficoltà.

Il server è installato sulla piattaforma AWS di Amazon, e come tipo di istanza è stata scelta la T2 perché fa parte di una tipologia di servizio espandibile in termini di potenza ed è anche adatto per i microservizi. Come tipologia è stata scelta la T2.xlarge che corrisponde a un pacchetto di 4 CPU virtuali e 16 GB di ram.
Esso è stato sviluppato usando l'accoppiata Spring Boot e Spring Cloud.
Spring è la tecnologia che sta alla base di entrambe, e negli ultimi anni grazie all'avvento dei microservizi e di questi due nuovi *framework* è tornata alla ribalta espandendosi a macchia d'olio.
Spring Boot fornisce lo strumento ideale per creare microservizi. Esso una volta avviato riesce a mettere su un sistema funzionante, ma quest'ultimo ha bisogno di essere configurato adeguatamente e per farlo è stato usato Spring Cloud.

Fonte 7 - https://smartbear.com/SmartBear/files/d9/d916bdd6-b349-495a-8d7d-53ce604e4e58.png (11)

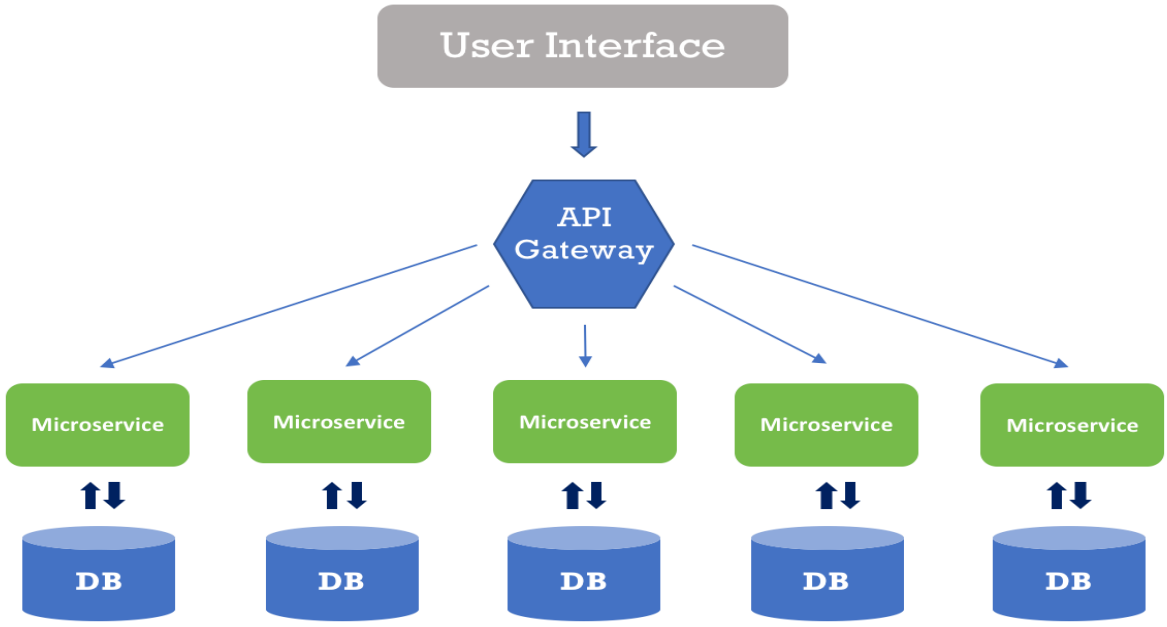


Figure 7 - Architettura a microservizi

Nell'architettura a microservizi c'è un proxy interno, rappresentato nella Figura sette sotto forma di API Gateway, che coordina gli altri microservizi. La comunicazione come detto prima avviene tra i vari microservizi attraverso l'utilizzo di http, quindi di fatto internamente è come se avessimo un sistema distribuito. La configurazione che viene eseguita con Spring Cloud è molto importante perché permette di risolvere parecchie criticità proprie dei sistemi distribuiti.

La memorizzazione dei dati è fatta attraverso un tipo di database non relazionale. Quindi un database che non si basa più sul concetto di relazione o di join, a cui di solito siamo abituati, ma bensì sul concetto di memorizzazione di documento.
Nello specifico è stato utilizzato MongoDB, esso è orientato ai documenti in stile JSON, e

questo ne rende elevata la compatibilità e l'integrazione con il resto delle applicazioni.
Ad esempio nella comunicazione tra clients e server, in questo progetto di tesi, vengono effettuate le chiamate tramite l'architettura software REST che è in grado di comunicare attraverso gli standard sia JSON che HTTP. Quindi quando viene utilizzato JSON non serve effettuare particolari conversioni per memorizzare i dati, ma vengono semplicemente incanalati dentro MongoDB.
Inoltre MongoDB supporta le *query* geo-spaziali e siccome l'applicazione lavora con dei dati inerenti le posizioni degli utenti, che possiamo considerare come dati geometrici, questa proprietà torna molto utile.
Per esempio grazie a questa caratteristica se avessimo a disposizione 4 punti in cui l'utente ha stazionato, noi potremmo chiedere al database di formare un poligono e disegnarlo su una sfera. Si potrebbero addirittura disegnare più poligoni, in base a quante posizioni dell'utente abbiamo, e vedere quanti di loro si intersecano fra di loro. Dunque tutta questa potenzialità torna utile qualora in futuro si volessero fare delle analisi geo-spaziali più dettagliate e più mirate.

2.2 Applicazione Android

All'inizio di questo progetto di tesi esisteva già una versione client dell'applicazione. La versione in questione era stata realizzata in ambiente Android. Essa era in grado già di svolgere diverse funzionalità, ovvero:

- Localizzazione dell'utente in ambito geografico.
- Riconoscimento del tipo di movimento dell'utente.
- Capacità di riconoscere un *geofence* in ingresso e in uscita.
- Possibilità, tramite UI, di interagire con altri utenti tramite una apposita sezione commenti.
- Possibilità di aggiornare il proprio profilo.
- Possibilità di aggiungere manualmente un tipo di viaggio con un determinato tipo di mezzo.
- Possibilità di visualizzare, sempre tramite UI, le statistiche riguardo l'utilizzo della applicazione.

Era stato deciso, inizialmente, di sviluppare l'applicazione client in Android perché il mercato di smartphone con a bordo questo sistema operativo era, ed è, ancora abbastanza ampio. Infatti, secondo il tool di analisi di traffico web *Statcounter*, tra luglio 2017 e luglio 2018 si stima che il mercato dei cellulari Android, in Francia, riesce a coprire fino a un picco massimo del 67.96%.

Fonte 8 - http://gs.statcounter.com/os-market-share/mobile/france/#monthly-201707-201807

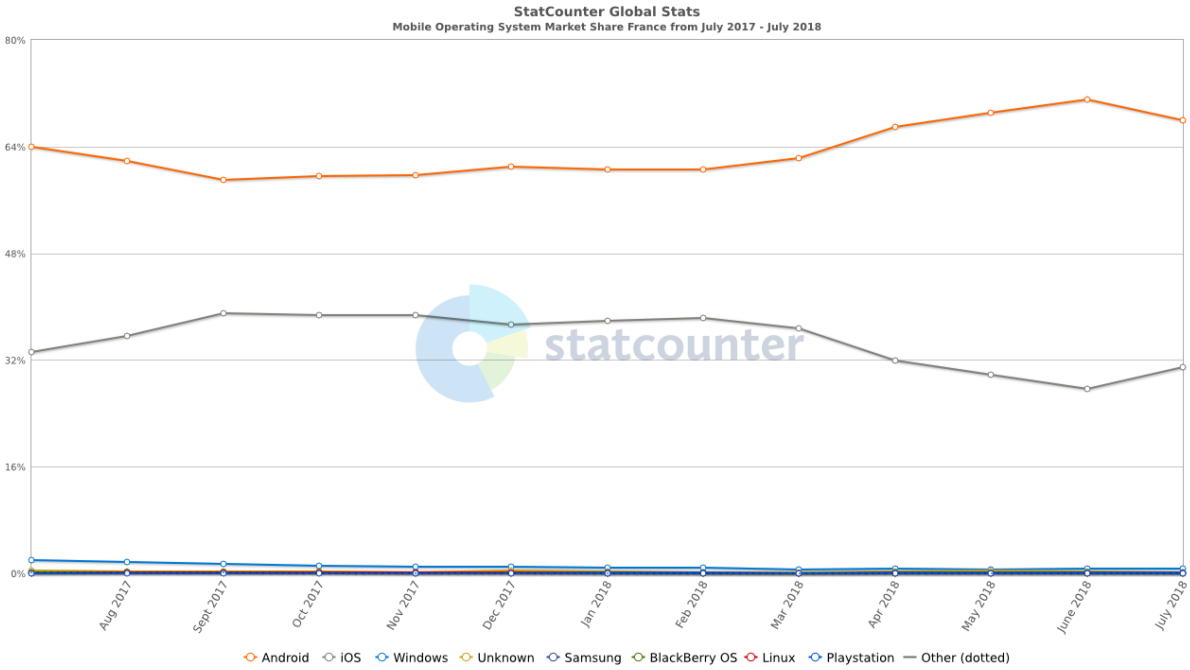


Figure 8 – Distribuzione sistemi operativi mobili in Francia (Luglio 2018 - Luglio 2018)

La rimanente maggioranza del 30.95% è coperta invece da *device* con a bordo il sistema operativo iOS.
Un altro motivo per cui si era deciso di sviluppare in Android è per via dei costi di budget iniziale. Sviluppare in ambiente Apple richiede sempre un budget leggermente più alto. Basti pensare che ad esempio per sviluppare in iOS serve prima di tutto una macchina con a bordo il sistema operativo macOS, e solitamente quest'ultima a parità di hardware con una macchina non Apple è sempre più costosa.
In secondo luogo per pubblicare delle applicazioni nello store di Apple serve un account registrato come sviluppatore iOS che richiede il pagamento di una tassa annuale di 99 dollari l'anno, mentre per pubblicare nello store di Google è sufficiente una tantum iniziale di ingresso e nient'altro.

Qualche tempo dopo il cliente della Tonic Minds s.r.l. ha pensato bene di espandere il target degli utenti a cui è rivolta l'applicazione, visto e considerato che comunque il 31% circa del bacino di utenza con smartphone Apple non era poco. Dunque è stata richiesta successivamente anche la realizzazione di una versione per iPhone, che è proprio l'argomento principale di questo progetto di tesi.
Una delle *challenge* più importanti è stata sicuramente quella di riprodurre il comportamento dell'applicazione Android nell'ambiente iOS nel modo più fedele possibile. Una sfida non da poco considerando che i due sistemi operativi nascono e si evolvono da due *background* diversi.

A seguire nei prossimi paragrafi verranno descritte le funzionalità Android che sono state analizzate e studiate approfonditamente prima di passare allo sviluppo vero e proprio dell'applicazione per iPhone.

2.2.1 Localizzazione utente in ambito geografico

Questa funzionalità è stata realizzata utilizzando la funzione di *service* che in Android agisce in background e non ha bisogno di alcuna interfaccia grafica.
In pratica all'avvio dell'applicazione viene lanciato un servizio di localizzazione che resterà

attivo per tutto il tempo necessario svolgendo le sue operazioni possibilmente su un *thread* separato dal *main thread*.
Dentro il servizio stesso la posizione dell'utente verrà recuperata grazie all'utilizzo delle api di geo-localizzazione di Google. Questa *location* verrà recuperata sotto forma di latitudine e longitudine e memorizzata opportunamente in un DB locale. La frequenza di campionamento per quanto riguarda la posizione è circa ogni 3 minuti.
Il vantaggio dell'utilizzo dei servizi in background è che sono disaccoppiati dal ciclo di vita dell'applicazione, lo sviluppatore ha la libertà di tenere questi servizi attivi a prescindere dal ciclo di vita dell'*app*, dunque anche qualora essa venisse chiusa.
Questa funzionalità torna particolarmente utile in questo caso e semplifica di molto la vita al programmatore, cosa che purtroppo non è così immediata e semplice da replicare in ambiente iOS. Ma vedremo poi nei prossimi capitoli come questo problema è stato risolto.

2.2.2 Riconoscimento del movimento e tracciatura del percorso

Il riconoscimento del tipo di movimento dell'utente viene effettuato utilizzando la classe *ActivityRecognitionClient*, essa permette di riconoscere vari tipi di movimento dell'utente tra cui: camminare, andare in bici, in macchina e ritorna anche un certo grado di confidenza. Quando questa confidenza non è abbastanza, perché ci sono pochi dati a disposizione, allora viene ritornato *unkown* come tipologia acquisita.

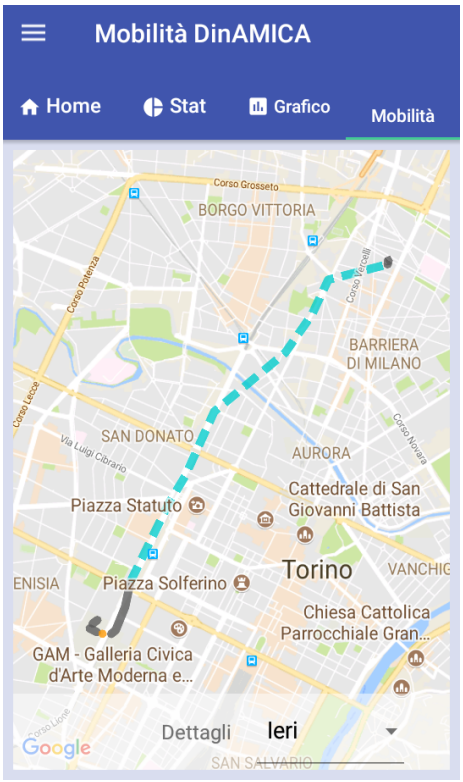


Figure 9 - Esempio di percorso di mobilità

Il tipo di movimento verrà associato ovviamente a uno spostamento fra due posizioni che l'utente ha compiuto. Le due posizioni consecutive, calcolate tramite l'api descritta nel paragrafo precedente, saranno utilizzate per tracciare un percorso sulla mappa.
Verrà dunque tracciata una linea di collegamento fra le due posizioni, e il colore della linea dipenderà dal tipo di attività rilevata.

Solitamente questo tipo di api, utilizzano l'approccio "*power saving*", ossia invece di intuire il tipo di movimento sfruttando sensori ad alto consumo come il GPS ne useranno degli altri

come ad esempio il giroscopio e l'accelerometro.
Vedremo che in ambiente iOS questo approccio è anche utilizzato ma con delle librerie diverse, e bisognerà calibrare anche li la frequenza di aggiornamento dei dati per modulare opportunamente il consumo di batteria e ottenere dati coerenti con bassi consumi.

2.2.3 Geofencing

La *GeofencingAPI* messa a disposizione in Android è la libreria che viene utilizzata per la realizzazione della funzione in questione.
Il geofencing di per se nasce come concetto di vicinanza o prossimità a un determinato luogo. Poi questo tassello di vicinanza viene scalato in base alle necessità e al grado di precisione che si vuole avere. Il *geofence* diventerà quindi un cosiddetto luogo di interesse, e dunque starà al programmatore interpretarlo e memorizzarlo opportunamente.

Nel nostro caso il concetto di *geofence* va letto come punto di partenza o di arrivo di un determinato tragitto di mobilità.
Ad esempio se l'utente decide di andare dal punto A al punto B, molto probabilmente per arrivare a destinazione seguirà i *path* più vari possibili, starà all'applicazione capire se durante questi micro-spostamenti i punti intermedi che verranno catturati saranno punti stazionari o punti di transito.
Qualora l'applicazione li valuterà come punti stazionari essi saranno memorizzati come geofence, con una relativa ora di ingresso e di uscita, e visualizzati sotto forma di circonferenze sulla mappa.

Fonte 9 - https://developer.android.com/training/location/geofencing#CreateAdd

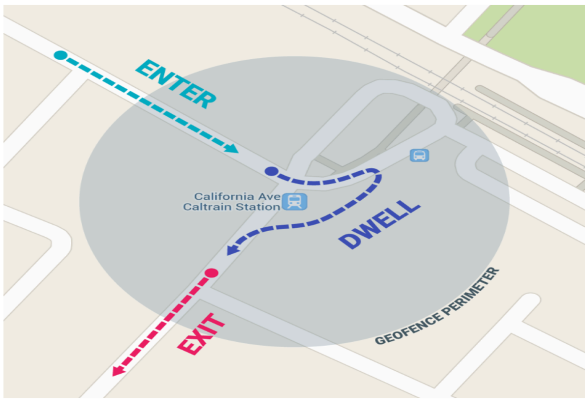


Figure 10 - Esempio di un geofence in Android

Nello specifico il geofence viene calcolato in Android seguendo un approccio molto semplice a livello concettuale. Ogni 3 minuti avevamo detto che la posizione veniva recuperata dal servizio in background, poi se per 5 volte consecutive la distanza tra la prima posizione e tutte le altre 4 a seguire è inferiore a 100 metri allora significa che probabilmente l'utente è li per fare una pausa, oppure perché quello è effettivamente un suo punto di destinazione finale; solo a questo punto l'applicazione riconosce quel luogo come *geofence* e ne rileva la posizione e ne associa le informazioni di contesto.

2.2.4 Funzionalità accessorie

Come specificato all'inizio del capitolo l'applicazione Android è in grado di fornire all'utente anche altre funzionalità, ognuna con uno scopo specifico.

Partiamo intanto dalla scherma principale, la *home screen* nella figura sottostante. Qui è possibile abilitare o disabilitare con uno *switch*, in basso a sinistra, il servizio di geo-localizzazione.

Questo rappresenta il comando globale dell'intera applicazione che permette all'utente di decidere spontaneamente quando abilitare o meno il monitoraggio automatico dei suoi spostamenti in ambito cittadino. Qualora decidesse di tenerlo attivo l'applicazione lavorerà in background, usando le strategie descritte nei paragrafi precedenti per ricavarsi le posizioni e i modi di viaggio; dunque non ci sarà bisogno di tenere l'applicazione aperta in *foreground*, ma l'utente continuerà ad utilizzare il suo cellulare esattamente come farebbe di solito.

Se invece l'utente volesse fornire manualmente informazioni riguardo il suo percorso urbano può sempre farlo attraverso il tasto in basso a destra con accanto indicato: "aggiunta di un nuovo viaggio".
In questo caso il servizio di monitoraggio generale viene disattivato perché sarà l'utente a fornire deliberatamente informazioni sul suo tipo di spostamento.
In pratica si aprirà una seconda schermata, e li sarà possibile scegliere da un elenco il possibile mezzo di trasporto con cui l'utente si sta spostando ed eventualmente aggiornarlo ogni qualvolta lo reputasse opportuno.
Può anche segnalare che il suo viaggio è finito e dunque cliccare sul bottone in basso a sinistra "Sono arrivato". Questa è una visione ad alto livello di quello che avviene, perché ad esempio ogni volta che l'utente clicca su uno dei possibili mezzi di trasporto indicati nell'elenco quello che in realtà avviene dietro le quinte è che il software fa tutta una serie di chiamate per recuperare la posizione reale dell'utente e inoltre attivare la geo-localizzazione soltanto in una finestra temporale minima per disabilitarla subito dopo al fine di ridurre al massimo il consumo della batteria.

Sempre nella *home screen* è possibile visualizzare il quantitativo di ore effettuate di monitoraggio dall'utente. Per evitare di essere falsate da errori locali o manomissioni da parte di un utente malevolo, questo valore è direttamente calcolato e scaricato dal server tramite l'uso di una chiamata REST usando il protocollo OAuth per l'autenticazione e l'autorizzazione.

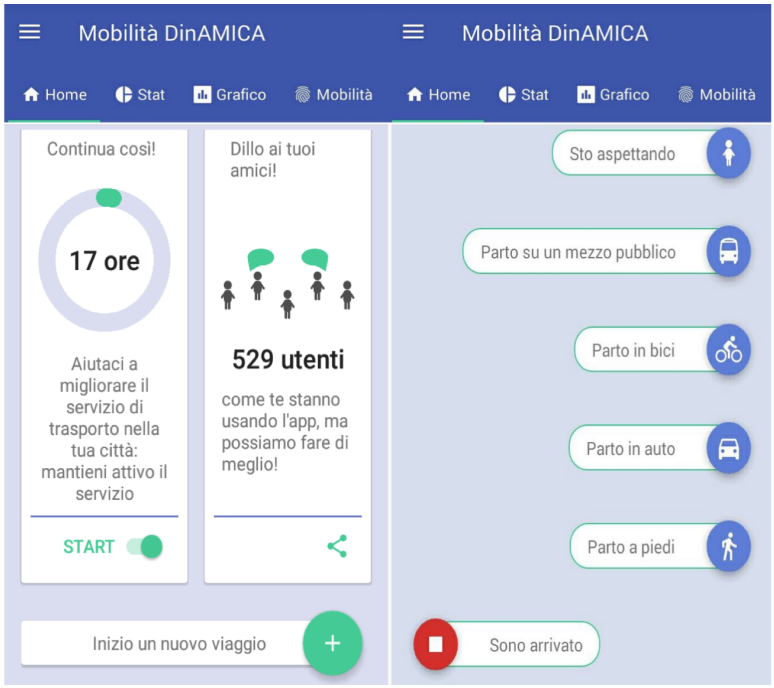


Figure 11 - Schermata home e aggiunta di un nuovo viaggio

Nella schermata successiva dell'applicazione l'utente può scegliere di visionare le statistiche dei suoi viaggi, in due tipologie grafiche differenti:

- la modalità sotto forma di *pie-chart*.
- la modalità sotto forma di *histogram*.

Nella prima schermata a sinistra, mostrata nella figura sottostante, è possibile selezionare un range di tempo in cui l'utente può analizzare dettagliatamente la distribuzione dei suoi spostamenti in ambito cittadino. È indicato anche quale tipo di mezzo di trasporto è stato usato maggiormente, e il numero totale di ore in movimento. Nella schermata a destra invece, una volta selezionato il tipo di mezzo di trasporto, è possibile ottenere una media di viaggio settimanale.

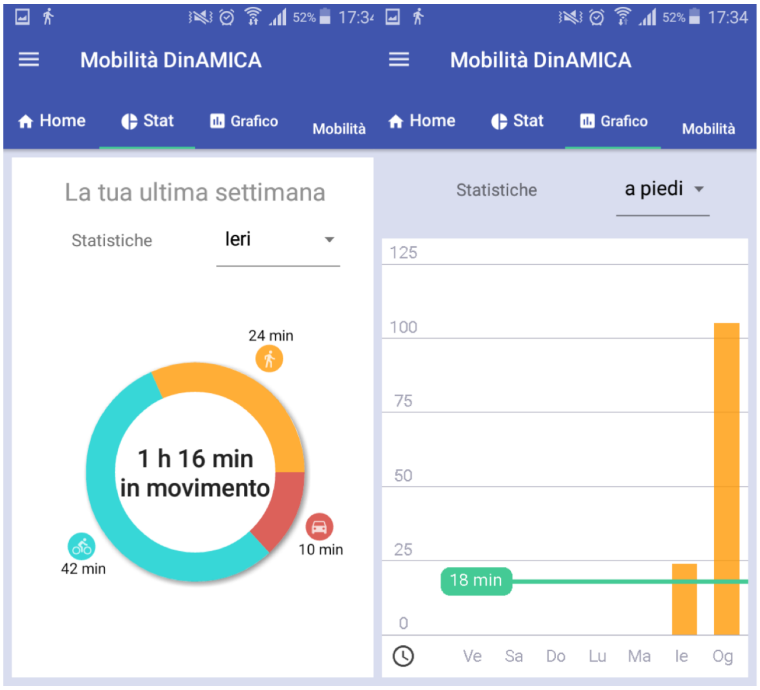


Figure 12 - Statistiche di viaggio

Anche in questo caso per evitare falsificazioni o manomissioni dei dati reali da eventuali utenti malevoli i dati non sono presi dal database locale del device, ma sono scaricati dal server sempre attraverso tecnologia REST e OAuth.

Un'altra funzionalità disponibile in ogni *activity* dell'applicazione, è quella del menu a tendina. Esso permette l'accesso alle informazioni inerenti al profilo, alla sezione commenti, e allo storico dei viaggi dell'utente.
Per raggiungerla bisogna che l'utente clicchi sul tasto in alto a sinistra, rappresentato da 3 linee azzurre parallele. Apparirà dunque quello che è chiamato in termine tecnico: *navigation drawer*, rappresentato nella figura sottostante con il primo *screen-shot* a sinistra.

La schermata inerente alla modifica del profilo non è qualcosa che l'utente è strettamente obbligato a compilare. Può farlo qualora ne ha voglia ma di sicuro sarà molto utile all'azienda che sta raccogliendo tutti i dati di mobilità. Più informazioni avrà l'azienda e meglio potrà filtrare le informazioni e classificare gli utilizzatori anche per range di età o per occupazione sociale etc. Inoltre l'utente aggiungendo informazioni come il nome o il sesso o l'età sarà più facilmente riconoscibile qualora volesse usufruire della sezione commenti.

Quest'ultima è sempre raggiungibile dal menù laterale, e praticamente raccoglie la lista degli eventuali commenti postati dalla community o meglio da tutti gli utilizzatori di questa applicazione. Questa schermata è molto utile qualora gli utenti volessero scambiarsi informazioni in tempo reale su possibili problematiche legate al traffico cittadino, oppure incidenti, o ritardi, o ancora deviazioni. A sua volta sarà anche possibile usare un filtro per

avere una visualizzazione più rapida in base alla categoria di appartenenza del commento. Tutta la lista recuperata sarà comunque rappresentata in ordine decrescente in modo cronologico dal più al meno recente.

L'ultimo *screen-shot* sulla destra mostra come l'utente ha anche la possibilità di visualizzare lo storico dei suoi viaggi, nello specifico quelli manuali che lui stesso ha aggiunto tramite la funzione: "aggiungi viaggio manuale" spiegata precedentemente in questo paragrafo.

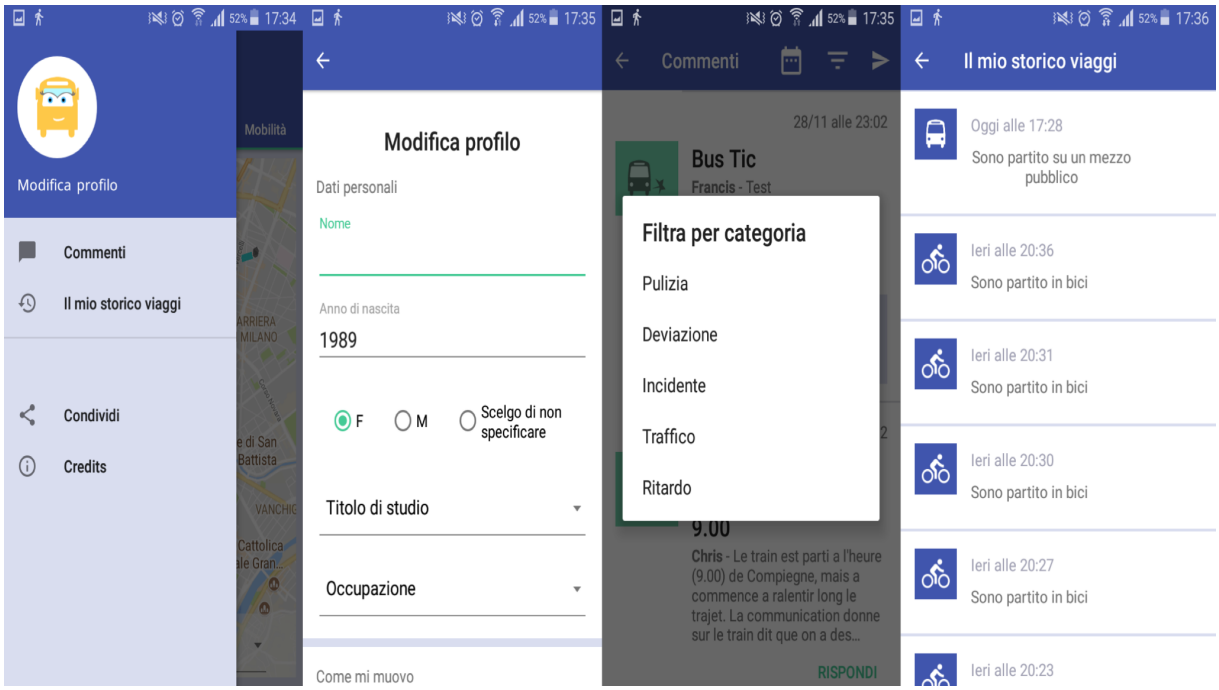


Figure 13 - Menu a tendina con schermate accessorie

Infine sempre dalla *tab-bar* orizzontale in alto, presente in tutte le *activity* dell'applicazione android, è possibile cliccare su "Mobilità". Questo aprirà la vista su un'altra schermata che servirà per mostrare all'utente gli spostamenti che ha fatto nell'ultimo lasso di tempo in giro per la città.

