Indice

1. Introduzione

1.1 Descrizione del progetto

Obiettivo del progetto è realizzare un'applicazione in grado di gestire, attraverso l'uso della tecnologia NFC e dei dispositivi mobili (smartphone e tablet), tutte le informazioni associate ad un'automobile in modo da collezionare uno storico della vettura, anche al fine di facilitarne la compravendita garantendo l'integrità e l'attendibilità dei dati. Questi saranno relativi alla cronologia dei guasti, alle manutenzioni, e ad altre informazioni, sia tecniche, come quelle riguardanti Carta di Circolazione (targa, telaio, cilindrata , ecc..), che geolocalizzate e di navigazione (posizione, percorso del viaggio, velocità e distanza percorsa).

Il telaio della vettura viene univocamente associato all'identificativo alfanumerico (UID) tipico di un tag NFC, che può essere istallato, con un costo irrisorio, dalla casa automobilistica; tale associazione viene salvaguardata dall'istallazione in un punto della vettura non accessibile e che non consenta un'agevole asportazione del tag stesso. L'utente, dopo una prima fase di autenticazione, può accedere da qualunque dispositivo mobile ai dati (memorizzati su un database esterno) della sua automobile, può aggiornare le info e, quando vuole, far partire una navigazione GPS tutto a portata di "tap NFC". Da qui il nome scelto per l'applicazione "Tap on Board".

1.2 Il sistema Android

Introduzione e panoramica generale

Quella a cui stiamo assistendo negli ultimi anni nel campo tecnologico è una vera e propria rivoluzione dal punto di vista dell'elettronica di consumo. La tendenza evidente è quella di miniaturizzare il più possibile i dispositivi elettronici al fine di offrire all'utenza un'esperienza sempre più completa.

In questo contesto uno dei sistemi più rappresentativi di tale evoluzione è senza dubbio Android.

Esso vede le fasi iniziali del suo concepimento nel 2003, anno in cui fu fondata l'omonima società, suscitando l'interesse di chi vedeva nel mercato mobile delle nuove opportunità.

Nel 2005 fu acquisita da Google ed è in questo frangente che nasce il nuovo sistema operativo basato sul kernel Linux.

Inizialmente concepito e ottimizzato per smartphone, con la versione 3.0 Honeycomb, esso tenta di abbracciare anche il segmento tablet, arrivando all'unificazione delle due tipologie di prodotti con la versione successiva, 4.0 Ice Cream Sandwich.

In pochissimi anni il software ha raggiunto una maturità tale da poter essere utilizzato tanto a livello del consumatore, quanto a livello professionale.

Android dal punto di vista tecnico è composto, come accennato precedentemente, alle fondamenta dal sistema operativo Linux, sopra il quale le applicazioni girano principalmente per mezzo della Dalvik virtual machine, una macchina virtuale che consente di massimizzare la compatibilità hardware, agendo come livello di astrazione per il sistema sottostante.

Il linguaggio di programmazione di riferimento è Java.

La scelta tecnica di utilizzare una macchina virtuale per eseguire il codice delle applicazioni Android è controversa, infatti il suo punto di forza è al contempo la sua più grande debolezza. Il suddetto strato di astrazione è la chiave per aiutare il programmatore a preoccuparsi il meno possibile dell'architettura fisica su cui l'applicazione dovrà funzionare, e questo è un requisito importantissimo se si considera la mole di dispositivi su cui Android è installato. Allo stesso tempo però, il medesimo aspetto comporta inevitabilmente un overhead computazionale che non sempre è trascurabile. Negli ultimi anni, la notevole evoluzione dell'hardware su cui gira il sistema operativo sta rendendo questo aspetto sempre più marginale, oltre al fatto che nel corso degli anni sono state effettuate diverse ottimizzazioni lato software. Ne è un esempio l'architettura a registri, con cui è stata da subito progettata la Dalvik Virtual Machine e il cui approccio la differenzia dalla Java Virtual Machine (la quale è caratterizzata da una metodologia basata su stack), che riduce l'utilizzo della memoria di sistema. Altro esempio è l'introduzione dalla versione di Android 2.2, della compilazione JIT (Just In Time), che consente al codice di essere compilato al volo prima di essere eseguito e che rispetto all'emulazione pura offre evidenti vantaggi prestazionali.

Nonostante i vari accorgimenti, le prestazioni possono essere ancora inferiori rispetto alla compilazione fatta a priori, e questo è noto agli sviluppatori del sistema operativo. Nei contesti in cui la necessità di performance diventa critica, esiste la possibilità di sviluppare codice nativo in C/C++, richiamando poi le librerie così create attraverso la Dalvik Virtual Machine.

L'ultima innovazione portata su Android versione 4.4 per incrementare ulteriormente le prestazioni è l'introduzione del runtime ART che sostituisce, a livello sperimentale e con l'esplicita attivazione da parte dell'utente, la Dalvik Virtual Machine. L'approccio della nuova tecnologia passa quindi da compilazione JIT a compilazione AOF(Ahead Of Time). Anziché essere compilato al volo, il codice dell'applicazione verrà compilato al momento dell'installazione, riducendo sensibilmente l'overhead di cui sopra. Le applicazioni richiedono un tempo leggermente superiore rispetto al passato per essere installate, oltre ad un quantitativo di spazio del 10/20% in più rispetto alla vecchia architettura, dal momento che l'app risiede sulla memoria interamente compilata. Di contro ART migliora sensibilmente le performance delle applicazioni, l'efficienza energetica delle stesse e del multitasking del dispositivo. In ogni caso lo stato

attuale dello sviluppo è ancora in fase sperimentale, quindi al momento la Dalvik continua ad essere il sistema di riferimento su cui si basa Android.

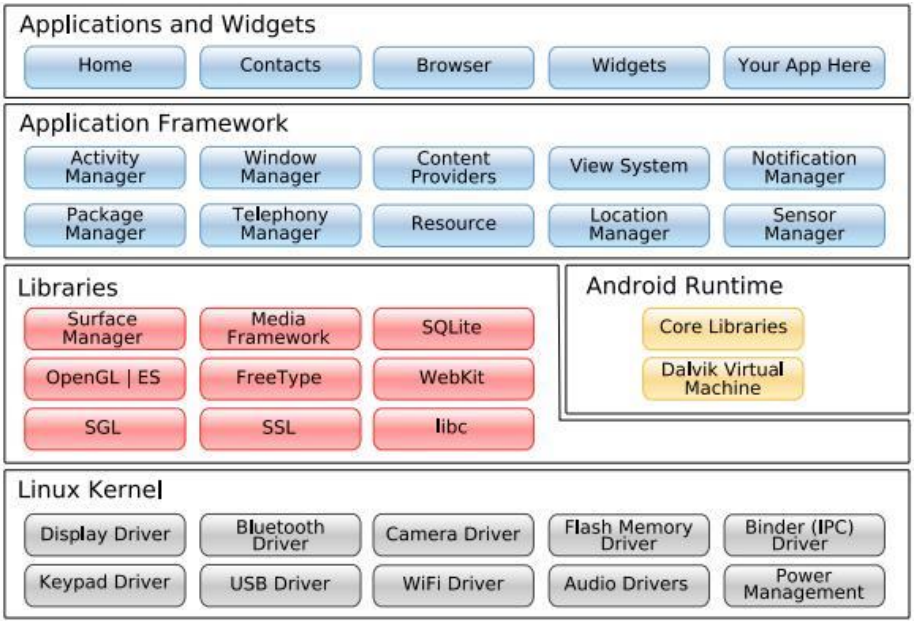
La straordinaria popolarità di Android è in massima parte dovuta ai suoi punti di forza tecnici che lo rendono un sistema al contempo versatile e, nonostante le considerazioni appena fatte, performante su un gran numero di dispositivi anche molto diversi fra di loro.

Chiaramente lo sviluppo di Android procede di pari passo con lo sviluppo del lato hardware sul quale esso dovrà poi essere eseguito. Molte delle opportunità offerte agli sviluppatori di applicazioni mobile sono date più che dalla pura potenza di calcolo, che nei dispositivi in esame è ancora relativamente limitata, dal nuovo approccio all'interazione che essi presentano all'utente. Il più ovvio e immediato elemento è il touch screen che permette allo sviluppatore di superare la limitazione dell'input tradizionale (come mouse e tastiera), di volta in volta plasmando una nuova schermata ottimizzata per il contesto.

Non meno importanti come sostegno all'interazione con l'utente sono gli svariati sensori, che permettono la creazione di nuovi applicativi prima impossibili da realizzare, se non con strumenti appositi, o più semplicemente ne rendono l'utilizzo più facile e soprattutto più intuitivo.

L'architettura di Android

Il diagramma seguente mostra le componenti principali del sistema operativo di Android. Ogni sezione è descritta più dettagliatamente di seguito.



Applicazioni

Android funziona con un set di applicazioni di base, che comprende un email client, un programma SMS, calendario, mappe, browser, contatti e altro. Tutte le applicazioni sono scritte in linguaggio Java.

Framework per applicazioni

Gli sviluppatori hanno pieno accesso alle stesse framework API usate dalle applicazioni di base. L'architettura delle applicazioni è progettata per semplificare il riutilizzo dei componenti; ogni applicazione può rendere pubbliche le sue capacità e tutte le altre applicazioni possono quindi farne uso (sono soggette ai limiti imposti dalla sicurezza del framework). Questo stesso meccanismo consente all'utente di sostituire i componenti standard con versioni personalizzate.

Alla base di ogni applicazione si trova un set di servizi e sistemi, tra cui:

- Un gruppo ricco ed estensibile di Viste che possono essere usate per costruire un'applicazione; contiene liste, caselle di testo, pulsanti, e addirittura un browser web integrato
- Dei Content Providers che permettono alle applicazioni di accedere a dati da altre applicazioni (come i Contatti), o di condividere i propri dati
- Un Manager delle risorse, che offre l'accesso a risorse non-code come strings localizzate, grafica, files di layout
- Un Manager delle notifiche, che permette a tutte le applicazioni di mostrare avvisi personalizzati nella status bar
- Un Manager delle attività, che gestisce il ciclo di vita delle applicazioni e fornisce un backstack di navigazione comune

Librerie

Android comprende un set di librerie C/C++ usate da varie componenti del sistema di Android. Questi elementi sono presentati allo sviluppatore attraverso il framework per applicazioni di Android. Queste sono alcune delle librerie principali:

- System C library - un'implementazione BSD-derived della libreria standard C system (libc), disegnata per dispositivi basati su Linux
- Media Libraries - basate sull'OpenCORE di PacketVideo; le librerie supportano la riproduzione e la registrazione di molti popolari formati audio e video, compresi file di immagini
- Surface Manager - gestisce l'accesso al display subsystem e compone layer grafici 2D e 3D da applicazioni multiple
- LibWebCore - un motore di browser moderno che fa funzionare sia il browser Android sia la visualizzazione web implementata
- SGL - il motore grafico 2D sottostante
- 3D libraries - un'implementazione basata su APIs OpenGL ES 1.0; le librerie usano sia accelerazione hardware 3D (quando disponibile) sia quella inclusa, un rasterizer software 3D altamente ottimizzato
- FreeType - rendering di bitmap e vector font
- SQLite - un motore di database relazionale potente e leggero disponibile per tutte le applicazioni

Runtime

Android comprende un set di librerie centrali che forniscono la maggior parte delle funzionalità disponibili nelle librerie di base del linguaggio di programmazione Java.

Ogni applicazione di Android gira col suo proprio processo, con la sua propria istanza sulla virtual machine Dalvik. Dalvik è stata scritta in modo che un dispositivo possa eseguire VMs multiple in modo efficiente. La VM Dalvik esegue i files nel formato Dalvik Executable (.dex), che è ottimizzato per utilizzare un minimo spazio di memoria. La Vm è register-based, e funziona con classi compilate da un compilatore Java, trasformate in un formato .dex dallo strumento interno "dx".

La VM Dalvik si appoggia sul kernel Linux per funzioni di base come la gestione di threading e situazioni di livelli minimi di memoria.

Kernel Linux

Android si appoggia sulla versione 2.6 di Linux per servizi del sistema centrale come sicurezza, gestione della memoria, esecuzione, network stack, e driver model. Il kernel funziona anche da abstraction layer tra l'hardware e il resto del software.

Anatomia di un'applicazione Android

Ci sono quattro blocchi fondamentali che costituiscono un'applicazione Android:

- Activity
- Intent Receiver
- Service
- Content Provider

Non tutte le applicazioni hanno bisogno di tutti e quattro, ma la tua applicazione sarà sicuramente scritta con una qualche combinazione di questi blocchi.

Una volta deciso quali componenti servono per l'applicazione, bisogna elencarli in un file chiamato AndroidManifest.xml. Si tratta di un file XML in si dichiara quali sono i componenti dell' applicazione e quali sono le loro funzionalità e i requisiti.

- **Activity**

Le Activities sono le più comuni tra i quattro blocchi che costituiscono applicazioni Android. Un'Activity è normalmente una singola schermata nella tua applicazione. Ogni Activity è implementata come una classe singola che estende la classe di base Activity. La classe presenterà un'interfaccia utente composta di Views e risponderà agli eventi. La maggior parte delle applicazioni sono composte da schermate multiple. Ci si può spostare tra le schermate avviando una nuova activity. In alcuni casi un'activity può restituire un valore all'activity precedente.

Quando si apre una nuova schermata, quella precedente si ferma e viene archiviata in un'apposita sezione. L'utente può navigare indietro attraverso le schermate precedenti nell'history stack. Le schermate possono anche essere tolte dall'history stack, quando è inappropriato che rimangano memorizzate. Android mantiene history stacks per ogni applicazione lanciata dallo schermo principale.

- **IntentReceiver**

Android usa una classe speciale chiamata un Intent per spostarsi da schermata a schermata. Un intent descrive cosa un'applicazione vuole che venga eseguito. Le due parti più importanti della struttura di dati intent sono l'azione e i dati su cui agire.

Esiste una classe collegata chiamata un IntentFilter. Mentre un intent è effettivamente una richiesta di fare qualcosa, un intent filter è una descrizione di quanti intent (o quanti intent receiver, sarà più chiaro in seguito) un'activity è capace di gestire. Le Activities pubblicano IntentFilters nel file AndroidManifest.xml.

La navigazione da una schermata all'altra viene eseguita risolvendo intents. Per proseguire nella navigazione, un'activity richiama startActivity(myIntent). Il sistema guarda gli intent filters per tutte le applicazioni e sceglie l'activity il cui intent filters si accorda meglio con myIntent. La nuova activity viene informata dell'intent, causandone il lancio.

Si usa un IntentReceiver per rendere del codice nella tua applicazione eseguibile in reazione ad un evento esterno, per esempio, quando suona il telefono, o è disponibile la rete di dati, o quando scocca la mezzanotte. Le applicazioni possono anche inviare i loro propri intent broadcasts ad altre con Context.broadcastIntent().

