POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Sviluppo di un'applicazione Android per gli utenti di un servizio di car-sharing

Relatore
prof. Giovanni Malnati

Candidato
Jessica Fenudi

Supervisore Aziendale
ing. Paolo Doz

Anno accademico 2017-2018

Alla mia famiglia

Sommario

Il presente lavoro di tesi mira ad analizzare e progettare il prototipo di un'applicazione mobile per Android rivolta agli utenti di un servizio di car-sharing. Verranno studiati i requisiti generali richiesti da un tale servizio, con particolare attenzione alle funzionalità di base che l'applicativo mobile dovrà offrire; saranno valutate le principali applicazioni create ad hoc per i servizi di car-sharing attualmente attivi in Italia. Verranno presentati gli strumenti messi a disposizione dal sistema Android e le librerie che hanno permesso lo sviluppo di alcune delle funzionalità integrate nel prototipo in esame. Infine, saranno analizzati i risultati raggiunti e i possibili miglioramenti da apportare in futuro al fine di ottenere un prodotto di qualità.

Indice

Introduzione

La crescente diffusione delle auto private come modalità di trasporto preferita dalla collettività urbana, per il maggior comfort offerto, la sua maggiore flessibilità e capillarità, ha reso indispensabile la riorganizzazione generale del sistema dei trasporti nelle città, con interventi volti a ridurre i livelli d'inquinamento atmosferico e la congestione stradale. L'esigenza di soluzioni di mobilità alternative è stata motivata anche dalla scarsa efficienza dei servizi di trasporto pubblico, i quali risultavano inadeguati a soddisfare le necessità dell'utenza, perché spesso numericamente insufficienti o addirittura carenti riguardo orari e tratte da percorrere. È in questo contesto che si iniziano a studiare servizi di mobilità innovativi allo scopo di cambiare il modo in cui gli individui soddisfano i propri bisogni di spostamento, per sviluppare un modello di mobilità sostenibile. In Italia i numerosi interventi richiesti alle amministrazioni locali per la riduzione dell'inquinamento e della congestione nelle città, hanno aperto la strada allo sviluppo di nuove forme di mobilità sostenibile, in cui il fattore dominante è la condivisione anziché l'utilizzo individuale dei mezzi.

Negli ultimi anni è stata registrata una forte crescita ed evoluzione dei sistemi di mobilità condivisa, supportata dall'innovazione tecnologica che ha reso l'accesso ai servizi e ai mezzi in condivisione molto più semplice e veloce, grazie anche all'utilizzo dei dispositivi mobili quali smartphone e tablet, senza la necessità di richiedere l'assistenza da parte di personale dedicato. Inoltre, lo sviluppo di piattaforme digitali flessibili e adattabili a qualsiasi tipo di servizio di mobilità condivisa, ha contribuito alla crescita di questo settore, permettendo a numerose società e organizzazioni, sia pubbliche che private, di usufruire di queste soluzioni pronte all'uso per realizzare la propria offerta. Questo ha portato alla diffusione di svariate tipologie di servizi, tra cui il bike-sharing, il car-sharing e altri, che prevedono la condivisione di un mezzo da utilizzare in base alle necessità, risparmiando dal punto di vista economico e ottenendo gli stessi benefici del mezzo di proprietà.

In particolare il car-sharing è un servizio che prevede il noleggio di un'automobile da parte di diversi individui, i quali la utilizzano in sequenza e per periodi di tempo generalmente brevi, volto a soddisfare differenti esigenze di spostamento, laddove le soluzioni già presenti in ambito locale non siano adeguate, oppure in mancanza del mezzo di proprietà. Il car-sharing come forma organizzata di condivisione dell'automobile ha iniziato a svilupparsi in Svizzera già dal 1948, pur essendo al tempo caratterizzato da elementi differenti rispetto a quelli acquisiti attualmente. Si è diffuso in Italia all'inizio degli anni 2000 con l'obiettivo di ridurre l'uso eccessivo dell'auto privata e favorire il miglioramento della vita nelle città. L'originalità introdotta dal car-sharing e in generale da tutti i servizi di mobilità condivisa, sta nell'acquisto dell'uso dei veicoli e non dei veicoli stessi. Si passa dunque ad un modello di consumo di beni e servizi fondato sull'accesso anziché sulla proprietà.

Attualmente il mercato offre diverse soluzioni create su misura per una certa tipologia di auto, talvolta con veicoli progettati e realizzati appositamente per il servizio, come nel caso di Sharen'go, un progetto di CS Group frutto della collaborazione tra l'Italia e la

Cina, che ha portato alla realizzazione delle city car elettriche ZD1 e oggi anche ZD2, destinate sia al car-sharing che all'uso privato. Esistono anche produttori di automobili che hanno promosso il proprio servizio di car-sharing come la società tedesca Daimler AG, proprietaria del marchio Mercedes-Benz, che ha creato il servizio Car2Go presente ormai in diverse città del mondo. Car2go si basa su una flotta mono-prodotto di Smart Fortwo e Smart Forfour con allestimenti speciali. In Italia si è diffuso enormemente anche il servizio *Enjoy*, il prodotto di *Eni*, pressoché simile a *Car2go*, ma con alcune differenze riguardanti soprattutto le tariffe e le auto utilizzate; in questo caso la scelta infatti è ricaduta sulle Fiat 500 e Fiat Doblò Cargo.

Tali servizi richiedono la progettazione di una piattaforma tecnologica completa e adattabile alla specifica soluzione da attuare, che includa gli strumenti utili sia agli operatori di car-sharing, che devono tenere sotto controllo le proprie vetture in tempo reale, per garantire un sistema quanto più efficiente possibile, sia agli utenti che usufruiscono del servizio. È proprio a questo livello che Abinsula intende sviluppare la sua idea di business, realizzando un progetto in grado di supportare le attività di un qualunque operatore di car-sharing che voglia inserirsi in un mercato in continua crescita ed evoluzione come questo, per offrire servizi dedicati al consumatore finale oppure alle aziende.

L'azienda

Abinsula s.r.l. è un'azienda operante nel settore informatico dal 2012, anno in cui è stata fondata da alcuni professionisti, specializzati nella progettazione e sviluppo di software per dispositivi *embedded*. L'azienda, distribuita tra le sedi di Sassari, Torino, Cagliari e Barcellona, si occupa della realizzazione di sistemi e applicazioni per diversi settori: in particolare quello *automotive*, *mobile*, *smart agriculture*, *utility* e l'IoT per l'industria (IIoT), avvalendosi soprattutto di tecnologie e architetture open-source. Negli ultimi anni essa si è distinta per lo sviluppo di software nell'ambito *automotive* grazie alla sua notevole capacità di offrire soluzioni strategiche e innovative; infatti, alcuni tra i suoi più importanti clienti di questo settore, hanno deciso di affidarle diverse fasi critiche di sviluppo, all'interno di progetti destinati a grandi produttori automobilistici europei ed internazionali. Inoltre, Abinsula ha provveduto a formare un team stabile di Ricerca e sviluppo, in grado di creare e manutenere una propria distribuzione linux open-source, chiamata Ability, che ha le caratteristiche adatte ai dispositivi *embedded* (utilizzo ridotto di memoria e alta configurabilità). È utilizzata come piattaforma di base per numerosi progetti, che spaziano dai sistemi di *infotainment* destinati alle automobili, fino ad arrivare ai 'smart meters' (contatori della luce intelligenti) per il settore delle *utility*.

Partendo proprio dalle sue esperienze nella progettazione di architetture hardware e software in campo *automotive*, Abinsula ha maturato l'idea di sviluppare un proprio progetto sperimentale e di ricerca industriale, denominato *Splash* (*System PLatform for cAr SHaring*), destinato al settore del car-sharing. Il progetto è fondato sulla necessità di studiare e realizzare un prodotto standard, composto da tutti gli elementi utili per sostenere questo servizio, che si contraddistingua nell'essere configurabile e adattabile secondo le richieste del potenziale cliente che in questo caso rivestirà il ruolo di gestore del servizio.

Con il presente lavoro si vuole progettare e realizzare la configurazione che l'applicazione mobile per Android [1], prevista dalla piattaforma Splash e destinata all'utente finale, dovrà avere in prima istanza, da adattare e personalizzare successivamente in base alle esigenze degli specifici clienti. Nel Capitolo 1 verrà descritto il fenomeno della *sharing*

mobility nel suo complesso per contestualizzare lo sviluppo e la diffusione del car-sharing come strumento di mobilità sostenibile. Inoltre verranno presentati i requisiti generali del servizio e una descrizione sui principali servizi attivi in Italia. Nel Capitolo 2 verrà presentata l'architettura generale del sistema Splash progettata da Abinsula, per conoscere i dispositivi e le tecnologie a supporto del servizio, con i quali l'applicazione mobile dovrà interagire. In questo capitolo sarà riportata l'analisi dei requisiti di base definiti per tale applicazione. Nel Capitolo 3 sarà valutato il linguaggio Kotlin impiegato per il suo sviluppo: verranno analizzate le caratteristiche principali del linguaggio e il supporto offerto in particolare per il framework Android. Inoltre, verrà presentata la struttura logica dell'applicazione e la progettazione delle funzionalità previste dalla stessa. La descrizione dell'architettura interna dell'app e l'implementazione di alcune delle funzionalità presenti, invece, saranno oggetto del Capitolo 4, in cui sarà fornita una descrizione approfondita del codice sviluppato. Verranno descritti gli strumenti utilizzati durante lo sviluppo dell'applicazione nel suo complesso considerando sia le principali librerie adottate, sia lo strumento Git fondamentale per il versionamento del codice sorgente. Infine nel Capitolo 5 verrà fornita un'analisi della soluzione realizzata mettendo in luce i punti di forza e fornendo alcuni spunti per sviluppi futuri.

Capitolo 1

Sharing mobility e Car-sharing

1.1 Concetti generali

Con il termine *sharing mobility* o mobilità condivisa si indica una tendenza in continua crescita ed evoluzione che si è affermata in molte città del mondo e consiste nell'utilizzo di un mezzo di trasporto (automobile, bicicletta, scooter ecc.) in condivisione con altre persone. Lo sviluppo di strumenti di mobilità urbana condivisa è legato alla diffusione di un fenomeno ancora più ampio che va sotto il nome di *sharing economy* o economia della condivisione, il quale promuove un modello di consumo collaborativo di beni e servizi piuttosto che il possesso degli stessi: i consumatori utilizzano prodotti e accedono ai servizi in maniera temporanea quando ne hanno bisogno, pagando per il loro utilizzo e con gli stessi benefici della proprietà. Inoltre, la diffusione di servizi di mobilità condivisa è stata motivata dall'attenzione rivolta da enti locali, nazionali e internazionali verso nuove forme di spostamento sostenibile per affrontare temi quali l'inquinamento atmosferico e acustico delle città e la congestione del traffico stradale, che affliggono ormai da tempo i grandi centri urbani. Il modello tradizionale della mobilità urbana, caratterizzato dal trasporto collettivo di massa, non essendo in grado di rispondere alle richieste sempre più esigenti degli utenti in termini di capillarità degli spostamenti, orari, flessibilità e comfort, ha subito una profonda crisi finanziaria che ha portato ad una conseguente riduzione degli investimenti e quindi ad un abbassamento della qualità del servizio offerto. Questo ha spinto gli utenti a preferire la modalità di trasporto individuale e dunque all'utilizzo massivo delle auto private causando un impatto negativo sull'ambiente e sulla qualità della vita urbana.

Nell'ottica dello sviluppo di un sistema di trasporto sostenibile il fenomeno della *sharing mobility* riveste un ruolo fondamentale, insieme ai numerosi provvedimenti delle amministrazioni pubbliche che spingono i cittadini a preferire l'utilizzo di mezzi di trasporto elettrici, ibridi e meno inquinanti per contribuire alla riduzione delle emissioni, oppure che scoraggiano totalmente l'utilizzo del mezzo privato preferendo soluzioni alternative quali quelle offerte dal trasporto pubblico locale o dai servizi in condivisione. Il contributo decisivo che la *sharing mobility* può dare in questo senso è quello di creare le condizioni per indurre le persone a cambiare i loro comportamenti e le loro scelte in termini di spostamenti, orientandoli verso una nuova cultura della mobilità basata su servizi innovativi che sono condivisi e integrati tra loro.

Il cambiamento promosso dalla *sharing mobility* consiste nella valutazione per ciascun spostamento del mezzo di trasporto migliore sia dal punto di vista individuale, considerando fattori quali il tempo di percorrenza e i costi di viaggio, sia dal punto di vista generale

causando minori impatti e ottenendo una maggiore efficienza; talvolta le soluzioni di trasporto migliori possono includere i servizi pubblici, l'andare a piedi oppure in bicicletta, non solo l'utilizzo di mezzi condivisi. Se si analizza il comportamento degli individui nel modello di trasporto individuale, si può affermare che i possessori di un mezzo privato lo usano principalmente per la sua versatilità e la comodità offerta nella pianificazione del viaggio: l'auto privata è sempre disponibile e il proprietario ogni volta che ne ha necessità organizza in autonomia lo spostamento. Inoltre, possedere un veicolo implica che lo si utilizzi quanto più possibile per qualsiasi esigenza di mobilità, un pò per abitudine e un pò perché non si ha una vera percezione dei costi di un viaggio effettuato con il proprio mezzo; infatti, spesso si tiene conto soltanto dei costi diretti, tra cui quelli del carburante o del pedaggio e questo risulta in un calcolo errato delle proprie convenienze che impedisce di valutare soluzioni più efficienti in termini economici e di tempo.

Il vantaggio dei servizi in condivisione invece è la trasparenza dei costi, per cui gli individui hanno la possibilità di programmare uno spostamento conoscendo quanto si paga realmente per una certa tratta; tale valutazione può portare ad esempio alla minore percorrenza di chilometri non necessari e all'utilizzo più efficiente dei veicoli (si utilizzano quando non esistono soluzioni alternative o vantaggiose). Questo approccio moderno alla mobilità risulta benefico per l'ambiente e la società in generale, consente un maggior risparmio economico e una rivalutazione del trasporto pubblico e di altre modalità: chi compra o scambia un servizio di mobilità e abbandona il mezzo privato può scegliere il treno per spostarsi dalla periferia verso il centro città, la bicicletta nel caso debba compiere brevi tragitti, oppure i veicoli condivisi come quelli offerti dal car-sharing nel caso di spostamenti occasionali o a media e bassa frequenza per percorrere medie distanze. Per superare il modello di mobilità individuale è necessario offrire non solo servizi innovativi ed efficienti ma puntare alla multi-modalità, ovvero garantire che per ogni spostamento si possa accedere alla soluzione migliore. In questo senso la tecnologia ha un ruolo determinante nel suggerire all'utente la combinazione ottimale tra vari sistemi di trasporto in funzione dello spostamento. Esistono già degli applicativi detti aggregatori o *Multimodal Journey Planner*, i quali integrano in un'unica soluzione i diversi servizi di mobilità offerti sia dai principali operatori di *sharing mobility*, sia dalle aziende di trasporto pubblico, mettendoli a confronto secondo parametri quali i costi e i tempi di percorrenza, ma anche quello dell'impatto ambientale che ne deriva dal loro utilizzo.