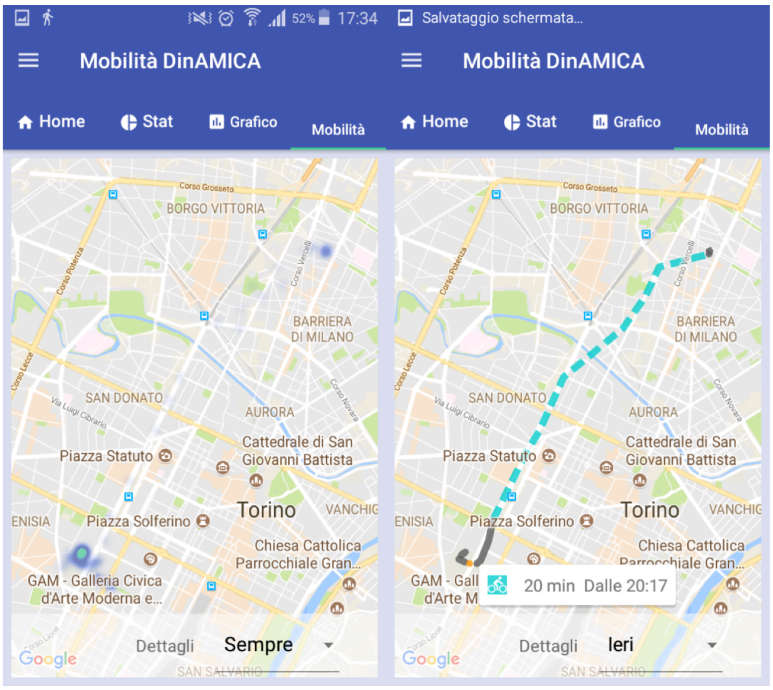


Figure 14 - Mappa di calore e spostamenti cittadini

L'applicazione in automatico sarà in grado di capire, usando la geo-localizzazione dell'utente, la sua posizione attuale. Inoltre per primo verrà visualizzata la mappa di calore ossia una mappa della città con una rappresentazione visuale dei posti in cui l'utente è stato più spesso. Sostanzialmente i punti più frequentati dall'utente verranno evidenziati con una macchia di calore più ampia, fitta, e omogenea, invece quelli meno frequentati saranno rappresentati sempre in sovraimpressione sulla mappa ma con un livello di aggregazione dei punti di calore più basso.

Con la *drop-down-list* in basso alla mappa sarà possibile cambiare il filtro di visualizzazione e selezionare un determinato giorno della settimana. A secondo del giorno della settimana scelto verrà visualizzata l'impronta di mobilità dell'utente nell'arco delle 24 ore. In base a come l'utente si è spostato vi saranno tratte disegnate sulla mappa con colori diversi, ad esempio la tratta in bici rappresentata nella figura soprastante, a destra, fa vedere che è di colore ciano. Per capire il tipo di movimento associato al colore della linea basterà un semplice tocco sul tragitto dopodiché apparirà in sovraimpressione una annotazione con la durata dello spostamento, la tipologia, e l'orario di inizio.

2.3 Applicazione in iOS

Essa è stata resa disponibile sullo store di Apple a partire da aprile 2018. Rappresenta il perno principale di questo lavoro di tesi, infatti le saranno dedicati interamente il resto dei capitoli.

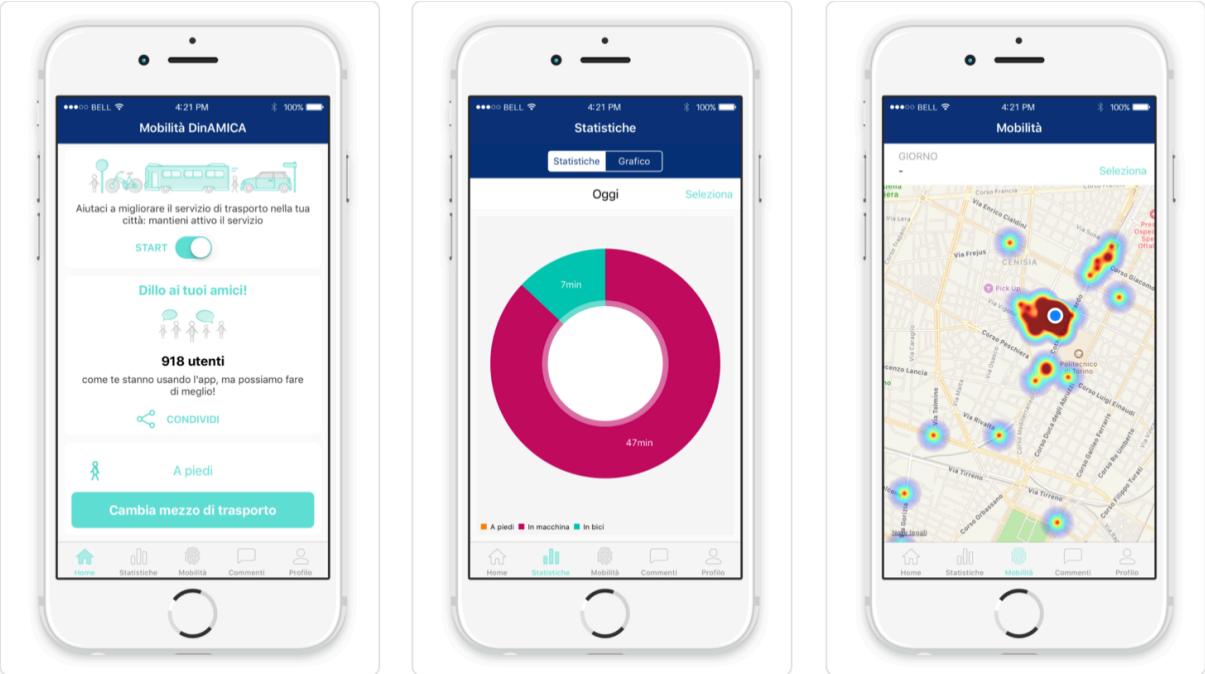


Figure 15 - Screenshots Mobilità Dinamica per iOS

A prima vista si nota subito che la grafica è stata completamente rinnovata, e anche per alcune funzionalità si può dire altrettanto. Si è cercato di rendere disponibili tutte le funzioni che erano presenti in Android, adattandole al meglio alle esigenze che il sistema operativo richiedeva. Inoltre l'approccio *user-experience* grazie alla nuova grafica è stato migliorato e reso più adatto allo *user* medio Apple.

Gli *screenshots* nella figura soprastante sono alcune delle schermate principali della versione iOS dell'applicazione, e sono reperibili dal seguente link ufficiale: linkAppMobilitàDinamica[1].

A seguire nel capitolo tre affronteremo l'idea di base da cui si è partiti, spiegando più nel dettaglio le librerie utilizzate e perché sono state prese determinate scelte. Nel capitolo quattro invece scenderemo ulteriormente nel dettaglio, descrivendo alcune delle implementazioni più significative e come alcune problematiche sono state superate.

[1] https://itunes.apple.com/it/app/mobilit%C3%A9-dynamique/id1364437932?mt=8

3. Capitolo 3 – Funzionalità e Librerie

3.1 L'idea di base

Come accennato già nei capitoli precedenti l'idea di base era completare l'infrastruttura del software già messo in piedi dall'azienda Tonic Minds.
Serviva ampliare la parte client inserendo come bacino di raccolta dati anche quello degli utenti Apple.
Durante lo sviluppo di questo progetto di tesi possiamo dire che la parte server è stata leggermente migliorata a livello di prestazioni, pur comunque lasciando la parte ad alto livello invariata, e invece la parte client in Android è stata aggiornata di altre due versioni successive.
Nella fase finale di sviluppo dell'applicazione iOS è stato dunque necessario allinearsi agli ultimi miglioramenti inseriti in Android; un esempio è stata l'aggiunta della funzione di geo-localizzazione anche sui commenti che gli utenti inserivano.
In questo modo lato server è divenuto possibile filtrare i commenti anche in base alla zona geografica in cui l'utente si trovava.

Nel capitolo due abbiamo visto ad alto livello le funzioni che il software su piattaforma Android era in grado di fare.
Lo scopo principale da raggiungere in questo lavoro di tesi è stato dunque replicare tutte le funzionalità dell'app disponibile su Google Store e traslarle in una versione iOS. Addirittura qualora fosse stato possibile migliorarle.
Prima di tutto è stato necessario studiare l'ambiente iOS in profondità, capirne le potenzialità ma anche le sue limitazioni.
Si è passati dunque allo studio delle librerie più adatte da utilizzare per comporre i primi tasselli del mosaico.
E poi infine si è iniziato a comporre e collegare le varie parti, utilizzando diversi *ViewController*.

Una volta aperta l'applicazione vengono chieste allo *user* determinate autorizzazioni inerenti la privacy, i sensori di movimento e i sensori del gps.
Nella schermata iniziale l'utente può decidere, attraverso uno *switch* globale, quando permettere all'applicazione di utilizzare i suoi dati di mobilità e quando non può farlo.
Una volta che sono state date tutte le autorizzazioni necessarie, il codice deve essere in grado di funzionare sia in modalità *foreground* ma soprattutto in modalità *background*.
Per modalità *foreground* si intende quella in cui l'applicazione è aperta e l'utente la sta utilizzando in modo attivo, ossia interagendo con la *user-interface*.
Per modalità *background* si intende quella in cui l'applicazione è stata già aperta precedentemente ma non viene chiusa definitivamente, bensì resta aperta e continua a funzionare mentre l'utente fa altre operazioni con il cellulare o semplicemente spegne lo schermo per metterlo in modalità *stand-by*.
Uno degli obiettivi principali è catturare la posizione dell'utente attraverso l'uso del gps interno al device, e di interpretare il movimento che l'utente sta compiendo per arrivare da un punto A a un punto B, dunque di interpretare tutti i suoi spostamenti in ambito cittadino.
Questo codice che interpreta e colleziona informazioni di mobilità urbana deve essere in grado di essere eseguito più tempo possibile durante la giornata, senza intaccare il classico utilizzo che l'utente medio farebbe del suo dispositivo in un giorno normale.
Di base la logica sembra molto semplice, la difficoltà sta nel fatto che questo approccio non è di facile realizzazione in ambiente iOS.
Il motivo è che iOS ha molte restrizioni dal punto di vista di quello che puoi fare o non puoi fare in *background*, in pratica puoi soltanto tenere le applicazioni in questo stato solo e unicamente se si rispettano determinate casistiche contemplate da loro stessi. Le casistiche

sono limitate, come vedremo.
In Android invece per realizzare un comportamento del genere, dal punto di vista dell'architettura software, è sufficiente far partire un *service* in background e li dentro è possibile eseguire qualsiasi tipo di codice. Dunque non essendoci restrizioni viene semplificato di molto il lavoro del programmatore però dal punto di vista della durata della batteria a volte non è proprio il massimo.

Uno dei motivi per cui Apple ha limitato le tipologie di *background-mode* è infatti l'allungamento della durata della batteria, oltre che il miglioramento della *user-experience* (12).

Nei prossimi paragrafi verranno elencate le modalità in *background* che Apple contempla nei suoi dispositivi. Verranno spiegate le difficoltà incontrate lungo il corso dello sviluppo in iOS e come si è riusciti a superarle, o eventualmente aggirarle.
Infine verranno introdotte le librerie fondamentali che sono state utilizzate nella realizzazione del prodotto finale, e i motivi che stanno dietro queste scelte.

3.2 La background mode di Apple

Nel sistema operativo iOS se l'utente non sta usando l'applicazione in modo attivo, esso decide di spostarsi in uno stato detto di *background*.
A volte questo stato può essere rappresentativo di una breve transizione prima di cambiare ulteriormente e passare a un altro stato detto di sospensione.
Quest'ultima è molto importante infatti è uno dei modi in cui la vita della batteria viene estesa dando allo stesso tempo la possibilità al sistema di spostare la sua attenzione e dunque le sue risorse sulle applicazioni dette in *foreground*.
Il passaggio allo stato di sospensione è molto lineare e semplice ma ci sono delle casistiche in cui questo non va bene, anzi vogliamo esplicitamente che l'app continui a funzione in *background*.
I motivi possono essere di vario tipo, ad esempio applicazioni per le escursioni in montagna che al loro interno hanno delle rappresentazioni cartografiche relative agli spostamenti dell'utente, o il caso semplice in cui c'è la necessità di scaricare un documento ma l'utente vuole fare altro durante l'attesa, oppure quando si vuole ascoltare della musica con il cellulare in tasca mentre l'iphone è messo in modalità *stand-by*.
Apple fornisce delle linee guida che i programmatori possono adottare in modo che qualora siano costretti ad utilizzare la funzionalità *background* possono comunque limitare il consumo spropositato della batteria.
Le strategie consigliate da iOS si possono suddividere in tre principali macro punti:

- Applicazioni che sono in *foreground* e devono finire un piccolo task dunque richiedono un piccolo tempo aggiuntivo al sistema nel momento in cui l'app passa allo stato di *background*.
- Applicazioni che richiedono di funzionare in continuazione in *background* per determinati compiti specifici. In quel caso è possibile dichiarare il loro supporto in fase di sviluppo, precisamente nella sezione *Background-modes* della scheda *Capabilities* del file di tipo *.xcodeproj* dentro l'I.D.E. di Xcode.
- Applicazioni che iniziano il *download* quando l'app è attiva e poi però vogliono delegare la continuazione di esso al sistema, in questo modo l'utente qualora volesse può dedicarsi ad altro.

È sempre sconsigliato da Apple l'utilizzo della modalità in *background* almeno che non è strettamente necessario perché ricade in una delle categorie descritte sopra, oppure se uno degli obiettivi è migliorare la *user experience*. Questo perché in generale quando si va in

background è perché l'utente sta usando un'altra applicazione oppure perché ha messo in *stand-by* il telefono, quindi nella maggior parte dei casi l'utente pensa che quella applicazione non richiede più l'attenzione di prima o perché sostanzialmente non sta più facendo un lavoro significativo. Quindi spesso è importante distinguere la reale necessità di applicare questa modalità oppure no perché altrimenti se non lo si fa con consapevolezza si rischia scaricare la batteria più velocemente. (13)

3.2.1 Eseguire task in background con un tempo limitato

Ipotizziamo di dover eseguire un processo dentro l'applicazione e poi che improvvisamente venga premuto il tasto *home*. É comune ritrovarsi nel caso in cui l'app debba ultimare quello che il processo stava eseguendo in precedenza e dunque può darsi che abbia bisogno di un po' di tempo aggiuntivo.
Nel sistema operativo iOS è contemplato un modo per effettuare una richiesta di tempo extra al fine di dare la possibilità al *task* di portare a termine il suo compito una volta che l'app è andata in *background*. Usualmente vengono dati circa centottanta secondi, ma la documentazione ufficiale purtroppo non da un valore esatto a riguardo, ma soltanto indicativo.

In realtà non è neanche una background mode al 100%, tecnicamente parlando, perché qualora lo fosse andrebbe dichiarata in fase di progettazione dentro la scheda Capabilities di Xcode.

Nella documentazione Apple è scritto sostanzialmente che una volta che si esce dalla modalità *foreground*, e l'app era nel bel mezzo di un *task*, essa può chiamare una determinata funzione per richiedere un determinato piccolo *extra-time*. Per farlo serve chiamare una delle seguenti funzioni:

- **beginBackgroundTaskWithName:expirationHandler:**
- **beginBackgroundTaskWithExpirationHandler:**

Una volta completati l'app deve chiamare il metodo **endBackgroundTask:** in modo da avvisare il sistema che il task è finito e l'app può essere sospesa. Qualora il programmatore dimentichi di chiamare **endBackgroundTask:** automaticamente dopo un certo periodo di tempo in background, iOS chiamerà la *closure*[2] definita quando abbiamo chiamato la funzione **beginBackgroundTask(expirationHandler:)** per dare la possibilità di stoppare il codice in esecuzione. Apple raccomanda di assegnare il *task* a un *background-task* prima che esso inizi. (14)

3.2.2 Scaricare contenuti in background

Quando vengono scaricati documenti, o in generale dei file, dentro l'applicazione la classe che bisognerebbe usare è la URLSession. Essa permette di iniziare il download ma allo stesso tempo permette al sistema di tracciare il processo che lo gestisce, e questo torna molto utile nel caso l'applicazione venisse sospesa o terminata.
Qualora fossimo nell'ultima situazione, ma il download è ancora in corso il sistema continua a scaricare i dati dalla rete per poi rilanciare l'app in modo opportuno nel momento in cui il trasferimento è completato. (15)

Concettualmente è molto semplice, ma a livello pratico bisogna usare diverse strutture e chiamarle nei momenti opportuni.

---

[2] La *closure* in Swift sono un blocco di programma che viene chiamato in determinate condizioni. E per fare una analogia con altri linguaggi di programmazione possiamo dire che sono simili ai *block* in C o alle *lambda-functions* in C#.

La *URLSession* va configurata con un identificatore univoco in tutta l'applicazione. Bisogna settare la proprietà *sessionSendsLaunchEvents* a *true* in modo che una volta che il download finisce il sistema può risvegliare l'applicazione. Per i download che non richiedono particolari urgenza si può settare la proprietà *isDiscrectionary* in modo che sarà il sistema a decidere quando è meglio continuare il download, ad esempio quando agganciato a una rete wifi.
Le proprietà menzionate prima sono all'interno dell'oggetto di tipo *URLSessionConfiguration* che verrà passato come parametro all'atto della creazione dell'oggetto *URLSession*.
Infine bisogna fornire dentro il costruttore della *URLSession* un delegate, ossia quella porzione di codice che si occuperà di ricevere gli eventi dal trasferimento in background.
Dall'oggetto *urlSession* è possibile attraverso il metodo **downloadTask(with:)** creare il task di download, passandogli come parametro direttamente un URL o una URLRequest. Una volta fatto questo è possibile richiamare il metodo **resume()** per far iniziare il *background-task*. Un esempio è mostrato nella figura sottostante.

```
private lazy var urlSession: URLSession = {
        let config = URLSessionConfiguration.background(withIdentifier: "UniqueSession")
        config.isDiscretionary = true
        config.sessionSendsLaunchEvents = true
        return URLSession(configuration: config, delegate: self, delegateQueue: nil)
    }()
    let backgroundTask = urlSession.downloadTask(with: url)

    backgroundTask.resume()
```

Listing 1 - Esempio esecuzione background task

Fatto questo bisogna poi gestire i tre casi che possono verificarsi durante il download:

- La sospensione dell'app fatta dal sistema operativo iOS.
- La terminazione dell'app.
- Il completamento del download e/o gli eventuali errori.

Nel primo caso il sistema operativo per liberare risorse decide di sospendere l'applicazione. Quello che succede è che quando il download finisce il sistema chiama il metodo **application(_:handleEventsForBackgroundURLSession:completionHandler:)** contenuto dentro il *set* di metodi *UIApplicationDelegate*. Questo riceve come terzo parametro il completionHandler ossia la porzione di codice che verrà eseguita alla fine di tutta questa sequenza di chiamate, e poi riceve come secondo parametro il *sessionIdentifier* che nel nostro caso corrisponderebbe a quello creato nel Listing 1.
Successivamente verrà chiamato il metodo **urlSessionDidFinishEvents** che fa parte del protocollo[3] *URLSessionDelegate* che al suo interno recupererà il *completionHandler* e lo eseguirà.
Il secondo caso è causato dall'utente che he terminato l'applicazione manualmente, quindi questa è la differenza principale rispetto al primo caso. Mentre per quanto riguarda la gestione del risveglio viene trattata allo stesso modo, ossia: ricreazione delle sessione tramite l'*identifier*, e poi la serie di configurazioni e chiamate fino all'ultimo step che corrisponde all'esecuzione del *completionHandler*.
Nel terzo caso, quando il download viene completato è possibile gestire l'evento implementando il metodo **urlSession(_:downloadTask:didFinishDownloadingTo:)**, in base al valore di *downloadTask* è possibile capire se c'è stato ad esempio un errore 404 oppure se il download è andato a buon fine. Se il download è andato a buon fine poi si può procedere per esempio con lo spostamento di quel file in un'altra *directory*. (16)

[3] Il protocollo è un concetto importantissimo in Swift. Esso definisce un set di metodi, proprietà e altri requisiti che sono adatti per eseguire un determinato task o funzionalità. Esso può essere adottato da una *class*, una *struct*, o una *enumeration*.

3.2.3 Implementare task lunghi in background

Per *tasks* che richiedono lunghi tempi di esecuzione bisogna prima di tutto fare una richiesta specifica per quanto riguarda i permessi. I permessi servono a garantire il rispetto della privacy dell'utente e a renderlo consapevole di quello che l'app potrebbe fare in background. Se il permesso non è stato accettato dall'utente allora quel determinato servizio in background dell'app non potrà essere reso disponibile.
In iOS solo determinate applicazioni sono autorizzate ad eseguire lunghi compiti in *background*:

- Applicazioni che ricevono *update* regolari da accessori esterni.
- Applicazioni che informano tutto il tempo l'utente riguardo la posizione, un esempio sono appunto le app di navigazione.
- Applicazioni che registrano l'audio.

- Applicazioni che supportano il protocollo internet *Voip*.
- Applicazioni che hanno bisogno di scaricare e processare in continuazione nuovi contenuti regolarmente.
- Applicazioni che riproducono audio, ad esempio un *music player*.

In questi casi il tipo di servizio che viene garantito in background deve essere dichiarato in fase di progettazione anche dentro l'apposita scheda *Capabilities* delle impostazioni del progetto Xcode. In questo modo il sistema saprà che quella applicazione ricade in una categoria particolare e dunque non verrà sospesa dal sistema.

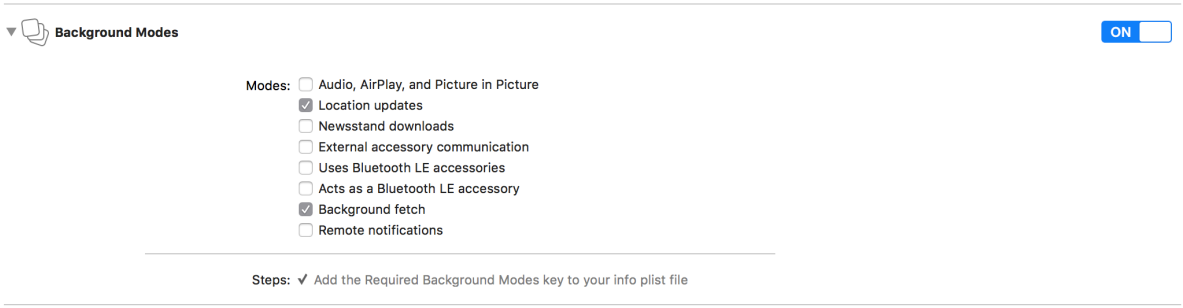


Figure 16 - Sezione background-modes dentro la scheda Capabilities del progetto Xcode

Aggiungendo i *check,* sulle capacità in background che ci interessano, verrà creata automaticamente una chiave chiamata *UIBackgroundModes* dentro il file Info.plist[4].
In questo file è possibile indicare varie impostazioni e autorizzazioni del progetto, ma nel nostro caso bisogna andare a fornire tramite una *check-box* il valore per la chiave *UIBackgroundModes*. I valori possibili selezionabili rientrano con corrispondenza 1:1 nell'elenco puntato precedente, ossia ci sarà un valore per ogni *background-mode*. Ad esempio nel caso specifico di un'applicazione che comunica con un accessorio sarà "*bluetooth-central*", o nel caso della localizzazione sarà "*location*", e così via. (13)
Ognuna delle modalità farà sapere al sistema che quella determinata applicazione potrebbe essere svegliata o rilanciata in certi momenti per rispondere a degli eventi rilevanti.
Prendiamo ad esempio il classico caso di una applicazione audio che inizia a riprodurre un

[4] L'estensione .Plist indica una property list all'interno del progetto di Xcode, essa è un *dictionary* rappresentato da coppie di chiavi e valori. Nonostante a basso livello sono scritte in XML, l'IDE di Xcode dà una visualizzazione più intuitiva ed editabile che è possibile modificare senza conoscere per forza l'XML.

brano e poi l'utente preme il tasto home. Se la modalità *background-audio* è stata impostata in modo corretto allora il sistema saprà che dovrà continuare a riprodurre l'audio che stava eseguendo prima. In caso contrario l'audio che era in stato di riproduzione verrà interrotto.

È molto importante capire in quali categorie l'applicazione che si sta sviluppando ricade, perché qualora venga scelta la modalità sbagliata o ancor peggio qualora si cercasse di imbrogliare dichiarando una modalità che poi di fatto non corrisponde al compito effettuato dall'applicazione, il rischio è che una volta completata la progettazione e inoltrata la richiesta di pubblicazione essa potrebbe essere rifiutata dall'Apple Store. (12)

Le modalità disponibili di background per task particolarmente lunghi sono diverse, e ognuna avrebbe bisogno di essere approfondita separatamente, ciononostante nell'ambito di questo progetto di tesi è stata usata unicamente quella relativa la localizzazione. Per quest'ultima ragione nel prossimo paragrafo approfondiremo la teoria che sta dietro il *tracking* della posizione dell'utente, tralasciando le altre modalità di background.

3.2.3.1 Monitoraggio della posizione dell'utente

Esistono diversi modi di monitorare la posizione dell'utente e alcuni di questi non richiedono nemmeno che l'applicazione sia eseguita incessantemente in *background*. Le principali possibilità di *tracking* sono:

- Solo quando l'applicazione è attiva e la sua *user-interface* altrettanto.
- Per cambiamenti significativi.
- In *background*.

Se l'applicazione non ha bisogno di avere una precisione elevata della posizione allora la modalità suggerita maggiormente da Apple è la prima, ovvero *significant-change*.
Qualora l'applicazione venisse terminata e fosse successivamente rilevato un cambio di posizione significativo il sistema rilancerà l'applicazione automaticamente. Ovviamente lato programmazione questo caso andrà gestito opportunamente. Questa modalità è disponibile solo in sistemi con una versione di iOS maggiore o uguale a quattro.
Se invece stessimo considerando la realizzazione di un'applicazione di navigazione come ad esempio Google Maps o similari, allora in quel caso potremmo optare per la modalità numero uno ossia la *foreground-only* perché una volta che l'app è stata sospesa o chiusa non abbiamo interesse a continuare il monitoraggio della posizione. (13)

L'ultima invece è quella che effettivamente serviva nel caso dell'applicazione in oggetto di questa tesi, Mobilità Dinamica. Come spiegato nel paragrafo precedente bisogna prima di tutto impostare i parametri giusti nel file Info.plist, in modo che iOS permetterà l'utilizzo del sensore gps anche in modalità di *background*.

Successivamente bisogna trovare un approccio mirato e intelligente affinché l'uso della batteria venga minimizzato ottenendo però allo stesso tempo dati coerenti e abbastanza precisi per il target che il software vuole raggiungere.
Il dettaglio implementativo di questa soluzione verrà descritto in modo accurato nel capitolo quattro.

3.3 CoreLocation e MapKit

CoreLocation e MapKit sono due librerie molto importanti che sono state utilizzate in questo progetto. Possiamo affermare che sostanzialmente sono due facce della stessa medaglia perché CoreLocation si occupa di tutto ciò che riguarda la cattura reale della posizione dell'utente mentre MapKit si occupa della parte UI ossia della visualizzazione della posizione sulla interfaccia grafica.

L'oggetto principale di CoreLocation è CLLocation e al suo interno abbiamo diverse proprietà come ad esempio: *coordinate*, *horizontal-accuracy*, *timestamp*, *speed*, *altitude* etc.
La proprietà *coordinate* al suo interno è rappresentata in due dimensioni ossia latitudine, longitudine.
La *accuracy* è un parametro molto importante che bisogna settare propriamente in base alle proprie necessità. Più si aumenta la precisione della *accuracy* più batteria verrà consumata.
Come accuracy esiste sia la *horizontal* che la *vertical* ed entrambe sono espresse in metri.
Qualora venisse restituito un valore negativo significa che quel valore è invalido.
I valori invece che è possibile settare in fase di configurazione sono i seguenti:

- KCLLocationAccuracyBestForNavigation.
- KCLLocationAccuracyBest.
- KCLLocationAccuracyNearestTenMeters.
- KCLLocationAccuracyHundredMeters.
- KCLLocationAccuracyKilometer.
- KCLLocationAccuracyThreeKilometer.

Sono in ordine di precisione, dalla più precisa alla meno precisa. In base alla scelta che si prende il drenaggio della batteria cambia perché in base all'accuracy richiesta verrà utilizzato internamente uno dei seguenti approcci:

- La triangolazione delle torri dei cellulari.
- Nodi wifi.
- Sensore gps.

La prima è quella che essendo meno precisa è non molto accurata mentre l'ultima è quella più precisa che si può ottenere. La seconda invece rappresenta una buona via di mezzo perché il cellulare è in grado di capire in base ai nodi wifi che trova nel suo vicinato di capire più o meno dove si trova.
In genere il cellulare in base alla accuracy che viene settata cerca sempre di fare del suo meglio per avvicinarsi il più possibile a quella richiesta. In ogni caso di queste tre modalità interne non abbiamo direttamente la possibilità di sceglierne una in particolare, ma sarà lui stesso a decidere quella che per lui rappresenta il miglior compromesso.
Un altro elemento importante è il timestamp che ci servirà per capire quando effettivamente quella determinata rilevazione della posizione è stata effettuata.
Un oggetto CLLocation è preso sempre attraverso il CLLocationManager.
Una volta creato il CLLocationManager bisogna settare sempre il delegato in modo da far sapere alla libreria dove mandare gli updates. Successivamente bisogna configurare l'istanza di CLLocationManager in modo da specificargli quale tipo di location desideriamo. Fatto questo si può fare lo *start* in modo da iniziare a ricevere tutte le posizioni di cui abbiamo bisogno.