- **Service**

Un Service (servizio) è codice di lunga vita che gira senza un'interfaccia utente. In un'applicazione tipo media player, ci sono probabilmente una o più activities che permettono all'uten-

te di scegliere le canzoni e iniziare ad ascoltarle. Tuttavia, la riproduzione della musica non dovrebbe essere gestita da un'activity, perché l'utente si aspetterà di continuare a sentire le canzoni anche quando naviga su un'altra schermata. In questo caso, l'activity media player può avviare un servizio usando Context.startService(), in modo che si esegua in background e la musica continui. Il sistema manterrà il servizio in esecuzione fino a quando non è finito. Ci si può connettere ad un servizio (avviarlo se non è ancora in esecuzione) con il metodo Context.bindService(). Una volta connesso ad un servizio, puoi comunicarci attraverso un'interfaccia esposta dal servizio stesso. Per il servizio della musica, ciò permette di mettere in pausa, di tornare indietro, ecc.

- **Content Provider**

Le applicazioni possono immagazzinare i loro dati in files, in un database SQLite, o in qualsiasi altro meccanismo che abbia senso. Un content provider, tuttavia, è utile che i dati dell'applicazione vengano condivisi con altre applicazioni. Un content provider è una classe che implementa un set standard di metodi che permettono ad altre applicazioni di immagazzinare e recuperare il tipo di dati che è gestito dallo stesso.

Ciclo di vita di un'applicazione Android

Nella maggior parte dei casi, ogni applicazione Android gira sul suo proprio processo Linux. Questo processo è creato per l'applicazione quando deve essere eseguito parte del suo codice, e continuerà ad essere in esecuzione fino a quando non sarà più necessario e il sistema dovrà recuperare la sua memoria per usarla per altre applicazioni. Una caratteristica inusuale ma fondamentale di Android è il fatto che la vita del processo di un'applicazione non è direttamente controllata dall'applicazione stessa. E' invece determinata dal sistema attraverso una combinazione delle parti dell'applicazione che il sistema sa essere in esecuzione, attraverso l'importanza che queste hanno per l'utente e quanta della memoria complessiva è ancora disponibile nel sistema.

E' importante che gli sviluppatori di applicazioni comprendano come differenti componenti delle applicazioni (in particolari Activity, Service, e IntentReceiver) incidono sulla vita del processo dell'applicazione. Non usare correttamente questi componenti può causare la fine di un'applicazione in esecuzione mentre sta svolgendo compiti importanti.

Per determinare quali processi devono essere portati a termine quando il sistema è a corto di memoria, Android li colloca in una "gerarchia di importanza" basata sui componenti in esecuzione e sui loro stati. I tipi di processi sono questi, in ordine di importanza:

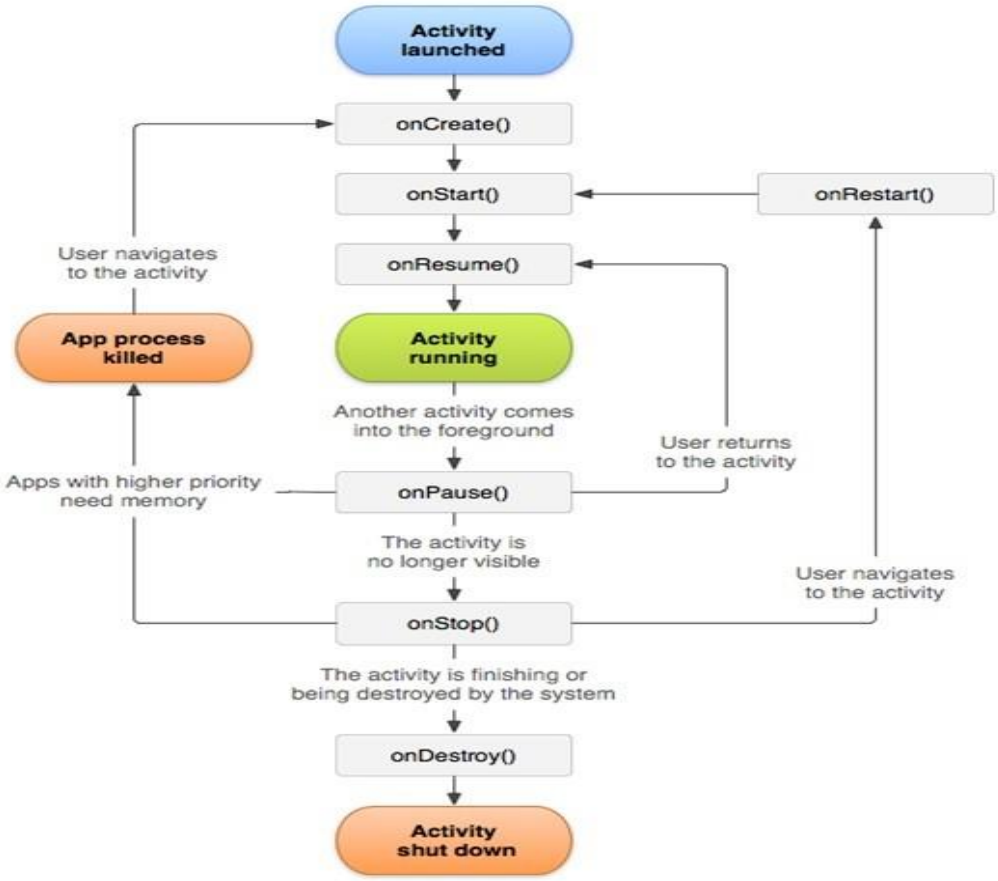
- Un processo di primo piano (foreground process) è un processo indispensabile per l'attività che l'utente sta correntemente svolgendo. Vari componenti di un'applicazione possono portare il processo che li contiene ad essere considerato di primo piano, in modi diversi. Un processo è considerato di primo se sussiste almeno una di queste condizioni:Sta eseguendo una Activity sullo schermo con cui l'utente sta interagendo (il suo metodo onResume() è stato richiamato); ha un IntentReceiver correntemente in esecuzione (il suo metodo IntentReceiver. nReceiveIntent() è in esecuzione); ha un Service che sta correntemente eseguendo codice in una delle sue richiamate (callbacks: Service. onCreate(), Service.onStart(), o Service. onDestroy()). Ci saranno sempre solo un paio di processi con questa massima priorità nel sistema.
- Un processo visibile (visible process) è un processo che gestisce una activity che è visibile dall'utente sullo schermo ma non in primo piano (il suo metodo onPause() è stato richiamato). Un processo tale è considerato molto importante e non verrà portato forzatamente a termine a meno che non sia necessario per mantenere in esecuzione tutti i processi di primo piano.
- Un processo di servizio (service process) è un processo che gestisce un Service che è stato avviato con il metodo startService(). Anche se questi processi non sono direttamente visibili dall'utente, generalmente svolgono compiti che gli interessano (come la riproduzione di mp3 sullo sfondo o upload o download di dati di rete), quindi il sistema manterrà

sempre questi processi in esecuzione almeno che non ci sia abbastanza memoria per mantenere tutti i processi di primo piano e visibili.

- Un processo di sfondo (background process) è un processo che gestisce un'Activity che non è correntemente visibile all'utente (è stato richiamato il suo metodo onStop()). Questi processi non hanno un impatto diretto sull'interazione dell'utente. Quando implementano correntemente il ciclo di vita delle loro activities, il sistema può portarli a termine in ogni momento per recuperare memoria per uno dei tre tipi di processo precedenti. In genere ci sono molti processi di questo tipo in esecuzione, vengono quindi tenuti in una lista LRU per assicurare che il processo che è stato visto dall'utente più recentemente sia portato a termine per ultimo in caso di carenza di memoria.
- Un processo vuoto (empty process) è un processo che non gestisce nessun componente attivo dell'applicazione. L'unica ragione per mantenere un tale processo a disposizione è considerarlo come una cache. Il sistema porterà sovente a termine questi processi per bilanciare le sue risorse complessive.

Quando viene deciso come classificare un processo, il sistema sceglie il livello più importante tra tutti i componenti correntemente attivi nel processo.

Diagramma ciclo di vita di un'Activity:



1.3 Near Field Communication (NFC)

L'NFC è una tecnologia nata dall'evoluzione di RFID (Radio Frequency Identification), insieme a altre tecnologie di connettività senza fili. Al contrario però dei chip RFID, NFC permette una comunicazione bidirezionale, permettendo quindi di mettere realmente in contatto Initiator e Target (ovvero chi esegue la connessione e chi la riceve), quando questi vengono accostati "tapping" a corto raggio (circa 5 cm). In questo modo è possibile scambiare dati tra prodotti eterogenei (purché ovviamente entrambi compatibili con detta tecnologia) addirittura senza

interazione o comunque con una minima interazione con l'utilizzatore, a seconda dello scopo e del contesto. Il primo smartphone a godere di modulo NFC è stato il Nokia 6131, lanciato nel 2006 mentre, su Android, è stato il Samsung Nexus S nel 2010 e da allora si è sempre più diffusa in questa piattaforma al punto di essere presente sulla quasi totalità dei nuovi dispositivi in commercio. NFC opera per mezzo di onde di frequenza 13.56 MHz, il suo transfer rate spazia tra i 106 kbps e i 424 kbps mentre il suo Setup-time è minore di 0.1 s.

NFC può funzionare in tre modi:

- In modalità passiva: non è necessario che entrambi i dispositivi che prendono parte della comunicazione debbano essere alimentati da batterie esterne, infatti solo uno dei due è incaricato di far partire la comunicazione generando un campo elettromagnetico che possa generare potenza su un dispositivo passivo. Questa caratteristica rende possibile e pratico inserire tag NFC nei più svariati contesti, come per esempio all'interno di una carta di credito, oppure come nel nostro caso sotto un cruscotto o in una plancia di una autovettura.

- In modalità attiva: i devices abilitati NFC possono comunicare con altri terminali e punti di contatto (alimentati ma non necessariamente dispositivi mobili) per lo scambio dati. Essi consentono agli utenti di accedere a tutti i dati di un "Tap Point", come per esempio: un tornello di trasporto pubblico o la lettura di informazioni da un cosiddetto "Poster intelligente", che contiene informazioni da condividere (ad esempio l'indirizzo Web della pubblicità, informazioni su una promozione commerciale, ecc..)

- In modalità peer-to-peer (P2P): i dati possono fluire in entrambe le direzioni tra due dispositivi mobili che utilizzano il formato NFC Data Exchange (NDEF). Collegamento di dispositivi in modalità NFC P2P è in netto contrasto con i processi ingombranti di accoppiamento Bluetooth. Infatti lo scambio dati può avvenire in maniera immediata senza la fase di ricerca e accoppiamento tipica della comunicazione Bluetooth.

Questa tecnologia tende, in maniera sempre più accentuata, a semplificare le attività quotidiane, garantendo un'esperienza di utilizzo che le semplifichi portando sempre di più ad interagire con l'ambiente circostante, tramite mezzi elettronici, con un semplice gesto.

1.4 Interfacciamento al database MySQL e il formato JSON

Uno dei metodi di archiviazione dati più utilizzati nell'ambito delle tecnologie informatiche è certamente il database.

Fra i più diffusi protagonisti di questo tipo di tecnologia vi è sicuramente MySQL, il database relazionale sviluppato e distribuito dalla Oracle Corporation.

Esso si compone di un client con interfaccia a riga di comando, e di un server, entrambi compatibili sia con i sistemi Unix-like come GNU/Linux, sia con i sistemi Windows. Esistono, inoltre, diversi tipi di MySQL Manager, ovvero di strumenti per l'amministrazione di MySQL. Uno dei programmi più popolari per amministrare i database MySQL è phpMyAdmin (richiede un server web come Apache HTTP Server ed il supporto del linguaggio PHP). Si può utilizzare facilmente tramite un qualsiasi browser.

L'utilizzo del database MySQL si rivela una scelta particolarmente vantaggiosa quando si vuole implementare il database su di un server esterno al sistema Android, in quanto è facilmente sfruttabile per mezzo di script PHP presenti sul lato server, raggiungibili per mezzo del protocollo di rete HTTP.

Infine, per massimizzare la compatibilità e la leggibilità dei dati scambiati è una buona scelta utilizzare il formato JSON (JavaScript Object Notation) per codificare le informazioni interpellate di volta in volta nelle tabelle del database. Quest'ultimo risulta per le persone facile da leggere, mentre per le macchine risulta facile da generare e analizzarne la sintassi. Si basa su un sottoinsieme del Linguaggio JavaScript ed è un formato di testo completamente indipendente dal linguaggio di programmazione, ma viene usato come supporto da tutti i più famosi linguaggi di programmazione. Questa caratteristica fa di JSON un linguaggio ideale per lo scambio di dati. I tipi di dati supportati da questo formato sono:

- booleani (true e false);
- interi, reali, virgola mobile;
- stringhe racchiuse da doppi apici (");
- array (sequenze ordinate di valori, separati da “ , ” ,racchiusi in parentesi quadre);
- array associativi (sequenze coppie chiave-valore separate da “ , ” racchiuse tra graffe);
- null.

1.5 Geolocalizzazione e le API di Google Maps

Google offre un vasto assortimento di API, in particolar modo API web per gli sviluppatori web. Le API sono basate sui prodotti di Google più popolari tra cui Google Maps, Google Earth, Google Apps e YouTube. Google Maps, in particolare, è un servizio che consente la ricerca e la visualizzazione di mappe geografiche e foto satellitari di molte zone della Terra con diversi gradi di dettaglio. Oltre a questo è possibile ricercare servizi in particolari luoghi, tra cui ristoranti, monumenti, negozi, inoltre si può trovare un possibile percorso stradale tra due punti. Per i dispositivi mobili che supportano la piattaforma Java (Android, iOS, Windows Mobile, Palm OS o Symbian OS) e che dispongono di una connessione a Internet via GPRS, UMTS o HSDPA esiste dal 2006 un'apposita versione Google Maps Mobile, che permette di accedere alle mappe e di usufruire del servizio di navigazione satellitare GPS.

L’ API di Google Maps Directions è un servizio che calcola l'itinerario tra i luoghi utilizzando una richiesta http, passando parametri URL come argomenti al servizio. Una volta processata la richiesta viene restituita una risposta JSON o XML per l'analisi e l’elaborazione da parte dell'applicazione. È possibile cercare le indicazioni per diverse modalità di trasporto (automobile, trasporto pubblico, a piedi ecc..) e specificando tappe intermedie oppure evitando pedaggi, autostrade e traghetti. Le indicazioni restituite possono specificare l’origine, la destinazione e le tappe del percorso sia come stringhe di testo che come coordinate geografiche. Inoltre vengono fornite informazioni riguardanti la durata, la distanza e le indicazioni stradali di ogni tratto del viaggio.

Sfruttando questi strumenti insieme al servizio di localizzazione integrato nel device Android si possono immagazzinare dati riguardanti la posizione dell’utente, registrare i viaggi eseguiti e far partire la navigazione assistita satellitare direttamente dal proprio smartphone.