Da un'analisi del 1° rapporto nazionale sulla *sharing mobility* [2] presentato dall'Osservatorio Nazionale Sharing Mobility, progetto promosso dal Ministero dell'Ambiente e della Tutela del Territorio e del Mare insieme alla Fondazione per lo Sviluppo Sostenibile, sono stati individuati i servizi di mobilità condivisa attivi in Italia fino al 2016 che nel corso dell'anno 2017 si sono espansi ed evoluti grazie proprio all'innovazione tecnologica, secondo quanto riportato anche nell'ultimo rapporto nazionale prodotto nel 2018 e relativo al 2017 [3]. Infatti, rispetto al passato alcuni dei servizi analizzati hanno subito una radicale trasformazione. L'Osservatorio[1] è un'iniziativa nata come piattaforma di collaborazione tra operatori di servizi di *sharing mobility*, istituzioni pubbliche e private e mondo della ricerca, con l'obiettivo di analizzare, sostenere e promuovere il fenomeno della mobilità condivisa (l'osservatorio ad oggi vanta circa 80 membri). Oggetto dell'analisi presentata nel 1° rapporto nazionale sono la mappatura completa delle tipologie di servizi di *sharing mobility* attivi nel territorio italiano, la descrizione del loro funzionamento ottenuta dai dati forniti direttamente dai componenti dell'Osservatorio e la valutazione dell'impatto di

[1] fonte:
http://osservatoriosharingmobility.it/

questi servizi sulla sostenibilità ambientale. Il numero di servizi di *sharing mobility* in Italia [3, sezione 2.1], considerando tutte le tipologie presenti tra il 2015 e il 2017, è cresciuto di una quota pari mediamente al 17% ogni anno, per un totale di 357 servizi erogati al 31 dicembre 2017. Considerando nello specifico le diverse zone d'Italia, nel territorio del sud è stata registrata la crescita più forte con il 57% di servizi attivi in più al termine del periodo esaminato, mentre per il centro e per il nord l'aumento è stato del 31%. Tuttavia, il nord è la zona che vanta il maggior numero di servizi di mobilità condivisa, con Milano che è la città in cui, sia in termini di offerta di servizi che di domanda di mobilità condivisa, vi è la maggior diffusione della *sharing mobility* a livello nazionale.

Le caratteristiche che un servizio di mobilità condivisa deve avere per essere considerato tale sono diverse e l'Osservatorio Nazionale ne ha individuato cinque [2, sezione 1.2]. La condivisione di un mezzo di trasporto tra più individui è la caratteristica chiave nei servizi di *sharing mobility*; essa può avvenire in maniera contemporanea, ovvero quando più persone si trovano su un mezzo nello stesso momento e percorrono in parte o interamente lo stesso tragitto. Oppure esiste la condivisione in sequenza e prevede che il mezzo sia utilizzato a rotazione da diverse persone. In queste due modalità ricadono tutte le forme di trasporto in cui non si fa uso del mezzo di proprietà per gli spostamenti, tra cui anche il taxi, il servizio di noleggio con conducente, l'autobus e il treno. La seconda caratteristica su cui deve basarsi un servizio di *sharing mobility* è la tecnologia. Avere a disposizione un'intera piattaforma Web è importante non solo per la fruibilità del servizio, il quale diventa più veloce ed efficiente, ma anche per il perfezionamento complessivo dell'offerta e per ragioni di scalabilità. Le piattaforme digitali infatti fanno si che i servizi crescano e si trasformino nel tempo in modo da soddisfare la domanda di mobilità. Per i gestori del servizio è utile analizzare i dati sugli spostamenti degli utenti e sul loro grado di soddisfazione così da offrire soluzioni in linea con le loro necessità. Il terzo aspetto che determina il successo di un servizio di mobilità condivisa e che consente di sorpassare il modello individuale è la disponibilità d'uso dei mezzi secondo la necessità, sia in termini di tempo che di fasce orarie (di giorno, di notte, nei giorni festivi e feriali). La disponibilità va intesa anche come numero di mezzi offerti insieme alle diverse tipologie di veicoli che compongono la flotta: di base si tratta di mezzi nuovi dotati di tecnologie di ultima generazione e le persone possono scegliere tra quelle disponibili a seconda delle esigenze, ad esempio preferendo le piccole auto per gli spostamenti in città, i furgoni per effettuare traslochi, gli scooter o le biciclette per le zone con difficoltà di parcheggio. Oltre alla disponibilità e flessibilità d'uso, una caratteristica importante nei servizi di *sharing mobility* è il concetto di *community*. Si tratta di un elemento fondamentale che mette in contatto sia gli utenti del servizio tra loro, sia gli stessi con gli operatori in modo che essi collaborino per la costruzione e il miglioramento del servizio stesso; alcune tipologie di servizi ad esempio prevedono che l'utente contribuisca alla personalizzazione del tracciato di base secondo le proprie esigenze di spostamento, così come alla scelta della tariffa migliore. Non tutte le forme di condivisione però basano il proprio funzionamento interamente sulla *community*; tuttavia essa è sempre presente come elemento di socialità per cui gli utenti possono interagire per scambiarsi informazioni e anche fare segnalazioni ai gestori (molti servizi oggigiorno sono attivi sui social network). L'ultima caratteristica attribuibile ad un servizio di *sharing mobility* è lo sfruttamento della massima capacità dei veicoli che, al contrario, nel modello di trasporto individuale rimane spesso inutilizzata; un'auto privata tipicamente viaggia con una o due persone a bordo quando invece essa è progettata per un massimo di quattro o cinque e talvolta anche sette occupanti (a seconda dell'auto), oppure viene lasciata parcheggiata per la maggior parte della sua vita utile. I servizi in condivisione mirano ad utilizzare la capacità residua di un veicolo, ad esempio, aggiungendo il maggior numero di passeggeri ad uno stesso viaggio, oppure trasportando

quanti più utenti possibili a rotazione nell'arco di una giornata, di un mese o di un anno.

1.2 I servizi di sharing mobility in Italia

Prima di presentare nello specifico il car-sharing è utile conoscere e analizzare le diverse tipologie di servizi in condivisione finora diffuse nel territorio italiano, seppur in diversa misura. In questo modo è possibile comprendere come tali servizi vengano realizzati nel rispetto delle cinque caratteristiche evidenziate dal 1° rapporto nazionale. Le tipologie di servizi sono le seguenti:

Bike-sharing è un servizio in cui le biciclette di una flotta vengono condivise tra gli utenti che ne hanno sottoscritto l'apposito abbonamento o che hanno versato una cauzione. Generalmente le bici sono dislocate in vari punti di un centro urbano, di un campus universitario, di un parco naturale o di un'area turistica e vengono collocate in apposite rastrelliere a formare una stazione, oppure esse vengono disposte all'interno dell'area operativa senza l'utilizzo di stalli. Tra i due casi appena descritti cambia la modalità di prelievo e restituzione delle bici. Per i servizi basati sulle stazioni fisse le biciclette possono essere individuate immediatamente (spesso sono tutte dello stesso colore e le stazioni risultano segnalate da appositi cartelli), oppure si utilizza lo smartphone per individuarle su di una mappa se le stazioni sono georeferenziate. Nel caso di servizi a flusso libero che non prevedono stazioni, ciascuna bicicletta è dotata di un sistema di localizzazione per cui anche in questo caso si trovano facilmente tramite app. Il sistema di sblocco/blocco è differente per le due modalità d'uso: le stazioni fisse servono a bloccare le bici e gli utenti iscritti utilizzano la propria *smartcard* per sbloccarle, invece nell'altro caso ciascuna bicicletta è dotata di un sistema di blocco/sblocco con il quale l'utente interagisce via applicazione mobile scansionando ad esempio un codice QR per sbloccarla, oppure digitando il codice riportato sulla bici. La tariffazione del servizio inizia generalmente nel momento in cui la bicicletta viene sbloccata e si basa sulla durata di utilizzo delle biciclette (tariffe orarie o variabili a seconda dei minuti di percorrenza o del tipo di abbonamento). In alcuni servizi che operano nel territorio nazionale è possibile prenotare le bici in anticipo tramite l'applicazione mobile. Esistono anche servizi di bike-sharing tra privati (detti peer-to-peer) in cui la condivisione avviene tramite una piattaforma digitale che permette di registrare la bici che si intende condividere, inserire la foto e il prezzo così che gli interessati possano noleggiarla effettuando una richiesta di prenotazione.

Car-sharing come nel caso del bike-sharing esistono principalmente due tipologie di car-sharing, ovvero quello tradizionale detto *station-based*, in cui le auto da condividere sono parcheggiate in stalli riservati e collocati in diversi punti strategici della città, e quello che si è affermato negli ultimi anni detto *free-floating*. I veicoli possono essere localizzati e prenotati dalle apposite applicazioni Web o mobile grazie al fatto che essi sono dotati di un sistema GPS. Nella tipologia *free-floating* i veicoli possono essere prelevati e rilasciati in qualsiasi punto della città all'interno dell'area operativa del servizio. Invece, nel caso *station-based* si distinguono i sistemi di tipo *round-trip*, dove l'auto va prelevata e riconsegnata presso lo stesso stallo, da quelli *one-way* in cui le due operazioni possono essere svolte anche in stazioni differenti. Grazie alla tecnologia il car-sharing si è evoluto nel tempo e le flotte messe a disposizione dai vari operatori vantano oggi un maggior numero di veicoli. Sono presenti anche dei

modelli di car-sharing un pò meno diffusi, tra i quali troviamo il modello peer-to-peer, che permette al proprietario di un veicolo di affittarlo ad altri utenti attraverso l'utilizzo di una piattaforma Web, il modello cosiddetto di nicchia, che comprende servizi dedicati ai campus universitari (per studenti e insegnanti) e in ultimo quello pensato per le aziende (corporate car-sharing). In Italia ad oggi la modalità di condivisione peer-to-peer non è contemplata espressamente dal codice della strada, pertanto alcune soluzioni presenti tentano di adeguarlo al modello di servizio di noleggio senza conducente, il quale invece è ammesso per legge. Generalmente, nel car-sharing la condivisione dei veicoli avviene per un breve periodo di tempo con la possibilità di estenderlo fino ad un massimo numero di ore consecutive nel caso di servizi *free-floating*; invece, le soluzioni a stazione fissa permettono di prenotare un'auto anche per più giorni. A seconda della tipologia di servizio e dello specifico operatore è previsto il pagamento di una quota d'iscrizione e successivamente si paga in base all'utilizzo dei veicoli. Il sistema di tariffazione è generalmente diverso per le due modalità: nel caso *station-based* vengono applicate tariffe variabili in base ai chilometri percorsi e alle ore di utilizzo del veicolo, mentre nel modello *free-floating* si hanno tariffe al minuto (e anche al chilometro nel caso si sforino i limiti di utilizzo consentiti). Per lo sblocco delle auto nelle due tipologie di servizio si utilizza una carta magnetica (consegnata agli iscritti) da appoggiare sull'apposito lettore del veicolo (posto ad esempio sul parabrezza), oppure si utilizza direttamente l'applicazione mobile per velocizzare l'operazione. La rete di un servizio di car-sharing ha generalmente un'estensione urbana che può diventare talvolta regionale o nazionale.

Ride-sharing/Car-pooling riguarda la condivisione di automobili private tra un gruppo di persone, le quali devono percorrere uno stesso itinerario e contribuiscono al pagamento delle spese di trasporto sostenute dal conducente del veicolo che ne è anche il proprietario. La piattaforma Web è fondamentale perché permette al conducente del mezzo che deve effettuare un viaggio di offrire un passaggio ad altre persone interessate a percorrere lo stesso tragitto. Inoltre, grazie ad essa si crea una vera e propria comunità di persone che insieme decidono come organizzare il viaggio e possono conoscere le esperienze degli altri utenti in merito ai diversi conducenti; esiste sempre un sistema di *feedback* in questa tipologia di servizi. In genere è necessario iscriversi in maniera gratuita per avere accesso alla *community* e consultare/creare le offerte di viaggio: il conducente inserisce la propria offerta e il sistema propone l'importo più adeguato da richiedere ai passeggeri, che tiene conto dei costi di carburante, pedaggio e delle altre spese relative allo spostamento. L'obiettivo di questa tipologia di servizi è il risparmio, non quello di trarre profitto dai viaggi: ad esempio la politica adottata dal servizio BlaBlaCar, che sta dominando il mercato italiano e quello europeo, è di questo tipo. I vari servizi si distinguono in base al tipo di tratta percorsa (tratte urbane e extraurbane) e per la tipologia di utenti, tra cui privati, gruppi di privati e aziende. Si sono diffusi diversi servizi di car-pooling, alcuni ideati per il trasporto di pendolari tra la provincia e la città, altri per il trasporto di lavoratori che condividono il tragitto casa-lavoro. Permettono di coprire destinazioni che spesso non sono direttamente o facilmente raggiungibili con il servizio di trasporto pubblico locale.

Servizi a domanda appartengono a questa tipologia diversi servizi di mobilità condivisa, alcuni dei quali prevedono l'impiego di un veicolo personale per trasportare i passeggeri verso una specifica destinazione; si parla in questo caso di Ride-sourcing che si distingue dal Ride-sharing perché i conducenti non condividono la destinazione

con i loro passeggeri ma sono degli autisti dotati di patenti specifiche e di autorizzazioni di tipo noleggio con conducente. Tramite le applicazioni mobile è possibile conoscere ancor prima del viaggio gli autisti, le auto e si può scegliere anche di condividere la tratta con altre persone. Appartengono a questa tipologia anche i servizi di Ride-splitting, in cui l'equipaggio può formarsi prima del viaggio tra gruppi di persone che intendono condividere le spese, ma si può formare dinamicamente durante la corsa a seguito della richiesta degli utenti che richiedono il ritiro; in questo caso il percorso cambia in base alle prenotazioni ricevute da altri utenti.

Microtransit e Navette sono servizi di trasporto condiviso privato che prevedono l'utilizzo di navette o furgoni e vengono forniti in due modalità: servizi a percorso e orari fissi, oppure servizi a percorsi flessibili con pianificazione su richiesta[2]. Anche in questo caso è previsto l'utilizzo di una piattaforma digitale tramite la quale gli utenti possono effettuare le richieste di trasporto, contribuendo in questo modo a rendere il servizio ancora più flessibile per il fatto che il tracciato viene adattato dinamicamente in base al numero di richieste.

Con l'avanzamento tecnologico è possibile che le tipologie di servizi appena descritte subiscano delle trasformazioni e che si sviluppino in futuro nuove forme di mobilità sostenibile e condivisa.