Esistono diverse tipologie di *monitoring* che è possibile fare attraverso CoreLocation:

- Basati sull'accuratezza.
- Basati solo quando c'è un cambiamento significativo.
- Basati sulla *region.*

Il primo essendo basato sull'accuratezza significa che riceveremo tanti updates in base alla distanza settata come filtro, per esempio se mettessimo KCLLocationAccuracyBestForNavigation riceveremmo praticamente in continuazione nuovi valori. Se invece impostassimo KCLLocationAccuracyKilometer riceveremmo aggiornamenti più o meno a ogni chilometro di spostamento dall'ultima posizione rilevata, e così via. (17)

Il secondo fa riferimento a un cambiamento significativo e per "significativo" non è ben spiegato nella documentazione Apple cosa intende, diciamo che dà solo una indicazione di massima che è un filtro di circa 500 metri. In realtà però in base ai test effettuati e agli approfondimenti possiamo affermare che l'istanza del CLLocationManager viene risvegliata ogni volta che il cellulare si connette a una nuova antenna mobile dunque a una cella mobile differente dalla precedente. Ovviamente questa non è la situazione in cui si ottiene una location molto accurata.

Il terzo parla di approccio sotto forma di *region* ossia quando si decide di fissare un'area circolare con un determinato raggio e allora a seconda del raggio prefissato scatta l'aggiornamento della posizione.
Per far funzionare la libreria CoreLocation una delle prime cose necessarie da fare è verificare le autorizzazioni necessarie, per farlo si usa il metodo **authorizazionStatus()** che restituisce un valore del tipo **CLAuthorizationStatus** esso a sua volta ci tornerà uno dei seguenti possibili valori: authorized, denied, oppure restricted. Restricted è un caso speciale che indica che lo stato è come se fosse denied ma l'utente non può fare nulla per cambiare questo particolare stato. Questo potrebbe accadere quando magari lo *user* possiede un cellulare aziendale e lui non è stato autorizzato dalla compagnia ad usare i servizi di *location* all'interno del dispositivo.
Richiedere all'utente l'autorizzazione per monitorare la sua posizione è uno degli *step* principali. Si possono fare due tipi di richieste di autorizzazione: la inUseAuthorization attraverso il metodo **requestWhenInUseAuthorization()** oppure la alwaysAuthorization attraverso il metodo **requestAlwaysAuthorization()**. Entrambe le chiamate sono eseguite in modo asincrono dal sistema.
La prima significa che siamo abilitati a ricevere update solo quando l'applicazione è in *foreground* ossia quando l'utente usa l'applicazione attivamente e la user interface è visibile; la seconda invece è quella che permette di ottenere la posizione dell'utente anche quando l'applicazione è in background. (17)

Ovviamente nel nostro specifico caso per l'applicazione Mobilità Dinamica è stato necessario richiedere la alwaysAuthorization. Essa è quella più indicata per la realizzazione dei nostri obiettivi, ma allo stesso tempo è anche la più invasiva. Infatti l'utente potrebbe sentirsi a disagio a dare questa autorizzazione, perché in un certo senso è come se lo stessimo spiando o comunque stessimo invadendo la sua privacy. Per quest'ultimo motivo è molto importante specificare in un messaggio popup o attraverso una *alert-box* il motivo di questa richiesta ricordandogli che comunque è sempre lui ad avere il controllo della sua *privacy* perché può sempre disabilitare o riabilitare il monitoraggio ogni volta che lo desidera attraverso lo switch principale.
Dal punto di vista implementativo Apple facilita leggermente le cose, evitando che lo sviluppatore dimentichi di spiegare il motivo per cui sta richiedendo quella determinata autorizazzione. Il modo in cui lo fa è che rende obbligatorio inserire una stringa dentro il file Info.plist, esattamente nel valore della variabile NSLocationAlwaysUsageDescription.

Per ricevere aggiornamenti è sempre possibile fare una sorta di *polling*, ovvero una interrogazione ciclica, oppure usare l'approccio dell'accuracy impostando un certo filtro e dunque come spiegato precedentemente ricevere nuove posizioni solo e unicamente quando si rileva il superamento di una certa distanza di soglia.
A livello implementativo infatti per attuare il secondo approccio si può settare la variabile desiredAccuracy di tipo CLLocationAccuracy o anche la variabile distanceFilter di tipo CLocationDistance. (17)

Una volta deciso quale tipo di strategia adottare basta chiamare la funzione **startUpdatingLocation()** per iniziare a ricevere nuove rilevazioni della propria posizione.

Fatto questo è importantissimo però ricordarsi anche di stoppare il servizio con l'apposita funzione **stopUpdatingLocation()**, tutte le volte in cui in base alla logica del software non è più richiesto ricevere nuovi aggiornamenti. Qualora questo si dimenticasse rischieremmo di consumare inutilmente risorse energetiche dell'iPhone.

Una volta ricevuta la specifica rilevazione bisogna processarla. La nuova posizione viene dunque notificata attraverso il delegato CLLocationManager attraverso una funzione che ha la seguente *signature*:

func locationManager(CLLocationManager, didUpdateLocations: [CLLocation])

dentro didUpdateLocations abbiamo un array di locations ognuna al suo interno avrà il proprio timestamp quindi è poi possibile ricostruire in ordine temporale il set di location che ci serve. Inoltre se si implementa questa funzione per processare le varie *locations* è importante anche implementare la corrispettiva funzione che si occupa di gestire le varie situazioni in caso di un eventuale errore, e la *signature* di questa funzione è la seguente:

func locationManager(CLLocationManager, didFailWithError: NSError)

perché in base al valore di didFailWithError è possibile capire se ad esempio la location è sconosciuta temporaneamente, oppure se l'utente ha rifiutato di ricevere gli update e così via.

Come accennato all'inizio di questo paragrafo è anche possibile iniziare un monitoring seguendo l'approccio della *significant location* e per farlo bisognerà utilizzare le funzioni con questa *signature*:

func startMonitoringForSignificantLocationsChanges()
func stopMonitoringForSignificantLocationChanges()

la cosa bella di questo approccio è che praticamente consumerà pochissima batteria, e addirittura si sveglierà e inizierà a ricevere eventi sulle nuove posizioni sia che l'applicazione sarà in background sia che l'applicazione sarà stata chiusa precedentemente. Qualora fosse stata chiusa precedentemente l'applicazione verrà addirittura rilanciata in background. (17)

Il lato negativo di questo approccio però è che l'accuratezza della posizione è molto bassa rispetto a quello che si voleva ottenere. Per questo motivo questa modalità è stata testata nel corso di questa tesi di laurea, ma poi alla fine è stata scartata.

L'ultima modalità di monitoraggio possibile era quella legata alla *region*, per attivarla bisogna utilizzare le funzioni con questa signature:

func startMonitoringForRegion(CLRegion)
func stopMonitoringForRegion(CLRegion)

come è possibile vedere c'è un parametro che rappresenta una regione circolare creata appositamente con un determinato raggio, una volta passata alla funzione come parametro e attivata la startMonitoringForRegion automaticamente una determinata area verrà monitorata. Questa funzione nello specifico è stata particolarmente utile per la realizzazione dei geofence, ossia di quel concetto introdotto già nel paragrafo 2.2.3, solo che in ambito iOS la logica di realizzazione adottata è stata leggermente diversa da quella usata in Android. Il dettaglio della soluzione sviluppata lo vedremo nel capitolo quattro.

Per quanto inerente la visualizzazione delle posizioni e dei geofence su interfaccia grafica bisognava scegliere fra due principali opzioni, utilizzare la libreria che permetteva di interagire con le mappe di Google Maps oppure utilizzare quella di *default* fornita da Apple attraverso MapKit.

Le mappe fornite da Google hanno raggiunto un livello di dettaglio elevatissimo, infatti come afferma il cartografo e ingegnere informatico Justin O'Beirne i motivi principali sono due:

1. Google ha iniziato a raccogliere dati 7 anni prima di Apple.
2. Sempre Google è riuscita a sviluppare un algoritmo che interpola i dati delle immagini satellitari con quelli delle foto scattate per il servizio Street View. In questo modo riesce a ricostruire i dettagli di diverse strutture anche in aree suburbane.

Nonostante tutto questo però Apple con la sua libreria MapKit, anche se non ha ancora raggiunto il livello di precisione di Google, è riuscita a migliorare tantissimo negli ultimi anni. (18)

Sono state effettuate delle prove e si è riscontrato che comunque per lo scopo ultimo di questa applicazione il livello di qualità fornita dalle Apple Maps andava più che bene, dunque alla fine si è deciso di utilizzare queste anche per avere una maggiore compatibilità con il device.

MKMapView fa parte della libreria MapKit e rappresenta la *view* principale su cui è possibile stampare le posizioni. È possibile richiamarla a livello di codice allocandola come oggetto, oppure semplicemente facendo il *drag & drop* sulla *storyboard*.
MKMapView è in grado di mostrare un array di elementi di tipo MKAnnotation. Esso MKAnnotation in realtà è un protocollo composto da tre sotto elementi:

1. Un campo di tipo CLCoordinate2D che rappresenta le coordinate del punto sulla mappa, infatti hanno dei sotto campi *latitude* e *longitude*.
2. Due campi di tipo String che rappresenterebbero rispettivamente il titolo, e il sottotitolo dell'annotazione.

Per quanto riguarda l'array di MKAnnotation bisogna agire su di esso sempre con gli appositi metodi **addAnnotation(MKAnnotation)** e **removeAnnotation(MKAnnotation)** perché l'array è accessibile direttamente solo in modalità lettura.
Potrebbero esserci parecchie annotazioni da visualizzare nello stesso istante sulla *user-interface* e facendo lo *scroll*, lo *zoom* o il *de-zoom* è abbastanza normale che le richieste crescano in modo esponenziale. Questo potrebbe comportare un dispendio incredibile di memoria da parte del sistema, per questo motivo Apple per evitare di sovraccaricare il dispositivo e rendere la navigazione della mappa veloce e fluida ha sviluppato all'interno di MKMapView un sistema di riciclo delle *view* delle annotazioni.
Tramite il parametro di tipo MKMapType dentro MKMapView è possibile settare il tipo di mappa che si vuole visualizzare: ibrida, dal satellite, o standard ossia quella in due dimensioni.
Tramite il parametro showUserLocation di tipo Bool è possibile decidere se far visualizzare o meno la posizione dell'utente sulla mappa. (17)

Le annotazioni con all'interno le informazioni sulle coordinate sono proprio gli elementi che sono stati usati internamente nel progetto per raffigurare i *geofence* sulla mappa.

Un altro componente da attenzionare è MKOverlay, essa è una classe derivata da MKAnnotation e viene utilizzata tipicamente per aggiungere delle *overlay view* sulla mappa, ovvero delle figure in sovrimpressione sulla MKMapView.
Nello specifico questa particolare interfaccia MKOverlay è stata usata per due scopi:

1. Il tracciamento delle linee di collegamento tra le varie posizioni che l'utente attraversava durante i suoi spostamenti cittadini.
2. La raffigurazione grafica sotto forma di mappa di calore di tutte le *locations* che il device dell'utente è riuscito a registrare.

Gli MKOverlay sono associati con i corrispettivi *render* che nello specifico si chiamano MKOverlayRenderers, ed essi vengono richiamati attraverso l'utilizzo dei delegati. I *renderers* sono quelli che effettivamente si occupano, una volta letto lo specifico *overlay*, di visualizzarlo attraverso la scheda grafica dell'iphone.

3.4 La mappa di calore

Come intravisto in alcuni *screen-shots* del capitolo due, una delle funzionalità da realizzare in iOS era la mappa di calore.
Nella versione Android già era disponibile questa funzionalità e si trattava di una semplice visualizzazione della Google Map che copriva quasi tutto lo schermo del device con la possibilità di vedere in sovrimpressione tutti i punti più significativi in cui l'utente si era recato. Ogni punto era rappresentato da un piccolo punto di calore, e più i punti erano vicini più si raggruppavano fra loro creando una macchia di calore uniforme e ampia man mano che si applicava il *pinch-to-zoom*[5]. Lo scopo era appunto dare all'utente una visione globale della sua impronta di mobilità, relativa a tutti i luoghi che aveva visitato da quando aveva iniziato ad utilizzare attivamente questa applicazione.

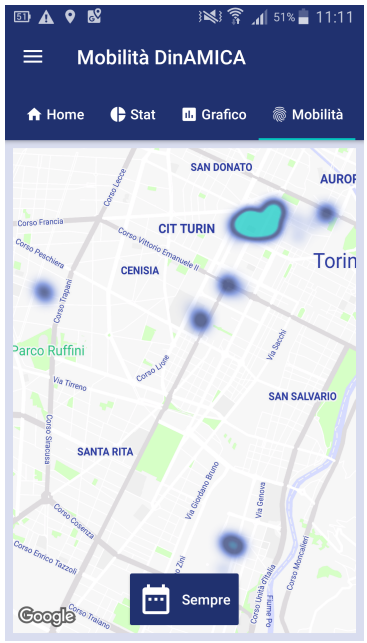


Figure 17 - Esempio mappa di calore Android su Torino

La visualizzazione è abbastanza chiara e auto esplicativa, per l'utente basta cliccare sul filtro "sempre" e automaticamente viene visualizzata una schermata simile a quella soprastante. In figura diciassette è facile notare come l'utente si è mosso maggiormente nella zona centrale della città di Torino, per questo motivo la chiazza di calore nei pressi di via Garibaldi è più grande.

In Android dentro le mappe di Google era già disponibile una libreria che permetteva di realizzare le mappe di calore, e per l'appunto si chiama HeatMapUtility.
Una volta settati alcuni parametri iniziali e ricavate le posizioni dell'utente era abbastanza semplice settare e realizzare la *heating map*, richiamando il metodo **HeatmapTileProvider.Builder()**, e passandogli la lista di oggetti con al loro interno la latitudine e la longitudine (19). Il fatto che questa libreria era già fornita da Google significava

[5] Movimento delle dita che ci permette di ingrandire o rimpicciolire con due dita la scala di visualizzazione di una immagina, un video, una mappa etc.

che non bisognava cercare *software* di terze parti compatibili con l'ambiente di sviluppo, e dunque molti problemi di compatibilità erano già risolti.

Per quanto riguarda iOS purtroppo una libreria di questo tipo non era fornita per MapKit dunque è stato necessario trovare una soluzione alternativa.
Inizialmente si è valutata una libreria interamente scritta in Swift che va sotto il nome di JDSwiftHeatMap[6], disponibile su GitHub sotto licenza MIT. Sembrava la più indicata inizialmente perché in quanto il progetto di questa tesi è stato quasi interamente sviluppato in Swift allora si è pensato potesse essere un effettivo vantaggio, anche per evitare futuri problemi di compatibilità. Purtroppo però, una volta integrata nel progetto e resa eseguibile, non aveva il comportamento sperato. Il problema era dovuto al fatto che invece di rendere la macchia di calore in sovrimpressione reattiva al *pinch-to-zoom*, quello che faceva era semplicemente aggiungere staticamente una immagine in overlay in una determinata porzione della mappa e nient'altro. Dunque è stata scartata.

Successivamente si è trovata invece un'altra libreria, integrabile sotto forma di *CocoaPods*[7], che si chiama DTMHeatmap[8], disponibile anch'essa su GitHub sotto licenza MIT. Questa, a differenza della prima, era realizzata in ObjectiveC[9] dunque prima di tutto è stato necessario comprenderne le dinamica di base, successivamente renderla eseguibile in ambiente Swift e infine adattarla dove necessario alle logiche del progetto di tesi.

```
- (void)viewDidLoad
{
    self.mapView.delegate = self;

    // Create DTMHeatmap
    self.heatmap = [DTMHeatmap new];
    [self.heatmap setData:myData];
    [self.mapView addOverlay:self.heatmap];

    // or...
    // Create DTMDiffHeatmap
    self.diffHeatmap = [DTMDiffHeatmap new];
    [self.diffHeatmap setBeforeData:beforeData
                          afterData:afterData];
    [self.mapView addOverlay:self.diffHeatmap];
}

- (MKOverlayRenderer *)mapView:(MKMapView *)mapView
            rendererForOverlay:(id<MKOverlay>)overlay
{
    return [[DTMHeatmapRenderer alloc] initWithOverlay:overlay];
}
```

Figure 18 - Estratto guida ufficiale per utilizzo della libreria DTMHeathmap

Nella figura soprastante è mostrato un estratto di codice suggerito dagli stessi autori della libreria su come andrebbe usato all'interno del progetto. A parte la differenza di sintassi tra ObjectiveC e Swift si è notato come le parti importanti fossero:

---

[6] Link = https://github.com/jamesdouble/JDSwiftHeatMap.
[7] È un manager di dipendenze di livello applicativo. I CocoaPods sono spesso usati per l'integrazione di librerie esterne in progetti software che sono in grado di eseguire codice scritto in ObjectiveC o Swift. Essi sono stati fortemente ispirati dal concetto di gemme dentro i progetti scritti in Ruby.
[8] Link = https://github.com/dataminr/DTMHeatmap.
[9] Linguaggio di programmazione sviluppato da Apple per ambienti iOS che veniva usato prima dell'avvento del più moderno Swift.

- Settare il delegato.
- Rendere funzionante il metodo **setData** richiamato dentro l'oggetto heatmap di tipo DTMHeatmap.

Ad alto livello sembra semplice e intuitivo ma le difficoltà sono nate dietro le quinte. Proviamo a farne una descrizione ad alto livello partendo dal metodo **setData** menzionato poco fa.

La sintassi di setData dentro la libreria ufficiale è descritta in questo modo:

```
- (void)setData:(NSDictionary *)newHeatMapData;
```

Figure 19 - Porzione di codice in ObjectiveC della libreria DTMHeatmap

secondo la documentazione ufficiale, molto esigua, veniva suggerito di passare un NSDictionary *wrapped*[10] dentro un tipo NSValue usando una particolare sintassi in modo da specificare anche un determinato *weight*[11].
Al fine di renderlo compatibile e funzionante con i tipi di variabile, leggermente diversi, esistenti in Swift si è deciso di utilizzare come parametro da passare alla funzione **setData** un Dictionary costituito da una coppia generica di valori rappresentati rispettivamente con NSObject e AnyObject.
Quest'ultimo, il dictionary, prima di tutto doveva essere popolato con i valori provenienti dal modello dei dati dell'applicazione.
Una delle difficoltà incontrate è stata capire come rendere compatibile il punto geografico immagazzinato dentro il modello dati tramite l'utilizzo di librerie di Swift con il punto geografico concepito per la libreria DTMHeatmap scritta in ObjectiveC.
Questo lavoro di conversione viene fatto direttamente dentro il metodo **loadHeatingPointsFromDataModel**. Questo metodo è richiamato dentro MapVC ma è definito dentro la classe DataManager.
Per risolvere questo problema di compatibilità si è usato un approccio misto. Dove possibile sono stati usati i tipi di variabili standard pensati per CoreData[12] mentre quando si riscontravano errori di compilazione o addirittura errori a runtime, nell'esecuzione della libreria stessa, ci si è resi conto che bisognava utilizzare alcune volte approcci un po' più a basso livello per rendere ObjectiveC compatibile con Swift.
Un esempio è mostrato in figura venti dove nella creazione della variabile *value* viene richiamato il costruttore NSValue e il parametro mapPoint viene passato utilizzando la e commerciale ossia il simbolo '&'.
Siccome di default i parametri delle funzioni in Swift sono passati per valore, in questo caso applicando questo specifico simbolo il parametro viene passato come un *in-out parameter*,

[10] Ossia inglobato.
[11] Questa variabile rappresenta il peso che viene dato al singolo punto di calore, più il valore era alto più sarebbe stata intensa la sua rappresentazione sulla mappa.
[12] CoreData è un framework sviluppato per macOS e iOS. È pensato come modello *relational entity-data model* e si occupa della realizzazione, gestione, serializzazione e persistenza dei dati all'interno dell'applicazione.

rendendolo dunque modificabile anche all'interno della funzione stessa.

```
                                                                cacheName: nil
    )
    try? fetchedResultController.performFetch()
    var heatingPoints = Dictionary<NSObject, AnyObject>()
    let results = fetchedResultController.fetchedObjects
    for element in results! {
        let coordinate = CLLocationCoordinate2D(latitude: element.latitude, longitude: element.longitude);
        var mapPoint = MKMapPointForCoordinate(coordinate)
        let type = NSValue(mkCoordinate: coordinate).objCType
        let value = NSValue(&mapPoint, withObjCType: type)
        heatingPoints[value] = element.weigth as AnyObject
    }
    completion(heatingPoints)
}
```

*Figure 20 - Porzione di codice scritta in Swift della funzione **loadHeatingPointsFromDataModel** dentro la classe DataManager*

Una volta che il *load* dei punti convertiti è terminato, viene chiamata la funzione **setUpHeatingPoints** alla quale vengono passati i punti convertiti precedentemente ma sotto forma di variabile heatingPoints, come è possibile vedere nella figura sottostante.

```
else { // SHOW HEATMAP
  startDate.text = "-"
  try? DataManager.shared.loadHeatingPointsFromDataModel(motionType: Travel.SOMotionType.NotValid,
                                                         dateFilter: dataFilter!, completion: { (heatingPoints) in
      self.setUpHeatingPoints(heatingPoints)
  })
```

*Figure 21 - Porzione di codice scritta in Swift dove viene richiamata la funzione **setUpHeatingPoints***

Il metodo **setData** poi viene richiamato a sua volta dentro la funzione **setUpHeatingPoints**, come mostrato in figura ventidue.

```
    private func setUpHeatingPoints(_ points: Dictionary<NSObject, AnyObject>) {
        mapView.remove(heatmap)
        mapView.removeOverlays(pathmap.getData())
        heatmap = DTMHeatmap()
        if (points.count>0) {
            heatmap.setData(points)
            mapView.add(heatmap)
        }
    }
```

*Figure 22 - Porzione di codice scritta in Swift della funzione **setUpHeatingPoints** in cui viene usata la libreria DTMHeatmap*

Questa funzione viene richiamata dentro la classe MapVC. Nel dettaglio quello che avviene dentro la setUpHeatingPoints è che prima di tutto va a rimuovere una eventuale mappa di calore precedentemente allocata e poi ne aggiunge una nuova, andando a caricare i vari punti contenuti nel Dictionary points.
Il vantaggio di Swift è che siccome è stato concepito per coesistere con applicazioni scritte in ObjectiveC, una volta integrata la libreria DTMHeatmap in modo opportuno ed effettuate le dovute conversioni di tipi, dove necessario, era possibile usufruire a pieno delle sue potenzialità.
Quando la heatmap viene aggiunta alla mapView, attraverso l'istruzione "mapView.add(heatmap)", la libreria che gestisce la visualizzazione richiama internamente la funzione **add(_ overlay: MKOverlay)** che difatti è definita dentro la classe MKMapView. Questo scatenerà il relativo delegato che si occuperà di fare il *rendering* dell'immagine sulla *View* della mappa. Quest'ultimo viene gestito sempre dentro la classe MapVC ma in una funzione differente mostrata nella figura sottostante:

```swift
/** Called internally everytime you try to add an overlay image to your MapView
 */
func mapView(_ mapView: MKMapView, rendererFor overlay: MKOverlay) -> MKOverlayRenderer {
    if overlay is DTMHeatmap {
        return DTMHeatmapRenderer(overlay: overlay)
    } else if overlay is CustomPolyline {
        let myCustomPolyLine: CustomPolyline = overlay as! CustomPolyline
        let lineView = MKPolylineRenderer(overlay: overlay)
        lineView.strokeColor = myCustomPolyLine.color
        lineView.lineDashPattern = myCustomPolyLine.dashPattern
        lineView.alpha = myCustomPolyLine.alpha
        return lineView
    } else if overlay is MKCircle {
        let renderer = MKCircleRenderer(overlay: overlay)
        renderer.fillColor = UIColor.black.withAlphaComponent(0.5)
        renderer.strokeColor = UIColor().UIColorFromRGB(rgbValue: 0x55DBCB)
        renderer.lineWidth = 2
        return renderer
    }
    return MKOverlayRenderer()
}
```

Figure 23 - Porzione di codice scritta in Swift dove viene gestito il rendering per la mappa di calore

Nello specifico qui vengono gestiti più tipi di *overlay* che potrebbero comparire sulla mappa, essi verranno spiegati più avanti. Per quanto riguarda quello inerente la mappa di calore, basta guardare la prima porzione di codice dove in pratica se viene riconosciuto che l'*overlay* è di tipo DTMHeatmap verrà chiamato l'opportuno DTMHeatmapRenderer e restituito come tipo di ritorno dalla funzione. Sarà il sistema a interpretarlo correttamente e visualizzarlo a *runtime* attraverso la scheda video.
Il risultato finale sarà una rappresentazione a macchia di calore di questo tipo:

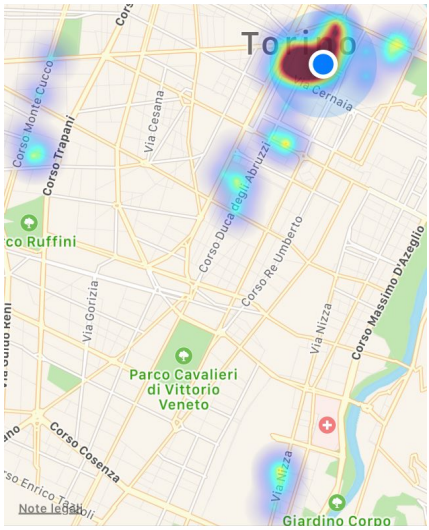


Figure 24 - Esempio mappa di calore su iOS

si può notare come in alcune zone della mappa la macchia di calore è più diradata e in altre è più intensa, raggiungendo addirittura la colorazione rossa. Il motivo è semplicemente perché c'è una correlazione direttamente proporzionale tra il numero di posizioni catturate dal device in cui l'utente si è recato e il colore più intenso e omogeneo usato nella rappresentazione della heatmap.

3.5 Rilevazione del movimento e co-processore M7

Una delle parti fondamentali del progetto in esame era capire come intercettare il movimento dell'utente nel modo più accurato possibile.
Il riconoscimento del tipo di attività compiuta dall'utente è di vitale importanza per il fine ultimo dell'app, perché questa informazione associata ai luoghi in cui si sposta vanno a fornire all'ente ultimo che analizzerà i dati una panoramica più precisa sulle abitudini dei cittadini.

Per realizzare questo obiettivo si è deciso di utilizzare un *set* di sensori già presenti a bordo dell'iphone ovvero:

- Accelerometro.
- Giroscopio.
- Magnetometro.

CoreMotion è la libreria ufficiale di Apple che si occupa della gestione dei dati provenienti da tutti e tre i sensori menzionati poco fa. Essa presenta tutto un insieme di funzioni che permettono l'accesso ai dati.
Per capire come maneggiare questi dati, prima di tutto, bisogna partire da una classe, contenuta all'interno della stessa libreria, che si chiama CMMotionManager.
Siccome di questi sensori ce n'è solo uno per tipo in tutto il device di solito a livello di programmazione è preferibile fare una singola classe che li gestisce, utilizzando un oggetto di tipo CMMotionManager, in modo da evitare la replica di codice in altre parti che porterebbe solo a ulteriore confusione.
Per recuperare le informazioni è possibile farlo in due modi:

- Tramite *polling* dunque con delle richieste esplicite.
- Tramite una funzione apposita che setta un determinato *rate* entro il quale riceviamo aggiornamenti sui dati dei sensori con una specifica frequenza. In quel determinato *rate* è possibile fornire una *closure* all'interno della quale possiamo processare i dati ricevuti in base alle nostre necessità.

A prescindere dall'approccio che si sceglie è importante ricordarsi che una volta ottenuta l'informazione bisogna sempre spegnere, tutte le volte che ne abbiamo l'opportunità, il ricevimento di nuovi updates relativi ai sensori di movimento.
È una regola fondamentale da seguire, altrimenti si rischia di consumare batteria inutilmente e l'utente sarà invogliato a chiudere l'applicazione o addirittura cancellarla definitivamente.

Per quanto riguarda l'integrazione di queste funzionalità nel progetto si è anche valutato inizialmente l'inserimento di un CocoaPod sotto il nome di SOMotionDetector[13] disponibile su GIT sotto licenza MIT. Essa dichiarava di essere in grado di riconoscere il tipo di movimento dell'utente e di saper distinguere tra le seguenti categorie:

- Utente fermo.
- Utente in cammino.
- Utente di corsa.
- Utente su un veicolo motorizzato.