2 Progetto Car System Stats

2.1 Scelte Progettuali

Le scelte progettuali adottate sono:

- piattaforma di lavoro: Android (un sistema operativo largamente diffuso su terminali mobili);
- linguaggi di programmazione: Java (per il client), Php (per il server), XML (per la grafica) e Sql (per il DBMS);
- tecnologia dei tag NFC: il formato Ntag203 (riconosciuto da quasi tutti i dispositivi), codificati secondo lo standard NDEF;
- database: MySql con motore di memorizzazione InnoDB;

2.2 Requisiti

L'applicazione richiede i seguenti requisiti:

- un dispositivo NFC-compatibile;
- un Tag NFC codificato NDEF;
- la prossimità con il tag NFC in fase di avvio dell'applicazione;
- una versione di Android 4.0 (minSdkVersion = 14), o superiore;
- una connessione internet attiva (non sempre necessaria grazie all'uso delle cache);
- un terminale che abbia la possibilità di essere localizzato attraverso il Network o tramite il servizio GPS integrato;

2.3 Sviluppo

Lo sviluppo dell'applicazione è stato svolto per mezzo degli strumenti messi a disposizione sul sito developers.android.com, e più nello specifico è stato utilizzato l'ambiente di sviluppo Eclipse assieme al plugin Android Development Tools (ADT) su piattaforma Windows. Come supporto per il lavoro e per ragioni di testing e debugging è stato utilizzato un Smartphone Sony Xperia s (4.1.2 JB) in mio possesso.

2.4 Installazione

La scelta precedentemente illustrata di utilizzare gli strumenti appositi per gli sviluppatori semplifica, oltre che la fase di sviluppo, anche quella di installazione. Infatti permette di creare un file TagOnBoard.apk, un contenitore di tutti i file necessari per far funzionare il programma codificati tramite una chiave.

Una volta accertatisi che il dispositivo che verrà utilizzato sia in possesso dei requisiti minimi e che il sistema operativo non blocchi applicazioni esterne non provenienti dal Google Play

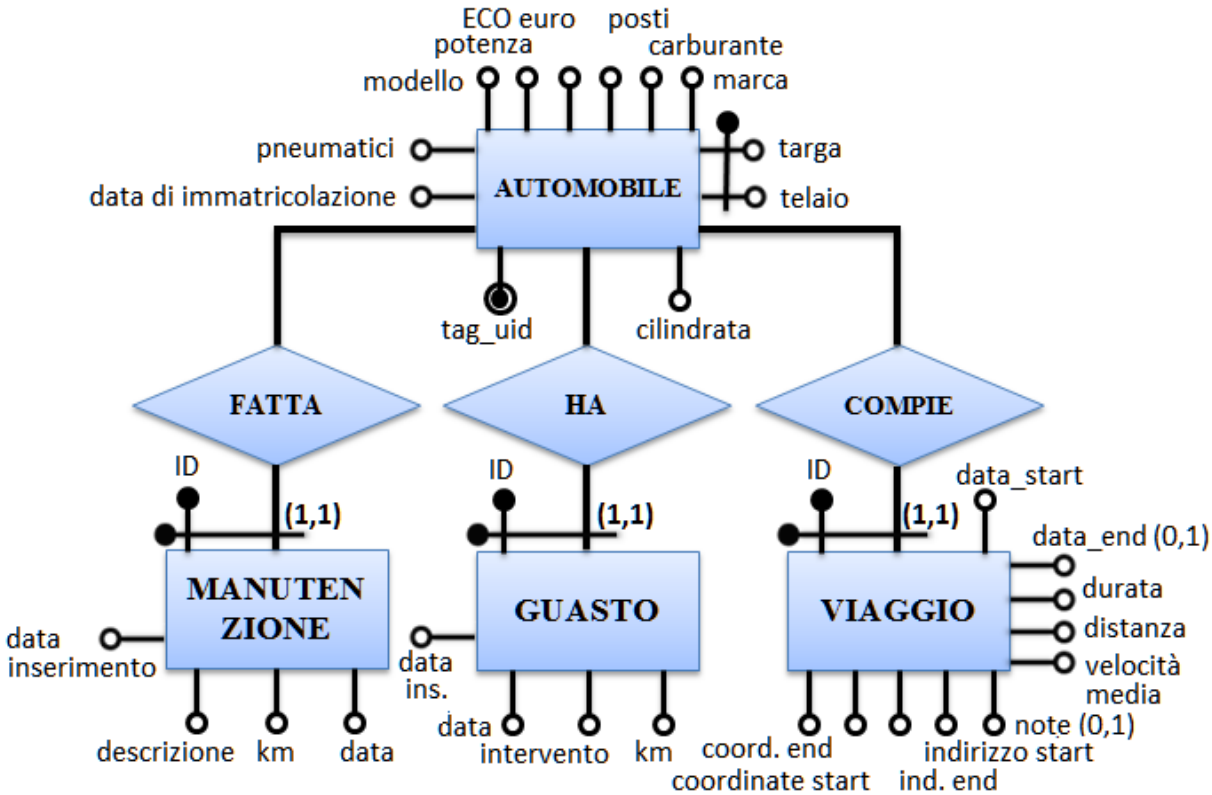
Store, l'installazione si riduce a un semplice click sul relativo file apk e la conseguente applicazione delle istruzioni su schermo.

In un secondo momento l'applicazione può essere direttamente scaricata dallo Store di Google e installata in automatico.

2.5 Progettazione concettuale

Modello Entità e Relazioni

I dati che gestisce l'applicazione sono riconducibili a 4 entità: Automobile, Viaggio, Manutenzione e Guasto. Per ogni auto interessa sapere tutti i guasti che può aver subito, le eventuali manutenzioni effettuate e i possibili viaggi compiuti, inoltre si vuole conoscere la data di immatricolazione, la marca, il modello, la cilindrata, la potenza, il carburante, il numero del motore, il numero dei passeggeri trasportabili, la classe ecologica, i pneumatici che può montare, la targa e il numero di telaio. Di ogni manutenzione e guasto si vuole sapere la data dell'intervento, i km in cui è stato fatto e la sua descrizione accurata. Invece dei viaggi su vuol sapere la data di partenza, l'eventuale data di arrivo, l'indirizzo e le coordinate geografiche di partenza e di arrivo, la durata, la distanza percorsa, la velocità media e se indicate anche le note del viaggio. Tutti i viaggi, i guasti e le manutenzioni sono identificate da un indirizzo unico nell'ambito della vettura a cui appartengono.



velocità media = (distanza(km) / durata(h));

Attivita e sottoprocessi

Avviata l'applicazione, avvicinando il cellulare al tag NFC, viene letto il codice identificativo del tag (univoco). Successivamente, attraverso una richiesta http (usando la classe JSON) al server, si verifica se quel tag è stato registrato e associato ad un veicolo. Il server, ricevuta la richiesta, interroga il database MySQL tramite uno script PHP e risponde: REGISTRATO o NON_REGISTRATO. Nel caso in cui la risposta sia: NON_REGISTRATO, parte una sotto-attività (Autenticazione Activity) che gestisce la registrazione e permette di inserire tutti i dati tecnici della vettura (Telaio, Targa, Anno Immatricolazione, Modello, Marca ecc...), questi ulti-

mi vengono associati al tag Nfc e memorizzati nel Database. Al contrario, nel caso in cui la risposta sia: REGISTRATO (ovverosia il tag è già presente nel DB), si procede con un'altra sotto-attività (menuUtente Activity) che, suddivisa in quattro schermate differenti (Fragment concorrenti) mostra i dati tecnici, la lista dei viaggi compiuti, la lista delle manutenzioni eseguite (tagliandi, cambio ruote, ecc..) e la lista dei guasti dell'autovettura. La prima schermata non è interattiva, permette solo la lettura dei dati che si riferiscono al libretto automobilistico, mentre nelle altre tre schermate, cliccando su un elemento che popola la lista videata (ListView), c'è la possibilità di aprire una finestra di dialogo che mostra in output i dettagli del viaggio, del guasto o della manutenzione e permette, quando possibile (non oltre i due giorni seguenti la data di inserimento) di modificare e cancellare l'elemento selezionato. Quest'opera di modifica viene compiuta da 4 attività chiamate: modificaViaggio, modificaGuasto e modificaManutenzione. Dal menuUtente Activity, premendo un tasto sulla barra superiore oppure usando il tasto fisico del menù contestuale di ogni pagina, si possono aggiungere eventuali viaggi, guasti o manutenzioni azionando i rispettivi processi aggiungiGuasto, aggiungiManutenzione e aggiungiViaggio Activity. Quest' ultima, in particolare, è più complicata delle altre due, infatti oltre ad aggiungere l'elemento alla lista, considerando che si tratta di un viaggio prima di aggiungerlo richiede le informazioni alle API di Google Maps, passandogli come parametri le coordinate geografiche prese dal GPS integrato oppure dal Network. L'applicazione inoltre viene predisposta per la gestione di dati relativi alla geolocalizzazione e per funzionare da navigatore satellitare. In qualsiasi caso di errore dipendente dalla connessione internet o dalla comunicazione con il server, viene avviata una schermata (erroreDiRete Activity) contenente il messaggio di errore e permette di tornare indietro per riprovare l'operazione.

Al Livello Server, ogni richiesta viene processata in maniera concorrenziale da vari script Php, i quali, per mezzo di un interfaccia codici in comune con il client, rispondono in formato Json alle richieste avute. Inoltre, la consistenza dei dai nel Database è data dall'unicità dell' id del tag che caratterizza ogni singolo record delle varie tabelle (Automobili, Guasti, Manutenzioni, Viaggi) presenti nel DB.

L'applicazione fa largamente uso del concetto di multi-threading, gestisce perfettamente la rotazione dello schermo (infatti durante le operazioni più onerose non si interrompe quando si capovolge il device) e fa uso delle cache (ovvero memorizza l'ultimo accesso e una copia delle risposte del server; inoltre, quando opportuno, utilizza tali risposte senza richiederle nuovamente dalla rete internet).

3. Car System Stats – Lato client

3.1 Avvio e riconoscimento Tag Nfc

Queste sono le schermate di avvio dell'applicazione, la prima in modalità schermo verticale l'altra orizzontale.

Durante la fase di avvio dell'applicazione, l'utente dovrà avvicinare il proprio Smartphone al cruscotto in particolare nella zona in cui sarà installato il Tag Nfc, in modo che il sistema possa riconoscere l'automobile.

L'applicazione può essere lanciata in due modi diversi:

- tramite icona;
- automaticamente avvicinandosi al Tag.

L'avvio tramite icona non fa altro che presentare una schermata con un disegno esplicativo e una didascalia che invita l'utente ad avvicinarsi al Tag NFC. Successivamente bisogna inizializzare il ForegroundDispatch per rendere l'Activity corrente l'unica in grado di gestire l' Intent NfcAdapter.ACTION_TAG_DISCOVERED scaturito dal Tag, quando viene accostato al sensore Nfc. L'Intent verrà poi gestito nel metodo onNewIntent().

```
@Override
protected void onCreate(Bundle savedInstanceState) {
. . . .
//inizializzazione ForegroundDispatch. Serve ad impostare questa attivity come quella predefinita per gestire  l'intent
    Intent intent = new Intent(this, getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP);
    pendingIntent = PendingIntent.getActivity(this, 0, intent, 0);
    IntentFilter tagDetected = new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
    tagDetected.addCategory(Intent.CATEGORY_DEFAULT);
    TagFilters = new IntentFilter[] { tagDetected };
  //abilito il ForegroundDispatch per questa Activity secondo il IntentFilter TAG_DISCOVERED
  myNfcAdapter.enableForegroundDispatch(this, pendingIntent, TagFilters, null);
}
```

NOTA: Anche in tutte le altre attività viene inizializzato e abilitato il ForegroundDispatch affinché il nuovo Intent possa essere catturato e ignorato senza che faccia ripartire l'applicazione dall'inizio.

Al fine di realizzare la seconda modalità è stato necessario inserire nel file AndroidManifest.xml la seguente stringa:

```
<intent-filter>
        <action android:name="android.nfc.action.TECH_DISCOVERED" />
        <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
<meta-data
        android:name="android.nfc.action.TECH_DISCOVERED"
        android:resource="@xml/nfc_tech_filter" />
```

che dichiara l'applicazione in grado di gestire l'Intent "android.nfc.action.TECH_DISCOVERED" se il tipo del Tag che lo ha generato è presente nella lista di quelli supportati nel file xml/nfc_tech_filter:

```
<?xml version="1.0" encoding="utf-8"?>
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
   <tech-list>
      <tech>android.nfc.tech.Ndef</tech>
   </tech-list>
   <tech-list>
      <tech>android.nfc.tech.NdefFormatable</tech>
   </tech-list>
</resources>
```

Con questo codice il sistema Android sarà in grado di mandare un Intent all'applicazione che così si auto-avvierà, per poi leggere l'identificativo univoco dello stesso, in modo da identificare l'autovettura.

All'interno della classe principale MainActivity viene effettuato un controllo per garantire che il dispositivo utilizzato disponga di un tag NFC funzionante e che esso sia abilitato. Se l'Nfc è disabilitato nelle impostazioni invita, tramite una finestra Dialog, ad attivarlo, premendo direttamente il bottone presente nella stessa finestra. Nel seguente frammento di codice è possibile vedere come è stata realizzata la funzione appena descritta:

Dentro onCreate() :

```
 myNfcAdapter = NfcAdapter.getDefaultAdapter(this);
if (myNfcAdapter == null) {
        Toast.makeText(this, "Il dispositivo non supporta NFC",
                                                Toast.LENGTH_LONG).show();
        finish();
        return;
}
```

Dentro onResume() :

```
if (myNfcAdapter.isEnabled()) {
        myTextView.setText("Avvicina il cellulare al simbolo NFC sul cruscotto.");
}
else if (!myNfcAdapter.isEnabled()) {
         myTextView.setText("NFC è disabilitato. \nAbilita per procedere");
          settDialog=MyDialogSettingsFragment.newInstance("NFC non attivo","Per procedere bisogna abilitare"
                        + "l'Nfc del dispositivo.", Service.NFC_SERVICE, 0);
        settDialog.show(getFragmentManager(),"setting");
}//se non ci sono messaggi a questo punto, il sensore e presente e funziona
```

Se questi controlli danno entrambi esito positivo, il processo si dichiara pronto a ricevere il nuovo Intent (come detto in precedenza attraverso il ForegroundDispatch) e appena lo riceve viene gestito con il metodo onNewIntent() sottostante:

```
//metodo invocato in automatico quando viene avvicinato il tag al sensore Nfc
@Override
protected void onNewIntent(Intent intent){
    String action = intent.getAction();
     Log.d("dentro handleintent onNewIntent ",action);
     if(NfcAdapter.ACTION_TAG_DISCOVERED.equals(action) ||
                              NfcAdapter.ACTION_TECH_DISCOVERED.equals(action)){
          Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
          String[] techList = tag.getTechList();
           String searchedTech = Ndef.class.getName();
          for (String tech : techList) {
               if (searchedTech.equals(tech)) {  //se il tag è codificato in NDEF
                    Toast.makeText(this, this.getString(R.string.ok_detection),
                                                              Toast.LENGTH_SHORT ).show();
                     showMyDialog();
                     new NdefReaderTask().execute(tag);
                     break;
               }
          }
     }
}
```

NOTA: Anche in tutte le altre attività è presente questo metodo, ma visto che il segnale deve essere ignorato al suo interno è presente solo un Log.d che mostra a video il messaggio di intent ignorato.