1.3 Car-sharing: nascita e diffusione

Il termine car-sharing deriva dall'inglese 'automobile condivisa' e indica un servizio di trasporto urbano con il quale più persone, a turno, condividono i veicoli di una flotta usufruendone per un certo tempo e pagando in ragione dell'utilizzo effettuato[3]. La prima forma di car-sharing nasce in Svizzera dalla cooperativa Sefage che nel 1948 aveva avviato a Zurigo un servizio di auto condivisa principalmente dovuto a ragioni economiche: le persone che non potevano permettersi di possedere un'auto ne condividevano una. In seguito si diffondono vere e proprie organizzazioni di car-sharing su ampia scala, più efficienti rispetto ai primi tentativi, mosse da ragioni di carattere ambientalista: è il caso delle esperienze di Zurigo e Lucerna dove, nel 1987, alcuni cittadini elvetici attenti alle tematiche ambientali e al risparmio energetico, condividevano già veicoli in multiproprietà [4, pag 30]. Successivamente il nuovo stile di mobilità proposto dal car-sharing inizia a diffondersi in altre realtà europee ed extra europee, diventando un vero e proprio servizio di trasporto complementare a quello pubblico, proponendosi già da allora come sostituto dell'auto privata (inizialmente per le seconde e terze auto di proprietà).

In Italia il car-sharing comincia ad essere preso in considerazione con il decreto del Ministero dell'Ambiente del 27 marzo 1998 [5, art. 4] rivolto ai comuni, i quali venivano invitati a *'promuovere e sostenere forme di multiproprietà delle autovetture destinate ad essere utilizzate da più persone, dietro pagamento di una quota proporzionale al tempo d'uso ed ai chilometri percorsi'*, per ridurre l'inquinamento dei centri urbani. La prima esperienza di un tale servizio si è avuta a Milano nel 2001 grazie all'iniziativa dell'associazione ambientalista Legambiente che proponeva una soluzione riservata unicamente ai

[2] fonte:
https://en.wikipedia.org/w/index.php?title=Shared_mobility&oldid=846219576

[3] fonte:
https://it.wikipedia.org/w/index.php?title=Car_sharing&oldid=94649498

membri. Questo è anche l'anno in cui parte il programma *Iniziativa Car Sharing* (*ICS*)[4], un accordo tra comuni ed enti locali, incoraggiato e finanziato dal Ministero dell'Ambiente con l'obiettivo di favorire la diffusione del car-sharing nelle città italiane, nel rispetto di uno standard nazionale a sostegno di un nuovo approccio alla mobilità: innovazione, efficienza e sostenibilità costituivano la base di questo programma. Il ruolo di *ICS* non era quello di erogare direttamente il servizio di car-sharing, ma la sua funzione era quella di incentivare e supportare l'avvio e la creazione di questi servizi da parte di associazioni pubbliche e private o da parte delle imprese, per la diffusione del fenomeno di *sharing mobility* a favore della sostenibilità ambientale nelle diverse città del territorio nazionale. Il car-sharing, dunque, è un servizio progettato per offrire una misura di trasporto alternativa ad altre esistenti, con un minor impatto sull'ambiente e sulla salute dei cittadini. In quest'ottica ecologista si intende svincolare l'utilizzo del veicolo dalla sua proprietà, con un conseguente risparmio in termini di costi fissi di gestione del veicolo, senza però rinunciare alla flessibilità del possesso privato. Aderirono alla convenzione *ICS* diverse realtà tra comuni, città metropolitane e province. I servizi nati nelle città iscritte all'iniziativa *ICS* (molte delle quali lo sono ancora), pur essendo gestiti da operatori differenti e non da uno unico a livello nazionale, sono tenuti al rispetto di certi standard: ad esempio è necessario garantire l'interoperabilità del circuito, ovvero che un utente iscritto al servizio in una città possa utilizzarlo anche in un'altra città in cui lo stesso è attivo. Questo viene realizzato tramite la condivisione da parte dei diversi servizi delle stesse tecnologie, tra cui gli strumenti di prenotazione, gli applicativi per gli utenti e altre procedure. La modalità di condivisione diffusa dai servizi di car-sharing del circuito *ICS* è quella *station-based*. I principali comuni che hanno aderito all'iniziativa 'IoGuido' di *ICS* sono stati diversi, alcuni dei quali oggi hanno chiuso l'attività, mentre altri risultano operativi ad esempio nelle città di Roma, Palermo e Parma. Il servizio 'Io Guido' della società Car City Club di Torino, comune che è stato uno tra i primi ad aderire al programma di *ICS*, è stato definitivamente interrotto nel 2017 con l'arrivo nel mercato di operatori privati. Inoltre, occorre menzionare anche i servizi GirACI, car-sharing avviato dalla società Aci Global collegata ad Aci, che prevedevano postazioni fisse in svariate città italiane (alcune delle quali oggi non offrono più tale servizio) e garantivano in più l'interoperabilità con 'IoGuido' di *ICS*. Per quanto riguarda Milano, nel 2004 la società GuidaMi erogava un car-sharing di tipo *station-based*. La sua gestione è passata nel 2015 ad Aci Global, mentre oggi è stata acquisita dalla start-up Ubeeqo che ha mantenuto il modello a postazioni fisse, ma offre una soluzione diversa orientata sia al consumatore finale, sia alle aziende.

Il programma realizzato da *ICS* ha posto le basi per lo sviluppo del car-sharing in Italia, che tuttavia fino al 2013 vantava ancora una bassa incidenza di questo servizio come modalità di trasporto alternativo, rispetto alle altre realtà nel mondo. Il 2013 è stato l'anno in cui ha iniziato a diffondersi la modalità di car-sharing a flusso libero o *free-floating*, offerta da operatori privati, la quale ha cambiato decisamente il trend di evoluzione del car-sharing nel territorio nazionale, attirando più utenti e facendo crescere esponenzialmente il numero di noleggi e di auto disponibili nelle città. L'innovazione tecnologica ha sicuramente favorito questa crescita con l'introduzione di nuovi sistemi per lo sblocco delle auto e per le prenotazioni. Un aspetto determinante nel successo di questa nuova tipologia riguarda soprattutto la rinnovata modalità d'uso delle auto nelle procedure di prelievo e rilascio. Infatti, come già descritto in precedenza, nel *free-floating* si ha una maggiore flessibilità e facilità di fruizione del servizio, considerando che l'utente

[4] fonte:
https://www.icscarsharing.it/ics/

non è costretto a restituire l'auto nella stazione di origine dello spostamento, ma è libero di parcheggiarla in qualsiasi area consentita, purché nel rispetto delle condizioni imposte dal servizio stesso. Milano è stata la prima città ad offrire un servizio a flusso libero con l'ingresso nel mercato degli operatori Enjoy e Car2Go, che in seguito si sono diffusi nella città di Roma, per poi affermarsi anche a Firenze e Torino. La società Enjoy eroga il servizio anche a Catania. Nell'anno 2015 è entrato nel mercato anche il servizio di car-sharing elettrico Sharen'go, dapprima a Milano e a seguire nelle città di Firenze, Roma e Modena. Sempre riguardo il car-sharing elettrico, a Torino opera dal 2016 il servizio Bluetorino che a differenza di Sharen'go è basato sul modello *station-based* con stazioni di noleggio a disposizione in alcuni punti della città. DriveNow è un altro servizio di tipo *free-floating* invece, attualmente attivo soltanto a Milano. Dai dati riportati nel 1° rapporto nazionale sulla *sharing mobility* (lo stesso trend è stato registrato anche con le analisi del 2° rapporto) è emerso che la modalità di condivisione *free-floating*, offerta dagli operatori privati, ha riscontrato un grande successo in poche grandi città, rispetto al servizio tradizionale *station-based* che invece, a livello territoriale, ha avuto una maggiore espansione considerando anche i centri italiani meno importanti [2, sezione 4.2.6, pag 105]; ne è un esempio il servizio E-Vai con postazioni distribuite a livello regionale in Lombardia, collocate nei principali punti delle province come stazioni ferroviarie, ospedali o aeroporti. Tuttavia, i servizi tradizionali vantano flotte numericamente più piccole e un numero di noleggi inferiore. Per quanto riguarda le analisi sugli spostamenti effettuati con i veicoli nei due modelli di condivisione, emerge che gli utenti preferiscono il servizio *free-floating* per percorrere brevi tragitti, i quali avvengono principalmente all'interno delle aree urbane, mentre nel caso *station-based* i veicoli vengono impiegati per le distanze medio-lunghe, con spostamenti anche extra-urbani. Confrontando la flotta di veicoli offerta nelle due modalità di car-sharing oggi in Italia, rispetto a quanto riportato nel 1° rapporto nazionale [2, sezione 4.2.6, pag 103], la situazione è rimasta pressoché invariata, a meno di alcune eccezioni: nel modello *station-based* la flotta dei veicoli è eterogenea, le auto si differenziano per produttore, tipo (utilitaria, station wagon e furgone), ma anche in termini di alimentazione. Invece, i servizi *free-floating* sono costituiti per la maggior parte da flotte omogenee. In alcuni casi, essi offrono diversi modelli di auto dello stesso produttore, a seconda della città in cui operano (Car2Go a Torino e Milano offre solo Smart Fortwo e Forfour, mentre a Roma è presente anche il modello cabrio). Fa eccezione il servizio DriveNow, il quale dispone di un parco auto con otto modelli di auto differenti tra BMW e MINI.

Requisiti generali del car-sharing

Per sviluppare un servizio di car-sharing al pari di quello erogato dai principali leader operanti oggi nel settore sono necessari diversi elementi, tra beni e servizi, in grado di rispondere alle esigenze dei due attori principali di questo sistema: la società che gestisce la flotta dei veicoli adibiti al servizio e gli utenti utilizzatori dello stesso. Il gestore deve poter registrare i veicoli specificandone le caratteristiche, conoscere la loro posizione in tempo reale, recuperare i dati sulla diagnostica del veicolo per monitorare continuamente il loro stato interno, tra cui il livello del carburante nel serbatoio (o di carica della batteria in caso di veicoli elettrici), il livello dell'olio motore, la pressione degli pneumatici e altri parametri come lo stato degli airbag e delle lampadine. Ottenendo quante più informazioni possibili, il gestore può intervenire tempestivamente sia per effettuare la manutenzione ordinaria dei veicoli, sia nel caso di malfunzionamenti o eventi di incidente, il tutto finalizzato all'ottimizzazione della flotta e del servizio offerto. Inoltre deve essere in grado di interagire

con il veicolo per effettuare lo sblocco delle portiere da remoto e dell'avviamento del motore, il quale risulta bloccato di default per motivi di sicurezza.

Considerando dall'altro lato gli utenti, anch'essi devono avere a disposizione gli strumenti adatti ad effettuare la ricerca dei veicoli, le prenotazioni e devono poter sbloccare le portiere del veicolo dal momento che le chiavi non vengono consegnate a loro direttamente, ma si trovano all'interno dell'auto. Come si è già evidenziato in precedenza, sfruttando la tecnologia, oggigiorno tutte queste operazioni possono essere svolte accedendo al sito Web dell'operatore di car-sharing e in maniera più semplice e veloce sfruttando il proprio smartphone in cui è installato l'applicativo mobile fornito dal gestore. Ancor prima di utilizzare il servizio gli utenti devono potersi iscrivere fornendo i propri dati e la documentazione richiesta. In questo modo, una volta registrati, il gestore può identificarli nelle fasi di prenotazione e sapere in qualsiasi momento quale utente sta utilizzando un certo veicolo della flotta, per poter garantire un servizio efficiente e disponibile. Inoltre, per ogni prenotazione il gestore deve poter conoscerne lo stato (attiva/conclusa), la durata e i chilometri percorsi con il veicolo interessato, così da poter procedere al calcolo e alla fatturazione delle somme dovute dagli utenti a fronte del loro utilizzo.

Analizzando il servizio di car-sharing nel suo complesso emergono, dunque, una serie di requisiti generali che richiedono l'impiego di una piattaforma informatica e telematica di supporto costituita dai dispositivi hardware/software da installare a bordo dei veicoli, i quali devono poter essere connessi alla rete Internet per consentire al gestore di comunicare con essi e agli utenti di utilizzarli. La piattaforma deve includere altresì il sistema centrale per la gestione delle iscrizioni, delle prenotazioni, dei pagamenti e in generale di tutto il servizio. Inoltre devono essere previsti gli applicativi software dedicati al gestore e agli utenti per l'interazione con il servizio e la piattaforma stessa. Per i dispositivi da installare a bordo devono poter essere previste differenti tecnologie: ad esempio nel caso di servizi *station-based* sono necessari dei lettori di tipo *contactless* per garantire agli utenti l'accesso al mezzo tramite *smartcard* fornita dal gestore, utili anche come soluzione di backup per i servizi che prevedono l'utilizzo dello smartphone durante le operazioni di apertura e chiusura dei veicoli.

Oltre all'aspetto tecnologico, che ha un ruolo dominante nella diffusione di un servizio di *sharing mobility*, bisogna considerare altri aspetti gestionali e organizzativi importanti che permettono agli operatori di portare il servizio nelle città, di farlo funzionare al meglio e di integrarlo con la mobilità urbana. Per erogare un servizio di car-sharing nelle città è necessario ottenere una delibera da parte del comune interessato che regoli il funzionamento del sistema: ad esempio il comune si impegna di riservare delle aree di parcheggio esclusive per i veicoli nelle flotte *station-based* e di consentire agli utenti di utilizzare gratuitamente i parcheggi pubblici nel caso di servizi a flusso libero, garantendo anche l'accesso a zone normalmente chiuse al traffico. Per quanto riguarda le operazioni di gestione e manutenzione dei veicoli (e degli eventuali punti di stazionamento se previsti), il gestore necessita di personale che operi a livello locale e che sia a disposizione della clientela. In ultimo, secondo quanto previsto dalla legge, il gestore deve dotare le auto di una copertura assicurativa definendo l'importo del massimale che copre i danni provocati a terzi, della eventuale franchigia da addebitare all'utente in caso di danni al veicolo a seguito di incidenti o di furto e incendio. La polizza assicurativa può includere anche garanzie accessorie a favore degli utenti che usufruiscono dei veicoli, quali ad esempio la copertura degli infortuni per il conducente e altri servizi.

1.4 Obiettivo della tesi

L'obiettivo di questa tesi è quello di contribuire alla realizzazione di un'applicazione mobile per Android che possa supportare gli utenti nell'utilizzo di un servizio di car-sharing. Si tratta di un prototipo capace di fornire le funzionalità di base previste dal servizio, che comprendono la registrazione degli utenti, la localizzazione dei veicoli, lo sblocco delle serrature senza l'utilizzo di una chiave e la prenotazione degli stessi; in aggiunta si vuole sperimentare la funzionalità di rifornimento del carburante. Questo prototipo verrà sfruttato da Abinsula per testare il funzionamento dell'intero sistema Splash e sarà successivamente perfezionato per presentarlo ai clienti che intenderanno fornirsi della piattaforma per creare il proprio servizio di car-sharing.