Come primo *step* si è cercato di comprendere a pieno la libreria scritta in ObjectiveC. Si è scoperto che la maggior parte del lavoro era fatta dentro il file SOMotionDetector.m. Dentro questa classe venivano definite delle costanti in questo modo:

[13] Link = https://github.com/arturdev/SOMotionDetector

```
CGFloat kMinimumSpeed        = 0.3f;
CGFloat kMaximumWalkingSpeed = 1.9f;
CGFloat kMaximumRunningSpeed = 7.5f;
CGFloat kMinimumRunningAcceleration = 3.5f;
```

Figure 25 - Porzione di codice della libreria SOMotionDetector nel file SOMotionDetector.m

esse erano utilizzate nel calcolo dei vari tipi di moto facendo dei confronti. Ovviamente però non bastava avere solo delle soglie per calcolare il tipo di attività ma serviva qualcosa di più. Quello che accadeva è che ad un certo punto veniva chiamato il metodo startDetection, esattamente in questa porzione di codice veniva istanziato un timer di tipo NSTimer con una frequenza pari al valore indicato sul parametro scheduledTimerWithTimeInterval, della medesima classe:

```
- (void)startDetection
{
    [[SOLocationManager sharedInstance] start];

    self.shakeDetectingTimer = [NSTimer scheduledTimerWithTimeInterval:0.01f
                                                                target:self
                                                              selector:@selector(detectShaking)
                                                              userInfo:Nil
                                                               repeats:YES];

    [self.motionManager startAccelerometerUpdatesToQueue:[[NSOperationQueue alloc] init]
                                             withHandler:^(CMAccelerometerData *accelerometerData, NSError *error)
    {
        _acceleration = accelerometerData.acceleration;
        [self calculateMotionType];
        dispatch_async(dispatch_get_main_queue(), ^{
```

Figure 26 – Porzione di codice del metodo startDetection nel file SOMotionDetector.m

settato quest'ultimo valore a 0.01 significava che il timer scattava con un intervallo pari a un centesimo di secondo lanciando la funzione **detectShaking**.
Questa funzione era in grado di utilizzare i valori catturati sui tre assi dell'accelerometro, e se il vettore somma di queste tre accelerazioni superava una certa soglia, ripetutamente, allora sostanzialmente significava che il cellulare in quel preciso momento era stato scosso e quindi veniva automaticamente settata la variabile isShaking.
Questa variabile era importante perché il suo valore veniva considerato nell'algoritmo per il calcolo del tipo di attività dell'utente.
Essa veniva determinata all'interno della funzione **calculateMotionType**:

```
- (void)calculateMotionType
{
    if (self.useM7IfAvailable && [SOMotionDetector motionHardwareAvailable]) {
        return;
    }

    if (_currentSpeed < kMinimumSpeed) {
        _motionType = MotionTypeNotMoving;
    } else if (_currentSpeed <= kMaximumWalkingSpeed) {
        _motionType = _isShaking ? MotionTypeRunning : MotionTypeWalking;
    } else if (_currentSpeed <= kMaximumRunningSpeed) {
        _motionType = _isShaking ? MotionTypeRunning : MotionTypeAutomotive;
    } else {
        _motionType = MotionTypeAutomotive;
    }
```

Figure 27 - Porzione di codice del metodo calculateMotionType dentro il file SOMotionDetector.m

Sostanzialmente, qualora un determinato tipo di hardware non fosse presente, passava all'esecuzione di un algoritmo che usava un approccio risolutivo misto. Affrontava il problema effettuando un confronto utilizzando sia delle soglie specifiche che anche il valore della variabile isShaking di cui abbiamo parlato prima. Il risultato finale era scritto dentro la

variabile _motionType.
Quello che non era ben chiaro era cosa fosse quel confronto fatto con dei valori relativi all'hardware del dispositivo. Andando a fondo nella libreria e nella documentazione si è scoperto che uno dei vantaggi di SOMotionDetector era proprio che supportava due tipi di riconoscimento del movimento. Il primo era relativo a quello che abbiamo descritto poche righe fa. E il secondo invece è relativo all'utilizzo di una tecnica più moderna attraverso l'oggetto CMMotionActivityManager.

Questa scoperta significava sostanzialmente che questa libreria poteva funzionare sia con dispositivi antecedenti o uguali a iphone5, sia con dispositivi più nuovi ovvero a partire dall'iphone 5S in su.

La differenza principale era che nei dispositivi più moderni Apple aveva introdotto un coprocessore di movimento M7. Quest'ultimo aveva lo scopo specifico di interpolare tutti i dati dei sensori dell'iphone e calcolare in continuazione, e in modo separato dal processore principale, il tipo di attività dell'utente ovvero come lui si spostava.

Il tutto avveniva dentro un'altra porzione di codice della medesima classe:

```
if (self.useM7IfAvailable && [SOMotionDetector motionHardwareAvailable]) {
    if (!self.motionActivityManager) {
        self.motionActivityManager = [[CMMotionActivityManager alloc] init];
    }

    [self.motionActivityManager startActivityUpdatesToQueue:[[NSOperationQueue alloc] init] withHandler:^(CMMotionActivity *activity) {
        dispatch_async(dispatch_get_main_queue(), ^{

            if (activity.walking) {
                _motionType = MotionTypeWalking;
            } else if (activity.running) {
                _motionType = MotionTypeRunning;
            } else if (activity.automotive) {
                _motionType = MotionTypeAutomotive;
            } else if (activity.stationary || activity.unknown) {
                _motionType = MotionTypeNotMoving;
            }
```

Figure 28 - Porzione di codice di startDetection dentro i file SOMotionDetector.m

quello che viene fatto nella porzione di codice soprastante è verificare se l'hardware è disponibile, perché ovviamente dipende dal modello di iphone su cui il software è installato, e fidarsi del valore restituito dal coprocessore M7 e recuperato tramite l'oggetto di tipo CMMotionActivityManager. Alla fine viene settata la variabile finale _motionType, in base al tipo di movimento rilevato.

Compresi approfonditamente i meccanismi interni della libreria SOMotionDetector si è pensato di integrarla nel progetto principale. Il problema però era che qualora l'avessimo integrata sotto forma di CocoaPod non avremmo avuto accesso a tutte le personalizzazioni del caso, di cui probabilmente avremmo avuto di bisogno.
Inoltre di questa libreria non esisteva una versione aggiornata con le API di Apple più moderne. Dunque si è deciso di tradurre il codice di questa libreria in Swift e integrarlo nel progetto.
Il dettaglio della soluzione implementativa verrà approfondito nel capitolo successivo.

4. Capitolo 4 – Implementazione

4.1 Fase di avvio: TutorialVC e PolicyVC

In Android l'elemento base da cui si parte per la realizzazione di una applicazione è la Activity, in iOS il corrispettivo delle Activity sono i ViewController. In questo progetto gestiamo un numero pari a 16 ViewController.

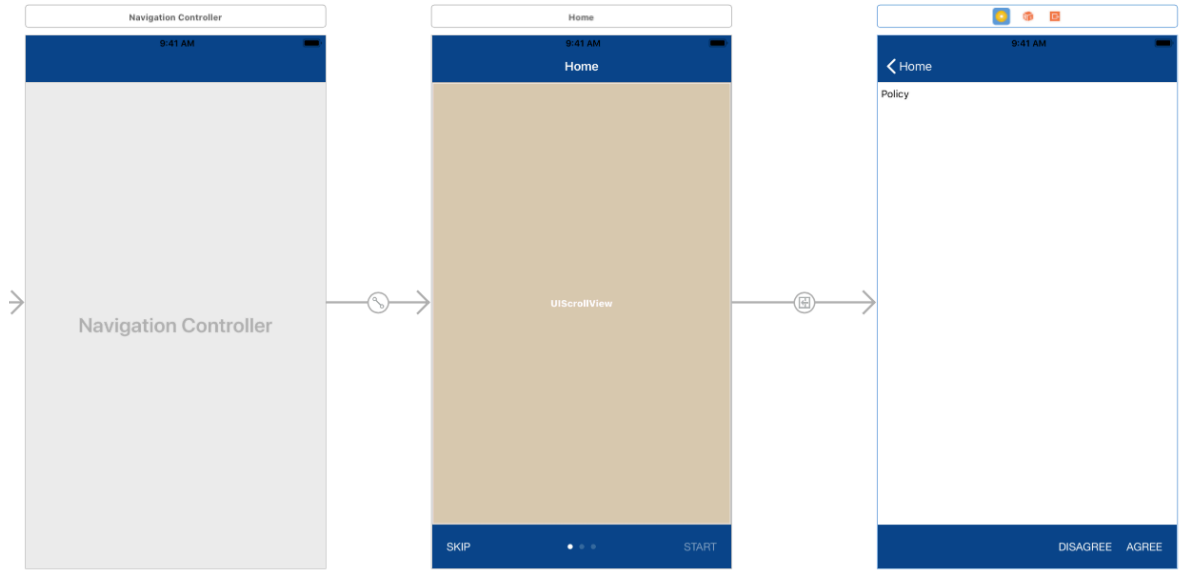


Figure 29 - Porzione di storyboard in cui viene gestita la fase iniziale

Tutte le schermate con le rispettive chiamate alle successive sono gestite, a livello visuale, nella storyboard della applicazione. Il file in questione si chiama Main.storyboard.
La figura sopra rappresenta la prima parte di schermate che viene mostrata all'avvio dell'applicazione. In pratica abbiamo a sinistra un NavigationController che, come è possibile vedere nella parte centrale sulla sinistra, ha una freccia in grigio. Questa freccia in grigio sta ad indicare che il software quando viene avviato parte da li. Quello è il suo primo punto di contatto.
Da questo NavigationController in base a cosa viene premuto dall'utente vengono effettuati dei Segue[14].
In questo caso i passaggi sono 'obbligati' perché la parte centrale, mostrata in figura ventinove, rappresenta la schermata di Home gestita dalla classe TutorialVC. Quest'ultima si occupa di visualizzare con dei walkthrough-screens[15] delle schermate introduttive in cui viene spiegato sommariamente un mini-tutorial di quello che l'applicazione è in grado di fare.
Successivamente si passa alla screen della *policy*, qui viene mostrato all'utente un documento inerente l'informativa sul trattamento dei dati personali ai sensi degli articoli di legge vigenti al momento. L'utente può decidere se accettare o meno, ma solo una volta accettata questa informativa può passare alla schermata principale dell'applicazione vera e propria.

Quello che accade dietro le quinte è descritto nel codice dentro la classe PolicyVC. Ecco una porzione di codice:

[14] In iOS il Segue serve per effettuare la transizione da un ViewController ad un altro.
[15] Per walkthrough-screens si intendono delle schermate che scorrono in orizzontale facendo swipe sullo schermo verso destra o verso sinistra.

```
28
29      @IBAction func disagreeButtonPressed(sender:UIButton) {
30          UIControl().sendAction(#selector(URLSessionTask.suspend), to: UIApplication.shared, for: nil)
31      }
32
33      @IBAction func agreeButtonPressed(sender:UIButton) {
34          UserDefaults.standard.set(true, forKey: AppDelegate.PRIVACY_ACCEPTED)
35          DataManager.shared.setCurrentUser()
36          subscribeAndLogin()
37      }
```

Figure 30 - Porzione di codice della classe PolicyVC

Nella funzione **agreeButtonPressed(sender:UIButton)** viene creato nel modello dei dati un utente, e successivamente viene fatta la sottoscrizione dell'utente e il login. Richiamando la funzione **subscribeAndLogin()**.

```
private func subscribeAndLogin() {
    self.showLoaderAnimated(animated: true)
    NetworkManager.shared.subscribeAndLogin { (success, response) in
        if(success) {
            let storyboard = UIStoryboard(name: "Main", bundle: nil)
            if let tabViewController = storyboard.instantiateViewController(withIdentifier: "main") as? UITabBarController {
                self.hideLoaderAnimated(animated: true)
                self.present(tabViewController, animated: true, completion: nil)
            }
        } else if (response == nil) { // NO RESPONSE FROM SERVER.
            let alert = UIAlertController(title: NSLocalizedString("error", comment: ""), message:
                NSLocalizedString("network_error", comment: ""), preferredStyle: .alert)
            alert.addAction(UIAlertAction(title: "OK", style: .default, handler: nil))
            self.hideLoaderAnimated(animated: true)
            self.present(alert, animated: true, completion: nil)
        } else { // GENERIC ERROR.
            let alert = UIAlertController(title: NSLocalizedString("error", comment: ""), message:
                NSLocalizedString("generic_error", comment: ""), preferredStyle: .alert)
            alert.addAction(UIAlertAction(title: "OK", style: .default, handler: nil))
            self.hideLoaderAnimated(animated: true)
            self.present(alert, animated: true, completion: nil)
        }
    }
}
```

Figure 31 - Metodo subscribeAndLogin nella classe PolicyVC

Dentro il metodo rappresentato in figura trentuno viene inizialmente richiamata la funzione **subscribeAndLogin** della libreria NetworkManager. Quest'ultima, memorizzata dentro la cartella Api del progetto, si occupa appunto di gestire le interazioni con la rete su un *thread*[16] a parte. Una volta che la sottoscrizione è andata a buon fine viene caricata dalla *storyboard* la schermata con l'identificativo 'main':

[16] Per *thread* si intende una porzione di un processo. Esso è anche detto processo leggero e la sua gestione cambia in base al sistema operativo che si sta utilizzando.

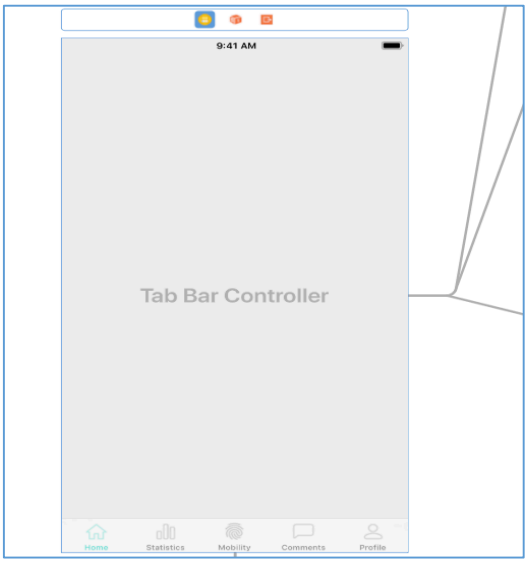


Figure 32 – TabBarController con identificativo 'main' nella storyboard

A questo TabBarController sono collegate le varie diramazioni verso le principali schermate dell'applicazione. L'utente cliccando dal menu in basso può decidere in quale porzione dell'applicazione navigare. I rispettivi ViewController che vengono chiamati sono rappresentati in modo riassuntivo in questa schermata:

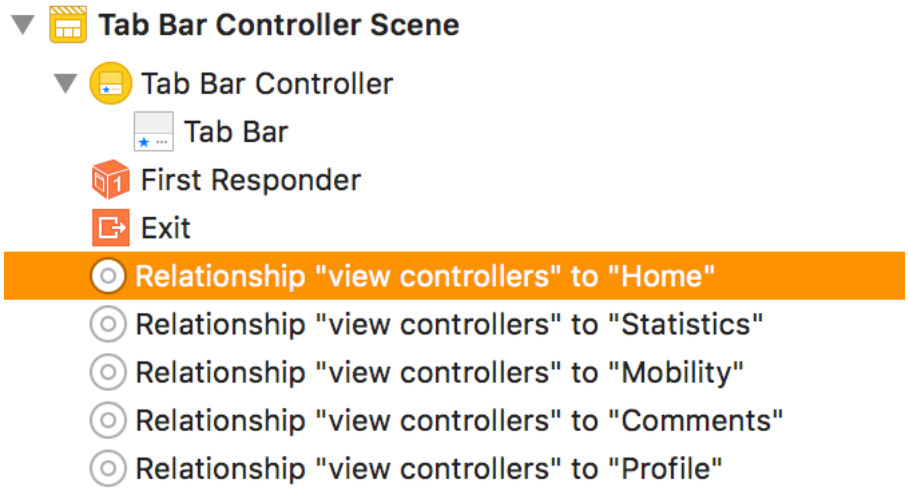


Figure 33 – Visione riassuntiva delle relazioni a cui il TabBarController è direttamente collegato

Come è possibile notare, la prima schermata è la Home e poi a seguire tutte le altre di cui parleremo più approfonditamente nei prossimi paragrafi.

Tornando al metodo **subscribeAndLogin** è possibile notare che nella parte 'else' del costrutto viene gestito il caso in cui la *login* non è andata a buon fine e dunque viene mostrata una a*lert*[17] all'utente indicandogli il tipo di errore, sia che venga da un problema con il server sia che venga da un errore generico interno all'applicazione.

[17] Per *alert* si intende una *alert-box* ovvero una finestra grafica, che spunta in sovrimpressione, con un messaggio che tipicamente segnala qualcosa di importante all'utente.

4.2 Schermata Home: HomeVC

La *home* rappresenta in un certo senso il biglietto da visita dell'applicazione perché è la prima interfaccia che l'utente visualizza subito dopo aver accettato la policy relativa alla privacy del trattamento dei dati personali.
Inizialmente quando il progetto per lo sviluppo in iOS ebbe inizio non erano stati ancora stabiliti dei canoni grafici specifici, dunque si è lavorato molto più sulle funzionalità principali dell'applicazione.
Per quanto riguarda il look grafico dunque ci si è basati su quello già realizzato in passato in Android, cercando di replicarlo in iOS nel modo più fedele possibile.
Per esempio una sfida considerevole era stata realizzare un elemento che in Android è molto comune ma che nel mondo Apple non lo è, si sta parlando del *three-line menu*[18]. In Android Studio è possibile crearlo quasi istantaneamente grazie all'IDE già in fase di creazione del progetto. In Xcode invece non è contemplato di *default*, quindi bisogna crearlo graficamente costruendolo da zero.
Poi a seguito di determinate interazioni con il cliente e di alcune intuizioni del *designer* ci si è resi conto che a livello di *user-experience* un tipico utente Apple è abituato ad avere una determinata struttura visiva, lui istintivamente sa che per navigare nelle varie parti della sua app ha bisogno di un certo *template*[19]. Dunque in fase finale si è deciso di rinnovare la veste grafica dell'intera applicazione, a partire ovviamente dalla *home*.
Come *template* di navigazione si è optato per il TabBarController citato nel paragrafo precedente, la prima diramazione è appunto la home-screen rappresentata in figura sottostante:

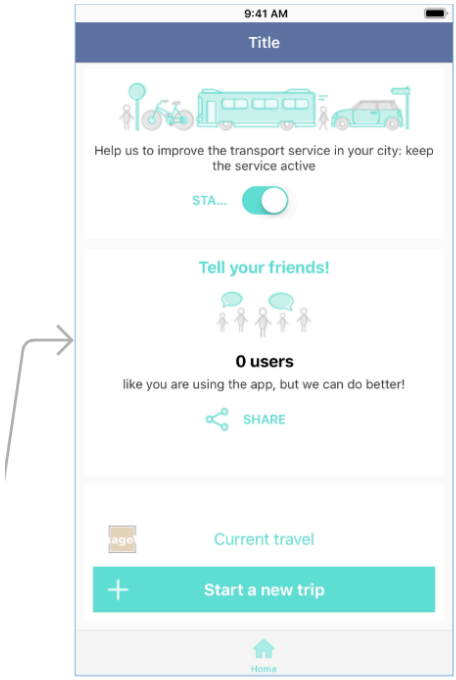


Figure 34 - Home Screen iOS

In iOS, solitamente, a ogni schermata presente nella *storyboard* corrisponde un determinato ViewController che si occupa di gestirla. La gestione si occupa non solo di quello che l'utente

[18] È il menu (chiamato in gergo tecnico *hamburger-menu*) che tipicamente appare nelle app Android. Esso è rappresentato da un simbolo di tre linee parallele, solitamente posizionate in alto a sinistra. L'utente cliccandoci sopra ha a disposizione più scelte di navigazione.
[19] Per *template* in questo caso si intende un determinato schema grafico.

può fare agendo direttamente sui controlli disponibili, ma anche di tutto quello che accade dietro l'interfaccia grafica.

In questo caso il ViewController responsabile di questa gestione è rappresentato dalla classe HomeVC. Esso svolge una infinità di compiti, perché ovviamente istanti prima che la schermata venga visualizzata deve essere tutto pronto affinché l'utente possa già utilizzare l'applicazione.

```
import UIKit
import CoreLocation

class HomeVC: BaseVC {

    @IBOutlet private var navBarItem: UINavigationItem!

    @IBOutlet private var swithcLocationContainer: UIView!
    @IBOutlet private var switchImgWidth: NSLayoutConstraint!
    @IBOutlet private var padTopSwitch: NSLayoutConstraint!
    @IBOutlet private var switchText: UILabel!
    @IBOutlet private var switchButton: UISwitch!
    @IBOutlet private var startLabel: UILabel!

    @IBOutlet private var shareContainer: UIView!
    @IBOutlet private var shareImgWidht: NSLayoutConstraint!
    @IBOutlet private var shareTitle: UILabel!
    @IBOutlet private var usersText: UILabel!
    @IBOutlet private var shareText: UILabel!
    @IBOutlet private var shareBtn: UIButton!

    @IBOutlet private var currentTravelView: UIView!
    @IBOutlet private var currentTravelImg: UIImageView!
    @IBOutlet private var addTravelImg: UIImageView!
    @IBOutlet private var currentTravelLabel: UILabel!
    @IBOutlet private var newTripContainer: UIView!
    @IBOutlet private var newTripButton: UIView!
    @IBOutlet private var newTripText: UILabel!
    @IBOutlet private var newTripButtonHeight: NSLayoutConstraint!
```

Figure 35 - HomeVC IBOutlets

Nella figura soprastante vediamo la porzione iniziale di HomeVC. Intanto si nota subito come esso erediti da BaseVC. BaseVC è la classe di base da cui tutti i ViewController gestiti dal TabBarController, ereditano determinate caratteristiche. Questo perché in BaseVC vengono fatte principalmente due cose:

- L'impostazione di alcune proprietà grafiche comuni a più schermate.
- L'implementazione del protocollo "UINavigationBarDelegate" che si occupa di gestire la visualizzazione delle singole *view* durante la navigazione dentro l'applicazione. In pratica a seconda del bottone che l'utente premerà nel menù in basso esso farà riferimento ad un solo elemento gestito dal TabBarController.

Tutte le variabili con accessibilità private invece rappresentano tutti i riferimenti agli *IBOutlet*[20] presenti nella grafica della *Home Scene*[21].

[20] Gli *IBOutlet* in Swift sono dei tipi particolari che hanno un legame 1:1 con gli elementi grafici nella storyboard.
[21] In questo caso si intende la Figura trentaquattro, ovvero la veste grafica nella storyboard relativa alla schermata di Home.

```
38
39      override func viewDidLoad() {
40          super.viewDidLoad()
41
42          // Start to track user.
43          PositionManager.shared.startService()
44
45          navBarItem.title = Bundle.main.localizedInfoDictionary!["CFBundleDisplayName" as String] as? String
46
47          newTripButton.layer.cornerRadius = 7
48          newTripButton.clipsToBounds = true
49
50          shareContainer.layer.cornerRadius = 7
51          shareContainer.clipsToBounds = true
52
53          swithcLocationContainer.layer.cornerRadius = 7
54          swithcLocationContainer.clipsToBounds = true
55
56          newTripContainer.layer.cornerRadius = 7
57          newTripContainer.clipsToBounds = true
58
59          switch UIDevice.current.screenType {
60          case UIDevice.ScreenType.iPhone4_4S, UIDevice.ScreenType.iPhones_5_5s_5c_SE:
61              usersText.font = usersText.font.withSize(14)
62              shareText.font = shareText.font.withSize(12)
63              shareTitle.font = shareTitle.font.withSize(14)
64              shareBtn.titleLabel?.font = shareBtn.titleLabel?.font.withSize(12)
65              padTopSwitch.constant = 7
66              shareImgWidht.constant = -20
67              switchImgWidth.constant = -20
68              switchText.font = switchText.font.withSize(12)
69              startLabel.font = startLabel.font.withSize(12)
70              currentTravelLabel.font = currentTravelLabel.font.withSize(14)
71              newTripText.font = newTripText.font.withSize(14)
72              newTripButtonHeight.constant = 40
73              break;
74          default:
75              break;
76          }
77
78          setConfigurationForTrip()
79      }
```

*Figure 36 - HomeVC **viewDidLoad()***

Andando avanti nella descrizione del funzionamento dell'HomeVC una delle prime cose che salta all'occhio è il richiamo del metodo startService() [Riga 43, Figura trentasei]. Esso è contenuto dentro un un oggetto condiviso fra tutte le classi del progetto, chiamato PositionManager. Come è intuibile dal nome esso si occupa di tutto ciò che riguarda la localizzazione, ma siccome ha un funzionamento parecchio importante e anche complesso esso verrà attenzionato nei prossimi paragrafi.

Successivamente vengono fatte tutta una serie di settaggi grafici relativi ai bordi tondeggiati di determinati *outlet* e poi vengono anche adattati i font di alcuni elementi nella scena qualora venga riconosciuto che il device ha uno schermo più piccolo del previsto, come ad esempio iPhone4 o iPhone5. E infine viene richiamata la funzione setConfigurationForTrip(), essa internamente usufruisce della classe UserDefaults[22], e tramite essa setta alcuni parametri interni inerenti la gestione di una altra modalità di funzionamento aggiuntiva che l'utente potrà sempre adoperare qualora lo desideri. La modalità in questione è la modalità di viaggio manuale, non se ne era parlato precedentemente ma verrà descritta a tempo debito.

[22] É una interfaccia creata appositamente in ambiente iOS che serve a interagire con un DB interno alla singola applicazione in modo che determinate informazioni possano essere scambiate e memorizzate velocemente sotto forma di chiave-valore senza dover per forza passare da CoreData.

Quella descritta finora era solo la parte relativa alla **viewDidLoad()**, essa nel ciclo di esecuzione della UIViewController è chiamata solo all'inizio, ossia quando la classe viene creata e il file *xib*[23] viene caricato. Successivamente un attimo prima che la *view* effettiva appaia sullo schermo viene invocato nel *life-cycle* del VC[24] la funzione **viewWillAppear()**:

```
80
81  |   override func viewWillAppear(_ animated: Bool) {
82          super.viewWillAppear(animated)
83          // Set the state of the switch considering if the positionManager is monitoring or not.
84          self.switchButton.setOn(PositionManager.shared.isUpdatingLocation, animated: true)
85
86          // Receive notification when the authorization to receive position changes.
87          NotificationCenter.default.addObserver(self, selector: #selector(self.setUpForConfigurationForAuthorization(_:)), name:
                NSNotification.Name(rawValue: PositionManager.kAuthPositionNotificationKey), object: nil)
88          NotificationCenter.default.addObserver(self, selector: #selector(self.currentTripChanged(_:)), name:
                NSNotification.Name(rawValue: PositionManager.kCurrentTripNotificationKey), object: nil)
89          // Receive notification when the switchStatus change.
90          NotificationCenter.default.addObserver(self, selector: #selector(self.currentSwitchStateChanged(_:)), name:
                NSNotification.Name(rawValue: PositionManager.kSwitchStatusChangedNotificationKey), object: nil)
91      }
```

*Figure 37 - HomeVC **viewWillAppear()***

Qui dentro vengono fatte principalmente due cose:

- Viene settato lo *switch-button* nella schermata principale in base al valore contenuto dentro una variabile di PositionManager. Sostanzialmente se il monitoraggio della posizione è stato avviato, a seguito di una serie di conferme e controlli che spiegheremo nel prossimo paragrafo, viene automaticamente abilitato lo switch della schermata home. In questo modo l'utente sa che da lì tutto sta funzionando bene, e qualora volesse disabilitare il funzionamento può sempre farlo.
- Viene utilizzato l'*Observer pattern*[25], impostando le varie funzioni che scatteranno per gestire tutta una serie di eventi. Questi ultimi possono essere:
    - Relativi al verificarsi di un viaggio manuale inserito dall'utente.
    - Relativi al cambio di autorizzazione dei servizi di localizzazione.
    - Relativi al cambio di stato dello switch, azionato dall'utente nel caso in cui volesse attivare o disattivare il servizio di geo-localizzazione.

Nel **viewDidDisappear()**, che sarebbe il complementare di **viewWillAppear()**, come è buona norma di programmazione vengono rimossi gli *observer* dal NotificationCenter[26]. In modo da non sovraccaricare il sistema.

Poi abbiamo la funzione **viewDidAppear()**, essa nel *life-cycle* viene chiamata dopo che la grafica della scena è apparsa. Ecco la porzione di codice:

```
105
106     override func viewDidAppear(_ animated: Bool) {
107         super.viewDidAppear(animated)
108         getDashboardData()
109     }
```

*Figure 38 - HomeVC **viewDidAppear()***

[23] È un elemento che fa parte della *graphical user interface.*
[24] Abbreviazione per indicare genericamente un ViewController.
[25] È un modello di programmazione che si basa sul concetto di eventi e osservatori. Quando determinati eventi vengono scatenati allora i relativi osservatori valutano e scattano eseguendo specifiche porzioni di codice. In iOS è possibile usare ad esempio il NotificationCenter.
[26] È un meccanismo di gestione e distribuzione di notifiche interno al framework Foundation di Apple.

Qui [Riga 108, Figura trentotto] viene richiamato il metodo **getDashboardData()** mostrato nella figura sottostante:

```
191
192     private func getDashboardData() {
193         NetworkManager.shared.dashboardData { (dashboardData) in
194             DispatchQueue.main.async {
195                 if let userCount = dashboardData.users {
196                     self.usersText.text = String(describing:userCount)+" "+NSLocalizedString("users", comment: "")
197                 }
198             }
199         }
200     }
```

*Figure 39 - HomeVC **getDashboardData()***

Qui dentro viene eseguita una funzione chiamata **dashboardData** che fa parte di un oggetto condiviso, anch'esso molto importante, chiamato NetworkManager. Esso rappresenta una libreria creata ad hoc che si occupa di gestire tutte le chiamate di rete che possono avvenire nell'applicazione.

Una cosa che risalta all'occhio è sicuramente che questa funzione non ha una sintassi comune. In pratica essa è stata creata appositamente sfruttando una caratteristica di Swift ovvero le *escaping closure*[27].
Esse sono molto comode quando bisogna eseguire determinati *task* in cascata, ad esempio come in questo caso in cui dentro **dashboardData** vengono eseguite delle specifiche operazioni di rete e di conversione atte al recupero dei dati della *dashBoard*[28] stessa.
Una volta che il recupero dei dati è ultimato, il risultato convertito in formato JSON viene passato dentro la variabile apposita ed elaborato nella porzione di codice fra le parentesi graffe a seguire. Viene dunque aggiornata la grafica della schermata home in base ai dati recuperati dalla rete, facendo attenzione a inserire questo *task* dentro la coda di esecuzione del *main-thread* in modalità asincrona.
Quest'ultima operazione è molto importante che venga eseguita sul *thread* principale, perché esattamente come in Android la grafica è sempre gestita da lui.
L'approccio più comune e corretto di programmazione è quello in cui i *task* lunghi vengano affidati a *thread* secondari ma tutti gli *update* grafici devono obbligatoriamente essere effettuati dal *thread main*, pena altrimenti diversi problemi a *run-time*[29] di inconsistenza dei dati o di accessi di memoria errati.