Nel codice sopra riportato riguardante l'azione del sensore NFC si richiama il metodo execute(String... arg0) della classe interna privata NdefReaderTask, che estende la classe android.os.AsyncTask. La lista dei parametri arg0 si riduce all'unico parametro necessario all'esecuzione, cioè il tag stesso. La creazione di questo nuovo Thread è necessaria poiché thread principale (UI Thread) si occupa solo della creazione e della gestione dell'interfaccia. La classe NdefReaderTask si occupa di leggere l'identificativo del tag e controllare se l' automobile sia già stata inserita nel database. In caso positivo verrà avviata l'attività MenuUtente, mentre in caso contrario apparirà la schermata di registrazione. L'identificativo del tag viene memorizzato in un array di byte e successivamente convertito in una stringa di cifre esadecimali per facilitare la memorizzazione. Qui di seguito il codice della funzione che trasforma l'array di byte, restituito dal metodo getId() della classe android.nfc.Tag, in formato Stringa esadecimale:

```
private String byteArrayToHexString(byte[] a) {
        StringBuilder sb = new StringBuilder();
        for(byte b: a) sb.append(String.format("%02x", b&0xff));
        return sb.toString();
}
```

Durante queste operazioni, per ovviare ad eventuali lag di rete, viene visualizzato il messaggio "Verifica Tag in corso" per mezzo della classe MyProgresDialogFragment . Come vedremo successivamente ogni finestra di dialogo usata in quest' applicazione estende la classe android.app.DialogFragment ed è dichiarata in modo statico per non risentire della rotazione dello schermo.

3.2 Thread e la classe AsyncTask

Quando una applicazione Android viene eseguita parte il thread principale dell'applicazione che si occupa di disegnare l'Activity iniziale e di gestire gli eventi dell'interfaccia grafica. In

pratica c'è un ciclo infinito (chiamato anche run loop) in cui Android controlla se ci sono nuovi eventi da gestire. Nel caso in cui ci siano nuovi eventi vengono invocati i corrispondenti listener sempre nel thread principale dell'applicazione. Ovviamente durante l'esecuzione di uno di questi listener non possono essere gestiti altri eventi dando la sensazione di applicazione non resposiva. Per evitare questi problemi, i task che possono essere lunghi (download da internet, caricamenti di molti dati, calcoli complessi) devono essere eseguiti in un nuovo thread, in modo che quello principale sia "libero" e possa continuare a gestire gli eventi che arrivano dall'interfaccia. Gli aggiornamenti dell'interfaccia devono però essere eseguiti sempre nel thread principale, se si prova a chiamare un metodo che aggiorna una View da un thread in background viene inviata una eccezione. Per risolvere questo problema si usa metodo runOnUiThread all'interno de thread secondario, l'argomento è un oggetto Runnable che contiene il codice di aggiornamento dell'interfaccia.

```
new Thread(){
        public void run(){
                System.out.println("operazione in background");
                runOnUiThread(new Runnable(){
                        @Override
                        public void run(){
                            System.out.println("Aggiornamento interfaccia");
                        }
                });
        }
}.start();
```

Questo codice non è proprio leggibile e facile da scrivere, ma esiste un modo alternativo che ci mette a disposizione Android: la classe AsyncTask. Questa classe definisce tre generics: AsyncTask<Params, Progress, Result>. Per capire a cosa servono questi parametri vediamo i metodi principali di AsyncTask:

- onPreExecute: eseguito sul thread principale, contiene il codice di inizializzazione dell'interfaccia grafica (per esempio la abilitazione una ProgressBar);
- doInBackground: eseguito in un thread in background si occupa di eseguire il task vero e proprio. Accetta un parametro di tipo Params (il primo generic definito) e ritorna un oggetto di tipo Result;
- onPostExecute: eseguito nel thread principale e si occupa di aggiornare l'interfaccia dopo l'esecuzione per mostrare i dati scaricati o calcolati nel task che vengono passati come parametro.

Il secondo generic della classe serve ad aggiornare l'avanzamento dell'eventuale ProgressBar.

3.3 La classe JSONParser

La classe JSONParser si occupa dell'interazione che l'applicazione software presente sul dispositivo mobile avrà con il server MySQL.

È composta da un solo metodo pubblico che restituisce un oggetto di tipo JSONObject:

```
public JSONObject makeHttpRequest(String url, String method,  List<NameValuePair> params)
```

Come si può notare dal codice qui sopra, il metodo accetta tre parametri:

- Una stringa che rappresenta l'url dello script PHP residente sul server e che verrà interpellato.
- Una stringa che consente di selezionare il metodo HTTP di interesse. In questa applicazione sono supportati i metodi "POST", "GET", che saranno rispettivamente: uno per aggiornare in maniera non tracciabile, l'altro per richiedere in modo esplicito le informazioni di interesse dal database.

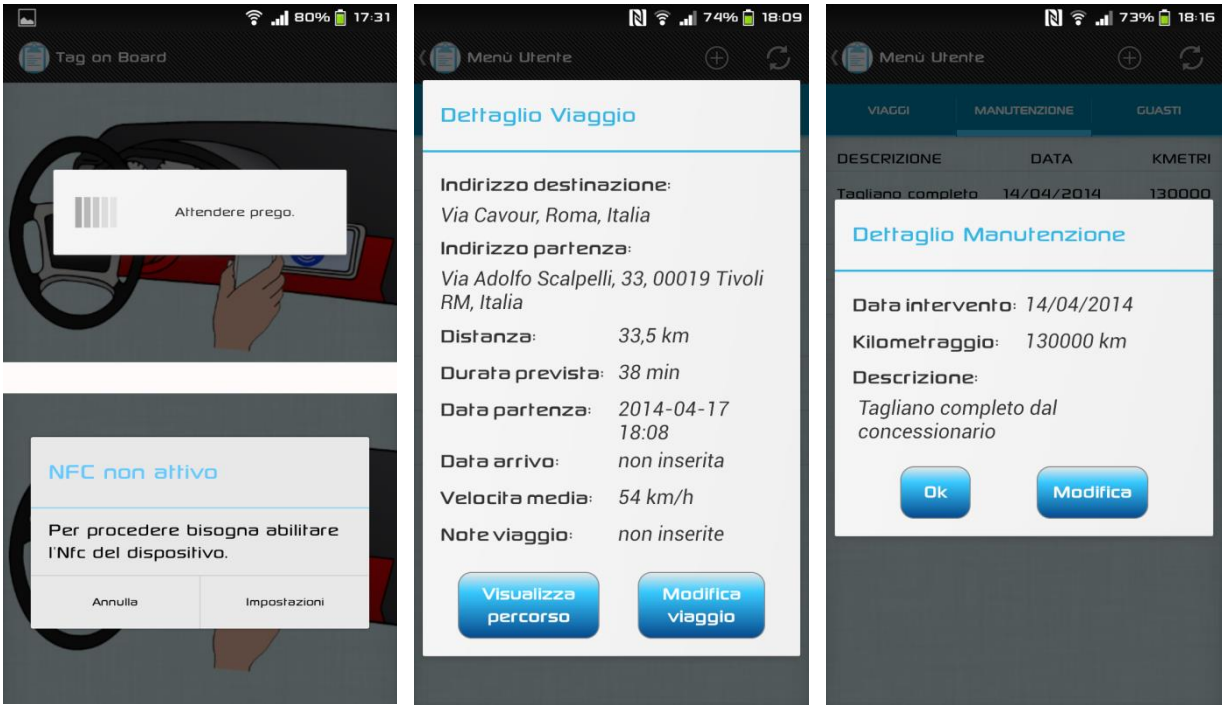
- Una lista di org.apache.http.NameValuePair che contiene i parametri sotto forma di coppia nome e valore (BasicNameValuePair).

Bisogna creare l'Url da inviare al Server estrapolando i parametri dalla lista params, successivamente viene fatta la richiesta http e infine si incapsula la risposta (un InputStream) in un oggetto Json.

```
HttpGet httpGet = new HttpGet(url);
HttpResponse httpResponse = httpClient.execute(httpGet);
HttpEntity httpEntity = httpResponse.getEntity();

InputStream      is = httpEntity.getContent();
//si trasforma is in una Stringa chiamata json tramite un BufferedReader
jObj = new JSONObject(json);
return jObj;  //si restituisce l'oggetto JSON
```

3.4 Finestre dialogo e la classe FragmentDialog

Queste sono tutte le finestre di dialogo utilizzate.



Sono 4 classi differenti che estendono tutte la classe FragmentDialog. Hanno un layout proprio e girano su un thread separato diverso dai quello dell'attività che li fa partire. Questa scelta, insieme all'inizializzazione statica, è stata fatta per renderli persistenti alla rotazione dello schermo grazie all'istruzione setRetainInstance(true). Infatti, in Android, quando cambia l'orientamento dello schermo l'Activity viene distrutta e fatta ripartire, quindi eventuali finestre che vengono inizializzate e risiedono sul UI-Thread vengono distrutti e in casi particolari lanciano un'eccezione che manda in crash l'intero programma.

La prima in alto a sinistra è una classica ProgressDialog e fa riferimento alla classe MyProgressDialog in questione:

```
public class MyProgresDialogFragment extends DialogFragment {
        private String tit = "Caricamento...";

    // Costruttore statico
  public static MyProgresDialogFragment newInstance(String t) {
        MyProgresDialogFragment m = new MyProgresDialogFragment();
        m.tit=t;
```

```
        return m;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);    // Lo dichiaro insensibile hai cambiamenti di configurazione
    }

    @Override
public View onCreateView(LayoutInflater inflater,ViewGroup container,Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.progress_dialog_holo, container, false);

        getDialog().getWindow().requestFeature(Window.FEATURE_NO_TITLE);
        TextView mess = (TextView)v.findViewById(R.id.message);
        mess.setText(tit);

        // Disabilitato il back button per non cancellarla
        OnKeyListener keyListener = new OnKeyListener() {
                                @Override
                                public boolean onKey(DialogInterface arg0, int arg1, KeyEvent arg2) {
                                        return arg1 == KeyEvent.KEYCODE_BACK;
                                }
                        };
        getDialog().setOnKeyListener(keyListener);

        // Disabilitata la cancellazione quando si tocca fuori la finestra
        getDialog().setCanceledOnTouchOutside(false);
        return v;
    }

    @Override
    public void onDestroyView() {
        if (getDialog() != null && getRetainInstance())
            getDialog().setDismissMessage(null);    // se la finestra è ancora in uso non la facciamo compa-
rire nello stack dei dialog da cancellare.
        super.onDestroyView();
    }
}
```

Viene visualizzata prima di un AsynTask per notificare all'utente che in background si stanno compiendo delle operazioni.

La seconda Finestra (MyDialogSettingsFragment) in basso a sinistra è una classica AlertDialog in cui viene rappresentato un avviso che illustra all'utente una particolare situazione: ad esempio notifica che il sensore Nfc è disabilitato oppure di attivare il Gps per avere una posizione più precisa. In questa finestra, come si può notare, è presente un titolo, la descrizione e due pulsanti uno per annullare e chiudere la finestra, l'altro per aprire la pagina delle impostazioni tramite questo script:

```
// Use the Builder class for convenient dialog construction
AlertDialog.Builder miaAlert = new AlertDialog.Builder(getActivity());
miaAlert.setPositiveButton("Impostazioni", new DialogInterface.OnClickListener() {
public void onClick(DialogInterface dialog, int id) {
                        if(tipo.equals(Service.NFC_SERVICE))
                                startNfcSettingsActivity();
                        else if(tipo.equals(Service.LOCATION_SERVICE))
                                startGpsSettingsActivity();
                }
```

```
});
protected void startNfcSettingsActivity() {
        startActivity(new Intent(android.provider.Settings.ACTION_WIRELESS_SETTINGS));
}

protected void startGpsSettingsActivity() {
        startActivity(new Itent(android.provider.
                                        Settings.ACTION_LOCATION_SOURCE_SETTINGS));
}
```

Nella figura centrale è mostrata il Dialog MyDialogViaggioFragment che descrive i dettagli di un viaggio memorizzato in una lista. Infatti, questo Fragment viene lanciato quando si clicca su un elemento nella ListView della classe ListaViaggiFragment. Oltre a far consultare tutte le informazioni (che per motivi di spazio non entrano nella tabella della lista) sono presenti due Button associati a queste funzioni: ModificaViaggioActivity e com.google.android.apps.maps. La prima serve ad aggiungere dettagli del viaggio (facendo eseguire la nuova attività passandogli i dati necessari per la modifica tramite il metodo Intent.putExtra(Indirizzo, Valore)); la seconda manda un Intent con l'url di Google Maps per visualizzare il percorso effettuato su una mappa e avviare la navigazione se necessario (questo Intent può essere risolto dall'App Maps di Google oppure tramite il browser web del dispositivo). Le textView presenti vengono settate tramite il parametro della classe Viaggio presente in questo file e nel rispettivo costruttore. Questo il risultato del onCreate:

```
public View onCreateView(LayoutInflater inflater,ViewGroup container,Bundle savedInstanceState) {
        setRetainInstance(true);
        View v = inflater.inflate(R.layout.dialog_viaggio_holo, container, false);
        getDialog().setTitle(title);
        TextView des_destinazione= (TextView) v.findViewById(R.id.des_destinazione);

        …. Altre textView……

        ind = viaggio.getId();
        des_destinazione.setText(viaggio.getEndAddr());

        ..... Altre inizializzazione....

        Button percorso = (Button)v.findViewById(R.id.visualizza_percorso);
        percorso.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
                //faccio vedere il percorso tramite google maps
                Intent ii = new Intent(Intent.ACTION_VIEW,
                        Uri.parse(http://maps.google.com/maps?f=d&saddr=  +  viaggio.getStartAddr() +
                                "&daddr="+ viaggio.getEndAddr() +"&dirflg=d&hl=it"));

                        startActivity(ii);
                        getDialog().dismiss();
                }
        });

        Button modifica = (Button)v.findViewById(R.id.modifica_dettagli);
        modifica.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                        Intent intent = new Intent(getActivity().getApplicationContext(),
                                                ModificaViaggioActivity.class);
                        intent.putExtra("ind_viaggio", ind);
                        intent.putExtra("des_inizio", viaggio.getStartAddr());
                        intent.putExtra("des_fine", viaggio.getEndAddr());
```

```
                intent.putExtra("des_distanza", viaggio.getDesDistanza());
                intent.putExtra("des_durata", viaggio.getDesDurata());
                intent.putExtra("data_inizio", viaggio.getDataInizio());
                intent.putExtra("data_fine", viaggio.getDataFine());
                intent.putExtra("note", viaggio.getNote());

                getActivity().startActivityForResult(intent, AGGIORNA_LISTAV_REQUEST
                getDialog().dismiss();
            }
        }); return v;     }
```

L'ultima figura a destra mostra l'output del file MyDialogFragment che per mezzo di un parametro "classe" nel costruttore visualizza il dettaglio di un guasto o di una manutenzione. Al livello strutturale si presenta come il file precedente ma cambiano ovviamente le TextView e i pulsanti sottostanti che rimangono ancora due, ma hanno funzioni differenti. Uno serve a chiudere la finestra di dialogo, l'altro a far partire una nuova attività di modifica dell'elemento. Quest'ultimo Button è visibile solo se non sono passati più di due giorni dalla data di inserimento dell'intervento sulla macchina, per mezzo di una funzione controllaData(). Il codice sottostante mostra questa funzione e come viene utilizzata per modificare la visibilità del pulsante.

```
private boolean controllaData(String data){
        long sax;
        SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd",Locale.getDefault());
        Date toCompare;
        try {
                toCompare = sdf.parse(data);
                Date today = new Date();
                sax = ((today.getTime() - toCompare.getTime())/60/60/24/1000);
                return sax<=1;
        }catch (ParseException e) { e.printStackTrace();}
        return true;
}

Button modifica = (Button)v.findViewById(R.id.modifica_dialog);
if(controllaData(dataCurr)){
        modifica.setVisibility(View.VISIBLE);
        modifica.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
                if(classe.equals("Guasto")){
                        Intent intent = new Intent(getActivity().getApplicationContext(),
                                                        ModificaGuastoActivity.class);
                        intent.putExtra("id", indirizzo);
                        intent.putExtra("intervento", descrizione);
                        intent.putExtra("data", data);
                        intent.putExtra("km", km);
                        getActivity().startActivityForResult(intent, AGGIORNA_LISTAG_REQUEST);
                }
                if(classe.equals("Manutenzione")){
                        Intent intent = new Intent(getActivity().getApplicationContext(),
                                                        ModificaManutenzioneActivity.class);
                        intent.putExtra("id", indirizzo);
                        intent.putExtra("descrizione", descrizione);
                        intent.putExtra("data", data);
                        intent.putExtra("km", km);
                       getActivity().startActivityForResult(intent, AGGIORNA_LISTAM_REQUEST)
                }
```

```
                getDialog().dismiss();
                }
        });
}
```

3.5 SharedPreferences e gestione delle cache

La classe android.content.SharedPreferences permette agli sviluppatori di salvare dei settaggi applicativi in un file e condividerli nella applicazione stessa o tra tutte le applicazioni.