L'idea di Abinsula si basa sulla fornitura di un prodotto personalizzabile secondo le esigenze di business del cliente. La personalizzazione della piattaforma in linea di massima è prevista sia per le singole funzionalità, sia per gli elementi che la compongono e può avvenire a diversi livelli: sono possibili modifiche ai dispositivi di bordo (ad esempio per la funzionalità di sblocco delle portiere) per adattarli ai veicoli della specifica flotta, ma anche allo stile grafico del portale Web e della stessa applicazione mobile, mantenendo la logica di basso livello inalterata. Se la flotta è già dotata delle apparecchiature di bordo e dunque il cliente intende utilizzare solo il software di front-end e l'applicazione mobile di Abinsula, sarà necessario adattare il software del server centrale in base alle interfacce previste dalle specifiche apparecchiature. Per quanto riguarda l'applicazione mobile sarà compito di Abinsula modificarla su richiesta dei clienti, perfezionando e aggiungendo eventuali funzionalità se necessario, nel caso in cui essi intendano integrarla nel proprio servizio. In alternativa, il cliente potrà sviluppare separatamente la propria applicazione sfruttando l'accesso alle API della piattaforma Splash.

1.5 Principali servizi di Car-sharing in Italia

In questa sezione vengono analizzate le soluzioni offerte dai principali operatori di car-sharing in Italia al momento della scrittura di questa Tesi, al fine di evidenziare le funzionalità delle applicazioni mobile prodotte per lo specifico servizio. Di seguito vengono presentati i servizi appartenenti alla tipologia *free-floating*, alcuni dei quali prevedono flotte omogenee di veicoli, altri invece offrono un parco auto diversificato. Inoltre, verranno analizzate le soluzioni offerte da due operatori che erogano servizi di tipo *station-based* per evidenziare alcune differenze.

1.5.1 Enjoy e Car2Go

Confrontando i servizi Enjoy[5] e Car2go[6] si nota come essi offrano funzionalità piuttosto simili. Per usufruire dei due servizi innanzitutto è necessario effettuare l'iscrizione, che nel caso di Car2go deve avvenire unicamente tramite il sito ufficiale (dall'app infatti si viene redirezionati sulla pagina Web); invece, la patente e i documenti possono essere forniti tramite app. Enjoy accetta l'iscrizione dal sito Web e dall'app ufficiale. Per iscriversi è

[5] fonte:
https://enjoy.eni.com/it/

[6] fonte:
https://www.car2go.com/IT/it/

richiesto l'inserimento di dati personali, tra cui un documento di identità (carta d'identità o passaporto), il codice fiscale, la data e il luogo di nascita, la patente di guida e i dati relativi ad una carta di pagamento. L'iscrizione al servizio è gratuita per Enjoy, mentre per Car2go è richiesto il pagamento di una quota, dopodiché per entrambi i servizi è necessario pagare unicamente per i viaggi effettuati. Per l'utilizzo dei veicoli gli utenti hanno a disposizione un codice PIN personale assegnato all'atto della conferma d'iscrizione. La tariffa di base proposta dai due servizi è al minuto e nel caso di Enjoy è prevista una tariffa al chilometro se si supera il limite massimo di chilometri consentito per noleggio. Agli utenti che hanno necessità di un'auto più a lungo Car2go offre dei pacchetti orari che consentono in questi casi di risparmiare rispetto alla tariffa al minuto. Per entrambi i servizi esiste un limite di tempo massimo per i noleggi fissato a 24 ore consecutive.

Le prenotazioni possono essere effettuate in anticipo dal sito Web (solo per Enjoy) o direttamente dall'applicazione: l'utente sceglie il veicolo d'interesse sulla mappa mostrata e ne visualizza i dettagli (luogo di ritrovamento e distanza, livello serbatoio, ecc.). Le due applicazioni permettono di raggiungere facilmente i veicoli mostrando la strada più veloce direttamente sulla mappa, ma non presentano un sistema di navigazione integrato. È possibile anche richiedere il noleggio dei veicoli incontrati lungo la strada senza la necessità di effettuare una prenotazione; entrambe le applicazioni supportano questa funzionalità. Considerando il sistema di sblocco dei veicoli, Enjoy ha implementato una soluzione differente per i due casi appena descritti: se l'utente ha effettuato una prenotazione, nel momento in cui ha raggiunto l'auto può richiederne l'apertura tramite il pulsante presente nell'app. Altrimenti, in assenza di prenotazione, esso deve inserire il numero di targa e il codice del veicolo a 4 cifre mostrato sul parabrezza . Invece, per le auto della flotta Car2go lo sblocco delle portiere richiede innanzitutto il codice PIN dell'utente e successivamente un codice a 3 cifre mostrato sul dispositivo telematico posizionato sul parabrezza. A questo punto il noleggio viene attivato e l'utente può utilizzare il veicolo. Nel servizio Enjoy il PIN è necessario per l'accensione del motore e va inserito sull'app dopo che il veicolo è stato sbloccato. Una volta usufruito dei veicoli, gli utenti sono tenuti a parcheggiarli nelle aree previste dai due servizi, purché si trovino all'interno dell'area operativa (tipicamente si può parcheggiare in tutti i parcheggi pubblici e anche in quelli riservati al servizio nel caso di Enjoy).

Un'altra funzionalità offerta agli utenti da entrambi gli operatori, è il rifornimento del carburante, pur essendo uno tra i servizi già inclusi di base nelle tariffe. Car2go permette di effettuare l'operazione presso le stazioni Q8 convenzionate che si trovano nell'area operativa, le quali sono facilmente individuabili dall'applicazione. Il rifornimento è gratuito e prevede il solo utilizzo della carta carburante fornita da Car2go, che si trova a bordo del veicolo. L'uso della carta permette a Car2go di tracciare l'operazione: l'utente è tenuto a digitare il PIN della carta nel terminale della stazione di servizio (mostrato sull'app o nel computer di bordo se presente) e il numero di chilometri del veicolo. Si nota come non avviene nessuna interazione dell'utente con l'applicazione mobile. La funzionalità di rifornimento consente agli utenti di guadagnare dei bonus nel caso si effettui il pieno ai veicoli con un certo livello di carburante. In questo modo gli utenti contribuiscono a migliorare l'efficienza del servizio.

Diverso è il caso di Enjoy in cui l'esecuzione del rifornimento prevede l'interazione tra l'utente e l'applicazione mobile. Le auto da rifornire vengono identificate da un'icona. Ci si reca presso la stazione Eni più vicina alla posizione del veicolo segnalata direttamente sulla mappa e si utilizza la procedura guidata presente nell'app; tramite app l'utente seleziona l'erogatore presso cui desidera effettuare il pieno e procede con il rifornimento. Il sistema è in grado di calcolare autonomamente il numero di litri necessari per riempire il

serbatoio e terminerà la procedura in automatico, segnalando opportunamente eventuali anomalie. Enjoy abilita l'operazione di rifornimento solamente agli utenti che abbiano effettuato noleggi per un certo importo e accredita dei 'voucher' di spesa per le successive prenotazioni, se l'operazione è stata eseguita correttamente. Anche in questo caso il prezzo del carburante è totalmente a carico di Enjoy. Non è previsto nessun rimborso per gli utenti che effettuano il rifornimento utilizzano una propria carta di pagamento.

1.5.2 DriveNow

DriveNow è un servizio di tipo *free-floating* erogato da BMW e attivo in Italia solamente a Milano. Rispetto ai due servizi descritti in precedenza il parco auto offerto comprende otto modelli di veicoli composto da BMW e MINI, tra cui uno completamente elettrico. L'iscrizione richiede gli stessi dati elencati in precedenza e avviene compilando il modulo disponibile sul sito. Dopodiché l'utente è tenuto a scansionare i documenti e la patente di guida direttamente dall'app. La quota d'iscrizione al servizio è più alta rispetto a Car2go così come le tariffe al minuto. Esistono diversi pacchetti adatti alla percorrenza di tragitti più lunghi (differenziati per auto): alcuni sono giornalieri, altri orari e includono un numero massimo di giorni, oppure ore consecutive di utilizzo, nonché un limite massimo di chilometri percorribili. Sono previsti dei pacchetti convenienza anche per i clienti abituali. È possibile prenotare un veicolo in anticipo tramite l'applicazione mobile dopo averlo scelto tra i veicoli più vicini proposti in automatico all'apertura dell'app. Dopodiché per aprire l'auto si utilizza l'app premendo l'apposito pulsante, il quale risulterà abilitato solamente se ci si trova ad una certa distanza dal veicolo. Oppure lo sblocco delle portiere si può effettuare tramite la carta clienti fornita da DriveNow come soluzione di backup, da avvicinare al lettore collocato sul parabrezza; in questo caso si verifica l'interazione utente-veicolo a differenza del precedente. Per l'accensione del veicolo è necessario digitare il PIN assegnato all'utente esclusivamente sul computer di bordo. Anche DriveNow permette di noleggiare i veicoli trovati lungo la strada senza necessità di effettuare una prenotazione; dal sito ufficiale si evince che l'operazione funziona con la sola carta cliente e non tramite app. Per il rilascio e la chiusura dei veicoli si utilizza sia l'app che la carta clienti. Le auto DriveNow possono essere rifornite o ricaricate presso le stazioni convenzionate utilizzando la carta fornita, consentendo agli utenti di ottenere minuti bonus se il livello del serbatoio o di carica raggiunge una certa soglia al termine del noleggio. L'operazione non richiede invece l'utilizzo dell'applicazione mobile.

L'applicazione DriveNow offre la possibilità di trovare facilmente il veicolo prenotato sfruttando il servizio Maps e rispetto ai servizi Enjoy e Car2go si può utilizzare la funzionalità 'Flash' che fa lampeggiare i fari dell'auto per identificarla su strada. Inoltre è possibile programmare in anticipo un viaggio per un'auto prenotata: basta digitare la destinazione nell'apposita sezione dell'app e automaticamente il sistema trasmette l'informazione al navigatore satellitare del veicolo, così da consultarlo durante il viaggio.

1.5.3 Sharen'go

Sharen'go[7] è un servizio di car-sharing a flusso libero attivo a Milano, Roma, Firenze e Modena con una flotta omogenea di soli veicoli elettrici dotati di cambio automatico. Al

[7] fonte:
https://site.sharengo.it/

pari dei servizi descritti in precedenza è richiesta l'iscrizione con gli stessi dati (l'iscrizione è a pagamento salvo convenzioni particolari), da eseguire sul sito Web o dall'app. I veicoli possono essere prenotati in anticipo e sbloccati tramite smartphone (non è previsto l'inserimento di codici ulteriori) oppure passando la carta Sharen'go sul lettore posto sul parabrezza. La carta può essere utilizzata per aprire direttamente un'auto trovata per strada, se disponibile, senza la necessità di prenotarla. Per mettere in moto è richiesto l'inserimento del codice PIN dell'utente sul display di bordo. Il PIN può essere recuperato facilmente sull'app oppure dal sito ufficiale. Il costo di ciascuna corsa viene calcolato tenendo conto di tariffe fisse al minuto, all'ora e al giorno, ma si possono acquistare dei pacchetti al minuto. Inoltre, per incentivare gli utenti ad utilizzare maggiormente il servizio, Sharen'go offre la possibilità di accumulare dei punti omaggio, un punto per ogni minuto di noleggio, da utilizzare successivamente nelle corse. Per i veicoli Sharen'go non è disponibile la funzionalità di ricarica ma l'operazione viene eseguita esclusivamente dagli operatori del servizio.

1.5.4 Bluetorino

Bluetorino[8] è un servizio di car-sharing *station-based* costituito da auto elettriche dello stesso tipo, progettate dal Gruppo Bolloré, con stazioni dislocate nei punti strategici della città di Torino. Il principio di funzionamento è diverso dai servizi analizzati in precedenza. Per accede al servizio è necessario essere abbonati. L'abbonamento si può effettuare dal sito Web oppure presso i chioschi self-service disponibili nelle aree di parcheggio Bluetorino; sono previsti abbonamenti giornalieri, settimanali, mensili e annuali. Le prenotazioni dei veicoli vengono gestite sia dall'applicazione mobile, sia dal sito Web, oppure chiamando il call center. L'applicazione consente di localizzare e prenotare il parcheggio di destinazione in cui terminare il noleggio. A differenza dei servizi descritti sopra, per l'apertura e la chiusura dei veicoli è possibile utilizzare unicamente la tessera Bluetorino ottenuta a seguito dell'abbonamento. La procedura di sblocco del veicolo avviene in due passaggi: l'utente avvicina la tessera alla colonnina di ricarica alla quale è collegato il veicolo, dopodiché posiziona la tessera sull'apposito lettore collocato al lato dell'auto in modo da poterla aprire. Per restituire il veicolo e terminare il noleggio il procedimento è lo stesso.

1.5.5 Ubeeqo

Ubeeqo[9] è una start-up nata in Francia che dal 2015 è diventata proprietà del Gruppo Europcar, azienda leader nei servizi di autonoleggio in Europa. Il servizio di car-sharing Ubeeqo è attivo in diversi paesi europei e in Italia è presente esclusivamente a Milano, a seguito dell'acquisizione da parte di Europcar della società GuidaMi che offriva già un servizio *station-based*. Il car-sharing Ubeeqo prevede postazioni fisse per le auto, le quali devono essere utilizzate in modalità *round-trip*, pertanto esse vanno prelevate e riconsegnate nel medesimo parcheggio. Tale servizio offre un parco auto differenziato e adatto a soddisfare svariate esigenze di spostamento, principalmente di lunga durata. È prevista un'iscrizione al servizio e le prenotazioni possono essere richieste accedendo alla pagina personale sul sito Web, oppure dall'app, così come avviene nella maggior parte

[8] fonte:
https://www.bluetorino.eu/

[9] fonte:
https://www.ubeeqo.com/it/

dei servizi. Tuttavia, il sistema di prenotazione di Ubeeqo è completamente differente da quello dei servizi analizzati finora, in quanto per prenotare un veicolo è necessario definire il periodo e la durata del noleggio ancor prima della partenza. Infatti, al momento della prenotazione il sistema richiede di specificare la data e l'orario di prelievo e di rilascio del veicolo, nonché il numero massimo previsto di chilometri da percorrere. In questo modo anche il costo del noleggio è noto in anticipo e il pagamento avviene all'atto di conferma della prenotazione. Si ha la possibilità di prenotare fino ad un mese prima della partenza scegliendo l'auto e il parcheggio dal quale la si vuole prelevare. È possibile effettuare prenotazioni per un tempo minimo di un'ora sino ad un massimo di 48 ore, con tariffe orarie diversificate per veicolo, consentendo così agli utenti di utilizzarli anche al di fuori dell'area urbana, per effettuare viaggi regionali e nazionali. Lo sblocco delle auto può essere eseguito tramite applicazione oppure usando la tessera clienti e per effettuare il rifornimento del carburante, anche in questo, si può utilizzare la carta carburante presente all'interno dei veicoli.