4.2.1 Modalità di Viaggio Manuale

Passiamo adesso ad alcune interazioni con la *user-interface* che l'utente può effettuare in modo attivo sulla schermata *home*, uno dei modi più comuni per gestire questo tipo di eventi è farlo utilizzando le **IBAction function**[30]. Una di queste IBAction è rappresentata dal metodo

[27] L'*escaping closure* è una *closure* di *escape. O*vvero secondo la documentazione ufficiale Apple è quando questa *closure* viene passata come argomento della funzione, ma è chiamata dopo che la funzione ritorna (20).
[28] Dall'inglese "plancia". In questo caso si intende il "cruscotto" di informazioni che bisogna visualizzare nella schermata principale.
[29] Per *run-time* si intende la fase in cui un determinato software viene eseguito.
[30] Le *IBAction function* sono delle funzioni che vengono create specificatamente con relazione, solitamente 1:1, tra il singolo IBOutlet e l'azione da scatenare associata a un determinato evento. In pratica si agganciano a un elemento grafico della scena ed è possibile scrivere al loro interno del codice che verrà eseguito una volta che l'IBOutlet subirà un cambiamento.

addTripButtonPressed che si occuperà di attivare la funzionalità anticipata prima, ossia quella del viaggio manuale.

Siccome esso è un metodo molto lungo, ne mostriamo una sola porzione riassumendo il comportamento però nella sua totalità.

```
@IBAction func addTripButtonPressed(_ sender:UITapGestureRecognizer) {

    var title: String?
    var message: String?
    if(UserDefaults.standard.integer(forKey: PositionManager.kCurrentTrip) == -1) {
        title = NSLocalizedString("New trip", comment: "")
        message = NSLocalizedString("New_trip_msg", comment: "")
    }

    //Create the AlertController and add Its action like button in Actionsheet
    let addTripActionSheet: UIAlertController = UIAlertController(title: title, message: message, preferredStyle: .actionSheet)

    if(UserDefaults.standard.integer(forKey: PositionManager.kCurrentTrip) != -1) {
        let stop = UIAlertAction(title: Travel.getConfigurationForMotionType(Travel.SOMotionType.MotionTypeUserStop).0,
            style: .default) {
            _ in
            PositionManager.shared.addNewTravel(type: Travel.SOMotionType.MotionTypeUserStop)
        }
        stop.setValue(Travel.getConfigurationForMotionType(Travel.SOMotionType.MotionTypeUserStop).1, forKey: "image")
        stop.setValue(UIColor.red, forKey: "titleTextColor")
        addTripActionSheet.addAction(stop)
    }

    let waiting = UIAlertAction(title: Travel.getConfigurationForMotionType(Travel.SOMotionType.MotionTypeNotMoving).0,
        style: .default) {
        _ in
        PositionManager.shared.addNewTravel(type: Travel.SOMotionType.MotionTypeNotMoving)
    }
    waiting.setValue(Travel.getConfigurationForMotionType(Travel.SOMotionType.MotionTypeNotMoving).1, forKey: "image")
    addTripActionSheet.addAction(waiting)
```

*Figure 40 - HomeVC **addTripButtonPressed()***

Intanto essa è collegata alla variabile newTripButton di tipo IBOutlet [Figura trentacinque]. Come parametro *sender* viene usata la UITapGestureRecognizer. Quest'ultima scelta è stata fatta di proposito, in modo che l'IBAction function scatti sulla intera *view* che contiene tutte le *sub-view* del bottone rettangolare, di sfondo ciano con la scritta bianca "Start a new Trip" [Figura trentaquattro].
In pratica lo scopo è dare una esperienza tattile più piacevole all'utente quando utilizza l'applicazione, perché in questo modo gli basterà cliccare in un qualunque punto contenuto dentro quel rettangolo piuttosto che dover pigiare specificatamente il simbolo di addizione sulla sinistra, che nella fase di editing grafico è significativamente più piccolo.
Ovviamente tutti questi dettagli sono invisibili all'utente perché sono relativi a settaggi del layout visivo che hanno a che fare con valori di rendering dell'immagine, valori di proporzione, pattern di grafica relativi o assoluti etc.

Prima di scendere nei dettagli della logica del metodo introdotto poco fa, spieghiamo cosa è la modalità viaggio manuale e come viene presentata graficamente.
Essa è una funzionalità fatta appositamente per rendere l'utente partecipe nella raccolta dati. Qualora lo desideri, sempre dalla schermata home è possibile cliccare sul tasto di cui parlavamo prima facendo apparire un menu a scelta dal quale l'utente potrà segnalare con quale mezzo si sta spostando. Un esempio nella figura sottostante:

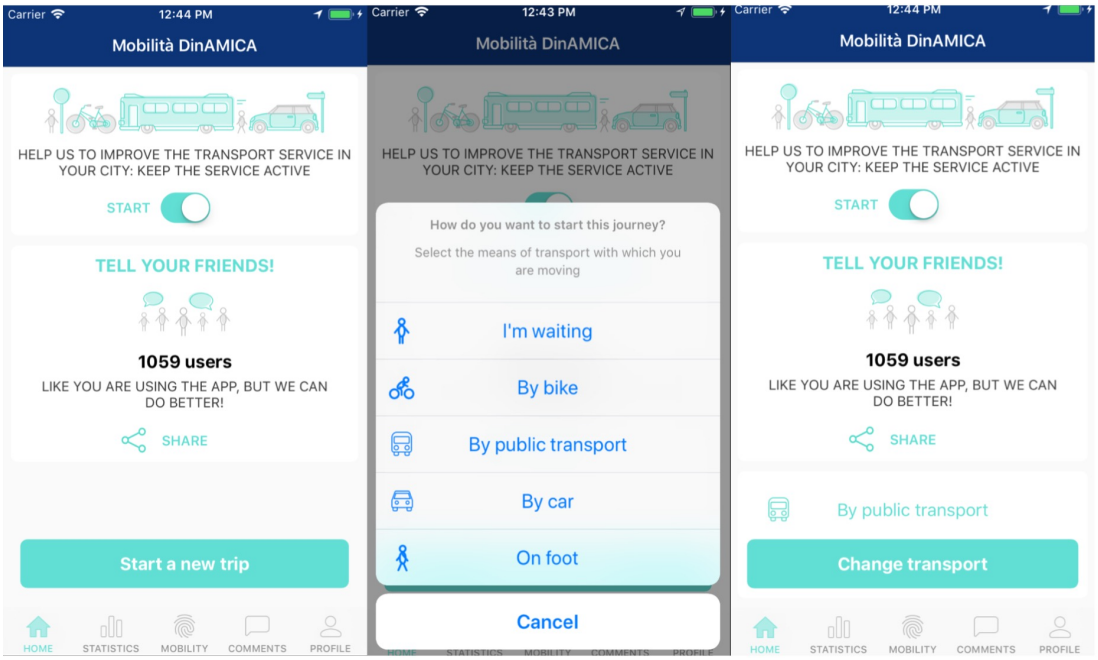


Figure 41 - Screens Viaggio Manuale

La sequenza [Figura quarantuno] evidenzia come una volta cliccato il bottone l'utente può decidere fra una lista di tipi di spostamento urbano: in bici, con la macchina, a piedi o addirittura con bus pubblico. Quest'ultima tipologia è una informazione in più che l'utente può fornire ma che il device, quando lavora in modalità automatica, non può acquisire. Il motivo è semplicemente dovuto al fatto che come spiegato nei capitoli precedenti il device utilizza i sensori interni per intuire il tipo di movimento dell'utente, e i sensori interni possono riconoscere tra le varie tipologie quella con un mezzo motorizzato, ma ovviamente non possono distinguere se il mezzo di trasporto è pubblico o privato.

Per quanto riguarda il funzionamento interno dobbiamo ritornare alla descrizione del comportamento di **addTripButtonPressed()** [Figura quaranta], esso si può riassumere in tre punti:

- Attraverso la classe UserDefaults valuterà il valore, contenuto all'interno del DB condiviso, relativo alla chiave PositionManager.kCurrentTrip. In base a questo capirà se l'utente aveva già iniziato un viaggio manuale precedentemente oppure no.
- Usando una variabile, chiamata addTripActionSheet di tipo UIAlertController, si occuperà di visualizzare dinamicamente il menù che si vede nello *screen-shot* centrale di Figura quarantuno.
  L'aggiunta delle singole righe del menù a tendina avverrà *on the fly*[31], anche, in base alle scelte dell'utente ma a livello implementativo il lavoro sarà fatto dal metodo **addAction()**, invocato sulla variabile addTripActionSheet.
  In base a cosa verrà cliccato sarà eseguito del codice atto a memorizzare opportunamente l'informazione.
- Appena l'utente compirà la sua decisione verrà eseguito del codice *ad hoc*, agendo sia sulla classe PositionManager che sulla classe HomeVC.
  La prima servirà per geo-localizzare l'utente e associare al tipo di movimento le coordinate geografiche in cui l'utente si trova in quel preciso istante.
  La seconda servirà per modificare la grafica della home una volta che le operazioni di

[31] Letteralmente dall'inglese "al volo". In gergo informatico può essere utilizzato per esprimere qualcosa modificata dinamicamente.

geo-localizzazione e salvataggio su CoreData saranno ultimate.
Qui sono state affrontate due criticità:

- La prima inerente alla sequenzialità delle operazioni. Ossia era di fondamentale importanza che nell'arco di alcuni microsecondi venissero fatte una lunga serie di operazioni: memorizzare la scelta dell'utente, inviarla alla classe che si occupa della localizzazione, interagire con il database CoreData per la memorizzazione, se possibile inviare i dati al server (quindi una eventuale interazione con la rete), e infine avvisare del completamento. La risoluzione di questa criticità è per la maggior parte affrontata e superata dentro la libreria PositionManager, per questo motivo verrà trattata separatamente nell'omonimo paragrafo.
- La seconda inerente la segnalazione alla classe HomeVC dell'avvenuta conclusione di tutti questi *steps*. Questo è fondamentale perché bisogna modificare la grafica e dare un riscontro visuale all'utente che la sua interazione è andata a buon fine oppure no. In iOS siccome viene utilizzato l'MVC[32] ogni ViewController è responsabile di modificare la grafica della sua scena sulla storyboard.
  Nello specifico l'avviso viene notificato grazie all'utilizzo del NotificationCenter e tramite l'*Observer pattern* applicato nella comunicazione tra il PositionManager e HomeVC.

Per rendere più chiaro il concetto di *Observer pattern* che viene adoperato nella comunicazione tra LocationManager e HomeVC dobbiamo partire dalla libreria PositionManager. Li dentro c'è una variabile statica di nome kCurrentTripNotificationKey, essa viene usata come base per costruire la relativa notifica nel NotificationCenter. Questa notifica verrà lanciata con il comando **NotificationCenter.post(…)** nel momento in cui quella sequenza di operazioni, indicata nella prima criticità, si conclude.
Dentro HomeVC nel metodo **viewWillAppear()** [Figura trentasette] viene impostato con il metodo **NotificationCenter.addObserver(…)** l'apposito *observer* collegato alla notifica kCurrentTripNotificationKey.
L'*observer* in questione è il metodo **currentTripChanged()** [Figura quarantadue] che scatterà immediatamente nel momento in cui la **post()** lancerà la notifica PositionManager.kCurrentTripNotificationKey.

```
237     @objc private func currentTripChanged(_ notification: NSNotification) {
238         DispatchQueue.main.async {
239             self.setConfigurationForTrip()
240         }
241     }
```

*Figure 42 - HomeVC **currentTripChanged()***

Dentro **currentTripChanged()** verrà richiesta in cascata l'esecuzione del metodo **setConfigurationForTrip()** [Figura quarantatré]. Da notare però che viene esplicitamente richiesta la sua esecuzione sul *main-thread*, per i motivi legati alla grafica spiegati precedentemente.

[32] MVC sta per Model View Controller che è il pattern di programmazione per eccellenza che viene usato in Xcode nello sviluppo di applicazione iOS.

```
202     private func setConfigurationForTrip() {
203         if(UserDefaults.standard.integer(forKey: PositionManager.kCurrentTrip) == -1) {
204             currentTravelView.isHidden = true
205             newTripContainer.backgroundColor = UIColor.clear
206             newTripText.text = NSLocalizedString("New_trip_btn", comment: "")
207         } else {    // There is a current trip
208             currentTravelView.isHidden = false
209             newTripContainer.backgroundColor = UIColor.white
210 
211             let trip = Travel.SOMotionType(rawValue: Int32(UserDefaults.standard.integer(forKey: PositionManager.kCurrentTrip))
212             currentTravelLabel.text = Travel.getConfigurationForMotionType(trip!).0
213             currentTravelImg.image = Travel.getConfigurationForMotionType(trip!).1
214 
215             addTravelImg.isHidden = true
216             newTripText.text = NSLocalizedString("New_trip_btn_transport", comment: "")
217         }
218     }
```

*Figure 43 - HomeVC **setConfigurationForTrip()***

Quest'ultimo metodo agirà direttamente sulla *user interface* dell'utente segnalando, sopra il bottone verde, il tipo di movimento che l'utente ha scelto. Inoltre modificherà la *label* del tasto di aggiunta che non sarà più "Start a new trip" ma "Change transport" perché una volta che l'utente ha deciso di collaborare attivamente, sarà lui a specificare nei suoi prossimi spostamenti quando cambierà mezzo di trasporto oppure se il viaggio è terminato.
Il risultato finale, a livello grafico, sarà qualcosa di simile a ciò che si vede nell'ultima *screen-shot* in Figura quarantuno, li è rappresentata la situazione in cui lo *user* ha precedentemente scelto la modalità di spostamento manuale "trasporto pubblico".

4.3 Position Manager

Essa rappresenta una delle librerie più importanti dell'applicazione perché si occupa principalmente della localizzazione dell'utente e allo stesso tempo sfrutta anche tante altre librerie fondamentali nella realizzazione degli obiettivi di questo progetto.
Interagisce con la parte relativa ai sensori di movimento per catturare la tipologia di spostamento dell'utente, usufruisce di CoreData per memorizzare *records* nel *database*, comunica con la rete esterna attraverso il NetworkManager, e infine con la libreria apposita si occupa sia della creazione che della coordinazione dei *geofence*.

Non ha un ViewController di riferimento, bensì è utilizzata da più ViewController dentro l'intera applicazione.
Siccome il gps è unico all'interno del device non aveva senso gestirlo in più classi, dunque come anche suggerito dalle documentazioni Apple si è deciso di farlo in una singola classe e rendere l'accesso condiviso a tutte le altre.
Nello specifico è stato usato il *Singleton pattern*[33], attraverso l'utilizzo delle proprietà statiche[34] in Swift.
Questo approccio permette di rendere la singola istanza visibile a tutte le altre. Ecco l'implementazione dentro la classe PositionManager:

[33] Uno dei pattern fondamentali in informatica. Esso viene utilizzato quando si vuole assicurare la creazione di una sola istanza di una determinata classe, per poi renderne l'accesso condiviso fra le altre classi del progetto.
[34] In Swift dentro le classi o le struct esistono tante forme di proprietà (nel senso informatico del termine). Ad esempio ci sono quelle di tipo *computed*, o addirittura quelle che fungono come una sorta di *observer* (attivate attraverso la *keyword* "didSet"). In questo caso però si fa riferimento alla *static property*.

```swift
import UIKit
import CoreLocation
import CoreMotion

class PositionManager: NSObject, CLLocationManagerDelegate {

    static let shared = PositionManager()
```

Figure 44 - PositionManager: static property

In Figura quarantaquattro è dove l'oggetto condiviso di PositionManager() viene creato e assegnato alla variabile statica shared. Quando viene creato il PositionManager() viene richiamato il costruttore che è stato debitamente sovrascritto:

```swift
private override init() {
    super.init()
    NotificationCenter.default.addObserver(self, selector: #selector(customWillTerminate),
                                           name: NSNotification.Name.UIApplicationWillTerminate, object: nil)

    // Activation of shaking_detection via accelerometer
    if (self.motionManager.isDeviceMotionAvailable && !self.motionManager.isDeviceMotionActive) {
        self.motionManager.deviceMotionUpdateInterval = 0.01
        self.motionManager.startDeviceMotionUpdates(to: self.queueShaking, withHandler: {  [unowned self]
            (deviceMotionData, error) in
            self.detectShaking(acceleration: deviceMotionData!.userAcceleration)
        })
    }
    // Activation location_manager setup
    self.locationManager = CLLocationManager()
    self.locationManager.delegate = self
    self.locationManager.allowsBackgroundLocationUpdates = true
    self.locationManager.pausesLocationUpdatesAutomatically = false
}
```

*Figure 45 - PositionManager **init()***

Qui è possibile vedere come prima di tutto viene settato un *observer* tramite il NotificationCenter ma stavolta di un evento particolare di sistema, l'evento fa riferimento alla notifica di nome NSNotification.Name.UIApplicationWillTerminate. Questo serve per dire al programma di reagire nel momento in cui l'utente decide di proposito di chiudere definitivamente l'applicazione, in quel caso verrà richiamata la funzione **customWillTerminate()** che si occuperà di stoppare il monitoraggio della posizione ma allo stesso tempo di inserire il record di chiusura nel database (CoreData) in modo da non avere informazioni inconsistenti. Questo è possibile farlo perché solitamente l'applicazione prima di chiudersi ha un ammontare di tempo di circa 5 secondi.

```swift
@objc func customWillTerminate () {
    // This method is executed before the app is gonna be killed in o
        applicationWillTerminate present in AppDelegate.swift is exec
    serialQueue4_Location_Geofence_Database_Management.sync {
        self.stopMonitoringUser()
        self.insertStartOrStopInDB(startOrStop: false, completion: {
            print("Value For Indicate -Service Stopped Before Killing
        })
    }
}
```

*Figure 46 - PositionManager **customWillTerminate()***

Successivamente nell'**init()** viene valutato se l'hardware ha a disposizione l'accelerometro, magnetometro, e giroscopio. Ossia i sensori che servono per intuire il tipo di movimento dell'utente. Qualora è presente l'accelerometro viene fatto partire su una coda a parte, chiamata queueShaking di tipo OperationQueue, una porzione di codice che si occupa di ricevere gli aggiornamenti dall'accelerometro con una frequenza pari a 0.1 secondi. Ogni

decimo di secondo dunque attraverso il metodo **detectShaking(…)** viene rilevato se l'utente sta scuotendo o meno il dispositivo.

```
@objc
private func detectShaking (acceleration:CMAcceleration) {
    self.shakingLock.lock()
    //Array for collecting acceleration for last one seconds period.
    // currentFiringTimeInterval is a counter for calculating completion of one second interval
    self.currentFiringTimeInterval += 0.01;
    if (self.currentFiringTimeInterval < 1.0) {// if one second time intervall not completed yet
        // Add current acceleration to array
        shakeDataForOneSec.add(acceleration)
    } else {
        // Now, when one second was elapsed, calculate shake count in this interval.
        // If there will be at least one shake then
        // We'll determine it as shaked in all this one second interval.
        var shakeCount:Int = 0;
        for element in shakeDataForOneSec {
            let boxedAcceleration:CMAcceleration = element as! CMAcceleration
            /**********************************
             *       Detecting shaking
             **********************************/
            let accX_2:Double = Double(powf(Float(boxedAcceleration.x),2))
            let accY_2:Double = Double(powf(Float(boxedAcceleration.y),2))
            let accZ_2:Double = Double(powf(Float(boxedAcceleration.z),2))
            let vectorSum:Double = sqrt(accX_2 + accY_2 + accZ_2);
            if (vectorSum >= kMinimumRunningAcceleration) {
                shakeCount = shakeCount+1;
            }
            /**********************************/
        }
        isShaking = shakeCount > 0;
        shakeDataForOneSec.removeAllObjects()
        self.currentFiringTimeInterval = 0.0;
    }
    self.shakingLock.unlock()
}
```

*Figure 47 - PositionManager **detectShaking()***

Il metodo rappresentato nella figura soprastante è stato tradotto dalla libreria SOMotionDetector, quella di cui si era parlato nel capitolo tre. In pratica si è convertito questo metodo dal linguaggio ObjectiveC a Swift, soprattutto per motivi di compatibilità. Qui la strategia di questo metodo è la stessa di quella descritta nel capitolo tre, sostanzialmente grazie al vettore somma delle accelerazioni rileva se il cellulare viene scosso. Questa informazione sarà utile per intuire il movimento qualora il device a bordo non possiede il coprocessore di movimento M7, oppure qualora l'utente non acconsente a dare l'autorizzazione per accedere ai dati catturati dal chip M7 (Ne discuteremo successivamente). Se invece l'autorizzazione ad accedere a questi dati verrà garantita dall'utente quello che accadrà sarà stoppare l'aggiornamento dei dati dell'accelerometro nella coda queueShaking, questo perché ovviamente controllare lo scuotimento, senza un chip specifico che fa il lavoro, significa sovraccaricare di cicli di controllo la CPU principale del dispositivo e questo ovviamente consumerebbe batteria.

Tornando al costruttore del PositionManager di Figura quarantacinque, tra la riga 101 e 104 (incluse) avviene il settaggio principale della libreria che si occupa di gestire la localizzazione. La libreria è la CoreLocation, e infatti se si va a guardare nelle prime righe di PositionManager troviamo l'import relativo.
Inoltre in Figura quarantaquattro, i più attenti avranno notato che viene implementato un

delegato specifico chiamato CLLocationManagerDelegate. Esso è fondamentale affinché vengano implementati, nella classe che adotta questo protocollo, i metodi minimi necessari per ricevere aggiornamenti dai sensori di localizzazione. Siccome in questo caso la gestione delle posizioni catturate la volevamo gestire dentro questa classe, si è deciso di settare come ricevitore degli aggiornamenti la classe stessa. Questa operazione è fatta proprio in riga 102 del costruttore [Figura quarantacinque]. Successivamente abbiamo due settaggi importanti del locationManager sempre nella medesima figura di prima, essi sono:

- `self.locationManager.allowsBackgroundLocationUpdates = true`
- `self.locationManager.pausesLocationUpdatesAutomatically = false`

questi due servono per due motivi:

- Specificare che i sensori del dispositivo devono essere in grado di ricevere aggiornamenti in background, ossia anche quando l'utente mette il cellulare in tasca e l'applicazione non è stata *killed*[35].
- Specificare che qualora il dispositivo rilevi una situazione di stazionarietà non mandi automaticamente il device in modalità pausa, bensì continui a ricevere aggiornamenti delle posizioni. Questa opzione è stata ampiamente studiata e testata in fase di sviluppo, sembrava interessante inizialmente perché prometteva di far risparmiare parecchia batteria secondo la documentazione Apple.
  Il problema però è che quando si settava a true, e dunque si specificava anche un metodo speciale da far scattare al "risveglio" del locationManager. Questo approccio non andava bene perché:
  - Il metodo non scattava in casi in cui avrebbe dovuto. Ossia capitava che l'utente tornava a muoversi ma il ricevitore di posizioni non si risvegliava, perdendo dati preziosi.
  - Quando si entrava nello stato di pausa, un timer ciclico fondamentale nell'algoritmo della cattura delle posizioni smetteva di contare e di essere eseguito in background. Dunque interferiva con uno dei funzionamenti di base del software, e questo non andava bene. Per quanto riguarda l'utilizzo del timer a breve spiegheremo come è stato adoperato.

Proseguendo nella descrizione delle parti principali passiamo direttamente al metodo più importante dell'intera libreria che tra l'altro è anche obbligatorio da implementare una volta che si è deciso di dotare la classe del protocollo CLLocationManagerDelegate.
La signature del metodo è rappresentata nella Figura sottostante:

```
553     func locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {
554         if (isAddingTravelManuallyEventFired) {
555             // Manage the event
556             serialQueue4_AddingTravelManually.async {
557                 self.group4AddingLocationTravelManually.enter()
558                 self.serialQueue4_AddingUserMode_And_UpdateVariable.sync {
559                     if (self.isAddingTravelManuallyEventFired) {
```

*Figure 48 - PositionManager **locationManager(_manager, didUpdateLocations)***

Di base, a seconda anche dei filtri fissati in fase di inizializzazione del locationManager [Figura quarantacinque] questo metodo scatta tutte le volte che il dispositivo rileva determinate nuove coordinate con il sensore gps. Tutte le posizioni rilevate vengono passate dentro il parametro "locations", esso è un *array* costituito da più elementi di tipo CLLocation.

[35] In gergo informatico con *killed* si intende una applicazione che è stata chiusa di proposito e definitivamente dall'utente. Il tutto attraverso il gestore delle app in esecuzione di iOS, attivabile solitamente cliccando due volte sul tasto Home del device.

Qui dentro **locationManager(_manager, didUpdateLocations)** vengono gestite varie criticità e varie logiche che sono la struttura portante dell'algoritmo di acquisizione.

Parliamo dell'array introdotto poco fa. Avrebbe potuto essere molto più semplicemente un parametro singolo, magari con il nome location (anziché locations) e contenente una unica variabile relativa ad una unica posizione rilevata, l'ultima.
Questo avrebbe reso la logica più semplice. Purtroppo però per motivi relativi a come Apple ha deciso di sviluppare la sua libreria, ha pensato bene di restituire una collezione di oggetti con più posizioni rilevate. Il motivo è perché spesso i sensori del device effettuano in sequenza più di una rilevazione simultaneamente, nonostante magari l'evento di richiesta di acquisizione dati venga scatenato una sola volta. Dunque per evitare di far scattare troppo velocemente il metodo **locationManager(_manager, didUpdateLocations)**, internamente il dispositivo raccoglie tutte le catture in un buffer interno e poi decide ad un certo punto di restituirle tutte insieme dentro l'array locations. Starà poi al programmatore selezionare l'ultima, qualora voglia la più recente, perché internamente le acquisizioni sono disposte nell'array in ordine cronologico. A tal riguardo per distinguere quando le singole location sono state effettuate, ognuna di loro all'interno ha un campo *timestamp* di tipo Date.

Per quanto riguarda dove questo metodo **locationManager(…)** viene eseguito, esso è semplicemente chiamato nel *main-thread*. Fin qua sembra tutto lineare, il problema però è stato capire come risolvere una serie di spinosità:

- Gestire le interazioni sincrone come ad esempio il riconoscimento del tipo di attività dell'utente. Bisognava garantire che esso fosse effettuato obbligatoriamente prima di salvare sul *database* l'informazione. In modo da memorizzare in un solo step, nel modello dei dati, tutte le informazioni relative le coordinate con associato il tipo di movimento dell'utente.
- Coordinare le operazioni di scrittura in modo sincrono sul DB CoreData per evitare inconsistenza dei dati. Nelle prove pratiche infatti si è riscontrato che la **locationManager(…)** non si comportava esattamente come indicato nella documentazione Apple, anzi a volte accadeva che poteva azionarsi più volte nel giro di poche frazioni di secondo. Questo comportamento bizzarro si suppone sia stato causato da *glitch random* dovuti alla sensibilità dei sensori interni al device, dunque una problematica intrinseca di come era stato concepito l'hardware.
- Gestire interazioni di tipo asincrono come ad esempio l'invio dei nuovi record sulla rete. Qui bisognava farlo al momento opportuno e su una coda di esecuzione *ad hoc*, in questo modo avremmo garantito durante queste chiamate il funzionamento principale del software senza causare *deadlock* o rallentamenti sulla grafica.

È difficile spiegare come sono state risolte tutte queste cose in una sola digressione, o attraverso l'utilizzo di qualche screenshot. Il motivo è dovuto al fatto che i flussi di esecuzione tendono a intersecarsi e ad azionarsi in modi diversi e imprevedibili, sia per via del funzionamento principe ovvero la modalità standard di background sia per il funzionamento in modalità manuale. E il problema principale è che tutte le acquisizioni passavano obbligatoriamente e unicamente da questo metodo **locationManager(…)**.

Per fare un po' di chiarezza e ordine, iniziamo da una visione ad ampio spettro delle logiche di implementazione adoperate. Uno strumento molto importante è stato il GCD, ovvero il Grand Central Dispatch che è una API che serve a gestire la concorrenza di più thread a basso livello. Esso tra le infinite funzionalità che offre permette anche di coordinare tre principali tipi di code: la Main che è di tipo seriale, le Global che sono di tipo condiviso e dunque viste da tutto il sistema, e le Custom che possono essere create dal programmatore e inoltre possono essere sia di tipo concorrente che seriale.

Dentro GCD esistono varie strutture di sincronizzazione, ma una delle più importanti che è stata usata all'interno di questo progetto è stata il DispatchGroup. Esso permette di raggruppare insieme più *task* appartenenti anche a *code* diverse di esecuzione, che siano esse sincrone o asincrone. Inoltre è possibile aspettare per il completamento di un *task* e decidere al termine di esso di farne eseguire un altro in sequenza.
Questo è stato utilissimo soprattutto nei casi in cui dovevamo sincronizzare due thread separati e pensati con due compiti o target diversi ma che dovevano coordinarsi e collaborare fra di loro senza ricadere in stati di inconsistenza.
Infine sempre dentro GCD esiste anche una struttura chiamata DispatchQueue, essa è stata largamente adoperata, oltre che in PositionManager, anche in altre parti del progetto di tesi. Questa classe permette di creare, usando una determinata sintassi, sia delle code di esecuzione seriali che concorrenti.