Il file dove vengono salvate queste applicazioni è presente nel path: /data/data/ <package_applicazione>/.

Il modo più semplice di interagire con questi dati prevede l'utilizzo del metodo che la classe Activity eredita dalla classe Context:

```
public abstract SharedPreferences getSharedPreferences (String name, int mode);
```

la quale accede a un'istanza della classe SharedPreferences associata a un nome e a un identificatore dei permessi di accesso da parte delle altre applicazioni (privati o pubblici in lettura / scrittura).

La classe SharedPreferences non mette a disposizione dei metodi relativi a ciascun tipo di dato, ma funge da factory di implementazione dell'interfaccia SharedPreferences.Editor che gestisce l'aggiornamento delle informazioni analogamente alla gestione di una transazione. È prevista l'esecuzione di un'operazione di commit() al fine di rendere effettive le modifiche apportate.

A tal proposito il codice allegato permette di salvare l' indirizzo esadecimale del tag dopo che la Macchina è stata aggiunta al DB oppure verificata la sua registrazione.

```
private void aggiornaSharedPreferences () {
        //aggiorno le sharedPreferences
        SharedPreferences prefs = getSharedPreferences(getString(R.string.My_shared_preferences),
Context.            MODE_PRIVATE);
        // Otteniamo il corrispondente Editor
        SharedPreferences.Editor  editor = prefs.edit();
        //salviamo le Preferences con il valore dell Id del tag
         editor.putString(getString(R.string.Data_Tag_uid), tag_uid );
        editor.commit();
}
```

Mentre nell'Activity che richiede quel dato associato alla My_shared_preferences scriviamo:

```
        // Leggiamo le Preferences
SharedPreferences prefs = getActivity().getSharedPreferences(
                                getString(R.string.My_shared_preferences),  Context.MODE_PRIVATE);
        // Leggiamo l'informazione associata alla proprietàe la memorizzo in String tag_uid
        String tag_uid  =  prefs.getString(getString(R.string.Data_Tag_uid), "...");
        //se non viene trovata viene sostituita con il valore di default: "..."
```

In questo modo quindi viene memorizzato l'ultimo accesso eseguito. Quindi se il tag che manda l'Intent è lo stesso dell'ultimo accesso l'applicazione non richiede i nuovi dati da Internet poiché sa con sicurezza che quell'automobile è già presente nell'archivio.

In quest'ottica viene usato anche un altro accorgimento: quando viene fatta una richiesta GET al server viene memorizzata, in caso affermativo, una copia del file JSON di risposta, attraverso l'uso dei File System.

Questo il codice utilizzato per memorizzare il File:

```
private void memorizzaInfo(String json) {
// ottenuto il json lo salvo su file nella direzione preposta per le cache
        String sdCard = getActivity().getCacheDir().toString();
```

```
        try {
        //creo l'istanza File che verrà memorizzato con il nome del 2° parametro
                File file = new File(sdCard, "infoAuto.json");
                FileWriter out = new FileWriter(file);
                //lo memorizzo
                out.write(json);
                out.close();
        } catch (IOException e1) {e1.printStackTrace();}
                // file salvato.
}
```

Il file .json, una volta salvato nelle cartella del programma dedicata alle cache, può essere preso e processato come se provenisse dalla rete. Ovviamente bisogna controllare che:

1) il file salvato in precedenza appartiene effettivamente al Tag (autovettura) corrente;

2) non sia stata fatta nessuna modifica alla lista dei dati precedentemente richiesti.

Per gestire quest'ultimo punto, il programma fa partire ogni Activity figlia (esempio: AggiungiGuastoActivity), tesa a modificare i dati, tramite quest'istruzione:

```
Intent intent = new Intent(getActivity().getApplicationContext(), AggiungiGuastoActivity.class);
intent.putExtra("tag_uid", tag_uid);
getActivity().startActivityForResult(intent, AGGIORNA_LISTAG_REQUEST);
```

L'Activity padre (esemio: MenuUtenteActivity) così può monitorare il valore di ritorno del processo avviato in precedenza, tramite il metodo:

```
void onActivityResult(int requestCode, int resultCode, Intent data)
```

In caso di risultato positivo (`Activity.RESULT_OK`) viene modificato un flag condiviso, che se impostato a "`AGGIORNA_OK`" fa richiedere forzatamente i dati al server per mezzo della rete dati del dispositivo. L'Activity figlia incaricata di una modifica dei dati nel DB, nel caso in cui la modifica vada a buon fine prima di terminare setta il suo valore di uscita in questo modo:

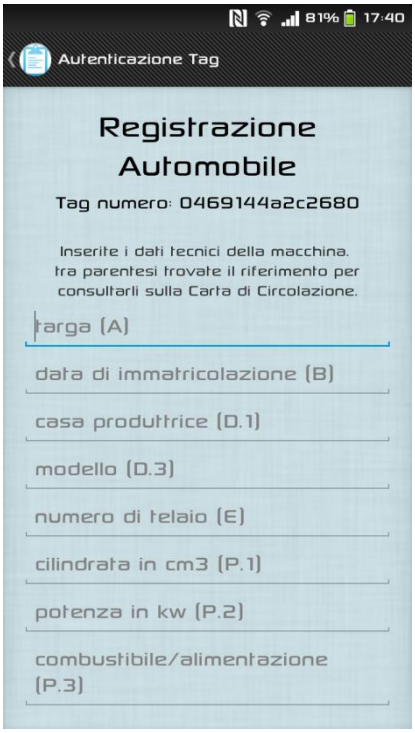
```
Intent intent = getIntent();
setResult(RESULT_OK, intent);
finish();
```

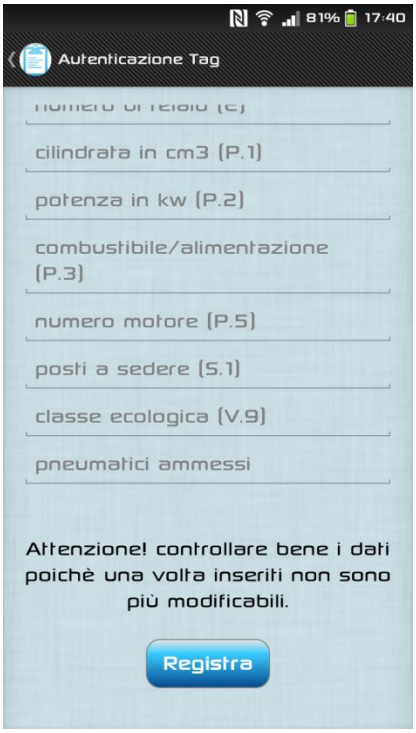
Da tutto ciò si evince una caratteristica fondamentale di quest'applicazione: richiedere i dati alla rete solo se strettamente necessario. I motivi di questa scelta sono molteplici tra cui:

- Rendere la visualizzazione molto più veloce, poiché i dati sono memorizzati direttamente sulla memoria del cellulare;
- Utilizzare, quando possibile, l'applicazione in modalità offline;
- Realizzare l'ottimizzazione delle risorse applicando un risparmio energetico.

3.6 Modulo di registrazione autovettura

L'Attività di registrazione e associazione del tag all'auto si presenta come segue:





Tutte le informazioni richieste dovranno essere aggiunte dall'utente, che può essere il proprietario dell'automobile oppure un addetto del concessionario stesso. Una volta inserite non possono essere più modificate. Vicino ad ogni descrizione c'è anche il codice facente riferimento la legenda della Carta di Circolazione. Tutti i valori devono essere non nulli e se non dovessero risultare validi verrà rifiutato con un messaggio di errore (Toast) "errore campo richiesto mancante, riprova".

Il numero identificativo del tag, invece, chiaramente non richiede immissione manuale da parte dell'utente, ma sarà visualizzato in formato esadecimale dopo essere stato letto dall'Activity precedente.

Alla pressione del tasto "Registra" parte un thread che si occupa della comunicazione con il server per mezzo della classe JsonParser. Per ovviare ad eventuali lag di rete, viene visualizzato il messaggio "Registrazione automobile in corso" per mezzo della classe MyProgresDialogFragment. Mentre l'utente visualizza il messaggio, il thread si occupa di prelevare le informazioni necessarie dalle varie EditText e successivamente fa la richiesta HTTP al server chiedendo di creare una nuova automobile con le informazioni appena fornite. Un messaggio informativo apparirà sullo schermo per notificare il successo dell'operazione oppure eventuali errori.

3.7 Schermata principale Menu Utente e i Fragments

La schermata principale dell'applicazione MenuUtente, permette di consultare i dati e scegliere l'operazione di interesse. Essa appare immediatamente al contatto con il tag NFC se la macchina era già stata registrata precedentemente, oppure dopo l'Activity di registrazione se il mezzo è stato appena aggiunto. Viene presa l'informazione dell'identificativo del tag dalle SharedPreference aggiornate. Questa infatti pur non essendo interessante per l'utente (quindi non visualizzata) è necessaria per la comunicazione con il server (per identificare le righe delle tabelle del DB).

MenuUtente estende la classe FragmentActivity una classe che deriva da Android.Activity ma permette di usare i fragments della libreria android.support.v4.app.Fragment. Tradizionalmente, le applicazioni Android sono costituite da due componenti fondamentali: un layout, ossia il

contenuto dello schermo, e una classe di attività associata, utilizzata per gestire le funzioni e gli handler per la gestione degli eventi del ciclo di vita della schermata. Il Fragment è un nuovo concetto che disaccoppia ulteriormente il layout visivo dal suo utilizzo: ciò consente di ottenere dei vantaggi, sia nell'utilizzo della gestione della grafica, sia nella stesura del codice, in quanto quest'ultimo, scritto per eseguire funzioni specifiche, può essere semplicemente riutilizzato in più punti e in più progetti. Come indicato dal sito ufficiale, si può pensare a un frammento come una sezione modulare di un'attività, che ha un proprio ciclo di vita, riceve i suoi eventi di input, e che è possibile aggiungere o rimuovere, mentre l'attività è in esecuzione.

La schermata principale del MenuUtente è suddivisa in quattro Fragments navigabili attraverso una barra di navigazione superiore e attraverso lo swipe (verso destra o sinistra) del dito sul touchscreen.

Le quattro schermate sono in ordine:

1. InfoAutomobileFragment: dove vengono visualizzate in una serie di TextView tutte le info dell'automobile inserite nelle registrazione (marca, modello, targa, ecc..);
2. ListaViaggiFragment: una ListView con tutti i viaggi dell'auto;
3. ListaManutenzioniFragment: una ListView con tutte le manutenzioni effettuate;
4. ListaGuastiFragment: una ListView con tutti i guasti subiti.

La navigazione fra i Fragment è gestita dalla classe TabsAdapter (che Eclipse genera in automatico quando si crea una nuova attività con questo modo di navigazione) e in particolare con l'istruzione:

```
protected void onCreate(Bundle savedInstanceState) {
// imposto il pager come layout di MenuUtente
        pager = new ViewPager(this);
        pager.setId(R.id.pager);
        setContentView(pager)
// inizializzo la barra di navigazione
        bar = getActionBar();
        bar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
        bar.setDisplayHomeAsUpEnabled(true);
// creo il TabsAdepter e aggiungo i fragments
        mTabsAdapter = new TabsAdapter(this, pager);
        mTabsAdapter.addTab(bar.newTab().setText(getString(R.string.title_section1)),
                                        InfoAutomobileFragment.class, null);
        mTabsAdapter.addTab(bar.newTab().setText(getString(R.string.title_section4)),
                                        ListaViaggiFragment.class, null);
        mTabsAdapter.addTab(bar.newTab().setText(getString(R.string.title_section2)),
                                        ListaManutenzioniFragment.class, null);
        mTabsAdapter.addTab(bar.newTab().setText(getString(R.string.title_section3)),
                                        ListaGuastiFragment.class, null);
```

Attraverso il metodo addTab viene aggiunta all'ActionBar di navigazione una nuova sezione con un titolo e la classe del Fragment a cui fa riferimento. Usando i Fragment il codice utente si sposta per lo più all'interno dei Fragment stessi e ben poco rimane nel file principale che deve solo gestire le transizioni tra layout ed il codice che permette ai Fragment di comunicare.