Capitolo 2

Il progetto Splash

2.1 Architettura generale del sistema

Splash è la piattaforma, aperta e modulare, che Abinsula intende sviluppare per la gestione di un sistema di car-sharing: come si è già detto in precedenza, tale servizio richiede numerose centraline che riconoscano gli utenti, che permettano di localizzare i veicoli e supportino le attività di prenotazione, il prelievo e il rilascio dei veicoli e i pagamenti per l'utilizzo degli stessi. Inoltre, è necessario progettare un sistema informatico e telematico in grado di recuperare le informazioni sullo stato dei veicoli, utili all'operatore di car-sharing per eseguire operazioni di gestione e monitoraggio della flotta; ciò richiede la presenza di applicativi Web per la sala di controllo. Nello specifico, la piattaforma Splash comprende gli sviluppi seguenti: un'architettura hardware completa del modulo di comunicazione con l'auto e di quello telematico per l'accesso alla rete, un modulo software *embedded* per il monitoraggio dello stato del veicolo, un server centrale, un pannello Web dedicato all'operatore per la gestione dell'intera flotta, degli utenti e dei pagamenti, e infine l'applicativo mobile da fornire agli utenti che dovranno accedere al servizio.

La configurazione attuale dell'architettura e le funzionalità ad ora disponibili rendono il progetto Splash adatto all'implementazione di servizi *free-floating*, non essendo prevista al momento la gestione di stazioni di servizio. La soluzione è utilizzabile per sistemi di medio-grandi dimensioni e può essere adottata da aziende che rivendono il proprio servizio di car-sharing a privati, ma anche dalle aziende che intendono offrire un servizio riservato ai propri collaboratori e dipendenti.

Abinsula ha identificato due possibili modelli di business per la piattaforma Splash:

on-premises viene venduta l'intera soluzione già personalizzata per lo specifico operatore di car-sharing e il software di gestione viene installato direttamente sui server del cliente. Si prevede un prezzo fisso per l'intera soluzione;

versione cloud in questo caso la gestione del sistema avviene sui server di Abinsula che offre il servizio in modalità SaaS (Software as a Service), con pagamento annuale che dipenderà dalla dimensione della flotta, oppure dal numero di noleggi registrati.

2.1.1 Unità installata a bordo del veicolo

Di seguito vengono riportate le caratteristiche principali individuate per un'architettura hardware di massima, la quale dovrà essere in grado di gestire contemporaneamente i tre

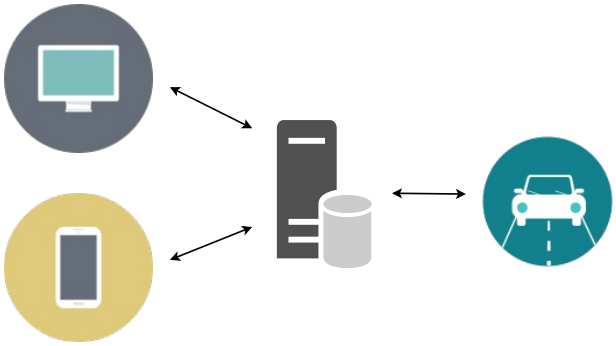


Figura 2.1: Architettura del sistema

aspetti fondamentali di una *On Board Unit* (*OBU*) per un'infrastruttura di car-sharing: il sistema di *infotainment*, il modulo per la comunicazione real-time con le periferiche dell'auto e quello telematico per comunicare con la sala di controllo. L'architettura hardware prevede i seguenti componenti:

- CPU Core basata su processore iMX-6.
- sistema di comunicazione a corto raggio: Bluetooth
- sistema di comunicazione a lungo raggio: 3G con opzione per upgrade a LTE
- sensori di movimento e posizione: GPS e accelerometro a 3 assi
- Display TFT 7" (opzione touch o tasti funzione in cornice da valutarsi)

Software embedded

Per il funzionamento dei componenti precedenti è necessario realizzare il relativo software che comprende:

- il software di base, che consiste nel porting del sistema Ability sullo specifico HW
- il software per HW real-time
- il modulo per la gestione della SIM e la comunicazione in rete
- lo sviluppo dell'interfaccia uomo-macchina (HMI) realizzata in linguaggio Qt

2.1.2 Server

Al centro della piattaforma troviamo il server incaricato di servire le richieste che arrivano dai vari client, in questo caso rappresentati dall'applicazione mobile e dal pannello di controllo Web, ma anche della gestione del database a supporto del sistema di car-sharing, nonché della comunicazione con il veicolo. La messa in opera di questi servizi è affidata

a due macro-componenti: un Web Service RESTFul che espone le API utilizzate dall'applicazione mobile e dal pannello Web per interagire con l'intero sistema e un componente responsabile della comunicazione con il veicolo. Quest'ultima avviene tramite il protocollo MQTT e prevede l'inoltro di comandi specifici verso il veicolo e la ricezione del suo stato; il componente è costituito da un MQTT broker responsabile della gestione di una coda di messaggi e da un modulo ulteriore che permette al Web Service di rimanere in ascolto di eventuali messaggi in arrivo sulla coda.

MQTT-Broker

MQTT è un protocollo di livello applicativo che si appoggia al TCP/IP e viene utilizzato, come già detto, per realizzare la comunicazione tra il veicolo e il server, la quale deve essere limitata il più possibile al fine di ridurre il traffico dati generato e incidere al minimo sui costi di connessione e sulla reattività del sistema. Tale protocollo, dunque, rispetta sia le specifiche dei dispositivi installati sui veicoli, per i quali è richiesto un basso consumo, sia quelle della rete a cui essi vengono interconnessi, la quale è caratterizzata da una bassa larghezza di banda; inoltre, esso riesce a garantire affidabilità d'invio e ricezione dei dati. In questo progetto è stato utilizzato il formato JSON per costruire il payload dei messaggi MQTT, ma è possibile inserire dati in altri formati (testo semplice, binario, altro). La peculiarità del protocollo MQTT sta nel tipo di comunicazione richiesta, che non avviene secondo il modello client-server di HTTP, ma il meccanismo adottato è di tipo *publish-subscribe*, che si avvale di un broker per lo scambio dei messaggi MQTT, sia per inviare appositi comandi dal server verso il veicolo, sia per riceverne da quest'ultimo lo stato (posizione GPS, stato portiere, livello carburante, altro). In particolare, esiste il concetto di 'topic' o argomenti, per cui i messaggi relativi ad un certo argomento (ad esempio 'apertura-portiere') vengono pubblicati (publish) su una coda gestita dal broker; spetta a chi è interessato iscriversi (subscribe) e ogni volta che viene pubblicato un nuovo messaggio relativo ad un determinato argomento, il broker lo distribuisce a tutti quelli lo hanno sottoscritto. Questo ha permesso di realizzare una comunicazione asincrona con il dispositivo a bordo del veicolo e di disaccoppiare il produttore e l'utilizzatore del dato (chi genera il dato non deve essere direttamente connesso e/o attivo nello stesso momento di chi lo utilizza).

Web Service REST

Questo componente è stato realizzato tramite l'utilizzo del Django REST framework[1], che ha consentito di sviluppare e documentare le API REST fruibili tramite pannello accessibile via Web, di attuare politiche di autenticazione e autorizzazione, nonché di utilizzare strumenti per la serializzazione e la validazione degli input. Inoltre, esso ha permesso di sfruttare il servizio di *Object-relational mapping* (ORM) per la mappatura degli oggetti presenti nel dominio dell'applicazione e le relative entità relazionali nel database gestito lato server.

Le Web API sono suddivise in 5 gruppi principali, ciascuno dedicato ad una delle funzionalità fondamentali previste dal sistema, tra cui l'autenticazione delle richieste effettuate dai vari client, la gestione degli account utente, le operazioni di ricarica portafoglio

[1] fonte:
http://www.django-rest-framework.org/

e i pagamenti, la ricerca dei veicoli, la gestione delle prenotazioni e dei rifornimenti di carburante. Django offre differenti meccanismi di autenticazione standard, con la possibilità di realizzarne uno personalizzato: il presente Web Service si avvale di uno schema di autenticazione basato su un *token* HTTP, ciò significa che tutte le richieste eseguite dai client dovranno includere il *token* rilasciato dal server nella prima fase di accesso e configurazione. Rispettando i principi REST, tutte le operazioni e i dati su cui esse operano, sono accessibili attraverso l'esecuzione di richieste HTTP verso particolari end-point: si tratta di risorse identificate da un URI univoco, per le quali vengono definite le operazioni consentite espresse attraverso i metodi standard previsti dal protocollo HTTP. Infatti, per ciascuna operazione delle quattro CRUD disponibili nei database relazionali è previsto il relativo metodo HTTP: *Create* corrisponde all'operazione POST per la creazione di una nuova risorsa, *Read* viene gestita tramite il metodo GET per ottenere una risorsa esistente, *Update* corrisponde al metodo PUT e prevede l'aggiornamento di una risorsa o la modifica del suo stato e infine *Delete*, realizzabile tramite il metodo DELETE, per la cancellazione di una risorsa. È opportuno notare che tutte queste operazioni consentono esclusivamente il trasferimento di una rappresentazione delle risorse presenti/da aggiungere sul server: durante le interazioni i dati vengono scambiati nel formato JSON che è basato su testo. Inoltre, così come previsto dal protocollo HTTP, le risposte ai metodi eseguiti sulle singole risorse includono un codice di stato per indicare il successo o l'insuccesso della richiesta; in questo modo i client possono prepararsi a gestire i vari casi. Più avanti, nella presentazione di ciascuna funzionalità, verranno mostrate in dettaglio le risorse a cui si deve fare accesso e quali informazioni sono previste per le relative richieste e risposte.

2.1.3 Pannello di controllo Web

L'applicativo Web è accessibile unicamente dall'operatore per il monitoraggio e la gestione della flotta e degli utenti. Viene realizzato con le moderne tecnologie Web (HTML5, jQuery) al fine di garantire un'esperienza utente semplice, moderna e positiva. Tra le funzionalità previste troviamo:

- gestione flotta: consente di esaminare il parco auto e di amministrarlo con la possibilità di aggiungere in qualsiasi momento nuove auto specificandone per ciascuna i dettagli (modello, targa, tipologia del motore, ecc.)
- gestione veicolo: per ciascun veicolo si ha la possibilità di visualizzarne le caratteristiche descritte sopra e in aggiunta la posizione geografica in tempo reale. Sarà possibile monitorarne lo stato, tra cui il livello di carburante nel serbatoio e dell'olio motore, così come altri dati da consultare per effettuare tempestivamente la manutenzione. L'operatore potrà aprire forzatamente le portiere da remoto nel caso in cui la stessa funzionalità sia temporaneamente non disponibile sull'applicazione mobile e l'utente abbia contattato il numero del servizio clienti. Inoltre, sarà possibile ottenere per ciascun veicolo le informazioni sui viaggi per i quali è stato utilizzato (data e ora di utilizzo, chilometri percorsi e tempo impiegato, numero totale viaggi per veicolo, ecc.)
- localizzazione veicoli su mappa: consente di visualizzare tutti i veicoli direttamente su una mappa, in modo da trovare facilmente il luogo in cui si trovano (parcheggiati o in uso)
- gestione utenti: è possibile accedere alla lista di tutti gli utenti registrati al servizio per consultare alcune informazioni personali (nome, cognome, residenza, ecc.),

tracciare le prenotazioni e i noleggi effettuati, nonché i rifornimenti e i pagamenti eseguiti per gli stessi

2.1.4 Applicazione mobile

Abinsula ha previsto, nell'ottica di un primo rilascio del prodotto, un sistema di accesso al servizio tramite applicazione per smartphone Android. Successivamente, sarà resa disponibile anche la versione per iOS e verrà valutata la possibilità di sviluppare per altre piattaforme mobile. Analogamente ai più moderni sistemi di car-sharing basati su app, tale componente è stato pensato principalmente per la localizzazione degli utenti e la ricerca dei veicoli della flotta, lo sblocco delle portiere e altre funzionalità indispensabili che verranno presentate in dettaglio nell'analisi dei requisiti seguente.

2.2 Analisi dei requisiti applicazione mobile

La fase di analisi dei requisiti è fondamentale per la buona riuscita di un software, ne precede il vero e proprio sviluppo. Visto che il lavoro di tesi mira alla realizzazione della configurazione base dell'applicazione mobile, di seguito verrà presentata l'analisi dei requisiti funzionali minimi di tale applicazione così com'è stata svolta in azienda, partendo dall'individuazione dei potenziali utenti finali del servizio e dai loro bisogni, fino ad arrivare alla definizione delle funzionalità che il software dovrà fornire. Ciò che ne scaturisce è la specifica dei requisiti, un documento che farà da guida per le successive fasi di progettazione e implementazione del prodotto. Nell'ottica di un primo rilascio, alcune delle funzionalità analizzate nel presente lavoro verranno ampliate e perfezionate in modo da rendere l'applicazione più matura e completa, al pari delle soluzioni già offerte dai principali operatori di car-sharing. Ulteriori funzionalità verranno esaminate e implementate da Abinsula qualora considerate una miglioria del prodotto e che ne consentono una maggiore 'rivendibilità' dello stesso.

Come già detto all'inizio, il car-sharing è un servizio che è nato per agevolare e migliorare la mobilità degli utenti che lo utilizzano e in primo luogo verrebbe da pensare che il principale utilizzatore di questo servizio è colui che viaggia continuamente per lavoro, vicino o lontano alla propria città, e che non possiede un'automobile. Sicuramente essi rappresentano una grossa percentuale degli utenti totali. Tuttavia, la diffusione del car-sharing è stata resa possibile da fattori quali il crescente inquinamento nelle grandi città, gli alti costi del carburante, l'inefficienza del sistema di trasporto pubblico e/o la sua distribuzione non uniforme, che hanno spinto sempre più persone a ricercare metodi di spostamento alternativi e sostenibili. Si tratta in generale di clienti che vogliono spostarsi in autonomia a qualsiasi ora del giorno e della notte, senza preoccuparsi di lasciare la propria auto incustodita. A questa categoria di utenti appartengono anche i collaboratori di aziende pubbliche e private, i quali possono accedere alle auto che la propria azienda gestisce in condivisione anche al di là delle attività aziendali stesse.