Dentro PositionManager sono state usate diverse code di tipo DispatchQueue:

- La serialQueue4_Location_Geofence_Database_Management.
- La serialQueue4_CalculateMotionType_And_UpdateVariables.
- La serialQueue4_UpdatingDatabase.
- La serialQueue4_AddingTravelManually.
- La serialQueue4_AddingUserMode_And_UpdateVariable.

Invece come approccio di sincronizzazione fra le varie code sono stati usati due DispatchGroup:

- Il primo chiamato group4UpdatingLocationWithTimer.
- Il secondo chiamato group4AddingLocationTravelManually.

Tutti questi elementi sono definiti all'inizio della classe PositionManager e sono trattati come variabili di istanza della medesima classe.

```
553     func locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {
554         if (isAddingTravelManuallyEventFired) {
555             // Manage the event
556             serialQueue4_AddingTravelManually.async {
557                 self.group4AddingLocationTravelManually.enter()
558                 self.serialQueue4_AddingUserMode_And_UpdateVariable.sync {
559                     if (self.isAddingTravelManuallyEventFired) { ••• }
608                 }
609                 self.group4AddingLocationTravelManually.wait()
610             }
611         } else {
612             if (self.timer4PickingUpThePosition != nil) {
613                 // If there is a geofence entering and the user decides to add manually a travel we risk
                        of inconsistency.
614                 // That's why we put this condition in order to avoid this problem.
615                 serialQueue4_Location_Geofence_Database_Management.async {
616                     self.group4UpdatingLocationWithTimer.enter()
617                     var isValueRefused4TimingReasons = false
618                     if (self.location != nil) { ••• }
635
636                     if (self.isGeofenceExitEventFired == true) { ••• }
648
649                     let checkLocation: CLLocation! = (locations.last)
650                     if (!isValueRefused4TimingReasons) { ••• }
749                     self.group4UpdatingLocationWithTimer.wait()
750                 }
751             }
752         }
753     }
```

Figure 49 - PositionManager ***locationManager(_manager, didUpdateLocations)*** *[Folded Version]*

Quella rappresentata in Figura quarantanove è una versione compressa della funzione **locationManager(…)**. In pratica non è possibile riportarla tutta in un unico *screen-shot* perché essendo lunga circa 200 righe di codice, con diversi costrutti innestati fra loro, rischierebbe di diventare illeggibile.
Dunque ho sfruttato la funzionalità di *folding* presente in Xcode, che permette di raggruppare alcune porzioni di codice mostrando solo l'inizio di un determinato costrutto con a fianco i tre puntini di sospensione. Di queste parti di codice compresse, verranno scelte a seguire le più significative in modo da poter spiegare i principi di funzionamento e le logiche di sincronizzazione adottate.

La prima scelta avviene nel costrutto *if* di livello gerarchico superiore.
Viene fatto un distinguo tra la fase in cui questo metodo scatta per motivi legati alla modalità automatica in *background*, e la modalità di funzionamento manuale. Di quella manuale ne abbiamo già parlato ampiamente nei paragrafi precedenti, ma a livello di codice praticamente una volta che la notifica scatta viene settata dentro PositionManager una variabile di istanza chiamata isAddingTravelManuallyFired di tipo Bool.
A seconda di quale evento scatena la chiamata di questa funzione verranno dunque eseguiti due flussi di codice differenti. Nonostante ciò, esiste una cosa che accomuna i due flussi e stiamo parlando delle due code: serialQueue4_AddingTravelManually e serialQueue4_Location_Geofence_Database_Management [Figura quarantanove].
Esse sono due code differenti perché ognuna si occuperà di task diversi, ma sono entrambe seriali e soprattutto agiscono in maniera asincrona su thread diversi.
Questa scelta è stata fatta di proposito in modo da disaccoppiare l'esecuzione del metodo **locationManager(…)** fatta sul *main-thread*, ed evitare di far eseguire a quest'ultimo dei task lunghi.

```
556         serialQueue4_AddingTravelManually.async {
557             self.group4AddingLocationTravelManually.enter()
558             self.serialQueue4_AddingUserMode_And_UpdateVariable.sync {
559                 if (self.isAddingTravelManuallyEventFired) {
560                     let retrievedLocation = locations.last!
561                     // Reset the locationManager to the power saving configuration as in the beginning
562                     self.setLocationManagerToPowerSavingConfiguration()
563                     // If the monitoring service ,before the addiction of a new trip , was not activated by the user we need to ac
                            location with the detected activity we need to start the locationService.
564                     if (!self.isUpdatingLocation) {
565                         // We check this variable because if the service was previously stopped we have to call programmatically t
                                start inside the DB ) and "startMonitoringUser" with the parameter isAfterManualTripAdditionWithServic
                                background monitoring service but with out firing the reading of a new location ).In this particular c
                                one added manually by the user immediately when this two operations will be completed
566                         self.serialQueue4_Location_Geofence_Database_Management.sync {
567                             self.insertStartOrStopInDB(startOrStop: true, completion: {
568                                 self.startMonitoringUser(isAfterGeofenceExit: false,
569                                                          isAfterManualTripAdditionWithServicePreviouslyStopped: true)
570                                 NotificationCenter.default.post(name: NSNotification.Name(
571                                                                     rawValue: PositionManager.kSwitchStatusChangedNotificationKey),
572                                                                     object: nil, userInfo:[PositionManager.kSwitchStatusChangedValueKe
573                             })
574                         }
575                     }
576                     if (retrievedLocation.horizontalAccuracy > 0 && retrievedLocation.horizontalAccuracy < self.horizontalAccuracy
608                 } else {
609                     self.group4AddingLocationTravelManually.leave()
610                 }
611             }
612             self.group4AddingLocationTravelManually.wait()
613         }
```

*Figure 50 - PositionManager **locationManager(...)**, primo flusso di esecuzione*

In Figura cinquanta è rappresentato il primo flusso di esecuzione, ovvero quello eseguito sulla coda serialQueue4_Location_Geofence_Database_Management.
Tramite il group4AddingLocationTravelManually di tipo DispatchGroup viene effettuata la sincronizzazione. Nel dettaglio vediamo che viene chiamata su di esso la funzione **enter()**, in

questo modo notifichiamo al gruppo che il *task* è iniziato. All'interno di questa coda asincrona però facciamo partire una chiamata innestata ad un'altra coda, la serialQueue4_AddingUserMode_And_UpdateVariable e stavolta in modalità sincrona. Il motivo di questa scelta è dovuto al fatto che vogliamo assicurarci che due operazioni avvengano in modo sincrono. Le operazioni a cui ci riferiamo sono l'aggiornamento della variabile relativa all'ultima posizione rilevata e l'aggiunta della userMode nel record nel database. La sincronia è fondamentale che sia garantita perché potrebbe capitare che lo stesso metodo **locationManager(…)** scatti nello stesso momento per via di altri motivi, come ad esempio quando i sensori di posizione rilevano un nuovo cambiamento in modalità automatica.

A seguire vengono fatte diverse operazioni tra le quali: il settaggio della modalità *power-saving* relativa al rilevamento automatico, l'inserimento nel database di CoreData del record di posizione dell'utente, la chiamata del metodo **startMonitoringUser(…)**, la notifica alla classe HomeVC per l'aggiornamento della grafica, la valutazione della qualità dell'accuratezza della posizione. Approfondiamo alcune di esse in particolare.

Prima di tutto alcune di queste operazioni sono a loro volta eseguite su code diverse, in modo da evitare *race-conditions* in cui il metodo **locationManager(…)** azionandosi per altri motivi possa andare a creare conflitti sul modello dei dati o ancora peggio sulla rigida sequenzialità di certe operazioni che vanno eseguite obbligatoriamente in cascata. Per evitarlo sono stati usati debitamente i metodi di sincronizzazione offerti dal DispatchGroup usando per l'appunto la funzione **enter()** per segnalare l'ingresso di un *task* nel gruppo e la funzione **leave()** per segnalarne l'uscita. Qui è stata apportata la massima cura e attenzione, soprattutto nel seguire tutti i flussi di esecuzione possibili assicurandosi di bilanciare il numero di chiamate alla funzione **enter()** con il numero di chiamate alla funzione **leave()**. Qualora il numero di chiamate non fosse stato bilanciato in modo esatto l'applicazione sarebbe andata in *crash*, dunque si può ben capire quanto fosse stata delicata questa fase.

La chiamata al metodo **startMonitoringUser(…)** è molto importante perché esso è polimorfico[36], cioè viene adoperato sia nel caso di modalità automatica sia nel caso di modalità manuale. Poi all'interno fa delle valutazioni e decide in modo appropriato come settare il timer delle prossime acquisizioni, e anche di resettare alcune variabili relative alla valutazione del Geofence.

Per quanto inerente la valutazione dell'accuratezza della posizione, essa è una parte di codice che è nata a seguito di alcuni test effettuati sul campo. In pratica in determinate condizioni come ad esempio quando l'utente era dentro un ascensore, in metro o comunque in posti dove la qualità del gps era molto bassa quello che accadeva era che il metodo **locationManager(…)** veniva richiamato ugualmente e leggendo i valori delle coordinate si è notato che erano completamente sfasati rispetto alla posizione reale del device.
Il motivo era legato alla accuratezza orizzontale troppo bassa delle singole *locations*, bisognava assicurarsi che fossero più precise possibili considerando ovviamente un possibile errore di precisione. Si è deciso dunque di adottare due politiche, una per la modalità automatica e una per la modalità manuale.
Se i sensori rilevavano una *location* con una precisione ad esempio di 150 metri, significava che quelle coordinate erano giuste nell'intorno di una circonferenza di raggio di 150 metri rispetto alla reale posizione del dispositivo.
Qualora il valore di questa precisione era sotto i 150 metri significava che era molto precisa per noi e dunque meritava di essere salvata nel modello dei dati. Altrimenti se questa

[36] Inteso in questo particolare caso nell'accezione italiana del termine e non in quella informatica che invece ha un significato differente.

precisione era maggiore di 150 metri, nella modalità manuale, veniva automaticamente scartata e l'utente avvisato con un messaggio di *alert* in cui si specificava la causa.
Nella modalità automatica invece si è deciso di dare una seconda possibilità ai sensori, valutando che se l'accuratezza orizzontale aveva un valore compreso fra i 150 e 200 metri non era del tutto da scartare. Il motivo poteva ad esempio essere legato a quando ci si sposta in macchina e il device non avendo un ottimo segnale del gps si aiuta con l'aggancio alle celle telefoniche. Li è usuale avere una precisione temporanea di questo tipo, dunque il dato veniva salvato e veniva alzata di 50 metri la soglia di accettazione di valori (ossia da sotto i 150 metri ai 200 metri), almeno fino a quando il device non fosse tornato a ricevere *location* con precisione più alta.

```
615             if (self.timer4PickingUpThePosition != nil) {
616                 // If there is a geofence entering and the user decides to add manually a travel we risk that for the i
                        of inconsistency.
617                 // That's why we put this condition in order to avoid this problem.
618                 serialQueue4_Location_Geofence_Database_Management.async {
619                     self.group4UpdatingLocationWithTimer.enter()
620                     var isValueRefused4TimingReasons = false
621                     if (self.location != nil) {
622                         print("Date old location stored\(self.locationDate)")
623                         // Before was : print("Date last location acquired: \(locations.last!.timestamp)")
624                         let nowDate = Date()
625                         print("Date last location acquired: \(nowDate)")
626                         let secondsSinceLastPoint = nowDate.timeIntervalSince(self.locationDate)
627                         if (secondsSinceLastPoint < (self.timeIntervalForPickingUpThePositionStandard - 5.0)) { ••• }
637                     }
638
639                     if (self.isGeofenceExitEventFired == true) { ••• }
651
652                     let checkLocation: CLLocation! = (locations.last)
653                     if (!isValueRefused4TimingReasons) { ••• }
752                     self.group4UpdatingLocationWithTimer.wait()
753                 }
754             }
```

*Figure 51 - PositionManager **locationManager(...)**, secondo flusso di esecuzione*

In Figura cinquantuno è rappresentato (in modo compresso) il secondo flusso di esecuzione del metodo **locationManager(…)**, questa parte è ancora più sofisticata e lunga della prima perché qui abbiamo a che fare con la modalità di funzionamento automatica in background dove la tipologia di movimento non è fornita dall'utente ma deve essere dedotta dai sensori di movimento.
Come se non bastasse qui viene gestita anche la valutazione e la creazione dei geofence.
Inoltre viene implementata una buona parte della logica di acquisizione automatica attraverso il timer di tipo NSTimer.
Anche in questo secondo flusso vengono usati debitamente sia le code seriali di tipo DispatchQueue che i costrutti di sincronizzazione attraverso i DispatchGroup.

In una prima parte viene gestito il caso in cui ci siano dei falsi positivi. Mi spiego meglio. In pratica in una fase iniziale di *testing* si è visto che la **locationManager(…)**, anche se a dire di Apple scattava con una frequenza ottimale ritornando tutte le ultime posizioni direttamente dentro l'array *locations* in realtà non era esattamente così.
Quando ancora eravamo in una fase prototipale si è visto che il metodo scattava più volte nell'arco di meno di cinque secondi, incrementando senza motivo il numero di *record* nel database con valori di coordinate pressochè identici. Quello che accadeva era dovuto a un difetto intrinseco dell'interazione fra l'hardware e la libreria ufficiale, e bisognava evitare di avere duplicati dentro CoreData. Per farlo si è dunque settato uno *slack* temporale di 5 secondi e tutte le "acquisizioni" che venivano effettuate a distanza di meno di 5 secondi dall'ultima venivano scartate perché considerate come falsi positivi.

Per poter spiegare la parte successiva bisogna prima dare una panoramica di come l'algoritmo automatico di acquisizione è stato pensato e come agisce internamente. Durante lo sviluppo di

questo progetto si è cercato di prendere il comportamento dell'algoritmo di acquisizione automatico in Android e renderlo funzionante in iOS in modo da avere alla fine dati più uniformi possibili.
L'idea sembra abbastanza semplice il problema però a parte le limitazioni di Apple accennate nei capitoli precedenti, è anche il modo di pensare completamente diverso della società di Cupertino. Loro sono abituati a concepire certi meccanismi in modo differente dalla diretta rivale di Google.

In Android per quanto riguarda l'acquisizione automatica il funzionamento lo possiamo riassumere in questo modo:

- Viene lanciato un service in background e attraverso l'uso di un timer ogni tot minuti viene prelevata la posizione dell'utente.
- Dopo che il sensore gps era stato attivato veniva stoppato subito dopo l'acquisizione delle coordinate, e il timer continuava a scattare ogni top minuti senza modificare il suo flusso di esecuzione nonostante l'app rimanesse in background con il gps disattivo quando non doveva acquisire dati.

In questo tipo di applicazioni le due cose che consumano più energia in assoluto sono il sensore gps e quelli di localizzazione. Quindi questo tipo di approccio appena spiegato, e adoperato in Android permetteva un grandissimo risparmio.

In iOS si è provato inizialmente a implementare la stessa logica di funzionamento del software ma purtroppo non era di facile realizzazione a causa di alcune limitazioni imposte da Apple durante lo sviluppo. Andiamo con ordine descrivendo qual è stato il problema che nello specifico è stato incontrato e poi descriveremo come è stato superato.

Prima di tutto nell'architettura del progetto di tesi si è partiti da questi due punti fondamentali:

- Chiedere all'utente l'autorizzazione all'accesso dei dati della sua posizione.
- Abilitare l'acquisizione anche in modalità di background.

Fatto questo si è proceduto con la configurazione della libreria CoreLocation. Essa permette di settare una distanza di soglia, come anticipato nel capitolo tre, e questa diventa il valore del raggio di una circonferenza immaginaria che viene disegnata attorno l'ultimo punto che il gps ha rilevato. Tutta questa circonferenza "immaginaria" copre una determinata area sulla mappa, appena quest'area viene oltrepassata dall'utente il sensore gps accorgendosi del cambiamento fa scattare la rilevazione.
Quando ancora eravamo in fase prototipale si sono fatte delle prove sul campo ed effettivamente la libreria si comportava in modo abbastanza fedele rispetto a quello che diceva nella documentazione ufficiale.
Non dimentichiamoci però che il nostro obiettivo era replicare il comportamento dell'algoritmo in Android e farne il *porting* in iOS nel modo più fedele possibile, per i motivi spiegati precedentemente.
Con CoreLocation di iOS settato in quel modo la posizione era correttamente acquisita ma l'approccio si scontrava con quello di Android. Il problema principale era che i dati rilevati erano fatti in base a una sorta di filtro di distanza e non di tempo.
Nello specifico il dato veniva acquisito perché l'utente si era spostato oltre una certa soglia circolare, e non perchè era trascorso un determinato lasso di tempo.
Qui si sono presentate due problematiche:

- Se la distanza, impostata come filtro dall'ultima posizione rilevata, fosse stata troppo grande l'utente avrebbe potuto muoversi tutto il tempo nell'intorno di quell'area circolare "immaginaria" senza fare scattare mai il rilevamento di una nuova posizione.

- Se il filtro fosse stato troppo piccolo il rilevamento della posizione sarebbe scattato troppo spesso.

Era difficile decidere quale valore impostare come filtro per acquisire dati coerenti con l'impronta di mobilità urbana del cittadino.

Un altro macro-problema che si collega direttamente a quest'ultimo era la realizzazione del geofence. Questo concetto già introdotto nel capitolo due era fin troppo simile, e facilmente confondibile con l'approccio suggerito da Apple descritto poco fa. Era molto importante distinguere i due casi perché l'amministrazione pubblica di Compiégne accedendo dal portale dell'applicazione web voleva comprendere quali erano i luoghi in cui l'utente aveva stazionato e quali erano semplicemente di passaggio.

Si è deciso di cambiare approccio, non seguendo più le linee guida della documentazione ufficiale. Ma cercando di trovare una strada che risolvesse tutte queste criticità.

Si è realizzato un timer ciclico attraverso la classe NSTimer, esso scattava ogni tot minuti e in *foreground* funzionava bene.
Ancora non rilevava nessuna posizione dunque si è deciso di intersecare con esso la libreria CoreLocation chiamando in sequenza due metodi sull'oggetto locationManager:

- Il primo di start che serviva ad attivare il sensore.
- Il secondo di stop una volta che la posizione era stata acquisita.

Quello che si è riscontrato è che l'algoritmo funzionava bene, ma solo in *foreground*. Questo difetto bisognava risolverlo, ma in un primo momento non si capiva da cosa fosse causato. Approfondendo lo studio inerente questa problematica si è scoperto che Apple ancora una volta non ci veniva in contro. In pratica per far funzionare la libreria CoreLocation anche in *background* era necessario seguire tutta una serie di punti:

- Avere l'autorizzazione a rilevare la posizione dell'utente.
- Essere abilitato il sensore a ricevere posizioni in *background*.
- Avere dichiarata la modalità di funzionamento *background* nelle *capabilities* del progetto.
- Essere attivo il sensore di gps mentre l'app è in *background* altrimenti essa veniva automaticamente stoppata nella sua esecuzione dal sistema operativo. Il flusso di esecuzione sarebbe stato ripreso solo nel momento in cui l'app fosse tornata in *foreground*.

I primi tre punti erano stati tutti seguiti alla lettera perché ben chiari nella documentazione, l'ultimo invece è stato scoperto dopo parecchie indagini e test pratici.

Il problema ormai era ben chiaro. Chiamando la funzione di stop sul gps mentre l'app era nello stato di *background* il timer smetteva di scattare.
Si è scoperto che Apple sostanzialmente applica questo tipo di scelta: "Se la modalità *background* che dichiari nelle *capabilities* non la tieni attiva quando l'applicazione è in quello stato allora io non ho motivo di far continuare il flusso di esecuzione del tuo programma". I motivi di questa scelta probabilmente sono legati a:

- Miglioramento delle performance dell'iphone, per dedicare risorse ad altre app.
- Miglioramento della durata della batteria.
- Miglioramento della sicurezza.

Questo comportamento in iOS è totalmente diverso da quello di Android. Diciamo che Android offre più libertà rendendo il lavoro del programmatore più semplice, ma Apple preferisce fare esattamente il contrario, rendere più restrittivo l'utilizzo di alcune librerie con il fine di ottimizzare e di garantire una sicurezza maggiore.

Adesso che la visione globale della problematica è stata fornita, possiamo continuare con la descrizione della risoluzione creata ad hoc per aggirare il problema delle limitazioni imposte da Apple spiegando i punti più salienti nel codice. L'obiettivo ultimo è stato raggiunto ma sempre nel rispetto delle regolamentazioni di Apple. Questo alla fine garantirà sicurezza e allo stesso tempo l'accettazione nell'Apple store dell'applicazione, infatti in fase di pubblicazione non si è avuto nessun problema, i controlli che di routine vengono eseguiti per integrità e validazione del codice sono stati superati con successo.

```
282     private func startMonitoringUser(isAfterGeofenceExit:Bool, isAfterManualTripAdditionWithServicePreviouslyStopped:Bool) {
283         if CLLocationManager.authorizationStatus() == .authorizedAlways {
284             // Anyway if is called after geofencing exit or not we need to put the bestAccuracyConfiguration. The only case w
                    isAfterManualTripAdditionWithServicePreviouslyStopped = true because when is true it's a special case and we
                    position has been already added.
285             if (isAfterManualTripAdditionWithServicePreviouslyStopped) {
286                 setLocationManagerToPowerSavingConfiguration()
287             } else {
288                 setLocationManagerToBestAccuracyConfiguration()
289             }
290             // Reset counters relative to Geofencing
291             locationCounter = 0
292             proximityCounter = 0
293
294             if (!isAfterManualTripAdditionWithServicePreviouslyStopped) {
295                 locationManager.stopUpdatingLocation()
296                 locationManager.startUpdatingLocation()
297             }
298             isUpdatingLocation = true
299
300             DispatchQueue.main.async {
301                 self.timer4PickingUpThePosition = Timer.scheduledTimer(
302                     timeInterval: self.timeIntervalForPickingUpThePositionStandard,
303                     target: self, selector: #selector(self.timedCode), userInfo: nil, repeats: true)
304             }
305         } else if (CLLocationManager.authorizationStatus() == .denied || CLLocationManager.authorizationStatus() == .authoriz
                == .notDetermined ) {
306             NotificationCenter.default.post(name: NSNotification.Name(rawValue: PositionManager.kAuthPositionNotificationKey)
                    CLAuthorizationStatus.notDetermined])
307             self.showAlert(title: NSLocalizedString("Warning", comment: ""), message: NSLocalizedString("authorization_errror
308         }
309     }
```

*Figure 52 - PositionManager: **startMonitoringUser(...)***

In Figura cinquantadue ci troviamo nel metodo **startMonitoringUser(…)** che viene utilizzato per gestire l'inizio del monitoraggio della posizione sia quando è attivata la modalità normale di background sia quando è attivata l'uscita da un geofence. Per quanto riguarda i geofence ne parleremo in seguito.
Nella parte evidenziata viene fatto partire il timer in modo ciclico con un intervallo temporale pari al valore contenuto dentro la variabile timeIntervalForPickingUpThePositionStandard.
Il timer viene azionato di proposito nel *main-thread* altrimenti non funzionerebbe in modo adeguato. Il codice che viene eseguito in modo ciclico è contenuto nel metodo indicato nel campo "selector", ed è chiamato timedCode.

```
172
173     /// This method is called always from the main thread. And b
            or de-activation. To understand better the lifecycle wat
174     @objc func timedCode() {
175         self.setLocationManagerToBestAccuracyConfiguration()
176         self.locationManager.stopUpdatingLocation()
177         self.locationManager.startUpdatingLocation()
178     }
```

*Figure 53 - PositionManager: **timedCode()***

Il codice in Figura cinquantatré è quello che viene chiamato a cicli in base a quando il timer scatta. Quello che notiamo è che vengono fatte tre operazioni fondamentali:

- Viene settata la sensibilità del locationManager al suo valore più alto con accuratezza massima.
- Viene in sequenza chiamato sul locationManager lo stop e subito dopo lo start.

Il primo punto è fondamentale per rendere sensibile il sensore gps a tutti gli spostamenti. Il problema successivo era farlo scattare istantaneamente, per non perdere la acquisizione della posizione. Istintivamente verrebbe di chiamare subito dopo la funzione di start sul locationManager. Il punto è che se avessimo fatto così non avremmo garantito l'esecuzione del flusso di codice che riguarda l'acquisizione. Avremmo solo ottenuto una buona probabilità che il rilevamento della posizione fosse compiuto. Ovviamente a noi non bastava avere una buona probabilità, ma ne volevamo la certezza.
Questo ostacolo è dovuto a un dettaglio "subdolo" contenuto nella libreria ufficiale di CoreLocation.
Analizzando a fondo la documentazione del metodo di start sul locationManager si scopre che esso una volta chiamato ritorna immediatamente al thread chiamante e che **didUpdateLocations(…)**, ovvero la porzione di codice che si occupa attraverso il delegato di leggere le coordinate del sensore gps, viene attivata solo in base a quando quella famosa circonferenza circolare "immaginaria" viene oltrepassata. In altre parole, anche se avessimo settato il tipo di filtro più sensibile possibile sul sensore gps c'era il rischio che se quest'ultimo non rilevava un minimo di cambiamento non faceva partire l'acquisizione.
A questa situazione ambigua però c'era una scappatoia; si scopre che l'attivazione certa dell'acquisizione, dunque di tutto il flusso dentro il metodo **didUpdateLocations(…)**, era garantita solo in un caso specifico. In pratica se eravamo nella situazione in cui il sensore gps era attivo e nel frattempo veniva chiamata la sequenza di istruzioni:

- locationManager.stopUpdatingLocation();
- locationManager.startUpdatingLocation();

allora questo garantiva con sicurezza lo scatenare dell'evento che avrebbe portato all'acquisizione. Ecco spiegato il motivo della scelta implementativa che sta dietro il **timedCode()** di Figura cinquantatré.

```
655                     if (!isValueRefused4TimingReasons) {
656                         if (checkLocation.horizontalAccuracy > 0 && checkLocation.horizontalAccuracy <
657                             //Normally was < 100 in the previous tests
658                             // Pretty good gps signal then i can store it
659                             // set last vector of locations in the instance var locations
660                             self.locations = locations
661                             // set the accuracy used in this case
662                             self.accuracy = checkLocation.horizontalAccuracy
663
664                             // Reset the locationManager to the power saving configuration as in the b
665                             self.setLocationManagerToPowerSavingConfiguration()
666
667                             // If before the accuracy was incremented for a value refused , now is imp
                                    good data collection
668                             self.horizontalAccuracyRequired = self.horizontalAccuracyDefaultConstant
```

*Figure 54 - PositionManager: frammento di codice **didUpdateLocations(...)***

In Figura cinquantaquattro si vede un'altra porzione di codice del **didUpdateLocations(…)**. In pratica una volta fatti tutta una serie di controlli su: gestione dei falsi positivi, sincronizzazione delle code sequenziali, accuratezza della posizione catturata etc., viene richiamata la funzione **setLocationManagerToPowerSavingConfiguration()**, evidenziata

nella figura soprastante. Questa parte dell'algoritmo è anche di fondamentale importanza perché garantisce la risoluzione di due problematiche di spicco:

- Il sensore gps veniva settato in una modalità che noi chiameremo di power-saving. E questa modalità veniva applicata solo dopo che ci eravamo assicurati di aver catturato un attimo prima la *location*.
- Il dispositivo che misurava le coordinate non andava in stop, quindi il funzionamento dell'algoritmo e del timer era garantito anche in background.

Adottando questa strategia, veniva rispettata la politica di CoreLocation perché adesso mantenendo attivo il gps in background il timer poteva scattare anche in quella modalità. Allo stesso tempo stavamo anche:

- Rispettando l'obiettivo che volevamo raggiungere, replicando il comportamento di acquisizione di Android su iOS.
- Rispettando la politica di CoreLocation secondo la quale per eseguire codice in background bisognava tenere il sensore gps attivo.
- Evitando sprechi di batteria inutili, quando non ci interessava prelevare una nuova posizione perché il sensore non era tecnicamente in stop ma a livello logico era come se lo fosse.

La parte che avevamo anticipato prima e che si interseca nella progettazione della classe PositionManager è la realizzazione dei *geofence*.

```
700                                 }
701             self.evaluateGeofenceActivationAndInsertLocationIntoDB(lct: retrievedLocation,
702                                                                     motionRecognized: definitiveMotionType,
703                                                                     completion: { (userMode,restartTimer) in
704             // This if below it to check if we are are entering into a geofence or exiting from a geofence.
                    same value of userMode i.e. 9 or 10
705             var definitiveMotionRecognized = definitiveMotionType
706             var definitiveUserMode:Int32 = userMode
707             var definitiveUserInteraction:Bool = false
708             if definitiveUserMode == 9 || definitiveUserMode == 10 {
709                 definitiveMotionRecognized = Travel.SOMotionType(rawValue: definitiveUserMode)!
710             }
711             // In this if below we check if the user has started a manual trip. In that case we trust of him
                    homeVC. Then we set the definitiveUserMode and definitiveUserInteraction with respect the us
712             if(UserDefaults.standard.integer(forKey: PositionManager.kCurrentTrip) != -1) && (UserDefaults.s
713                 // It means the user has put some manual motion type then we have to set userInteraction = t
714                 if definitiveUserMode != 9 && definitiveUserMode != 10 {
715                     // It means if the userMode is not related to a GeofenceEnterRegion or to a GeofenceExit
716                     definitiveUserMode = definitiveMotionRecognized.rawValue
717                     definitiveUserInteraction = true
718                 }
719             }
```

*Figure 55 - PositionManager: frammento di codice **didUpdateLocations(...)***

In Figura cinquantacinque, nella parte evidenziata, ci troviamo sempre nella **didUpdateLocations(…)** ma nel flusso numero due, ovvero quello relativo alla gestione della modalità automatica ampiamente discussa prima.