Come detto in precedenza, essendo questo il processo che chiama tutti gli altri che modificano il database, serve il codice che gestisce le cache attraverso il metodo:

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == AGGIORNA_LISTAM_REQUEST && resultCode == Activity.RESULT_OK) {
        // aggiornono il flag AggiornaListaManutenzioni con AGGIORNA_OK
       aggiornaSharedPreferences(AGGIORNA_LISTAM_REQUEST);
    }
    if (requestCode == AGGIORNA_LISTAG_REQUEST && resultCode == Activity.RESULT_OK) {
        // aggiornono il flag AggiornaListaGuasti con AGGIORNA_OK
```

```
        aggiornaSharedPreferences(AGGIORNA_LISTAG_REQUEST);
    }
    if (requestCode == AGGIORNA_LISTAV_REQUEST && resultCode == Activity.RESULT_OK) {
        // aggiornono il flag AggiornaListaViaggi con AGGIORNA_OK
        aggiornaSharedPreferences(AGGIORNA_LISTAV_REQUEST);
    }
```

3.8 Info Automobile Fragment

La prima schermata che viene visualizzata è questa sulla sinistra ed è formata da una serie di TextView in un Grid-Layout dove vengono inserite le informazioni tecniche della vettura. Questi dati inseriti in fase di registrazione non sono modificabili.

Già in questa prima schermata possiamo notare che, in tutti i Layout dei Fragment, è possibile fare lo scroll verticale per vedere i dati a piè pagina, non visualizzabili a causa della dimensione ridotta dello schermo.

Questa classe popola le sue TextView attraverso due AsynTask. Il primo a partire è VisualizzaInfoCache che prova a recuperare i dati dalle cache, controllando se l'ultimo accesso fatto sia congruente con l'auto attuale (confrontando l'identificativo del tag attuale con quello del file salvato in cache), se questo controllo fallisse partirebbe un nuovo thread che, questa volta, chiederebbe i dati al server.

Questo nuovo AsynTask è chiamato VisualizzaInfoDB, il quale si connette al server aggiorna i dati dell'interfaccia grafica e salva i nuovi dati nelle cache.

Il codice di questi due AsynTask è il seguente:

```
private class VisualizzaInfoCache extends AsyncTask<String, String, String> implements InterfacciaCodici {
        private int result;
        @Override
        protected void onPreExecute() {
                super.onPreExecute();
                bloccaRotazione();
        }
        @Override
        protected String doInBackground(String... params) {
                JSONObject json = leggiFileInfo();          //leggo il file nelle cache
                if(json == null){                           //non esiste il file
                Log.d("json == null"," il file listaGuasti non esiste ");
                result = 0;         //se impostato a 0 vuol dire che bisogna fare la richiesta al server
                return null;
                }
                try {
                        JSONArray auto = json.getJSONArray(AUTOMOBILI);
                        JSONObject j = auto.getJSONObject(0);

                if(j.getString(TAG_UID).equals(tag_uid)){
                        creaInfoAutomobile(json);  // creo le info ma non le faccio visualizzare poiché
```

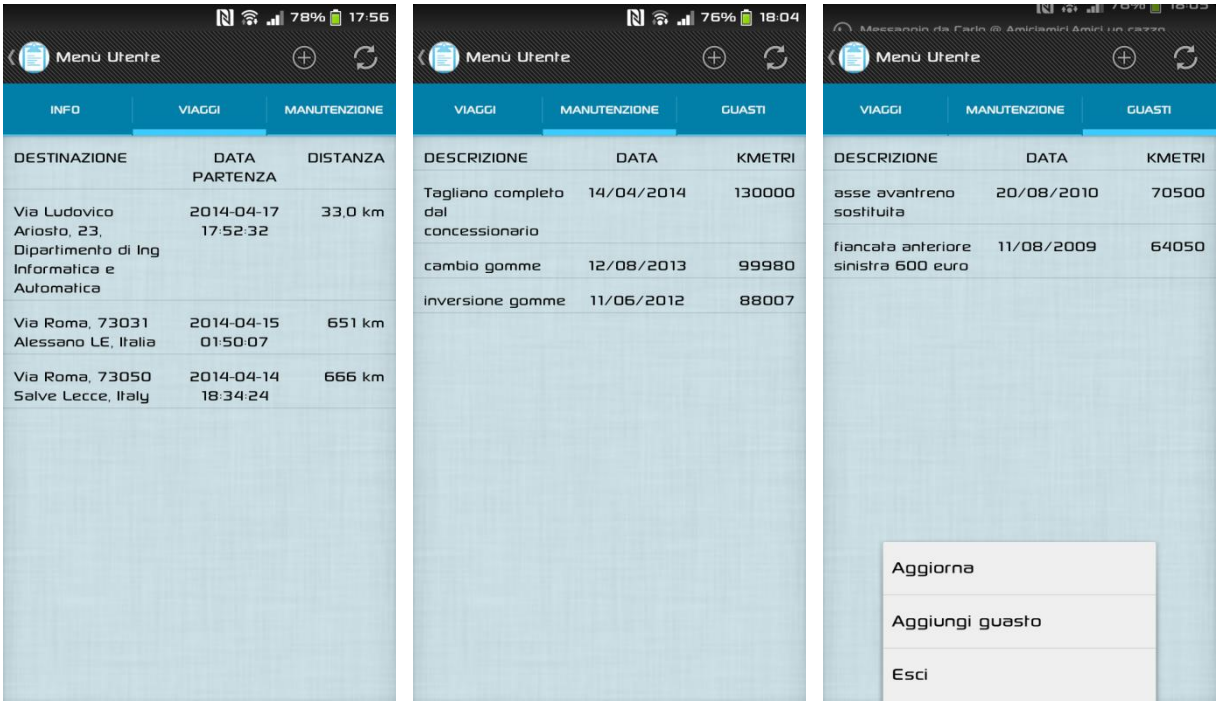
```
                                                    siamo dentro un thread secondario.
                    result = 1;          //non serve richiederle al server
                }
                else{ result = 0; }
            } catch (JSONException e) {     e.printStackTrace();}
            return null;
}
            @Override
            protected void onPostExecute(String r) {
                if(result==1){
                    visualizzaInfoAutoView();  //non aggiorno le textView  poiché siamo nel
                                                MainThread
                    Log.d("risultato postexe info: ", result +" info prese dalle cache");
                    // riattivo la rotazione dello schermo
                    getActivty().setRequestedOrientation(
                            ActivityInfo.SCREEN_ORIENTATION_UNSPECIFIED);
                    Pdialog.dismiss();       //chiudo il ProgressDialog
                }
                else{     // result = 0
                    Log.d("risultato postexe info ", result +" info prese dal DB");
                    new VisualizzaInfoDB().execute();   // richiedo le info al DB
                }
            }
}

 private class VisualizzaInfoDB extends AsyncTask<String, String, String> implements Interfac-
ciaCodici {
             private int success;
            @Override
            protected void onPreExecute() {
                super.onPreExecute();
                bloccaRotazione();
            }
            @Override
            protected String doInBackground(String... params) {
                JSONParser jsonParser = new JSONParser();
                List<NameValuePair> par = new ArrayList<NameValuePair>(2);
                par.add(new BasicNameValuePair(MY_GET, GET_AUTOMOBILE_DETAILS));
                par.add(new BasicNameValuePair(TAG_UID, tag_uid));
                JSONObject json = jsonParser.makeHttpRequest(getString(R.string.gestione_client_url),
                                                                "GET", par);
                    Intent intent;
                    //codice per errori di rete
                    if (json == null) {
                        intent = new Intent(getActivity(), ErroreDiReteActivity.class);
                        startActivity(intent);
                        return null;
                    }//Log.d("json main: ", json.toString());
                    try {
                        success = json.getInt(SUCCESS);
                        if (success == 1) { //l'auto è presente nel database
                            memorizzaInfo(json.toString());
                            Log.d("Informazione Auto: ", " scritta su file ");
                            creaInfoAutomobile(json);
                            return null;
                        }
                        else //success = 0 || 2 ----> errore
                            getActivity().runOnUiThread(new Runnable() {
```

```
                                        public void run() {
                                        Toast.makeText(getActivity().getApplicationContext(), "errore
nel controllo auto dal database", Toast.LENGTH_LONG).show();
                                        }
                                });
                        } catch(JSONException e) {e.printStackTrace();}
                                return null;
        }
        @Override
        protected void onPostExecute(String result) {
                getActivity().runOnUiThread(new Runnable() {
                        public void run() {
                                if(success == 1)
                                        visualizzaInfoAutoView();
                                }
                        });
                getActivity().setRequestedOrientation(
                                        ActivityInfo.SCREEN_ORIENTATION_UNSPECIFIED);
                Pdialog.dismiss();
        }
}
```

Per la prima volta si può notare una caratteristica importate di questa applicazione, cioè quella di bloccare la rotazione dello schermo nel metodo onPreExecute(), per poi sbloccarlo al termine del AsynTask nell'onPostExecute(). Tutto questo perché in Android quando si capovolge il dispositivo, se la rotazione è abilitata, l'Activity corrente viene distrutta e poi rigenerata con il nuovo layout capovolto. Si possono creare quindi dei problemi di inconsistenza ed essere lanciate delle eccezioni, in quanto il Thread secondario una volta finito può andare a modificare una variabile (ad esempio una TextView) non ancora inizializzata a seguito della rotazione. Questo accorgimento viene utilizzato in tutti gli AsynTask utilizzati.

```
private void bloccaRotazione(){
        //  viene memorizzata la posizione dello schermo attuale
         int rotation = getActivity().getWindowManager().getDefaultDisplay().getRotation();
         Log.d("blocca rotazione guasti: ", "rotation :  "+ rotation);
         switch(rotation) {
              case Surface.ROTATION_180:
                 // fisso la rotazione al valore attuale
                 getActivity().setRequestedOrientation(
                                 ActivityInfo.SCREEN_ORIENTATION_REVERSE_PORTRAIT);
                 break;
              case Surface.ROTATION_270:
                  getActivity().setRequestedOrientation(
                                 ActivityInfo.SCREEN_ORIENTATION_REVERSE_LANDSCAPE);
              break;
              case  Surface.ROTATION_0:;
                  getActivity().setRequestedOrientation(
                                        ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
                  break;
              case Surface.ROTATION_90:
                   getActivity().setRequestedOrientation(
                                        ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
                   break;
        }
 }
```

3.9 ListFragment dei viaggi, delle manutenzioni e dei guasti



Da questi screenshots si evince come queste tre schermate abbiano tutte le stesse caratteristiche, estendono tutte e tre la classe ListFragment e hanno come layout una ListView interattiva, ovvero quando si clicca su un elemento si avvia un Listener che fa aprire una finestra dialog con i dettagli da visualizzare.

Appena creati questi fragments si occupano di prelevare le informazioni per popolare la lista. Questi dati possono essere prelevati da due Thread (come è già stato illustrato per InfoAutoFragment), uno visualizzaListaCache parte per prima, cercando di trovarli nel percorso delle cache; l'altro visualizzaListaDB li preleva dal server facendo uso, ancora una volta, del metodo makeHttpRequest della classe JSONParser.

```
JSONParser jsonParser = new JSONParser();
        List<NameValuePair> par = new ArrayList<NameValuePair>(2);
        par.add(new BasicNameValuePair(MY_GET, GET_ALL_GUASTI));
        par.add(new BasicNameValuePair(AUTOMOBILE, tag_uid));
        JSONObject json = jsonParser.makeHttpRequest(getString(R.string.gestione_client_url),
"GET", par);
```

Questo, ad esempio, è il metodo http "GET" utilizzato per ottenere la lista dei guasti. I parametri necessari sono l'identificativo del tag dell'auto e il codice GET_ALL_GUASTI per comunicare al server cosa si vuole.

Al termine dell'elaborazione, se non ci sono errori, viene creata e visualizzata la lista, per mezzo del seguente codice:

```
runOnUiThread(new Runnable() {

        public void run() {
                ListAdapter adapter = new SimpleAdapter(
            getActivity().getApplicationContext(),
                        listaGuasti,    // ArrayList<HashMap<String, String>>()
                        R.layout.elemento_guasto,
                        new String[] { "id", "intervento", "data", "km", "dataCurr"},
                        new int[] { R.id.id_guasto, R.id.intervento, R.id.data_guasto,
                    R.id.km_guasto, R.id.dataCurr_guasto });

                setListAdapter(adapter);

        }});
```

I primi tre parametri passati al costruttore di SimpleAdapter indicano:

- Context da usare, quello dell'Activity padre;
- i dati da mostrare organizzati in una lista di mappe chiave-valore;
- il layout da usare per mostrare una singola riga.

Il quarto e quinto parametro sono due array della stessa lunghezza che indicano come mappare gli oggetti della lista dei dati alle View; il primo array di stringhe indica le chiavi nella mappa dei dati di un oggetto, mentre il secondo array di interi indica le corrispondenti View contenute nel layout (del terzo parametro) in cui mostrare i dati.

Il layout del singolo elemento viene descritto nel file XML elemento_guasto.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:orientation="horizontal" >
   <TextView
      android:id="@+id/id_guasto"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:visibility="gone" />
   <TextView
      android:id="@+id/dataCurr_guasto"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:visibility="gone" />
   <TextView
      android:id="@+id/intervento"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1.4"
      android:gravity="left"
      android:paddingLeft="10dp"
      android:paddingTop="10dp"
      android:textColor="#000000"
      android:textSize="15sp" />
   <TextView
      android:id="@+id/data_guasto"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1.2"
      android:gravity="center"
      android:paddingLeft="10dp"
      android:paddingTop="10dp"
      android:textColor="#000000"
      android:textSize="15sp" />
   <TextView
      android:id="@+id/km_guasto"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:gravity="right"
      android:paddingRight="10dp"
      android:paddingTop="10dp"
      android:textColor="#000000"
      android:textSize="15sp" />

</LinearLayout>
```

Il codice che abbiamo visto faceva riferimento alla classe ListaGuastiFragment ma la struttura è analoga anche per ListaViaggiFragment e ListaManutenzioniFragment.

3.10 I menù e l'ActionBar

I menù utilizzati in questo progetto sono di due tipi come si vede dalle immagini. La prima foto mostra il classico menù a scomparsa Android di prima generazione attivabile per mezzo del tasto Hardware dello smarthphone mentre la seconda immagine fa vedere il nuovo concetto di menù basato su l'ActionBar (disponibile dall'API Level 11 in poi), in cui è possibile inserire pulsanti (piccole icone in formato .png) on titoli direttamente sulla barra superiore dell'Activity. In quasi tutte le attività del programma sono usati questi menù che a seconda del contesto, se premuti i loro elementi, svolgono operazioni differenti.

I Layouts XML dei menù dove sono specificati gli Item, devono essere dichiarati nel metodo onCreateOptionsMenu(Menu menu) per associarli all'Activity.

L'Actionbar può essere richiamata attraverso il metodo getActionBar() a cui viene applicato setDisplayHomeAsUpEnabled(true) che darà la possibilità, premendo l'icona dell'applicazione in alto a sinistra, di richiamare l'activity padre, come espresso nella sua dichiarazione nel "manifest". Ad esempio l'attività padre di AggiungiViaggio è MenuUtente come si evince dalla sua dichiarazione:

```
<activity
        android:name="com.lorenzo1889.diariodibordo.AggiungiViaggioActivity"
        android:label="@string/title_activity_aggiungi_viaggio"
        android:parentActivityName="com.lorenzo1889.diariodibordo.MenuUtente" >
        <meta-data
           android:name="android.support.PARENT_ACTIVITY"
           android:value="com.lorenzo1889.diariodibordo.MenuUtente" />
```

Quando si preme una qualsiasi voce dei menù viene invocato il metodo onOptionsItemSelected(MenuItem item) in cui bisogna gestire l'azione associata a quel pulsante.