Il servizio Splash e di conseguenza l'applicazione mobile si rivolge a tutti coloro che hanno bisogno di spostarsi e cercano una soluzione alternativa alla vettura di proprietà o al servizio di trasporto pubblico locale se non soddisfa le esigenze di spostamento, a prezzi adeguati alle loro capacità di spesa. Tra questi abbiamo individuato:

lavoratori utilizzano i servizi di auto condivisa per raggiungere la propria sede di lavoro. Vivono soprattutto in zone centrali o semi-centrali e utilizzano il servizio per brevi spostamenti;

studenti usufruiscono delle auto condivise saltuariamente per recarsi al luogo di studio, all'aeroporto oppure alla stazione ferroviaria (soprattutto nel caso di studenti fuori sede);

turisti sono persone che si trovano in visita occasionale nella città, cercano un servizio alternativo al trasporto pubblico (autobus, taxi) e al classico noleggio;

dipendenti d'azienda utilizzano i veicoli della flotta aziendale condivisa, nel caso di aziende che dedicano il proprio parco auto ai propri collaboratori e dipendenti, per spostamenti anche di medio-lunga durata.

Dopo aver riconosciuto i possibili consumatori e citato alcuni dei probabili contesti di utilizzo, vediamo quali sono le funzionalità dell'applicazione che si vuole sviluppare e le informazioni richieste agli utenti.

Il servizio permetterà unicamente ai clienti registrati di prenotare in anticipo il proprio veicolo e di utilizzarlo per tutto il tempo desiderato. Al termine del noleggio l'utente potrà parcheggiare l'auto in tutti gli spazi pubblici consentiti e accessibili a tutti. Non sarà possibile utilizzare parcheggi privati di qualsiasi tipo oppure lasciare l'auto al di fuori dell'area in cui opera il servizio. Nel caso di un gestore particolare, ad esempio un'azienda, tali vincoli potrebbero essere soggetti a cambiamenti e sarà l'azienda stessa a definirli, ad esempio prevedendo un parco auto di sua proprietà o specificando privilegi e facilitazioni particolari. L'utente potrà registrarsi inserendo un *nickname* identificativo, il proprio nome e cognome, un indirizzo email valido e una password. A registrazione avvenuta esso avrà un profilo personale, da completare specificando le sue generalità, tra cui la data, il luogo, lo stato e la provincia di nascita e infine il codice fiscale; al momento, considerato che l'applicazione viene sviluppata come prototipo per valutare l'intera piattaforma, tra i requisiti non sono previste le informazioni sui documenti di identità degli utenti e sulla patente di guida. Le informazioni di base sul profilo utente potranno essere modificate in qualsiasi momento dalla stessa app. Per ogni utente, verrà creato un portafoglio elettronico basato su crediti, necessari per avviare le prenotazioni ed effettuarne il pagamento. L'applicazione mostrerà in una lista le varie opzioni di acquisto crediti, indicando per ciascuna di esse il numero di crediti e l'importo da pagare. Nella configurazione base del prodotto la funzionalità di ricarica potrà essere acceduta solamente durante la richiesta di una prenotazione; infatti sarà la stessa applicazione a suggerirla per portare a termine la prenotazione in caso di crediti insufficienti. L'applicazione dovrà consentire all'utente di acquistare i crediti in modo sicuro, attraverso una sezione apposita che integrerà, nella prima fase di sviluppo, il servizio PayPal come unica modalità di pagamento online.

Continuando con l'analisi delle funzionalità previste vi è quella di poter aprire l'auto prenotata utilizzando lo smartphone come chiave di sblocco. Infatti, le chiavi dell'auto si troveranno all'interno e sul parabrezza verrà posizionato uno schermo nel quale verrà visualizzato un codice QR che l'utente dovrà scansionare tramite l'applicazione stessa, per dimostrare che si trova vicino alla macchina. Qualora ci fossero dei problemi sia nello sblocco del veicolo che nella prenotazione l'utente potrà contattare l'assistenza disponibile ad un numero gestito dal fornitore del servizio. Dal momento che le automobili saranno distribuite in un'area abbastanza vasta, eccetto esigenze particolari del cliente, un'altra funzionalità offerta sarà la localizzazione dei veicoli della flotta con conseguente visualizzazione su una mappa. L'applicazione permetterà all'utente di scoprirne i dettagli per ciascuno di essi prima di confermare una prenotazione. Tra i principali dettagli sono stati previsti una foto del veicolo e la tipologia (familiare, decapottabile, altro), la targa, il

livello del carburante nel serbatoio, l'indicazione della via in cui si trova e la distanza per raggiungerla, nonché il numero di crediti al minuto richiesti per la prenotazione. Quest'ultima, considerata l'attuale configurazione del sistema, potrà essere effettuata specificando la data e l'ora d'inizio (sarà possibile prenotare fino ad un'ora prima della partenza) e la durata espressa in ore. La richiesta del periodo di prenotazione ancor prima dell'utilizzo del veicolo rappresenta una semplificazione che, in questa prima versione del prodotto, permetterà di strutturare lo sviluppo; tuttavia questo limite potrà essere superato con successive implementazioni che, ad esempio, non richiederanno l'indicazione del tempo di utilizzo della vettura.

Una volta avviata la prenotazione, verrà mostrato sulla mappa il percorso tra la posizione dell'utente e quella del veicolo, con la possibilità di visualizzare alcuni dettagli tra cui le indicazioni da seguire per raggiungerla. Si tratta di informazioni relative al solo percorso a piedi. In aggiunta, l'utente potrà accedere direttamente dall'applicazione ad una navigazione in tempo reale, grazie all'utilizzo del navigatore ufficiale *Google Maps* installato sul dispositivo. Tale funzionalità potrà risultare utile all'utente inizialmente per raggiungere l'auto appena prenotata, ma anche durante un noleggio nel caso in cui esso l'abbia temporaneamente parcheggiata per poi riprendere la corsa. Sempre riguardo la prenotazione esiste un tempo massimo per raggiungere l'auto, al termine del quale essa risulterà nuovamente disponibile. Nel caso di problemi durante il raggiungimento o altro, l'utente potrà decidere di cancellare la prenotazione entro un certo tempo (il limite attuale sarà impostato a 10 minuti prima dell'orario di inizio) e il sistema ricaricherà i crediti spesi.

Visto e considerato che i veicoli verranno utilizzati in condivisione presumibilmente durante l'arco di un'intera giornata, al fine di garantire un servizio sempre disponibile è necessario tenere sotto controllo il livello del carburante, incentivando l'utente ad eseguire il rifornimento. Si tenga presente che di base il rifornimento delle auto sarà già incluso nel servizio offerto e in certi contesti di utilizzo la funzionalità che si vuole implementare potrebbe essere addirittura rimossa dall'app. Nella versione del prodotto in esame l'operazione di rifornimento sarà opzionale e non obbligatoria per l'utente; tuttavia, per chi dovesse eseguire il rifornimento è prevista l'acquisizione di un certo numero di crediti in base all'importo pagato. Anche in questo caso le regole degli accrediti saranno definite in dettaglio dal singolo gestore. Per ciascun rifornimento l'applicazione chiederà all'utente di specificare il prezzo totale sostenuto ed eventualmente la foto della ricevuta, anche se sarà il sistema a verificare tali dati in base alle informazioni ricevute dall'auto e ad altre fornite da servizi esterni. Infine l'applicazione mostrerà in una lista tutti i rifornimenti effettuati, in modo tale che l'utente possa visualizzarne i dettagli.

Capitolo 3

Sviluppo e progettazione applicazione Android

3.1 Funzionamento di un'App nel sistema Android e tecniche multithreading

Le applicazioni mobile sono dei programmi software progettati e sviluppati per funzionare su appositi dispositivi mobili quali smartphone, tablet, smartwatch e simili. Proprio a causa dei supporti elettronici su cui vengono installate, esse necessitano di una struttura semplificata al fine di garantire leggerezza, velocità e consumi ridotti, differenziandosi così dai tradizionali applicativi per computer. In genere tali dispositivi, al contrario dei PC, sono caratterizzati da prestazioni e capacità di memorizzazione limitate, nonché da vincoli stringenti sui consumi energetici a causa della limitata energia fornita dalla batteria. L'interfaccia grafica è la parte fondamentale di un'applicazione mobile, la quale deve garantire un'interazione efficace ed efficiente tra l'utente e le funzionalità offerte. Progettare la giusta architettura grafica, intuitiva, semplice da utilizzare e con un design accattivante, è importante affinché si realizzi un prodotto di successo. Eppure c'è un altro aspetto da tenere in considerazione ovvero la velocità con cui vengono gestiti gli input dell'utente. Un'applicazione deve essere quanto più veloce possibile, con un'interfaccia fluida e reattiva: bisogna progettare i suoi componenti e le operazioni che essi devono eseguire evitando di rallentare, o addirittura di bloccare, l'interfaccia utente.

Pertanto, è opportuno comprendere il funzionamento di un'applicazione nel sistema Android, identificando gli elementi creati per la sua esecuzione e responsabili del funzionamento dei componenti di base della sua architettura, quali ad esempio `Activity` e `Service`, con particolare riguardo alla gestione dell'interazione dell'utente con l'interfaccia e all'aggiornamento della stessa. Quando un'applicazione viene avviata per la prima volta, Android crea un nuovo processo linux nel quale istanzia un thread di esecuzione principale, chiamato appunto 'main', nel quale eseguiranno tutti i vari componenti e attraverso cui verrà gestita l'interazione dell'utente. Inoltre vengono generati altri thread ma, a differenza di quello principale, gli utenti non interagiranno mai direttamente con loro; essi sono gestiti dal sistema operativo e servono per la sopravvivenza dell'applicazione nell'ecosistema Android. Il thread principale è di fondamentale importanza perché, per impostazione predefinita, si occupa della gestione delle *callback* che scandiscono il ciclo di vita dei vari componenti, di disegnare l'interfaccia grafica e di rispondere all'interazione dell'utente. In particolare esso è l'unico responsabile dell'aggiornamento dell'interfaccia: per questo motivo viene solitamente chiamato *UI Thread*. Esso esegue il suo lavoro sequenzialmente,

raccogliendo le attività da una coda di messaggi, nota come `MessageQueue`, nella quale vengono inserite delle richieste sotto forma di oggetti `Message`, per poi essere elaborate dal componente dell'app destinatario del messaggio. L'accesso alla coda è sincronizzato, consentendo a diversi thread di inserire nuovi messaggi solo se la coda non è occupata. In Android essa rimane bloccata non solo quando un thread sta leggendo, ma durante tutto il tempo di gestione dell'evento. Questo significa ad esempio che durante l'esecuzione della *callback* `onCreate()`, richiamata all'avvio di un'`Activity`, l'inserimento di nuovi messaggi dovrà attendere il suo completamento (che corrisponde all'esecuzione del codice al suo interno). Android ha definito che questo tempo di attesa debba essere al massimo dell'ordine dei 3-4 secondi; questo implica che tutte le operazioni eseguite nelle varie *callback* dovranno terminare entro questo tempo. Se l'elaborazione dovesse superare il limite, Android provvederà a termina l'applicazione generando un'eccezione *ANR (Application not responding)* e il processo verrà distrutto. Perciò tutte le operazioni che richiedono lunghi tempi di esecuzione come la connessione ad un server remoto, l'interrogazione di un database e il caricamento di immagini, andranno eseguite fuori dall'*UI thread* per evitare il blocco dell'interfaccia grafica. Inoltre l'Android toolkit non è thread-safe, quindi solamente l'*UI thread* può modificare l'interfaccia: se una delle operazione precedenti, eseguita da un thread in background, necessita di aggiornare lo stato di uno qualsiasi dei *widget* presenti nell'interfaccia, potrà farlo pubblicando un nuovo `Message` nella coda (Figura 3.1), il quale verrà processato all'interno del thread principale.

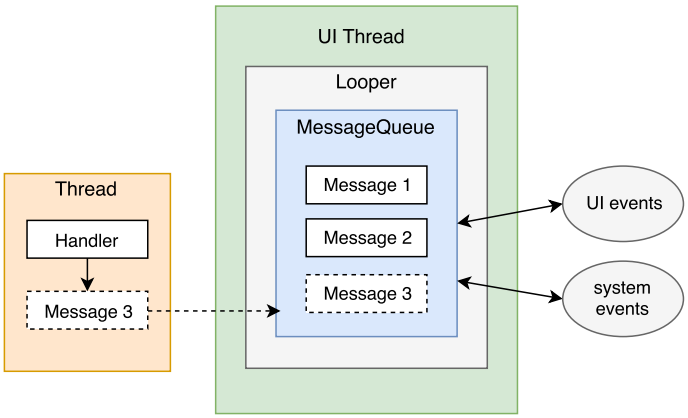


Figura 3.1: Comunicazione con l'*UI Thread*

Per spostare l'esecuzione di operazioni intensive fuori dal thread principale, oltre agli strumenti offerti da Java come la classe standard `Thread` e l'interfaccia `Executor`, Android mette a disposizione delle classi per creare thread di lavoro che ne semplificano la gestione del loro ciclo di vita, la loro comunicazione e in generale il multithreading. Tra queste troviamo la classe `Looper` specializzata nella gestione della coda di messaggi associata ad un singolo thread. Di base estendendo dalla classe `Thread` non è prevista l'associazione ad una coda; `Looper` ne permette la sua creazione, l'inserimento e l'estrazione di nuovi messaggi. Esso esegue un ciclo infinito nel quale rimane in attesa del prossimo messaggio, senza consumare cicli di CPU, per poi distribuirlo al gestore interessato. Il 'main' thread dell'applicazione funziona in questo modo. Normalmente un `Looper` può avere associati uno o più oggetti di tipo `Handler`, una classe che offre un'interfaccia per accedere in maniera sincronizzata alla coda di messaggi inserendo nuove richieste tramite appositi metodi. Il `Looper` si occuperà di estrarle una alla volta e consegnarle al rispettivo `Handler` attraverso cui sono state inserite. Possiamo considerare l'`Handler` come produttore di messaggi e il `Looper` come il consumatore.

Gli eventi inseriti nella coda di messaggi potrebbero essere relativi ad attività che richiedono l'accesso agli elementi dell'interfaccia grafica e, come abbiamo detto, questo non può avvenire al di fuori del 'main' thread altrimenti l'applicazione terminerebbe con un'eccezione. Sono necessari dei meccanismi che permettano al thread secondario di notificare il suo risultato a quello principale in modo che esso lo mostri. Un modo per realizzare questa comunicazione prevede l'inserimento di un evento nella coda gestita dal thread principale attraverso diversi metodi, tra cui `activity.runOnUiThread()` e `view.postRunnable()`, i quali però hanno dei vincoli: ad esempio il primo richiederebbe al thread secondario di mantenere il riferimento all'`Activity` in esecuzione, mentre il secondo può essere invocato fintanto che l'oggetto `View` è visibile.