In iOS esista già una funzionalità che fa parte di CoreLocation che si può utilizzare per la realizzazione dei *geofence*, intesi come descritti nel capitolo due e pensati con la stessa logica seguita in Android. Nello specifico la funzione utilizzata è un metodo dentro il locationManager che si chiama startMonitoring ma ha la seguente signature:

```
func startMonitoring(for region: CLRegion)
```

*Figure 56 - Signature: **startMonitoring(for region)***

Il funzionamento è abbastanza lineare, si realizza una CLRegion che rappresenta una area circolare da monitorare. Poi facendo partire la startMonitoring quell'area viene monitorata costantemente dal device. All'interno di essa viene settato il raggio, nel nostro caso abbiamo deciso che 100 metri potesse andare bene, e poi anche una coppia di coordinate. Viene impostato un identificatore univoco per ogni *geofence* per distinguerli all'interno del ciclo di esecuzione dell'app.
Poi la parte più importante del settaggio della CLRegion è la tipologia di evento che dovrà far scattare la relativa porzione di codice che si occuperà della gestione pratica.
Esso può essere di due tipi: .OnEnter oppure .OnExit, nel nostro caso una volta che il *geofence* veniva creato ci interessava monitorare con il tipo .OnExit, ossia far scattare una determinata funzione nel momento in cui l'utente usciva dalla zona di stazionamento durante il suo percorso cittadino.

Il metodo che viene richiamato nella didUpdateLocations(…) della Figura cinquantacinque è **evaluateGeofenceActivationAndInsertLocationIntoDB(…)**, anche qui viene usata la tecnica presente in Swift relativa alle *closure* attivate a seguire di una *completion*. Nella figura sottostante viene rappresentato quest'ultimo metodo:

```
181     private func evaluateGeofenceActivationAndInsertLocationIntoDB (lct:CLLocation,
182                                                                     motionRecognized:Travel.SOMotionType,
183                                                                     completion: @escaping (Int32,Bool) -> Void) {
184         // Default values for userInteraction and userMode with respect the specifications in the API
185         var userMode:Int32 = -1
186         var restartTimer:Bool = true
187 
188         if (isGeofenceExitEventFired == true) {
189             userMode = 10
190             // reset the geofenceExitEvent variable in order to be prepared for the next geofenceExitEvent
191             isGeofenceExitEventFired = false
192         }
193 
194         locationCounter += 1
195         if (locationCounter == 1) {
196             previousLocation = lct
197         } else {
198             // locationCounter > 1
199             // make a comparison between the new one location and the previous one location ancient maximum 3 previo
200             let metersOfDistance = lct.distance(from: previousLocation)
201             if (metersOfDistance <= thresholdGeofenceMeters) {
202                 proximityCounter += 1
203                 if (proximityCounter == thresholdProximityCounter) {
204                     // Add geofence
205                     let clLocationCoordinate2d = CLLocationCoordinate2D(latitude: lct.coordinate.latitude, longitude
206                     addGeotification(coordinate: clLocationCoordinate2d, radius: 100.0, identifier: "geofenceID", no
207                     // Stop the timer for picking up positions because if we entered in a geofence is useless to kee
208                     restartTimer = false
209                     // N.B. In the exit region you will find a proper code in order to react to the event, reset cou
210                     // Set userMode properly in order to save it into the DB as a geofence position
211                     userMode = 9
212                 } else {
213                     // Do not update the previousLocation because it means proximityCounter is < 3 and we have incre
214                     // Therefore we are going to compare the next upcoming locations with 3 times ancient previous c
215                 }
216             } else {
217                 previousLocation = lct
218                 locationCounter = 1
219                 proximityCounter = 0
220             }
221         }
222         completion(userMode,restartTimer)
223     }
```

*Figure 57 - PositionManager: **evaluateGeofenceActivationAndInsertLocationIntoDB(...)***

All'atto della chiamata di questo metodo, ovvero dentro **didUpdateLocations(…)**, gli viene passata l'ultima posizione rilevata e il tipo di movimento dell'utente. In base a queste informazioni il metodo di Figura cinquantasette valuta se è il caso di creare il *geofence* oppure no. Ad alto livello il funzionamento dell'algoritmo è il seguente. Ogni volta che c'è una

acquisizione automatica della *location* viene valutato se la successiva, che chiameremo *location+1*, si trova vicino a quella acquisita nello step precedente. Se è così allora viene incrementato un contatore accessibile all'interno dell'istanza PositionManager. Esso è chiamato proximityCounter, nel momento in cui il proximityCounter raggiunge il valore di soglia del thresholdProximityCounter allora significa che sono passati circa 21 minuti dunque quello viene considerato un *geofence* ossia un punto di arrivo o di partenza del percorso di un utente. Con il metodo **addGeotification(…)** viene poi effettivamente creato e iniziato il monitoraggio all'interno del geofence.
Un vantaggio considerevole è che una volta attivato il monitoraggio di una CLRegion il device resta attivo in background e consumerà pochissima batteria. Il risparmio dell'energia era un altro obiettivo importante da raggiungere. L'utente ovviamente sarà più invogliato ad usare l'app, e a fornire i suoi dati, se il software non intaccherà di molto il suo normale utilizzo giornaliero.
Tornando all'implementazione, alla fine di questo flusso di esecuzione quando verrà attivato il *geofence* bisognerà stoppare il timer perché non sarà più di alcuna utilità prendere nuove posizioni se l'utente non è più in movimento. A livello pratico viene fatto tramite la completion passandogli una variabile booleana con cui il metodo chiamante capirà se dovrà stoppare il timer oppure no.

```
718     self.insertMobilityFootPrintInDB(lc: retrievedLocation,
719                                      userInteraction: definitiveUserInteraction,
720                                      userMode: definitiveUserMode,
721                                      motionRecognized: definitiveMotionRecognized,
722                                      insertManual: false,completion: {
723             DispatchQueue.main.async {
724                 if (restartTimer) {
725                     self.timer4PickingUpThePosition.invalidate()
726                     self.timer4PickingUpThePosition = Timer.scheduledTimer(timeInterval: self.timeIntervalFor
727                                                                            target: self,
728                                                                            selector: #selector(self.timedCode
729                                                                            userInfo: nil, repeats: true)
730                 } else {
731                     // it means we have entered in the userMode = 9 i.e. GeofenceActivated so we don't need t
                           basically we must only invalidate and set nil
732                     self.timer4PickingUpThePosition.invalidate()
733                     self.timer4PickingUpThePosition = nil
734                 }
735             }
736             self.group4UpdatingLocationWithTimer.leave()
737         })
738     })
```

*Figure 58 - PositionManager: closure di **evaluateGeofenceActivationAndInsertLocationIntoDB(…)** contenuta in un frammento di **didUpdateLocations(…)**.*

In Figura cinquantotto è rappresentato una parte della closure di **evaluateGeofenceActivationAndInsertLocationIntoDB(…)** dove in base al parametro restartTimer viene deciso se invalidare il Timer o farlo ripartire. Esso infatti dovrà essere fatto ripartire nel caso in cui il metodo **evaluateGeofenceActivationAndInsertLocationIntoDB(…)** valuterà che l'utente non è entrato ancora in nessun *geofence*. Nell'altra parte del costrutto invece viene gestita l'invalidazione[37] di esso (il timer) quando sarà necessario. Entrambe le azioni del timer, come è possibile vedere sempre da Figura cinquantotto, saranno eseguite nel *main-thread*.

Per quanto riguarda la fase del risveglio, dopo che l'utente torna a muoversi e quindi esce dal *geofence*, è gestita dentro il metodo **handleOnExitEvent(…)** nella figura sottostante:

[37] Invalidazione è il termine tecnico che ha origine dalla funzione richiamata sull'oggetto Timer per richiedere il suo stop.

```
533     private func handleOnExitEvent(forRegion region: CLRegion!) {
534         print("Geofence onExit triggered!")
535         isGeofenceExitEventFired = true
536         // Remove geofence
537         removeGeotification(geotification: geotifications.last!)
538         // Start again monitoring the user in the normal way , i.e. re-activate the timer
539         startMonitoringUser(isAfterGeofenceExit: true, isAfterManualTripAdditionWithServicePreviouslyStopped: false)
540     }
```

*Figure 59 - PositionManager: **handleOnExitEvent(…)***

L'uscita è segnalata, a livello di istanza, attraverso la variabile booleana isGeofenceEventFired che viene settata a true. Successivamente viene richiamato il metodo **startMonitoringUser(…)** che farà ripartire il monitoraggio. Viene dunque fatta scattare la **didUpdateLocations(…)** e fatta ripartire la fase automatica in background. Ad un certo punto viene richiamata la funzione **evaluateGeofenceActivationAndInsertLocationIntoDB()** [Figura cinquantasette]; li dentro se la variabile isGeofenceEventFired è true viene impostato un valore della variabile userMode pari a 10. Questo valore avrà un significato particolare per il server, dove con il 9 si indica il record del geofence in ingresso e con il 10 il record del geofence in uscita. Infine viene chiamata la closure relativa alla completion [Figura cinquantotto] e all'atto della chiamata di quest'ultima viene passato il corrispondente valore di userMode. Dentro questa porzione di codice avviene l'inserimento effettivo del record con data e l'ora di uscita all'interno del database grazie alla funzione **insertMobilityFootPrintInDB(…)**[Figura cinquantotto].

4.4 Motion Recognition

Questo paragrafo tratta di come è stato realizzato il riconoscimento del tipo di movimento dell'utente per spostarsi da un punto A a un punto B della città.
Questa è una funzionalità altrettanto importante quanto il riconoscimento della posizione. Nello specifico come anticipato nei capitoli precedenti si è deciso di fare il *porting* della libreria SOMotionDetector e tradurla da ObjectiveC a Swift.

Per capire come è stato integrato il riconoscimento del movimento bisogna ricollegarsi al PositionManager ovvero il gestore della posizione dell'utente di cui si è discusso vastamente nel paragrafo precedente.
Quello che vorremo ottenere sono dei record completi nel database, ovvero delle tuple con dentro informazioni inerenti sia alle coordinate sia al tipo di attività dell'utente.

Dal punto di vista del funzionamento però siccome agiscono su sensori diversi, su logiche separate, su librerie differenti si è pensato bene di disaccoppiare i due flussi di esecuzione per poi farli incontrare nel punto giusto per la memorizzazione su CoreData.

Prima di tutto si è deciso di usare una coda a parte che si occupasse di questo task in particolare. Il nome della coda in questione è serialQueue4_CalculateMotionType_And_UpdateVariables ed è definita dentro PositionManager.

Questa coda viene richiamata dentro il metodo **didUpdateLocations(…)** nella parte di codice relativa al riconoscimento del tipo di spostamento in modalità automatica [Figura 60].

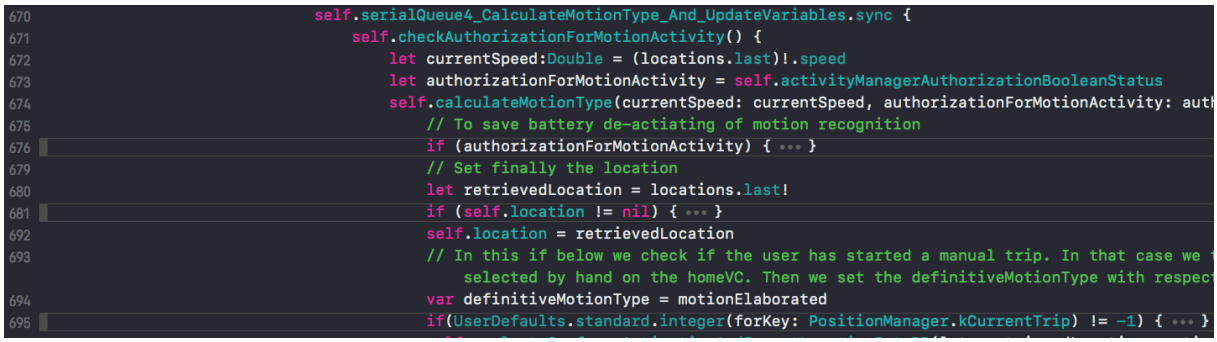


*Figure 60 - PositionManager: **calculateMotionType()** dentro **didUpdateLocations(...)**.*

Prima di tutto dentro la coda viene eseguita la funzione **checkAuthorizationForMotionActivity()**, perché senza le autorizzazioni necessarie non possiamo andare a prelevare i dati che desideriamo dai sensori di movimento.

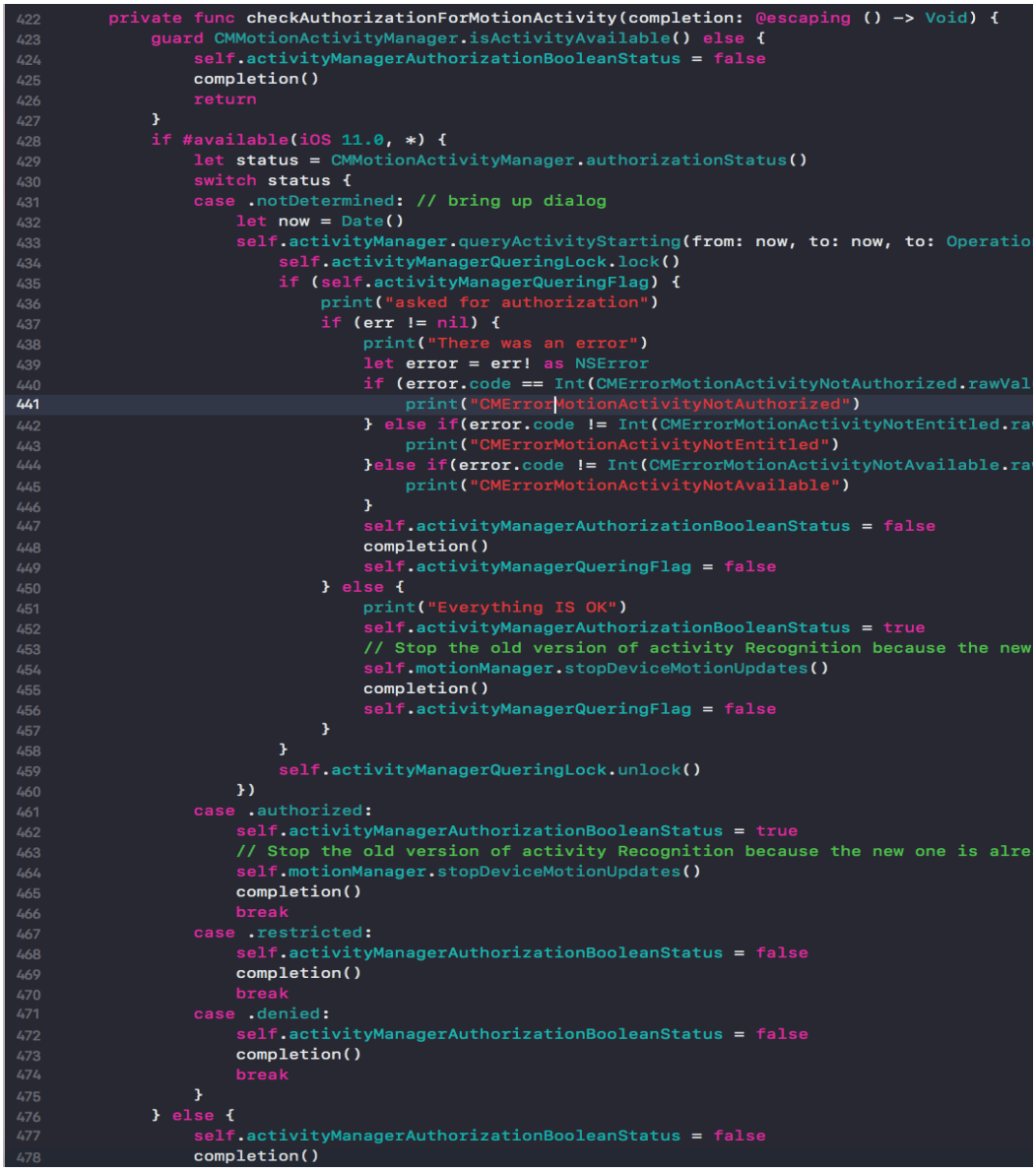
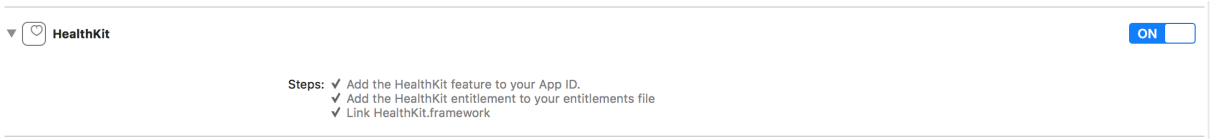


*Figure 61 - PositionManager: **checkAuthorizationForMotionActivity()**.*

Qui dentro [Figura sessantuno] vengono gestiti tutti i casi in cui potevamo ricadere ossia se l'autorizzazione veniva negata oppure se invece c'era un problema di restrizione oppure nel

caso migliore quando veniva accettata.
Ci si è dovuti scontrare nuovamente con le logiche particolari delle librerie di Apple. Per chiedere l'autorizzazione del coprocessore di movimento M7 non esisteva una semplice funzione per fare la richiesta che ritornasse direttamente la scelta dell'utente. A parte l'aggiunta dell'Health Kit dentro le capabilities del progetto [Figura sottostante]:



bisognava sull'activityManager interrogare il *set* di sensori attraverso il metodo **queryActivityStarting(…)** [Figura sessantuno, riga 433].
La richiesta era obbligatoriamente eseguita su una altra coda che doveva essere passata come parametro, la OperationQueue.
Questo metodo in base a come veniva interrogato era in grado di tornare, se ce n'erano, i dati dei sensori di movimento relativi anche agli ultimi giorni.
Questa interrogazione viene eseguita asincronamente, dunque una volta chiamata ritornava subito al metodo chiamante.
Dentro la closure di **queryActivityStarting(…)** come parametro interno c'era il campo "err" che serviva per capire se la richiesta era andata a buon fine oppure c'era stato un errore.
Qualora si fosse rilevato qualche problema o meno viene in ogni caso settata adeguatamente un set di variabili di istanza per segnalare lo stato delle autorizzazioni.

Se andava tutto a buon fine significava che:

- L'utente possedeva un dispositivo con a bordo il coprocessore di movimento.
- Veniva richiesto all'utente se voleva garantire l'accesso al chip M7, per avere una qualità migliore dei dati acquisiti.
- Se l'utente avesse accettato allora veniva stoppato il rilevamento del movimento in modo *legacy*[38], per risparmiare cicli di clock e dunque energia della batteria, e attivato il riconoscimento dell'attività dell'utente attraverso il più moderno coprocessore di movimento. A livello di codice la modalità legacy è gestita grazie all'oggetto motionManager, mentre quella moderna attraverso l'oggetto activityManager.
  Lo stop della modalità legacy nello specifico avviene attraverso la chiamata motionManager.**stopDeviceMotionUpdates()** rappresentato in Figura sessantuno [Riga 454].
- Una volta che l'utente aveva deciso di dare l'accesso al chip M7 avremmo ottenuto dati più precisi e allo stesso tempo avremo sgravato la cpu principale dall'interpolazione dei dati provenienti dai sensori. Adesso c'era un coprocessore apposito che si occupava di farlo.

Se invece ci fosse stato qualche errore esso sarebbe stato gestito opportunamente, utilizzando comunque altre strategie per intuire il movimento dell'utente.
Ad esempio se l'utente non avesse garantito all'applicazione l'accesso al sensore M7 allora in quel caso il riconoscimento *legacy* veniva lasciato attivo.
Dunque il rilevamento shaking (che era avvenuto nella fase di inizializzazione della applicazione) non veniva stoppato, perché quel valore assieme ad altri ricevuti da accelerometro, magnetometro, giroscopio andavano interpolati manualmente usando la

[38] Per *legacy* si intende il sistema precedente ovvero quello in cui il tipo di movimento dell'utente non poteva essere calcolato con l'utilizzo del chip M7 perché i dispositivi precedenti al iphone 5s non ne erano dotati. In questo specifico caso veniva usato un altro approccio.

funzione convertita appositamente in Swift dalla libreria SOMotionDetector.
In questo modo anche qualora l'utente non avesse dato l'autorizzazione in ogni caso avevamo sempre a disposizione un approccio che ci permettesse di intuire il movimento dell'utente. Questo ovviamente dava un valore più approssimativo, ovvero meno preciso di quello ottenuto con l'utilizzo del chip M7, però garantivamo che fosse rispettata la scelta dell'utente senza intaccare il funzionamento principale dell'applicazione.

Andando indietro nello stack delle chiamate ci si ritrova di nuovo in Figura sessanta. Dopo aver verificato il discorso delle autorizzazioni, discusso poco fa, viene finalmente chiamata la closure relativa al metodo **checkAuthorizationForMotionActivity()**, e poi li dentro viene invocato il metodo **calculateMotionType(…)** [Riga 674, Figura sessantuno].

```
374     private func calculateMotionType(currentSpeed: CLLocationSpeed,
375                                      authorizationForMotionActivity: Bool,
376                                      completion: @escaping (Travel.SOMotionType) -> Void)
377     {
378         var motionType:Travel.SOMotionType = Travel.SOMotionType.NotValid
379         if (authorizationForMotionActivity) {
380             activityManager.startActivityUpdates(to: OperationQueue()) { (activity) in
381                 print("SPECIAL RECOGNITION")
382                 if (activity?.stationary == true) {
383                     motionType = Travel.SOMotionType.MotionTypeNotMoving
384                     completion(motionType)
385                 } else if (activity?.walking == true) {
386                     motionType = Travel.SOMotionType.MotionTypeWalking
387                     completion(motionType)
388                 } else if (activity?.running == true) {
389                     motionType = Travel.SOMotionType.MotionTypeRunning
390                     completion(motionType)
391                 } else if (activity?.automotive == true) {
392                     motionType = Travel.SOMotionType.MotionTypeAutomotive
393                     completion(motionType)
394                 } else if (activity?.cycling == true) {
395                     motionType = Travel.SOMotionType.MotionTypeCycling
396                     completion(motionType)
397                 }
398             }
399         } else {
400             print("NORMAL RECOGNITION")
401             if (currentSpeed < kMinimumSpeed) {
402                 motionType = Travel.SOMotionType.MotionTypeNotMoving;
403                 completion(motionType)
404             } else if (currentSpeed <= kMaximumWalkingSpeed) {
405                 motionType = isShaking ? Travel.SOMotionType.MotionTypeRunning : Travel.SOMotionType.MotionTypeWalking
406                 completion(motionType)
407             } else if (currentSpeed <= kMaximumRunningSpeed) {
408                 motionType = isShaking ? Travel.SOMotionType.MotionTypeRunning : Travel.SOMotionType.MotionTypeAutomotive;
409                 completion(motionType)
410             } else {
411                 motionType = Travel.SOMotionType.MotionTypeAutomotive;
412                 completion(motionType)
413             }
414         }
415     }
```

*Figure 62 - PositionManager: **CalculateMotionType()***

All'atto della chiamata di quest'ultimo metodo gli vengono passati come parametri la velocità attuale, ricavata grazie al sensore gps dentro il metodo **locationManager(_manager, didUpdateLocations)**, e poi la variabile di istanza relativa all'autorizzazione per il chip M7 gestita precedentemente nella funzione apposita. Il codice rappresentato in Figura sessantadue è quello che effettivamente si occupa di ritornare il tipo di movimento associato a quell'ultima posizione ottenuta dal gps.
Vediamo che in base alla variabile autoritationForMotionActivity viene fatto un distinguo tra la porzione di codice relativa al riconoscimento con il coprocessore di movimento e quella invece senza.
Quella con chip M7 è abbastanza lineare perché viene letto il valore e poi richiamata la completion.
Quella senza chip M7 invece deve valutare la velocità e la variabile isShaking che viene

adoperata nel riconoscimento del movimento in modalità *legacy*.
Quest'ultima è gestita nel metodo detectShaking che, come anticipato nel capitolo tre, è quella che viene richiamata ogni centesimo di secondo in un thread a parte segnalando attraverso la variabile isShaking se in quel preciso istante il cellulare viene scosso oltre una certa soglia oppure no.
Ritornando alla sequenza di chiamate, una volta che calculateMotionType è stata ultimata si ritorna al flusso di esecuzione iniziale [Figura sessanta]. In questa porzione di codice siamo dentro completion e questo significa che il tipo di movimento è stato già calcolato, quindi per risparmiare energia della batteria, qualora il tipo di attività è stata ricavata con il chip di movimento M7, bisogna stoppare l'aggiornamento degli updates relativi alla continua lettura dei dati interpolati dal chip.
Nello specifico viene fatto nella Riga 676, dove se la condizione è affermativa viene richiamata la funzione activityManager.**stopActivityUpdates()**.
Poi nel prossimo ciclo di esecuzione automatico dell'applicazione in modalità background, ovvero quando ripartirà l'acquisizione di un'altra posizione geografica dentro **locationManager(_manager, didUpdateLocations)** allora li dentro verrà riattivata la ricezione dei dati del coprocessore dentro la funzione **calculateMotionType()** attraverso il metodo activityManager.**startActivityUpdates()** [Figura sessantadue , riga 380].

5. Capitolo 5 – Test energetici

5.1 Algoritmi precedenti

In fase di sviluppo si era implementata una strategia di acquisizione molto buona a livello di qualità dei dati, in pratica si era settato un filtro molto preciso su CoreLocation.
Il filtro in questione fa parte del set di valori che si possono impostare al locationManager affinché esso possa scattare ogni volta che quella determinata soglia viene superata.
Si era impostato il KCLLocationAccuracyNearestTenMeters, ovvero il localizzatore si attivava ogni volta che rilevava una distanza approssimativa di dieci metri rispetto all'ultima posizione rilevata e questo ovviamente garantiva un dato molto preciso e aggiornato, però allo stesso tempo consumava livelli di batteria molto alti.
Parliamo di circa il 40% della batteria, visualizzata su iPhone nelle impostazioni di sistema dopo una giornata intera di utilizzo.
Ovviamente questo approccio anche se prometteva dati molto precisi in realtà non andava bene per lo scopo ultimo di questo progetto perché serviva una applicazione che non andasse a infierire significativamente sui livelli di batteria ovvero doveva essere power-friendly in modo che l'utente fosse invogliato maggiormente a usare l'applicazione e a condividere i suoi dati di mobilità.

Successivamente si era usato un altro algoritmo, ovvero l'approccio in base al quale i dati venivano rilevati a seconda del cambio del tipo di movimento, ogni volta che c'era una transizione da un tipo di attività a un'altra si faceva partire la localizzazione e si memorizzava la tupla nel database. Questo perché si supponeva che nel momento in cui l'utente attivava questo tipo di evento eravamo nella piena fase di spostamento, dunque aveva senso rilevare la posizione e memorizzarla. Il problema è che anche in questo caso come in quello precedente si ottenevano dati di mobilità molto precisi ma con un utilizzo della batteria troppo elevato si parla sempre tra il 30% e il 40% dopo una giornata intera di utilizzo.

Per questi motivi alla fine si è optato per la realizzazione di un algoritmo che mediasse tra qualità dei dati e anche bassi consumi energetici, dunque l'algoritmo scelto è stato quello ampiamente spiegato nel capitolo quattro.

5.2 Algoritmo finale

L'applicazione in questione oltre le funzionalità descritte precedentemente è anche in grado di fornire all'utente:

- Una sezione commenti in cui condividere informazioni importanti con gli altri utilizzatori dell'applicazione. In questa sezione è possibile anche applicare dei filtri per una ricerca più veloce e immediata.
- Una sezione dedicata solo alle statistiche, giornaliere o settimanali. Divise per categoria o per data.
- Una sezione in cui è possibile aggiornare il proprio profilo. E anche quella in cui visualizzare i viaggi manuali inseriti dall'utente durante il suo utilizzo in modalità manuale.

Per quanto riguarda invece la funzionalità principale ovvero quella riguardante la cattura degli spostamenti del cittadino, il software è in grado di funzionare in due modalità:

- Modalità manuale.
- Modalità automatica.

La prima è quella che vede l'utente partecipe nella raccolta dati, quindi è lui che indica direttamente, tramite l'apposita *user-interface* nella home, i suoi cambi di mezzo per spostarsi da un punto all'altro della città.

La seconda è quella invece che è attivata non appena l'utente da il consenso e tutte le autorizzazioni necessarie, e agisce in background memorizzando nel modo più ottimale possibile l'impronta di mobilità del singolo utilizzatore.

Nella fase di testing ci siamo concentrati maggiormente sulla modalità automatica. È fondamentale che essa sia quanto più possibile trasparente sia all'utente che alle statistiche di utilizzo della batteria del cellulare. In questo modo l'utente non dovrà curarsi dell'app aperta in background, e sarà lui stesso più predisposto a tenerla attiva contribuendo alla raccolta dati.

L'algoritmo finale implementato è nato da tutta una serie di prove fatte sul campo e dalla sfida principale di replicare il comportamento Android su iOS, mantenendo su quest'ultimo i consumi bassi nonostante le restrizioni del sistema operativo.
Uno degli obiettivi prefissati era ottenere una percentuale di utilizzo della batteria quanto più piccola possibile. Sotto il 10% sarebbe stato un ottimo risultato, soprattutto paragonandolo ai consumi elevati delle versioni degli algoritmi precedenti discussi prima.