Di seguito un esempio del menù nel Fragment InfoAutomobile, il quale è senz'altro quello più complesso, infatti da questa schermata si possono aggiungere, premendo l'icona con il "+" cerchiato oppure azionando la voce "aggiungi <nome elemento>" nel menù classico, tutti gli

elementi (Viaggi, Guasti e Manutenzioni) che popolano le liste. Negli altri tre fragments si può aggiungere solo l'elemento appartenente alla lista visualizzata, ad esempio dentro ListaViaggiFragment si può aggiungere solo un Viaggio, non un Guasto o altro. Rimane invece invariata per tutti i frammenti il comportamento dell'Item "aggiorna" (espresso con l'icona raffigurante due freccette circolari oppure tramite il pulsante "aggiorna" nel menù a scomparsa) e dell'Item "esci". Il primo fa partire il thread per la richiesta dei dati direttamente al server, senza passare da eventuali cache. Il secondo come si può intuire, chiama il metodo finish() per terminare l'attività corrente.

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
        inflater.inflate(R.menu.info_automobile, menu);
 }
 @Override
 public boolean onOptionsItemSelected(MenuItem item) {
         if(!isNetworkAvailable() && item.getItemId() != R.id.option_add){
                 Toast.makeText(getActivity().getApplicationContext(), "Errore! Non sei connesso ad
internet", Toast.LENGTH_LONG).show();
                 return true;
         }
         Intent intent;
                 switch (item.getItemId()) {
                 case R.id.menu_refresh:
                          showMyDialog();
                          new VisualizzaInfoDB().execute();
                          return true;
                 case R.id.option_add_manutenzione:
                           intent = new Intent(getActivity().getApplicationContext(),
                                                          AggiungiManutenzioneActivity.class);
                          intent.putExtra("tag_uid", tag_uid);
                         getActivity().startActivityForResult(intent, AGGIORNA_LISTAM_REQUEST);
                          return true;
                 case R.id.option_add_guasto:
                 I        ntent = new Intent(getActivity().getApplicationContext(),
                                                                 AggiungiGuastoActivity.class);
                          intent.putExtra("tag_uid", tag_uid);
                         getActivity().startActivityForResult(intent, AGGIORNA_LISTAG_REQUEST);
                          return true;
                 case R.id.option_add_viaggio:
                          intent = new Intent(getActivity().getApplicationContext(),
                                                                 AggiungiViaggioActivity.class);
                          intent.putExtra("tag_uid", tag_uid);
                         getActivity().startActivityForResult(intent, AGGIORNA_LISTAV_REQUEST);
                          return true;
                 case R.id.item_refresh:
                          showMyDialog();
                          new VisualizzaInfoDB().execute();
                          return true;
                 case R.id.item_add_manutenzione:
                          intent = new Intent(getActivity().getApplicationContext(),
                                                          AggiungiManutenzioneActivity.class);
                          intent.putExtra("tag_uid", tag_uid);
                         getActivity().startActivityForResult(intent, AGGIORNA_LISTAM_REQUEST);
                          return true;
                 case R.id.item_add_guasto:
                          intent = new Intent(getActivity().getApplicationContext(),
                                                                 AggiungiGuastoActivity.class);
                          intent.putExtra("tag_uid", tag_uid);
```

```
                    getActivity().startActivityForResult(intent, AGGIORNA_LISTAG_REQUEST);
                    return true;
                case R.id.item_add_viaggio:
                    intent = new Intent(getActivity().getApplicationContext(),
                                                        AggiungiViaggioActivity.class);
                    intent.putExtra("tag_uid", tag_uid);
                    getActivity().startActivityForResult(intent, AGGIORNA_LISTAV_REQUEST);
                    return true;
                }
        return super.onOptionsItemSelected(item);
}
```

La funzione isNetworkAvailable() posta prima dello switch, restituisce true se il dispositivo è connesso ad internet, false altrimenti. Quindi se non c'è la rete attiva, poiché necessaria per aggiungere o aggiornare, viene visualizzato un messaggio di errore senza far partire inutilmente le sotto-attività.

3.11 Aggiungi guasto o manutenzione

L'Attività di aggiunta di un guasto e di una manutenzione si presentano come segue:

Queste due attività svolgono esattamente la stessa funzione, quella di aggiungere un intervento alla tabella delle manutenzioni o dei guasti nel Database.

Prenderemo in considerazione solo AggiungiManutenzioneActivity poiché l'altra è esattamente la stessa tranne per i campi propri di un Guasto e i codici da inserire nel Url inviato al server.

L'utente è invitato a immettere i tre campi seguendo le istruzioni a video in particolare per il formato della data che deve rispettare la convenzione indicata. Dunque il formato della data dovrà essere obbligatoriamente DD/MM/YYYY, dove YYYY sono le quattro cifre dell'anno, MM le due del mese e DD le due del giorno, separare dal carattere "/". Alla pressione del tasto "Aggiungi " viene effettuato il controllo, e se dovesse risultare in un formato non accettato, l'utente verrà avvertito con un messaggio e la richiesta al server non verrà inoltrata.

Se il controllo va a buon fine, il thread asincrono, che era preposto alla gestione dopo la pressione del tasto, continua e crea la lista di parametri utili per il metodo makeHttpRequest della classe JSONParser prima discussa. Il campo "method" sarà chiaramente "POST", mentre gli elementi di interesse sono: il codice: CREATE_MANUTENZIONE, l'automobile (idendificata

dal tag_uid), descrizione, data e kilometraggio. Essi vengono inseriti in una lista di NameValuePair e passati come terzo parametro al metodo, il quale a sua volta produrrà poi un oggetto di tipo org.json.JSONObject che rappresenterà la risposta del server.

In caso di esito negativo dovuto alla connessione di rete viene avviata la schermata ErroreDiReteActivity, per qualsiasi altro motivo invece viene mostrato un messaggio di errore all'utente. In caso di esito positivo viene mostrato un messaggio di successo.

```
Manutenzione m= new Manutenzione(null, tag_uid, e, d, o, null);
List<NameValuePair> par = new ArrayList<NameValuePair>(5);
par.add(new BasicNameValuePair(MY_POST, CREATE_MANUTENZIONE));
par.add(new BasicNameValuePair(AUTOMOBILE, m.getAuto()));
par.add(new BasicNameValuePair(DESCRIZIONE, m.getDescrizione()));
par.add(new BasicNameValuePair(DATA, m.getDataSQL()));
par.add(new BasicNameValuePair(KMETRAGGIO, m.getKm()));
JSONObject json = jsonParser.makeHttpRequest(getString(R.string.gestione_client_url),
                                                    "POST", par);
//codice per errori di rete
if (json == null) {
Intent intent = new Intent(AggiungiManutenzioneActivity.this, ErroreDiReteActivity.class);
        startActivity(intent);
        return null;
}
Log.d("Crea risposta", json.toString());
try {
        int success = json.getInt(SUCCESS);
        if (success == 1) {
                // "manutenzione aggiunta con successo",
                Intent intent = getIntent();
                setResult(RESULT_OK, intent);
                finish();
        } else
                //  "errore, manutenzione non aggiunta"

} catch (JSONException exc) {exc.printStackTrace();}
```

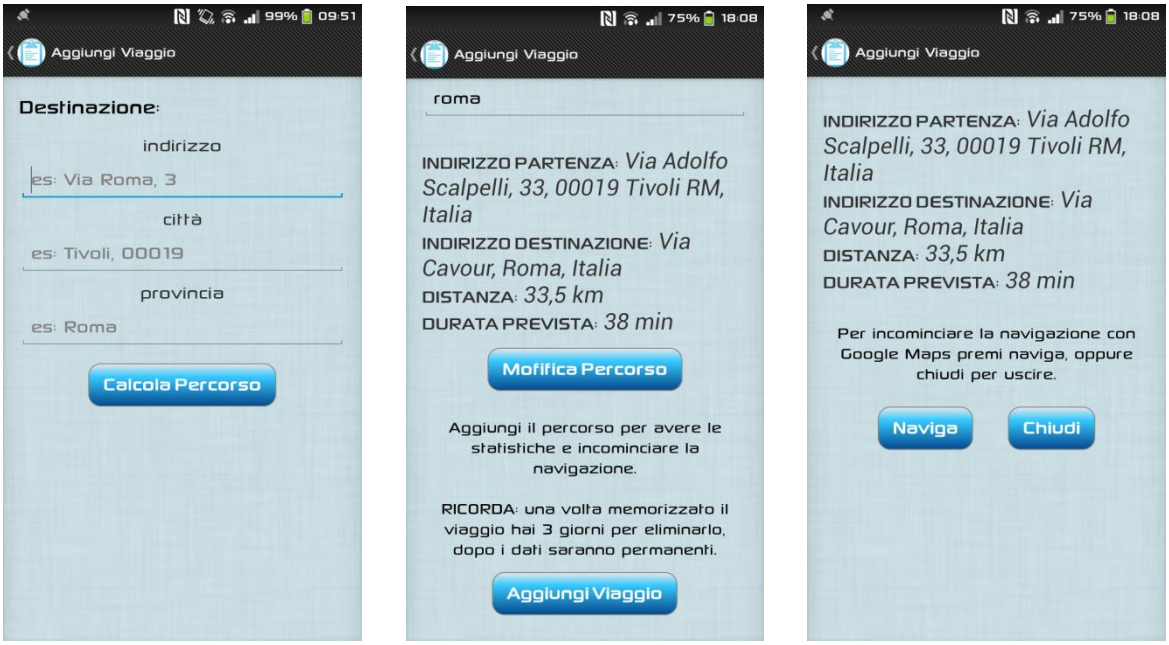
L'effettiva risposta del server in formato JSON non viene mai mostrata all'utente, ma può essere utile risalire ad essa per motivi di debug, pertanto la riga di codice:

```
Log.d("Crea risposta", json.toString());
```

si occuperà di mandare un messaggio reperibile direttamente dall'ambiente di sviluppo in fase di testing del codice.

L'intero processo normalmente richiede pochissimo tempo, meno di un secondo, ma è fortemente suscettibile alla reattività della rete internet in cui è presente il dispositivo, quindi per evitare che l'utente si ritrovi una schermata bloccata senza capire cosa stia succedendo, durante tutto il processo viene visualizzato il messaggio “Attendere prego” per mezzo di un FragmentDialog, in maniera del tutto analoga a quanto visto precedentemente.

3.12 Aggiungi Viaggio e le API di Google Maps



La procedura di aggiunta di un Viaggio si divide in tre fasi descritte in ordine, da sinistra a destra, dalle immagini soprastanti. La prima è quella di inserimento dei dati, viene suggerito all'utente di inserire l'indirizzo di destinazione seguendo le istruzioni. Quando si clicca su "calcola percorso" si avvia un Thread che raccoglie: i dati nelle EditText della destinazione, i dati memorizzati nelle variabili longitudine e latitudine provenienti dal servizio di localizzazione e fa la richiesta Http "GET" tramite la classe JSONParser al server di Google Maps. Quando viene avviata questa schermata si inizializza il servizio di localizzazione del dispositivo tramite l'istruzione:

```
// Acquire a reference to the system Location Manager
locationManager = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);

// Acquire a reference to the Listener of Location Manager
locationListener = new LocationListener() {
        public void onStatusChanged(String arg0, int arg1, Bundle arg2) {}
        public void onProviderEnabled(String arg0) {}
        public void onProviderDisabled(String arg0) {}
        // viene invocato quando la posizione è cambiata.
        public void onLocationChanged(Location l) {
           if (l.getLatitude() != 0 && l.getLongitude() != 0) {
                 location = l;
              latitudine = l.getLatitude();
              longitudine = l.getLongitude();
              Log.d("nuova coordinata location listener: ", "Location Are: " +latitudine+", "
                        +longitudine+ "from: "+l.getProvider());
           }
        }
};
```

Il LocationManager è uno strumento con cui è possibile ricevere notifiche sulla posizione del dispositivo. Si tratta di un servizio di sistema che richiede i seguenti permessi nel manifest:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Le informazioni relative alla posizione sono ottenute dal LocationManager in modi diversi, a seconda dei LocationProvider disponibili (GPS o basati sulla rete). La classe LocationManager permette di avere indicazioni sui LocationProvider disponibili e verificare quali soddisfano

meglio determinati criteri (classe Criteria) ed, eventualmente, selezionare quello che risponde alle nostre esigenze. Se non sono attivi nessuno dei due Provider viene visualizzata un AlertDialog (MyDialogSettingsFragment) che obbliga ad attivare il servizio di localizzazione altrimenti viene terminata l'Activity.

Per ricevere la notifica delle variazioni di posizione è necessario creare l'implementazione dell'interfaccia LocationListener e registrarsi a esso come ascoltatore con il seguente metodo: void requestLocationUpdates(String provider, long minTime, float minDistance, LocationListener listener). Dove il primo parametro è l'identificativo del provider, i parametri minTime e minDistance specificano una distanza ed un tempo minimo di notifica e l'ultimo parametro è il riferimento a un'implementazione dell'interfaccia LocationListener, che definisce una serie di operazioni. La più interessante è quella del metodo: void onLocationChanged(Location location), viene invocata in corrispondenza di una variazione nella posizione, le cui informazioni sono incapsulate in un oggetto di tipo Location. Location è una classe che oltre a restituire la latitudine e longitudine, fornisce dei metodi per calcolare la distanza tra due location diverse.

Questo il metodo usato per avere la posizione corrente:

```
private void getMyPosition(){
    Criteria criteria  = new Criteria();
    criteria.setAccuracy(Criteria.ACCURACY_FINE); //criterio per una posizione più precisa possibile
    criteria.setAltitudeRequired(false);
    criteria.setBearingRequired(false);
    criteria.setCostAllowed(true);
    criteria.setPowerRequirement(Criteria.POWER_LOW);
    String provider = locationManager.getBestProvider(criteria, true);
    Log.d("Best Provider:", "get location from " + provider);
    if (location == null){
        // avvio il listener per aggiornare di volta in volta la posizione
        locationManager.requestLocationUpdates(provider, 3000, 10f, locationListener);
        if (locationManager != null){
            // ultima posizione trovata
            location = locationManager.getLastKnownLocation(provider);
            if (location != null){
                latitudine = location.getLatitude();
                longitudine = location.getLongitude();
            }
 }} }
```

Una volta trovata la posizione di partenza e espressa quella di destinazione, come detto in precedenza, viene fatta la solita richiesta http "GET" questa volta a Google che però risponde sempre in formato JSON e la risposta viene visualizzata nella schermata della seconda fase (immagine 2). Qui l'utente può decidere se accettare il percorso, premendo su "aggiungi viaggio" o modificarlo, toccando su "modifica percorso" per ripetere la fase 1.

Quando si clicca su "aggiungi viaggio" parte il solito AsynTask che prende tutte le informazioni del file Json di Google e le invia al server dell'applicazione per aggiungere il viaggio corrente alla tabella dei Viaggi nel DB. Se l'aggiunta va a buon fine si giunge alla terza fase altrimenti viene visualizzato un messaggio di errore.

Nella terza fase (immagine 3) viene visualizzato un promemoria del viaggio appena aggiunto e si può decidere se attivare la navigazione o semplicemente chiudere la finestra. Quando si clicca su naviga percorso viene mandato un Intent con l'Url di Google Maps più l' indirizzo di destinazione, che può essere catturato e gestito o da un browser o dall'applicazione Maps. Questo il codice:

```
private void navigaPercorso() {
        Intent ii = new Intent(Intent.ACTION_VIEW,
        Uri.parse("http://maps.google.com/maps?f=d&daddr="+ v.getEndAdd()+"&dirflg=d&hl=it"));
        startActivity(ii);
}
```

3.13 Modifica e eliminazione di un elemento

Queste due attività vengono avviate a seguito della richiesta dell'utente fatta nelle relative finestre di dialogo. Come abbiamo detto in precedenza possono essere visualizzate solo se non sono passati due giorni dalla data di inserimento dell'elemento a cui fanno riferimento. Le EditText presenti vengono settate, quando possibile, con il vecchio valore che può essere modificato a piacimento rispettando il formato per la data. Successivamente quando si imposta il valore desiderato e si clicca su modifica viene avviato il thread che grazie all'indirizzo identificativo dell'elemento e i nuovi valori, fa la richiesta di modifica al Server simile a quella di aggiunta, vista precedentemente. In modificaManutenzione e modificaGuasto si possono modificare tutti i dati mentre in modificaViaggio si può solo aggiungere o cambiare la data di arrivo e le note del Viaggio.

Il Thread di eliminazione presente nei rispettivi file, viene avviato quando si preme sul bottone elimina (in basso a destra) oppure quando si clicca sull'icona del cestino presente sull'ActionBar. In questo AsynTask basta il solo identificativo dell'elemento per eliminarlo dalla tabella di appartenenza.

L'informazione sull'ID identificativo dell'elemento viene fornito dal database quando si richiede una lista e memorizzato in una TextView dell'elemento di appartenenza, reso invisibile all'utente.

3. Car Stats System – Lato server

3.1 Descrizione e struttura del Database

Il server è costituito da una serie di script Php che risiedono in uno spazio web nel dominio http://andproj.altervista.org/ e rispondono in formato Json alle varie richieste http (GET o POST). Il file principale è gestione_client che per mezzo di un interfaccia in cui sono presenti delle costanti in comune con il client, riesce ad individuare la richiesta e importare un altro file Php che possa rispondere. Ogni richiesta viene processata in maniera concorrenziale, questo è strettamente legato alla potenza del server su cui si effettua la connessione (in questo caso i server di Altervista permettono di avere gratuitamente un traffico di 15 GB al mese e 20mila/h queries). Inoltre, la consistenza dei dai nel Database è data dall'unicità dell' id dell'elemento e il numero del tag (della macchina) che caratterizzano ogni singolo record delle varie tabelle, infatti non potrà mai accadere che una specifica riga della stessa tabella possa essere richiesta e modificata nelle stesso istante da due dispositivi differenti.

Le Tabelle nel Database sono quattro strutturate come segue:

- Automobili: [tag_uid (chiave primaria), marca, modello, targa(chiave), dataImm, telaio(chiave), cilindrata, potenza, alimentazione, motore, posti, euro, pneumatici]
- Viaggi: [auto, id (chiave primaria), lng_start, lat_start, lng_end, lat_end, des_start, des_end, distanza, durata, des_distanza, des_durata, data_inizio, data_fine*, note*]
- Manutenzioni: [id(chiave primaria), auto, descrizione, data, km, dataCurr]
- Guasti: [id(chiave primaria), auto, intervento, data, km, dataCurr]

3.2 Connessione al database e configurazione

Tutte le comunicazioni dell'applicazione (che agisce da client nel sistema) con il server avvengono per mezzo di script PHP residenti sullo stesso.

La connessione al Database è gestita dal seguente script:

```
<?php
class DB_CONNECT {
        function __construct() {
                $this->connect();
        }
        function __destruct() {
                $this->close();
        }
        function connect() {
                require_once __DIR__ . '/db_config.php';
                $con = mysql_connect(DB_SERVER, DB_USER, DB_PASSWORD) or
die(mysql_error());
                $db = mysql_select_db(DB_DATABASE) or die(mysql_error());
```

```
            return $con;
        }
        function close() {
            mysql_close();
        }
}
?>
```

In cui la configurazione è letta dal file db_config.php:

```
<?php
define('DB_USER', "andproj");
define('DB_PASSWORD', "");
define('DB_DATABASE', "my_andproj");
define('DB_SERVER', "localhost");
?>
```

che contiene i parametri del database utilizzato in questo specifico contesto.

3.3 Il File gestione cliente e l'interfaccia codici

Il file gestione_cliente è l'unico che riceve direttamente la connessione Http da parte del client, per mezzo dell'interfaccia_codici è in grado di smistare le richieste e quindi richiamare il file Php in grado di svolgere la query SQL giusta.

```
<?php
        define('__ROOT__',(dirname(__FILE__)));
        require_once(__ROOT__.'/db_connect.php');
        require_once(__ROOT__.'/interfaccia_codici.php');
        $db = new DB_CONNECT();

$response = array();
// controlla i campi  richiesti: GET o POST
if(isset($_GET[MY_GET]) || isset($_POST[MY_POST]) ) {
   $richiesta = $_GET[MY_GET];
   $modifica = $_POST[MY_POST];
  // richiede lo script giusto per il codice espresso nella richiesta
   if($richiesta == GET_ALL_AUTOMOBILI)
        require __ROOT__.'/get_all_automobili.php'
   }
   else if($richiesta == GET_ALL_MANUTENZIONI){
        require __ROOT__. '/get_all_manutenzioni.php';
   }
   else if($richiesta == GET_ALL_GUASTI){
        require __ROOT__. '/get_all_guasti.php';
   }
   else if($richiesta == GET_ALL_VIAGGI){
        require __ROOT__. '/get_all_viaggi.php';
```

```
    }
    else if($richiesta == GET_AUTOMOBILE_DETAILS){
            require __ROOT__. '/get_automobile_details.php';
    }
    else if($modifica == CREATE_AUTOMOBILE){
            require __ROOT__.'/create_automobile.php';
    }
    else if($modifica == CREATE_MANUTENZIONE){
            require __ROOT__. '/create_manutenzione.php';
    }
    else if($modifica == CREATE_GUASTO){
            require __ROOT__. '/create_guasto.php';
    }
    else if($modifica == CREATE_VIAGGIO){
            require __ROOT__. '/create_viaggio.php';
    }
    else if($modifica == DELETE_MANUTENZIONE){
            require __ROOT__. '/delete_manutenzione.php';
    }
    else if($modifica == DELETE_GUASTO){
            require __ROOT__. '/delete_guasto.php';
    }
    else if($modifica == DELETE_VIAGGIO){
            require __ROOT__. '/delete_viaggio.php';
    }
    else if($modifica == UPDATE_MANUTENZIONE){
            require __ROOT__. '/update_manutenzione.php';
    }
    else if($modifica == UPDATE_GUASTO){
            require __ROOT__. '/update_guasto.php';
    }
    else if($modifica == UPDATE_VIAGGIO){
            require __ROOT__. '/update_viaggio.php';
    }
    else{
            $response[SUCCESS] = 0;
            $response[MESSAGE] = "codice errato gestione cliente";
            echo json_encode($response);
    }
} else {
            $response[SUCCESS] = 0;
            $response[MESSAGE] = "campo richiesto gestione cliente mancante";
            echo json_encode($response);
}
?>
```

3.4 Aggiunta di un elemento alla tabella

L'aggiunta di una nuova automobile al database è gestita dal seguente script PHP:

```
<?php
$response = array();
// controlla che tutti i campi  richiesti siano presenti
If  (isset($_POST[MARCA]) && isset($_POST[MODELLO]) && isset($_POST[TARGA]) && isset($_POST[TAG_UID]) && isset($_POST[DATA_IMM]) && isset($_POST[TELAIO]) && isset($_POST[CILINDRATA]) && isset($_POST[POTENZA]) && iset($_POST[ALIMENTAZIONE])
```

```
&& isset($_POST[MOTORE]) && isset($_POST[POSTI]) && isset($_POST[EURO]) && isset($_POST[PNEUMATICI])) {

    $marca = $_POST[MARCA];
    $modello = $_POST[MODELLO];
    $targa = $_POST[TARGA];
    $tag_uid = $_POST[TAG_UID];
    $data = $_POST[DATA_IMM];
    $telaio = $_POST[TELAIO];
    $cilindrata = $_POST[CILINDRATA];
    $potenza = $_POST[POTENZA];
    $alimentazione = $_POST[ALIMENTAZIONE];
    $motore = $_POST[MOTORE];
    $posti = $_POST[POSTI];
    $euro = $_POST[EURO];
    $pneumatici = $_POST[PNEUMATICI];
     // esegue la query
    $result = mysql_query("INSERT INTO automobili (marca, modello, targa, tag_uid, dataImm, telaio,
                                        cilindrata,  potenza, alimentazione, motore, posti, euro, pneumatici)
                    VALUES('$marca', '$modello', '$targa', '$tag_uid', '$data', '$telaio', '$cilindrata',
                                        '$potenza', '$alimentazione', '$motore', '$posti', '$euro', '$pneumatici')");
if ($result) {  //  la query è andata a buon fine
            $response[SUCCESS] = 1;
            $response[MESSAGE] = "automobile inserita";
            echo json_encode($response);
    } else {  // la query ha fallito
            $response[SUCCESS] = 0;
            $response[MESSAGE] = "errore, automobile non inserita";
            echo json_encode($response);
    }
} else { // manca un parametro nella richiesta
            $response[SUCCESS] = 0;
            $response[MESSAGE] = "campo Post creaAutomobile richiesto mancante'";
            echo json_encode($response);
}
?>
```

Il comportamento del server gestito da questo script è di immediata comprensione: la prima fase controlla se tutti i campi sono correttamente compilati, dopodiché invia la query MySQL con il metodo: mysql_query(), che inserisce i dati in una nuova riga della tabella.

Il passo successivo è il controllo del successo dell'operazione memorizzato nella variabile booleana $result, mentre l'ultima operazione è l'invio della risposta al client in formato JSON: echo json_encode($response).

Questo appena descritto è lo schema generale di come è organizzato il codice di ogni altro file Php importato in gestione_client.

Gli script che riguardano l'inserimento di un nuovo viaggio, manutenzione o guasto sono molto simili a questo soprastante, con le ovvie differenze della scelta dei nomi delle variabili e dei campi da inserire nelle rispettive tabelle.

3.5 Invio lista dei viaggi, delle manutenzioni e dei guasti

Come sopra, viene mostrato il codice di un solo componente, perché gli altri due sono analoghi. In questo caso mostriamo il codice riguardante la lista dei guasti, data una vettura identificata dall' id del tag, restituisce un file JSON.

```
<?php
$response = array();
if (!isset($_GET[AUTOMOBILE])) {
        $response[SUCCESS] = 0;
        $response[MESSAGE] = "campo richiesto mancante";
        echo json_encode($response);
} else {
        $tag_uid = $_GET[AUTOMOBILE];
        $result = mysql_query("SELECT * FROM guasti WHERE auto = '$tag_uid'
                                                ORDER BY km DESC");
        if (mysql_num_rows($result) == 0) { // se la risposta è nessuna riga la tabella è vuota
                $response[AUTOMOBILE] = $tag_uid;
                $response[SUCCESS] = 2;
                $response[MESSAGE] = "nessun guasto trovato";
                echo json_encode($response);
        } else {
                $response[GUASTI] = array();                    // crea l'array di guasti
                while ($row = mysql_fetch_array($result)) {     //legge le risposte del DB
                        $guasto = array();                      // crea un guasto
                        $guasto[INDIRIZZO] = $row["id"];        // inizializza il guasto
                        $guasto[AUTOMOBILE] = $row["auto"];
                        $guasto[INTERVENTO] = $row["intervento"];
                        $guasto[DATA] = $row["data"];
                        $guasto[KMETRAGGIO] = $row["km"];
                      $guasto[DATA_INSERIMENTO] = $row["dataCurr"];
                        array_push($response[GUASTI], $guasto);   // inserisce il guasto nell'array
                }
                $response[AUTOMOBILE] = $tag_uid;
                $response[SUCCESS] = 1;
                echo json_encode($response);                    // invia la risposta
        }
}
?>
```

3.6 Modifica ed eliminazione di un elemento

Lo schema è lo stesso utilizzato fin qui, la query MySQL prenderà in input l'identificativo univoco che rappresenta l'oggetto selezionato.

Ecco un esempio sull'eliminazione di una manutenzione, il file delete_manutenzione elimina l'elemento dalla tabella conosciuto il suo indirizzo (assegnato al momento della creazione dal DBMS stesso):

```
<?php
$response = array();

if (isset($_POST[INDIRIZZO])) {
   $id = $_POST[INDIRIZZO];

   $result = mysql_query("DELETE FROM manutenzioni WHERE id='$id'");

   if ($result) {
      $response[SUCCESS] = 1;
      $response[MESSAGE] = "manutenzione eliminata con successo";
      echo json_encode($response);

   } else {
      $response[SUCCESS] = 0;
      $response[MESSAGE] = "manutenzione non trovata";
      echo json_encode($response);
   }
} else {
   $response[SUCCESS] = 0;
   $response[MESSAGE] = "campo delete manutenzione mancante";
   echo json_encode($response);
}
?>
```

Di seguito invece il codice per la modifica di un viaggio, il file update_viaggio permette di modificare la data di arrivo e le note di un viaggio passandogli come parametro l'indirizzo dello stesso:

```
<?php
$response = array();

if(isset($_POST[INDIRIZZO]) && isset($_POST[DATA_FINE]) && isset($_POST[NOTE]) ) {

        $id = $_POST[INDIRIZZO];
        $data_fine = $_POST[DATA_FINE];
       $note = $_POST[NOTE];

        $result = mysql_query("UPDATE viaggi SET data_fine='$data_fine', note='$note'
                                                WHERE   id='$id'");
        if ($result) {
                $response[SUCCESS] = 1;
                $response[MESSAGE] = "viaggio aggiornato";
                echo json_encode($response);
        } else {
                $response[SUCCESS] = 0;
                $response[MESSAGE] = "errore, viaggio non aggiornato";
                echo json_encode($response);
        }
} else {
        $response[SUCCESS] = 0;
```

```
        $response[MESSAGE] = "campo update viaggio mancante";
        echo json_encode($response);
}
?>
```

Fonti

- Documentazione ufficiale Android, reperibile dal sito “developers. android.com”
- Zigurd Mednieks, Laird Dornin, G. Blake Meike & Masumi Nakamura - “Programming Android” - O'REILLY
- Documentazione Php sul sito “www.php.net”
- Documentazione Nfc sul sito “www.nfc-forum.org”
- Documentazione Google Maps sul sito “developers.google.com /maps”
- Carta di circolazione dell’automobile sul sito “www.up.aci.it”

Ringraziamenti

Giunto finalmente alla fine di questo percorso formativo ringrazio innanzitutto la mia famiglia, in particolar modo i miei nonni, i miei genitori e mia zia, che mi hanno sempre sostenuto e permesso gli studi universitari. Un ringraziamento particolare alla mia ragazza che mi è stata sempre accanto nei momenti di sconforto e mi ha spronato a proseguire negli studi. Infine ringrazio tutti gli amici che mi hanno aiutato ma anche tutti coloro i quali mi hanno dimostrato affetto e creduto nelle mie potenzialità.

Colgo l'occasione di ringraziare il relatore Prof. Gianni Orlandi e il suo assistente Prof. Andrea Proietti per i consigli e la disponibilità avuta nei miei confronti.

Grazie di cuore

Lorenzo Orsi