In alternativa ai meccanismi appena mostrati, esiste uno strumento a cui è possibile assegnare un lavoro da eseguire in maniera asincrona, interrompibile in qualsiasi momento e in grado di pubblicare i risultati nell'*UI thread*. Si tratta dell'`AsyncTask`, una classe astratta generica che va estesa fornendo l'implementazione di uno o più metodi che verranno poi invocati dal 'main' thread in momenti precisi della sua esecuzione. Tutte le sottoclassi devono definire i tre tipi di dati accettati dal task, chiamati `Params`, `Progress` e `Result`, che rappresentano rispettivamente il tipo dei parametri usati nel metodo asincrono, il tipo di progresso da mostrare durante il lavoro in background e il tipo del risultato atteso. Tutti gli `AsyncTask` condividono un insieme di thread creato da Android, detto *ThreadPool*, dal quale ne verrà scelto uno per la sua esecuzione che si svolge in 4 passaggi corrispondenti ai metodi elencati di seguito: `onPreExecute()` richiamato quando il thread è pronto, `doInBackground()` esegue il lavoro vero e proprio, `onProgressUpdate()` utile per mostrare lo stato di avanzamento dell'operazione e `onPostExecute()` invocato non appena il risultato è disponibile. Bisogna far attenzione però al caso in cui il codice eseguito dal task faccia riferimento agli elementi dell'interfaccia grafica: in questo caso infatti tutti gli aggiornamenti dovranno essere eseguiti solo se l'`Activity` non è stata distrutta e quindi ogni riferimento al task va cancellato in corrispondenza della *callback* `onDestroy`. Detto questo gli `AsyncTask` rappresentano un modo semplice e corretto di utilizzare un thread secondario, in particolare per operazioni di breve durata (pochi secondi al massimo).

Tipicamente la necessità di eseguire lunghe operazioni al di fuori del thread principale nasce soprattutto dall'esigenza di dover accedere ad una sorgente di dati talvolta indispensabili al funzionamento dell'applicazione, gran parte dei quali dovranno essere mostrati in appositi contenitori inclusi nell'interfaccia. Esistono così delle classi specifiche a cui delegare il caricamento asincrono delle informazioni, che restituiscono i risultati al componente chiamante una volta disponibili. Queste funzionalità sono incapsulate all'interno della classe `Loader` che `Activity` e `Fragment` possono utilizzare attraverso il proprio `LoaderManager`, non preoccupandosi della gestione di thread separati, riducendone quindi la complessità del codice ed eliminando i potenziali bug correlati ai threads. Per la consegna dei dati a fine caricamento è possibile registrare un `OnLoadCompleteListener` che invierà i risultati consegnati dal `Loader` chiamando il metodo `onLoadFinished`. Il monitoraggio dei dati sottostanti è demandato invece ad un osservatore il quale notifica eventuali cambiamenti in modo tale che tutte le viste contenenti queste informazioni possano essere aggiornate.

3.2 Il linguaggio Kotlin

3.2.1 Concetti generali

Kotlin [6] è un nuovo linguaggio di programmazione sviluppato da JetBrains, la compagnia che ha creato IntelliJ IDEA. Si tratta di un linguaggio conciso, sicuro, pragmatico, focalizzato all'interoperabilità con il codice Java che coinvolge tutti gli aspetti: dal linguaggio stesso al bytecode prodotto. Infatti esso offre la possibilità di costruire applicazioni per JVM e Android, compilando di fatto nel bytecode Java al quale si aggiunge la compilazione per JavaScript e la generazione di codice in linguaggio nativo per diverse piattaforme. Grazie all'elevato grado di interoperabilità tra Java e Kotlin è possibile effettuare una transizione senza problemi tra i due, o persino permettere ai due linguaggi di coesistere nello stesso progetto. Kotlin dunque può essere utilizzato in tutte le aree in cui è attualmente impiegato Java, come ad esempio lo sviluppo lato server, la costruzione di applicazioni Android e molto altro, avvalendosi di framework e librerie Java esistenti senza la necessità di modifiche. Esso ha come obiettivo principale quello di fornire un'alternativa più produttiva, semplice e sicura al linguaggio Java, estremamente popolare e diffuso in svariati ambienti ma con un certo numero di limitazioni. L'utilizzo di Kotlin in questi contesti mira ad alleggerire il lavoro degli sviluppatori, i quali si troverebbero a realizzare il software attraverso l'impiego di una quantità inferiore di codice e con meno complicazioni durante il processo di sviluppo.

Per gli sviluppatori Android, in particolare, il 17 Maggio 2017[1] Google ha annunciato l'aggiunta del pieno supporto per l'utilizzo del linguaggio Kotlin, disponibile nativamente a partire dalla versione di Android Studio 3.0, con la possibilità di installare un plugin a parte per le versioni precedenti di questo IDE. Attualmente Kotlin viene rilasciato sotto licenza Apache 2.0, dunque si tratta di un linguaggio completamente open-source e, secondo Google, rappresenta un sistema abbastanza maturo da impiegare nello sviluppo di applicazioni mobile per Android, oltre a Java e `C++`. Per iniziare a familiarizzare con il linguaggio e capire come funziona, JetBrains ha sviluppato uno strumento in grado di convertire il codice Java direttamente in Kotlin che, tuttavia, come tutti gli strumenti di conversione automatici, non è perfetto. Ne risulta infatti un codice funzionante che richiede però alcuni aggiustamenti per poter diventare un codice migliore.

Così come Java, Kotlin è un linguaggio a tipizzazione statica e forte, ciò significa che prevede l'assegnazione dei tipi alle variabili in fase di scrittura del programma, direttamente nel codice sorgente, con l'ausilio di parole chiave. Sarà dunque il compilatore a verificare che i metodi e i campi ai quali si tenta di accedere siano coerenti con il tipo utilizzato. Dall'altro lato, al contrario di Java, Kotlin non richiede di specificare esplicitamente il tipo per ogni variabile dichiarata ma, in molti casi, esso può essere determinato automaticamente dal contesto come ad esempio dal suo valore di inizializzazione. La capacità del compilatore di individuare implicitamente il tipo di un'espressione è detta *type inference*.

Un'altra novità importante di Kotlin è il supporto offerto dalla libreria alla programmazione funzionale, oltre che all'utilizzo del paradigma ad oggetti ben noto agli sviluppatori Java. Quest'ultimo è un linguaggio imperativo che pone l'enfasi sui dati e sul modo di processarli tramite procedure, metodi e funzioni. Partendo da un'insieme di dati si applicano

[1] fonte:
https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html

una serie di 'operazioni' eseguite in sequenza che li elaborano producendo il risultato finale. Nel linguaggio funzionale l'attenzione è rivolta alle funzioni, che possono essere argomento di altri metodi oppure usate come valore di ritorno. Inoltre, nel linguaggio funzionale le funzioni sono pure, non hanno effetti nascosti, quindi per certi valori degli argomenti il risultato di una funzione sarà sempre lo stesso indipendentemente da quando essa verrà eseguita. Usando questo paradigma si possono ottenere diversi benefici nella scrittura del codice che diventa breve ed essenziale; utilizzando le funzioni come valori si può ottenere una maggiore astrazione che consente di evitare la duplicazione di codice. Un esempio pratico è quello di una funzione che ricerca gli elementi all'interno di un'`ArrayList<T>`, dove `<T>` rappresenta un tipo generico, secondo una certa condizione. Definendo opportunamente la funzione più esterna che accetta come argomento la condizione da verificare per ogni elemento della lista, sotto forma di metodo, è possibile riutilizzare tale funzione per una serie di condizioni e su liste di oggetti con tipi diversi, tra cui la ricerca di tutti gli elementi multipli di un numero o di parole che contengano una certa lettera e così via. È evidente come questa tecnica riduca la quantità di codice del software ma anche il tempo impiegato per la sua realizzazione.

Analizzando il linguaggio in dettaglio possiamo scoprire gli strumenti che lo caratterizzano, alcuni dei quali sono stati impiegati nello sviluppo dell'applicazione Splash. Tra questi troviamo:

lambdas espressioni o funzioni anonime letterali che non devono essere dichiarate ma definite direttamente nel punto in cui vanno richiamate. La sintassi prevede la definizione del blocco della funzione tra parentesi graffe, tralasciando l'indicazione del nome e l'utilizzo della parola chiave `fun`: qualsiasi parametro ricevuto in ingresso dalla lambda deve essere scritto prima dell'indicatore `->`, con la possibilità di ometterne il tipo, mentre il corpo và espresso dopo tale indicatore. A differenza delle funzioni normali le lambda non richiedono di specificare il tipo ritornato dalla computazione, ma esso viene rilevato automaticamente dal compilatore. Un esempio comune di utilizzo delle lambda nella programmazione Android è il metodo `setOnClickListener` della classe `View`. Esso si aspetta di ricevere come parametro l'implementazione anonima dell'interfaccia `View.OnClickListener` per definire il comportamento desiderato dall'applicazione al verificarsi dell'evento di *click* su una `View`. Questo richiederebbe normalmente la scrittura di un certo numero di righe di codice, indipendentemente dalla porzione che verrà eseguita al verificarsi dell'evento *click*. In Kotlin la realizzazione dei *listener* viene semplificata utilizzando le espressioni lambda per definire l'implementazione dei metodi eseguiti al verificarsi di un certo evento, come nel caso del metodo citato sopra;

higher-order functions funzioni che accettano altre funzioni come parametri e/o restituiscono una funzione come risultato. Esistono diversi modi per passare una funzione come argomento di un'altra: dichiarandola in precedenza e utilizzando il suo nome come riferimento, preceduto da `::`, oppure fornendo il blocco di codice da eseguire direttamente nell'argomento; quest'ultima operazione richiede l'utilizzo di una funzione letterale che può essere un'espressione lambda, oppure una funzione anonima qualora sia necessario specificare esplicitamente il tipo di ritorno. Nonostante si tratti di un potente strumento, bisogna fare attenzione al fatto che le funzioni higher-order standard, a causa del loro funzionamento interno, possono incidere negativamente sulle prestazioni del software, soprattutto per i dispositivi Android vecchi o di fascia bassa. Infatti, il passaggio di un'espressione lambda come parametro di una higher-order si traduce nella creazione di un oggetto funzione generico: se la lambda fa riferimento ad una variabile dichiarata esternamente ad essa, viene

creata una nuova istanza dell'oggetto funzione ogni volta che la lambda viene passata come argomento, per poi essere rimossa dal garbage-collector dopo l'esecuzione. Invece, se si utilizzano semplici funzioni lambda esse verranno create una sola volta come singleton e riutilizzate per le successive esecuzioni;

inline functions possibilità di aggiungere il modificatore `inline` nella dichiarazione delle funzioni higher-order in modo da istruire il compilatore ad incorporare il codice dell'espressione lambda nel corpo della funzione chiamante, evitando tutte le penalità sopra citate durante l'esecuzione. Tuttavia, tale modificatore deve essere usato con cautela, soprattutto per funzioni lunghe che, se richiamate più volte, possono dare luogo ad una significativa crescita del codice sorgente di un programma;

null-safety il sistema dei tipi di Kotlin ha lo scopo di eliminare dal codice il pericolo di accesso a riferimenti nulli che in molti linguaggi di programmazione come Java sono causa di eccezioni (le cosiddette `NullPointerException` o `NPE`). Kotlin permette di distinguere tra tipi di variabili che accettano valori nulli (nullable references) e quei tipi che non possono contenere riferimenti nulli (non-null references). Grazie a questa distinzione è possibile accedere in modo sicuro alle proprietà di queste variabili evitando che il programma si arresti a causa di un'eccezione a tempo di esecuzione, considerando che la maggior parte degli errori vengono individuati dal compilatore al momento della compilazione. I campi delle variabili con tipo nullo possono essere acceduti tramite il cosiddetto *safe-call operator*, indicato con '`?.`', il quale garantisce l'accesso solo nel caso in cui la variabile sia non null. Inoltre, esiste l'operatore '`?:`' detto *elvis operator*, che permette di preparare un risultato da restituire di default nel caso il valore della variabile sia nullo;

coroutine dedicate all'esecuzione di operazioni asincrone non bloccanti i cui dettagli saranno presentati in seguito.

Oltre a queste funzionalità offerte di base dal linguaggio, esiste un plug-in specifico per il framework Android che semplifica e riduce la scrittura delle operazioni più frequenti. Si è già parlato delle funzioni lambda come supporto all'attività di aggiunta dei *listener* per gli elementi che costituiscono la vista di un'applicazione, ma vale la pena citare l'estensione che permette di rimpiazzare il metodo `findViewById` con delle proprietà sintetiche generate dal compilatore. Android prevede l'assegnazione di un identificativo univoco per ciascun elemento di tipo `View` definito in un layout, così da farvi riferimento durante la scrittura del codice tramite il metodo `findViewById`, sfruttando proprio tale identificativo. Grazie all'estensione di Kotlin è possibile importare un'opportuna proprietà sintetica per il tipo di vista desiderato e basterà richiamare la `View` con il proprio `id` per utilizzarla, evitando così la chiamata alla `findViewById`. Inoltre, dal momento che l'invocazione di questo metodo può risultare lenta soprattutto in caso di viste con una gerarchia complessa, tale estensione memorizza ciascuna vista in una cache dopo il primo utilizzo, suddividendole in base al contenitore a cui appartengono, effettuando la chiamata al metodo `findViewById` una sola volta.

3.2.2 Coroutine: supporto per la concorrenza

Disponibili a partire dalla versione Kotlin 1.1 in via sperimentale, le coroutine rappresentano un modo nuovo, semplice ed economico di eseguire operazioni asincrone non bloccanti in un sistema multi-thread. Essendo sperimentali, le prossime versioni della libreria sicuramente presenteranno miglioramenti e novità, perciò in questa sezione verrà analizzato

il loro funzionamento presentando alcuni vantaggi pratici introdotti nello sviluppo per Android, da intendersi come una possibile alternativa ai meccanismi offerti di base dal linguaggio; come tutti gli strumenti, il loro impiego è consigliato laddove risultino più convenienti e performanti di altri.

Nella sezione precedente si è discusso delle tecniche per la gestione del multithreading offerte da Java, in particolare nell'ambiente Android, con l'obiettivo di migliorare le performance e soprattutto la *user experience*, ottenendo una maggiore reattività dell'interfaccia grafica che ne rappresenta uno degli aspetti determinanti. Tuttavia, sebbene esse siano state ben progettate, spesso risultano difficili da adottare e talvolta richiedono la scrittura di una grande quantità di codice. Inoltre, seguono uno stile di programmazione concorrente bloccante, in cui è previsto che un thread, incaricato dello svolgimento di una lunga operazione, rimanga in stato di 'blocking' in attesa del suo completamento. Questo aspetto è da tenere in considerazione visto che la creazione di nuovi thread è onerosa e introduce un certo overhead nel sistema, che può rendere l'applicazione lenta proprio a causa del codice bloccante. Bloccare un thread è gravoso, specialmente in condizioni di alto carico e soprattutto in sistemi come quelli mobile dove ogni processo può avere un numero massimo di thread contemporaneamente: bloccandone uno si rischia di ritardare altre operazioni più importanti. Attraverso le coroutine Kotlin propone e incoraggia un modello di concorrenza completamente diverso, basato sul concetto di 'sospensione' e non di bloccaggio. La coroutine è un algoritmo in grado di eseguire operazioni in background all'interno di un thread, capace di sospendere e riprendere la sua esecuzione senza mai bloccare il thread nel quale viene eseguita. Anche in questo caso emerge il desiderio degli sviluppatori del linguaggio di rispondere alle esigenze dei programmatori fornendo uno strumento volto a migliorare l'implementazione della concorrenza e mantenendo la complessità all'interno delle librerie. Ad esempio una particolarità importante è quella di poter scrivere all'interno di una coroutine codice concorrente ma seguendo una logica sequenziale, lasciando alla libreria il compito di realizzare la concorrenza. Le coroutine permettono di lavorare con diversi meccanismi di programmazione concorrente presenti in altri linguaggi, molti dei quali sono già disponibili nella libreria o possono essere implementati facilmente; uno tra questi è quello basato sul pattern *async/await* presente in `C#`.