Nelle prove che verranno descritte in questo paragrafo sarà utilizzata direttamente la versione ultima dell'applicazione, ovvero quella pubblicata già sull'Apple store.
È stato anche fatto un confronto in parallelo tra la versione dell'applicazione in Android e la versione in iOS.
In questa fase sono stati utilizzati come dispositivi i seguenti device:

- Samsung Galaxy S5 con a bordo Android 6.01 e una batteria di 2800 mAh.
- Iphone 5SE con a bordo iOS 11.2.2 e una batteria di 1642 mAh.

5.2.1 Confronto qualitativo: Android vs iOS

Stabilire quale delle due versioni dell'applicazione fosse più o meno precisa nella raccolta dati non è stato semplice.
Il problema è che in questo caso non si può fare un banale confronto numerico.
Si ha a che fare con dati geo-localizzati catturati con due algoritmi simili per certi aspetti, ma diversi allo stesso tempo per via della struttura intrinseca differente sia hardware che software.
Inoltre come se non bastasse quando si parla di coordinate geografiche basta che la posizione venga catturata con un sensore gps avente una sensibilità leggermente diversa oppure in un istante di tempo lievemente sfasato, per ritrovarsi con due posizioni distanti magari di qualche centinaio di metri, con valori numerici diversi ma allo stesso tempo corretti.
Un esempio classico è quando si sta facendo un percorso in auto e si attraversa un intero viale e l'applicazione Android per esempio rileva la posizione vicino al marciapiede di destra nell'istante in cui l'utente è all'altezza del civico 10 e invece l'applicazione iOS rileva la posizione sul marciapiede di sinistra ma nell'istante in cui l'utente è all'altezza del civico 30.
Entrambe le rilevazioni sono corrette, ma in termini numerici è difficile stabilire quale delle due sia più giusta dell'altra.
Per questi motivi in questo paragrafo si è deciso di fare un confronto qualitativo grafico.

Entrambi i cellulari sono stati trasportati in tasca in fase di testing, e sono state appuntate le strade e i mezzi di trasporto utilizzati durante gli spostamenti.
Alla fine a termine della giornata si sono aperte entrambe le app, che fino a quel momento avevano lavorato in background, e poi guardando la mappa con i tracciati dello spostamento sono state tirate le conclusioni analizzando la qualità dei percorsi catturati.

I test sono stati effettuati in un arco di tempo di una ventina di giorni. Sono stati considerati dei *pattern* di utilizzo che potessero rispecchiare al meglio la totalità delle tipologie di utenti. Si è preso di riferimento un lasso di tempo pari a 13 ore circa, e le classi prese in considerazione sono state quattro:

- Utente pressochè statico.
- Utente leggermente dinamico.
- Utente mediamente dinamico.
- Utente dinamico.

La prima categoria potrebbe rappresentare l'utente che sostanzialmente ha l'applicazione attiva, ma quel giorno specifico decide di essere stazionario dalla mattina alla sera.
I motivi potrebbero essere vari, ad esempio perché fa *smart-working* dall'abitazione privata. Qualunque sia il motivo comunque si è rilevato che l'applicazione dopo un tot di tempo che può essere 15 minuti in Android e 21 minuti in iOS rileva il geofence e fa scattare la relativa libreria che si occupa di gestirlo.
Una volta entrato in questo particolare stato, se l'utente non si muoverà per tutta la durata delle 13 ore significa che non verranno più raccolte informazioni significative per lo scopo di questo progetto, quindi eviteremo di fare un confronto grafico ma ci limiteremo a dire che sostanzialmente entrambe le app sono state in grado perfettamente di localizzare propriamente il geofence e memorizzare i dati di contorno a riguardo.

Per quanto riguarda invece le altre tre categorie, esse potrebbero rappresentare un effettivo scenario di utilizzo di un cittadino quindi ci impegneremo ad illustrare i risultati ottenuti nei prossimi sotto paragrafi.

5.2.1.1 Utente leggermente dinamico

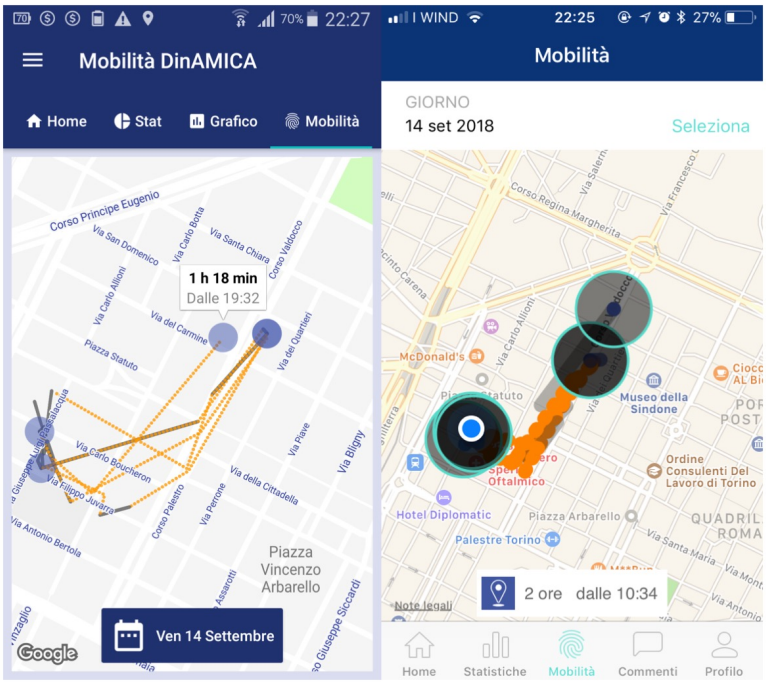


Figure 63 - Scenario di utilizzo 1: Android - iOS

L'utente leggermente dinamico potrebbe essere rappresentativo di un cittadino che è uno studente oppure anche di un lavoratore che ha l'ufficio vicino casa. Prendiamo il caso dello studente, magari ha vicino le aule studio e i suoi spostamenti sono limitati perché passa molto tempo in biblioteca.

Di conseguenza le sue traiettorie di spostamento sono poche e saranno limitate a connettere pochi geofence ma ciascuno con una lunga durata associata.
Nello scenario rappresentato in figura sessantatré si nota subito che dai test effettuati le posizioni sono coerenti con i percorsi realmente seguiti.
Le linee tratteggiate colorate di arancione sono i percorsi effettuati a piedi. I piccoli cerchi invece rappresentano i *geofence* in cui l'utente ha stazionato, cliccando su di essi apparirà una annotazione in sovraimpressione con indicato il tempo trascorso all'interno di un raggio di 100 metri. A sinistra della figura abbiamo lo *screenshot* del percorso catturato in Android e a destra quello catturato in iOS.
Facendo una analisi comparativa vediamo subito che i *geofence* sono anche coerenti eccetto uno. In pratica si è riscontrato che in Android c'è un geofence che è leggermente traslato rispetto alla reale posizione in cui l'utente si è recato. Quest'ultimo geofence è preso con epicentro sul museo del Polo del 900 quando invece il vero epicentro è quello raffigurato in iOS esattamente nell'intorno di Corso Valdocco 15. La distanza tra le due posizioni non è molta, parliamo di circa 200 metri. Però in generale su iPhone questo *geofence* è catturato con più precisione.
Possiamo dire che le traiettorie a piedi sono abbastanza simili in ambedue le catture. Se si aumenta il livello di zoom si noterà che in entrambe i device ci sono alcune linee nere semi trasparenti, quelle servono ad indicare che in alcune piccole tratte il device ha catturato la posizione ma non è sicuro di come l'utente si è realmente spostato, magari perché su due posizioni limitrofe ha trovato due tipi di attività diverse quindi non essendo sicuro del tipo di movimento lo segnala in questo modo.

5.2.1.2 Utente mediamente dinamico

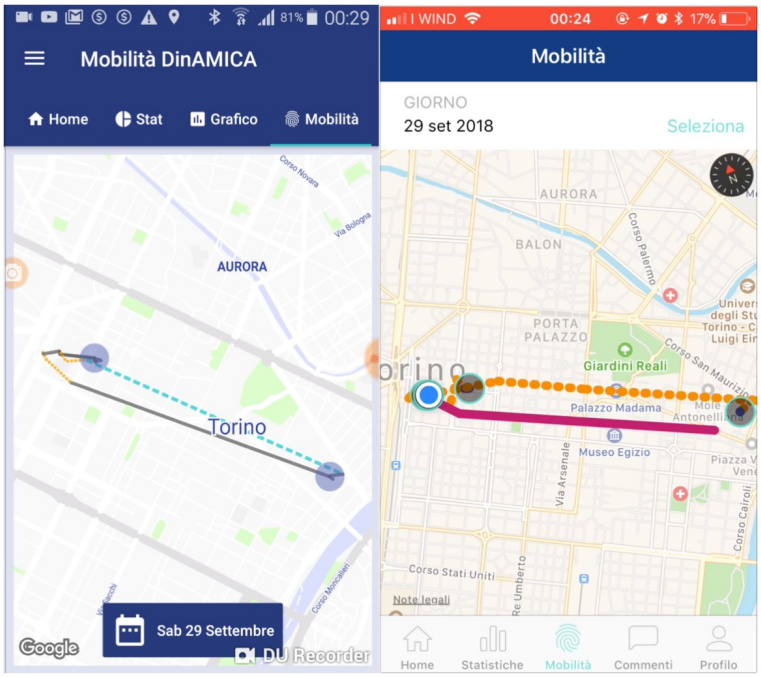


Figure 64 - Scenario utilizzo 2: Android – iOS

L'utente mediamente dinamico potrebbe essere quello che percorre distanze più grandi, per esempio il tragitto casa-lavoro e viceversa, dunque per spostarsi usa mezzi a motore oppure se la distanza non è eccessivamente ampia potrebbe anche utilizzare la bici. In questo caso le catture sono maggiori rispetto al caso precedente, perché il device deve prelevare più posizioni e intuire più movimenti, almeno fin quando non raggiunge il punto stazionario in cui poi ovviamente entrerà automaticamente in un *geofence* dopo un certo lasso di tempo.

Comparando le due schermata si nota subito che i *geofence* sono corretti sia nella prima che nella seconda. Anche le posizioni e le traiettorie seguite sono coerenti.
L'unica differenza che risalta all'occhio è che Android sembra intuire meglio il movimento in bici durante il viaggio di andata. La tratta in bici è infatti rappresentata nello screenshot di sinistra tramite la linea tratteggiata in color ciano, mentre nel device Apple benché riesce a catturare una ottima traiettoria geografica si confonde leggermente indicando il tipo di movimento a piedi anziché in bici. Il motivo per cui fa questo piccolo errore di intuizione potrebbe essere legato ai tempi di campionamento, magari nell'istante in cui è andato a catturare l'attività dell'utente, quest'ultimo era stazionario al semaforo o magari aveva poggiato il piede a terra per non investire i pedoni.
D'altro canto iOS però si riscatta nel percorso di ritorno, infatti in questo caso l'utente ha preso un mezzo motorizzato per tornare indietro, e iOS stavolta è il dispositivo che riesce a intuire meglio il tipo di movimento, rappresentato con linea continua rossa sullo screenshot di destra. Magari qui Android si è semplicemente confuso fra due tipi di attività diverse e quindi ha preferito indicare un generico nero trasparente che al solito sta a significare "non sono sicuro di come ti sei spostato dal punto A al punto B".

5.2.1.3 Utente dinamico

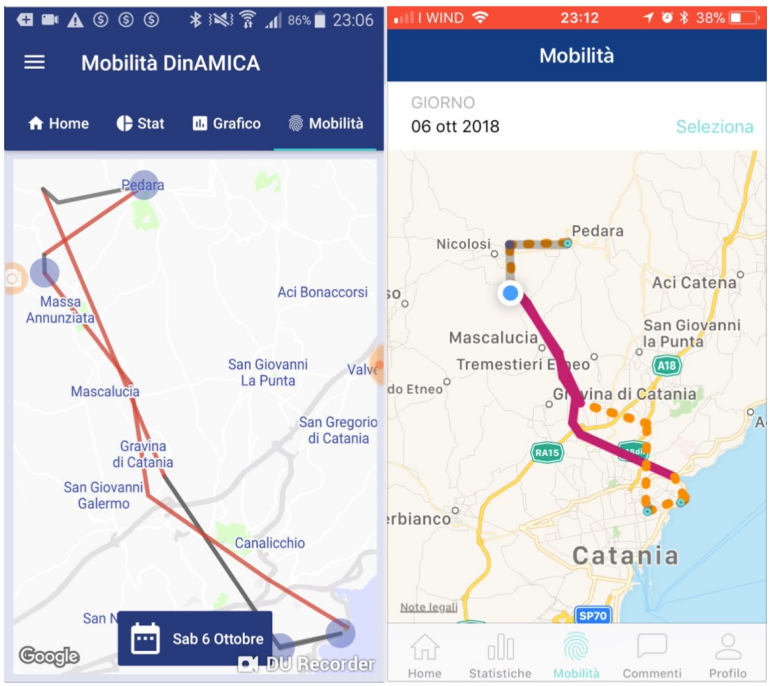


Figure 65 - Scenario di utilizzo 3: Android – iOS

L'utente dinamico potrebbe essere colui che percorre delle tratte ancora più ampie del caso precedente, e che poi staziona in *geofence* differenti e lontani fra loro.
Un esempio potrebbe essere un turista che magari ha affittato la macchina e si sposta in luoghi diversi della città, però poi vi staziona qualche ora per visitarli in loco.
Per farsi una idea della differenza delle distanze percorse fra questa classe di utente e quello precedente basta guardare le diverse scale di dettaglio che ci sono tra la mappa rappresentata in Figura sessantaquattro e quella in Figura sessantacinque.
Nell'ultima sono evidenziati i nomi dei comuni, mentre nella sessantaquattro si riconoscono distintamente i nomi dei quartieri all'interno dello stesso comune.

Le posizioni sono assolutamente coerenti con i luoghi visitati e anche i *geofence*. Vediamo che entrambe le applicazioni hanno rilevato il movimento principale, ossia quello in auto

indicato attraverso la linea continua di colore rosso.
Poi nelle piccole visite vicino al *geofence*, l'utente si è spostato a piedi e vediamo che iOS riesce a intuirlo adeguatamente rappresentandolo con la ormai ben nota linea tratteggiata arancione, mentre Android sembra avere qualche dubbio in alcune piccole tratte.
Infine vediamo che in Android per alcuni spostamenti ha preferito usare una linea nera quando non era sicuro, invece iOS in quelle stesse tratte ha percepito bene il tipo di movimento dell'utente, che era in auto, e ha deciso di segnalarlo.
D'altra parte anche iOS non è perfetto, ci sono alcuni percorsi in cui l'utente era in macchina ma ha segnalato che l'utente era a piedi, questo potrebbe essere dovuto al fatto che quando il device ha cercato di intuire il tipo di spostamento in una determinata traiettoria l'utente magari era fermo in coda e quindi si spostava molto lentamente o a "singhiozzo".
Entrambe i device presentano alcune piccole linee in nero trasparente, il che è normale perché può succedere dopo una intera giornata di utilizzo che in alcune catture fatte su due posizioni geografiche successive vengano rilevati due tipi di attività diverse.

5.2.2 Confronto numerico: iOS vs Android

In questo sotto paragrafo discutiamo dell'analisi comparativa a livello di consumi energetici. I device sotto esame sono gli stessi a cui abbiamo fatto riferimento nel confronto qualitativo del paragrafo 5.2.1.

Anche qui la fase di testing è durata circa venti giorni, e l'app è stata messa sotto osservazione durante l'arco di una giornata media di utilizzo.
Nello specifico la finestra temporale presa in considerazione è di 13 ore circa in *background* e qualche minuto in *foreground.*

Le stesse impostazioni di monitoraggio in modalità automatica sono state fissate sia su Android che su iOS. In questo modo si è riusciti ad analizzare il comportamento degli algoritmi.
Inoltre questo studio ci ha aiutato a capire se effettivamente il prodotto finale riesce a comportarsi bene, soprattutto in una fase di utilizzo assiduo sotto le mani di un utente dinamico. Infine ci ha permesso di riflettere sui pregi e difetti di entrambe le app.

A seguire abbiamo un istogramma riassuntivo dei risultati ottenuti:

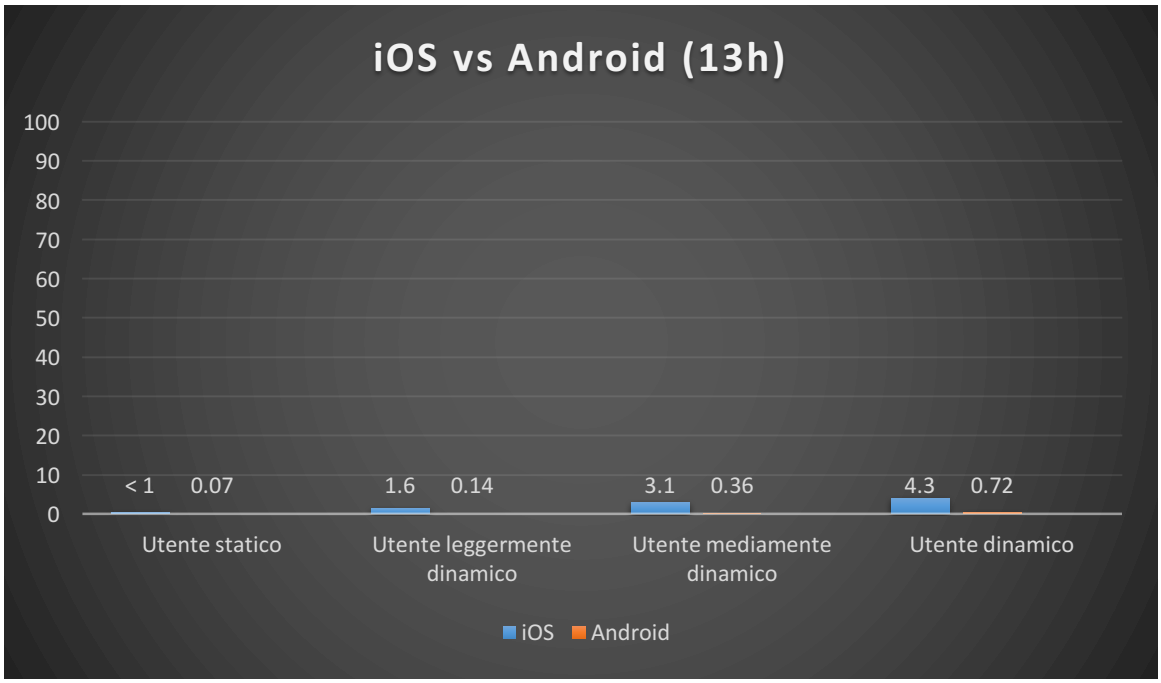


Figure 66 - Istogramma energetico medio: iOS vs Android

In Figura sessantasei abbiamo una rappresentazione grafica dei consumi energetici percentuali rilevati su entrambi i dispositivi a seguito di un utilizzo appartenente alle categorie indicate sull'asse dell'ascisse. Le classi in esame più significative sono le stesse spiegate e considerate nel confronto qualitativo del paragrafo 5.2.1.

Quello che salta all'occhio è sicuramente come nella prima classe il consumo sia pressochè nullo. In questo caso per quanto riguarda iOS, a differenza di Android, non ci è data sapere la percentuale di batteria esatta consumata, perché il sistema operativo permette di avere delle statistiche percentuali, sulla sezione batteria apposita, che sono rappresentate solo a scatti di numeri interi nel campo discreto che vanno appunto da 1 a 100, e qualora il valore sia sotto l'1% non viene indicato nelle statistiche. Il consumo così basso è sicuramente dovuto al fatto che il *geofence*, come anticipato nei capitoli precedenti, è un concetto che una volta implementato e attivato permette al dispositivo di risparmiare tantissima energia. Quest'ultima affermazione è valida per entrambe le piattaforme.

Nella seconda classe di utilizzo, ovvero quella dell'utente "leggermente dinamico" vediamo che in iOS il consumo è sempre molto contenuto. Sicuramente perché essendo state percorse piccole tratte e poi entrando subito all'interno di geofence vicini, quello che accade è che l'app ricade in uno stato simile a quello della prima classe risparmiando dunque parecchia energia.

La terza classe di utilizzo già ha una percentuale di consumo in iOS un po' più significativa. Questo valore inizia ad avvicinarsi a quello della quarta classe ovvero dell'utente "dinamico". La spiegazione di questo comportamento è dovuta al fatto che stavolta l'utente inizia a spostarsi più assiduamente, visitando magari dei luoghi di interesse e stazionandovi per una oretta o più. Inoltre è più probabile che l'utente si sposti in mezzi diversi, magari piccole tratte le percorre a piedi mentre quelle più lunghe con mezzi motorizzati. É dunque normale che l'applicazione rilevando più posizioni e diversi tipi di attività abbia dei consumi più alti.

In generale vediamo che tra Android e iOS il consumo energetico è paragonabile solo nel caso di utente statico. Per il resto sembra che Android riesca a risparmiare meglio sugli utilizzi giornalieri. Il motivo è sicuramente da ricercare alla fonte, e probabilmente è legato anche

intrinsecamente all'architettura. In Android infatti a differenza di iOS è possibile stoppare il servizio di localizzazione pur mantenendo allo stesso tempo l'algoritmo principale attivo. Con "stoppare" si intende proprio disattivare la localizzazione del gps a livello di codice, e questa azione a livello energetico corrisponde a premere il tasto off sull'icona del localizzatore dentro le impostazioni di sistema. Questo tipo di approccio non è stato possibile applicarlo alla lettera su iOS a causa delle innumerevoli restrizioni del sistema, legate soprattutto a quali modalità in background era possibile tenere attive e quali invece erano proibite. Nonostante queste limitazioni imposte da Apple, si è però raggiunto un obiettivo importante se lo compariamo alle prime versioni dell'algoritmo di acquisizione. Quest'ultimo nelle sue forme primordiali consumava percentuali fra il 30% e il 40% della batteria. Valori fin troppo alti per un utilizzo giornaliero in background, quasi paragonabili ai consumi che si potrebbero avere usando applicazioni di navigazione satellitare come Google Maps o similari.
Inoltre come fine ultimo ci eravamo posti di trovare un compromesso che permettesse di abbattere questi valori al di sotto di una soglia giornaliera del 10%. Possiamo dire di aver rispettato abbondantemente questo obiettivo, infatti anche nel caso di utente dinamico, siamo riusciti ad ottenere un consumo di solo il 4.3% a fronte di 13 ore di monitoraggio.

6. Capitolo 6 – Conclusione

6.1 Considerazioni finali e Sviluppi futuri

Possiamo affermare che in questo progetto di tesi gli obiettivi prefissati sono stati tutti ampiamente raggiunti.

Si è riusciti a fare il porting di tutte le funzionalità da Android su iOS pur rispettando le politiche limitative di Apple.

Si è riusciti a mantenere i consumi della batteria molto bassi nonostante le parecchie difficoltà incontrate nel replicare l'algoritmo di cattura da un ambiente più libero e malleabile a un ambiente più *custom* e soggetto a restrizioni.

I risultati dimostrano che entrambe le app sono in grado di prelevare l'impronta di mobilità del cittadino e fornirne uno spaccato sotto forma di mappa di calore oppure di traiettorie percorse differenziando i tipi di spostamento nel contesto urbano.

È possibile interagire con altri utilizzatori dell'app attraverso l'apposita sezione commenti. È anche disponibile una coppia di schermate che mostra le statistiche giornaliere o settimanali, scaricando i dati direttamente dal server a micro-servizi.

In generale possiamo dire che i dati catturati sono molto coerenti in tutte le categorie di utente prese sotto esame. Durante la fase di testing si è riscontrato che su iOS le posizioni geografiche e i *geofence* sono leggermente più precisi rispetto alla versione Android che invece a volte commette delle incertezze di qualche centinaio di metri.

Le attività rilevate durante gli spostamenti dell'utente possono essere di quattro tipi: stazionario, a piedi, in bici, in auto. Sono catturate quasi tutte modo abbastanza uniforme in entrambe le versioni delle applicazioni, eccetto che per quella in bici che sembra leggermente più riconosciuta in Android rispetto iOS. Il motivo di quest'ultimo risultato probabilmente è da attribuire al fatto che i due algoritmi sono simili ma hanno frequenze di campionamento leggermente diverse.

L'applicazione è disponibile sullo store da aprile 2018. In fase di pubblicazione non ha ricevuto nessun richiamo da parte di Apple, ed è stata rilasciata senza alcun problema.

Per quanto riguarda dei possibili sviluppi e/o migliorie future si potrebbe considerare l'idea di introdurre un meccanismo a premi all'interno dell'applicazione. Per fare un esempio si potrebbe stabilire una regola secondo la quale a scaglioni di 100 o 200 ore di monitoraggio l'utente riceve dei *voucher*. Un'altra idea, considerando che ormai la moneta digitale sta prendendo piede ai giorni nostri, potrebbe essere che al posto dei *voucher* l'utente riceva dei gettoni digitali in termini di scambio delle sue informazioni di mobilità. Questi gettoni a loro volta potrebbero essere riutilizzati dall'utente stesso all'interno dell'ambito cittadino, magari per accedere a delle mostre nei musei oppure per pagare servizi quali ad esempio car sharing o bike sharing.

Bibliografia

1. **Parks, Madeline.** The History of Public Transportation. *Gogocharters.* [Online] [Riportato: 08 08 2018.] https://gogocharters.com/blog/history-of-public-bus-transportation/.
2. **Wikipedia.** Bicicletta . *Wikipedia .* [Online] [Riportato: 09 08 2018.] https://it.wikipedia.org/wiki/Bicicletta .
3. —. Storia dell'automobile. *Wikipedia.* [Online] [Riportato: 09 08 2018.] https://it.wikipedia.org/wiki/Storia_dell%27automobile.
4. —. Sustainable transport. *Wikipedia.* [Online] [Riportato: 09 08 2018.] https://en.wikipedia.org/wiki/Sustainable_transport.
5. *Urban spatial structure, daily mobility and energy consumption: a study of 34 European cities.* **Néchet, Florent Le.** 2012, Cybergeo : European Journal of Geography.
6. *Air quality in Europe.* **Agency, European Environment.** 2017.
7. **Pronello, Cristina.** Agenda Digitale. *https://www.agendadigitale.eu/infrastrutture/mobilita-sostenibile-risparmiare-tempo-e-denaro-con-l-aiuto-dei-dati-il-progetto-opticities/.* [Online] [Riportato: 10 08 2018.]
8. **Li, Corrina.** Improving urban mobility starts with better travel data. *Medium.* [Online] [Riportato: 23 08 2018.] https://medium.com/sidewalk-talk/improving-urban-mobility-starts-with-better-travel-data-e2a180c9c6c0.
9. **UITP (International Association of Public Transport).** *URBAN PUBLIC TRANSPORT IN THE 21ST CENTURY* . s.l. : UITP MyLibrary, 2017.
10. **Herrlin, Johan.** Expert View: Mobilising data for the future of urban transport. *SmarTrailWorld.* [Online] [Riportato: 27 08 2018.] https://www.smartrailworld.com/expert-view-mobilising-data-for-the-future-of-urban-transport.
11. **Huston, Tom.** What is Microservices Architecture? *Smartbear.* [Online] [Riportato: 30 08 2018.] https://smartbear.com/learn/api-design/what-are-microservices/.
12. **Eller, Brody.** Background Modes Tutorial. *Raywenderlich.* [Online] https://www.raywenderlich.com/5817-background-modes-tutorial-getting-started.
13. **Apple.** Background Execution. *Documentation Archive.* [Online] [Riportato: 08 09 2018.] https://developer.apple.com/library/archive/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html.
14. **Abhimuralidharan.** Finite-Length Tasks in background iOS swift. *Medium.* [Online] [Riportato: 07 09 2018.] https://medium.com/@abhimuralidharan/finite-length-tasks-in-background-ios-swift-60f2db4fa01b.
15. **Sharma, Piyush.** Downloading in background using Swift 3.0. *Medium.* [Online] [Riportato: 07 09 2018.] https://medium.com/@piyush.dez/downloading-in-background-using-swift-3-0-f770531db619.
16. **Apple.** Downloading Files in the Background. *Documentation Archive.* [Online] [Riportato: 07 09 2018.] https://developer.apple.com/documentation/foundation/url_loading_system/downloading_files_in_the_background.
17. **Hegarty, Prof. Paul.** Developing iOS 9 Apps with Swift . *Lecture 17: Segues, Core Location, and MapKit.* [Video Lecture]. s.l., California, Stati Uniti : Stanford University, 2016.
18. **Tonelli, Marco.** Ecco come funziona Google Maps. *La Stampa Tecnologia.* [Online] [Riportato: 17 09 2018.] http://www.lastampa.it/2018/01/03/tecnologia/ecco-come-funziona-google-maps-CfWX5a50TUDIjgUVu8BFvL/pagina.html.

19. **Google.** Google Maps Android Heatmap Utility. *Google Maps Platform.* [Online] [Riportato: 18 09 2018.] https://developers.google.com/maps/documentation/android-sdk/utility/heatmap.
20. **Apple.** Closures. *Swift.org.* [Online] [Riportato: 27 09 2018.] https://docs.swift.org/swift-book/LanguageGuide/Closures.html#//apple_ref/doc/uid/TP40014097-CH11-ID546.