Le coroutine sono classificate come `Thread` 'leggeri', dunque con il loro stesso obiettivo ma con un impatto ridotto in termini di prestazioni sul sistema. Più propriamente esse non sono dei veri e propri elementi `Thread` come quelli conosciuti nel mondo Java, sono economiche nella creazione e non introducono nessun overhead di gestione; infatti, generalmente l'esecuzione di una coroutine non avviene all'interno di un nuovo thread nativo creato appositamente per tale scopo, ma viene affidata ad uno tra quelli condivisi e messi a disposizione dalla libreria. Ognuno di essi potrà ospitare diverse coroutine e di conseguenza si riduce il numero di thread necessari al funzionamento di un programma. Questo modello di esecuzione non prevede un limite per il numero di coroutine avviate contemporaneamente, al contrario degli oggetti `Thread` per i quali è il sistema operativo a imporre dei limiti al numero massimo di thread eseguiti in parallelo. L'altro grande vantaggio risiede nella possibilità di sospendere una coroutine mentre è in attesa del completamento di un'operazione, liberando di fatto il thread su cui stava eseguendo, per poi riprendere l'esecuzione al termine dell'attesa in un thread libero tra quelli presenti. Per lanciare una coroutine e abilitare la sospensione è necessario utilizzare le cosiddette 'suspending functions', funzioni marcate con il modificatore `suspend` che, come quelle tradizionali, possono accettare parametri in ingresso ed eventualmente restituire un risultato, con la differenza che devono essere richiamate all'interno del contesto di una coroutine o di un'altra funzione `suspended`.

Kotlin mette a disposizione la libreria `kotlinx.coroutine` in cui sono definiti diversi *builder* per la creazione di una coroutine, ognuno dei quali ha uno scopo ben preciso:

`launch()` permette l'esecuzione di un'operazione asincrona senza tener conto del risultato e restituisce un oggetto di tipo `Job`, utile per cancellare una coroutine, o se necessario per attendere il suo completamento;

`async()` concettualmente è simile a `launch()`, ma restituisce il valore `Deferred<T>`, un future non bloccante che permette di accedere al risultato dell'operazione; per ottenerlo è disponibile il metodo `.await()`, che attende il termine della computazione senza bloccare il thread nel quale viene invocato, restituendo il risultato o lanciando un'eccezione in caso di errori;

`runBlocking()` a differenza dei precedenti è stato progettato per eseguire codice bloccante che in alcuni casi può risultare utile: ad esempio se bisogna attendere il completamento di una funzione `suspended` prima che un programma venga terminato.

Una caratteristica di questi metodi è la possibilità di specificare tra i parametri il contesto in cui essi verranno eseguiti. Abbiamo già detto che nelle applicazioni mobile è importante aggiornare lo stato dell'interfaccia utente unicamente nel thread principale, l'*UI Thread*, mentre le computazioni lente possono essere affidate a dei thread in background. Per venire incontro a tali esigenze, Kotlin mette a disposizione due contesti per lo sviluppo su Android, entrambi istanza di `CoroutineContext`: il primo è `UI` che, come suggerisce il nome, rappresenta l'*UI Thread*, mentre l'altro è `CommonPool` che identifica il contesto di uno dei thread gestiti dalla libreria. Combinando in modo opportuno le funzioni descritte e avvalendosi dei contesti è possibile sfruttare le potenzialità delle coroutine per implementare la concorrenza nello sviluppo Android, in alternativa alle classi presentate in precedenza: ad esempio la creazione di un *task* che esegue in background e necessita di pubblicare un risultato nell'*UI Thread* può avvenire sfruttando le coroutine, piuttosto che utilizzando un `AsyncTask`.

```
fun visualizzaImmagine(url:String) = launch(uiContext) {
        mostraProgresso(true) // ui thread
        val task = async(poolContext) {
            provider.caricaImmagine(url) }
        val result = task.await() // sospensione, no ui thread
        mostraProgresso(false)
        imageView.setImageBitmap(result) // ui thread
}
```

Figura 3.2: Esempio di un *task* eseguito tramite coroutine

La Figura 3.2 mostra un esempio di codice Kotlin che realizza una tra le funzionalità più comuni di un'applicazione Android: scaricare un'immagine da remoto visualizzandola all'interno di un'`Activity`. In questo esempio si suppone di voler realizzare una semplice applicazione che esegue il caricamento di immagini da/verso uno *storage* attraverso la rete Internet. L'applicazione gestisce la lista delle immagini già caricate e per ciascuna di esse memorizza una descrizione testuale, mostrata a video, più l'URL in cui è disponibile. Al click di un elemento della lista verrà richiamato il metodo `visualizzaImmagine()`, mostrato in Figura 3.2, responsabile di eseguire il download dell'immagine e di mostrarla

a schermo. Queste due operazioni vengono gestite in due thread distinti: il download sarà affidato ad un thread diverso da quello principale, mentre il risultato và caricato nell'`ImageView` dall'*UI Thread*. Il codice proposto effettua tale distinzione tramite due coroutine, quella più esterna creata da `launch()` nel contesto `UI` e l'altra avviata con `async()` nel contesto `CommonPool`. Considerando che il metodo `visualizzaImmagine()` verrà invocato dentro il thread principale, l'operazione asincrona svolta da `async()` e la sospensione causata da `.await()` necessitano di essere eseguite dentro il contesto di un'altra coroutine, creata appunto da `launch()`. In sostanza, quando questa funzione verrà richiamata la sequenza delle operazioni svolte seguirà l'ordine in cui esse sono state scritte: il metodo `caricaImmagine()`, appartenente ad una classe preposta all'esecuzione della richiesta `HTTP` all'URL specificato, effettuerà il download asincrono dell'immagine all'interno di una coroutine. Quest'ultima rimarrà sospesa in attesa del risultato e, non appena esso sarà disponibile, verrà mostrato all'interno dell'`ImageView`. Il metodo `mostraProgresso()`, invece, mostra un `ProgressBar` che indica lo stato di avanzamento dell'operazione.

```
class DownloadImageTask(imgView:ImageView) : AsyncTask<String,
    Void, Bitmap?> {

        override fun onPreExecute() {
               mostraProgresso(true) // ui thread
        }
        override fun doInBackground(vararg params:
            String?):Bitmap? {
               val url = params[0]
               return provider.caricaImmagine(url) //no ui thread
        }
        override fun onPostExecute(result: Bitmap?) {
               mostraProgresso(false) // ui thread
               imgView.setImageBitmap(result)
        }
}
```

Figura 3.3: Esempio di un *task* eseguito tramite `AsyncTask`

La stessa funzionalità realizzata tramite un `AsyncTask`, invece, richiederebbe la creazione di una classe e l'implementazione dei suoi metodi (Figura 3.3). Il download verrà eseguito nel metodo `doInBackground()`, invocato nel thread in background. Quest'ultimo, a differenza del caso precedente, rimarrà bloccato in attesa del completamento dell'operazione. Il risultato sarà disponibile nel metodo `onPostExecute()`, invocato nell'*UI Thread*, il quale visualizzerà l'immagine nello schermo. È evidente come le coroutine siano un'alternativa efficiente agli `AsyncTask` in Kotlin, sia per i vantaggi già presentati riguardo la sospensione, sia per la ridotta quantità di codice da scrivere a favore di una migliore leggibilità e manutenibilità dello stesso.

3.3 Progettazione delle attività

Prendendo in considerazione quanto si è detto nell'analisi dei requisiti discussa nel capitolo precedente, è stato possibile delineare la struttura logica del prodotto attraverso la definizione del flusso che l'utente dovrà seguire nell'utilizzo delle funzionalità offerte dall'applicazione e delle informazioni necessarie per la loro esecuzione. A tal proposito, è stato utile costruire il diagramma dei casi d'uso mostrato in Figura 3.4, il quale permette di descrivere in maniera chiara e coesa come sono percepite tali funzionalità dall'utente stesso, di individuare gli scenari tipici e mostrare come avviene l'interazione tra gli attori e il sistema; oltre al diagramma generale, per ciascuna funzionalità verranno illustrate le precondizioni nelle quali sono eseguibili, la sequenza delle azioni svolte dagli attori e dal sistema, nonché le informazioni richieste durante le interazioni. L'obiettivo di questa fase è stato quello di definire un modello che mostri il comportamento del sistema quando sollecitato da un attore, da seguire durante tutto il processo di sviluppo per coordinare il lavoro.

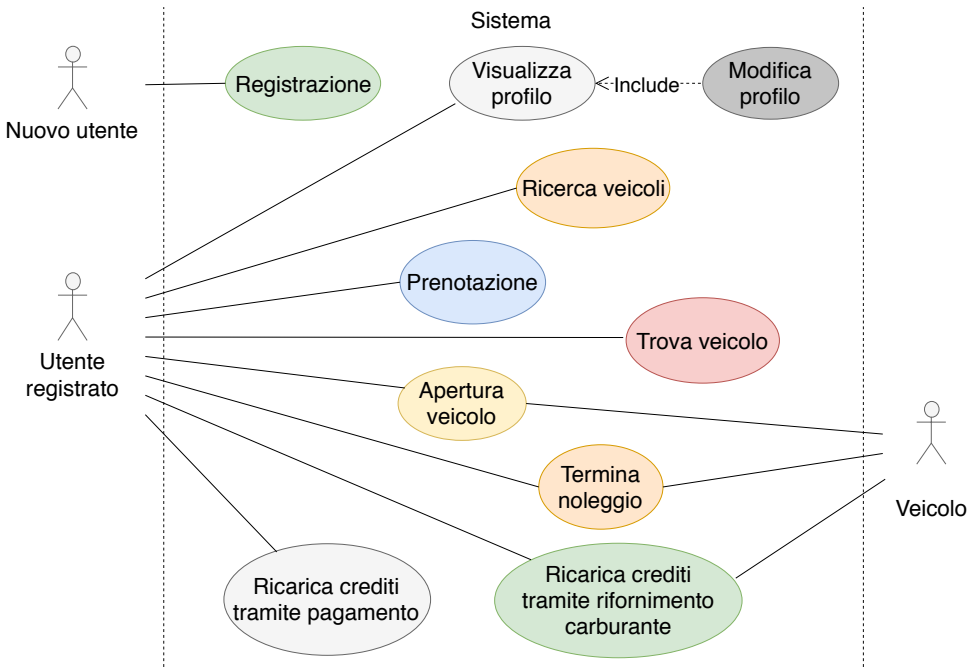


Figura 3.4: Diagramma dei casi d'uso

3.3.1 Registrazione

L'utente che intende utilizzare il servizio Splash dovrà eseguire la procedura di registrazione fornendo un *nickname* che lo identificherà nel servizio, il proprio nome e cognome, un indirizzo email valido e la password di accesso. Il sistema invierà un link per l'attivazione dell'account all'indirizzo email specificato. L'utente dovrà confermare che è in possesso di tale indirizzo cliccando sul link ricevuto.

Nome	Registrazione
Attori partecipanti	Utente
Precondizioni	L'utente ha installato l'applicazione
Flusso degli eventi	1. L'utente si registra al servizio fornendo le informazioni necessarie 2. Il sistema invia all'utente un'email di conferma 3. L'utente clicca sul link per l'attivazione dell'account 4. Il sistema salva l'account dell'utente
Eccezioni	5. Se i dati non sono validi, al passo 2 il sistema non invia l'email di conferma 5.1 Il sistema permette all'utente di reinserire le informazioni e riprovare Si ritorna allo scenario principale a partire dal passo 2

3.3.2 Visualizzazione e aggiornamento profilo

L'attuale configurazione del sistema prevede un profilo utente di base, da completare in qualsiasi momento, in cui saranno richieste all'utente le sue generalità, tra cui la data, il luogo, lo stato, la provincia di nascita e il codice fiscale. Queste informazioni permettono al gestore di catalogare gli utenti e verificare, per quanto possibile, che essi abbiano almeno compiuto la maggiore età, requisito indispensabile per guidare. Successivamente sarà il gestore del servizio a definire altre politiche di controllo, come ad esempio la richiesta dei dati sulla patente di guida.

Nome	Aggiornamento profilo
Attori partecipanti	Utente
Precondizioni	L'utente è registrato
Flusso degli eventi	1. L'utente accede al profilo personale 2. L'utente modifica uno o più campi tra quelli previsti e conferma 3. Il sistema memorizza le nuove informazioni e aggiorna il profilo dell'utente
Eccezioni	4. Se i dati non sono validi, al passo 3 il sistema non esegue il salvataggio dei dati 4.1 Il sistema permette all'utente di reinserire le informazioni e riprovare Si ritorna allo scenario principale a partire dal passo 3

3.3.3 Ricerca veicoli

L'applicazione permette agli utenti di effettuare la ricerca di tutti i veicoli messi a disposizione dall'operatore: viene utilizzata una mappa per mostrare i veicoli localizzati che vengono distinti in base al loro stato di prenotazione (disponibile, occupato). Nella prima configurazione del prodotto il sistema esegue la ricerca senza richiedere all'utente l'inserimento di filtri particolari.

Nome	Ricerca veicoli
Attori partecipanti	Utente
Precondizioni	L'utente è registrato
Flusso degli eventi	1. L'utente richiede di visualizzare i veicoli presenti 2. Il sistema localizza l'utente e esegue la ricerca 3. Il sistema mostra sulla mappa tutti i veicoli che ha trovato differenziando tra quelli disponibili e quelli già prenotati
Eccezioni	4. Se il sistema non riesce a localizzare l'utente, al passo 2 non esegue la ricerca 4.1 Il sistema invita l'utente ad attivare il servizio di localizzazione e riprovare

3.3.4 Prenotazione veicolo

L'utente che desidera utilizzare un veicolo della flotta ha la possibilità di prenotarlo in anticipo. Tale operazione può essere avviata selezionando il veicolo dalla mappa presente nella schermata principale dell'applicazione. Il sistema offre all'utente la possibilità di esaminare i dettagli del veicolo prima di procedere con la prenotazione; viene mostrata una foto del veicolo e la tipologia, il livello del carburante, l'indicazione della via in cui risulta parcheggiato e la distanza dalla posizione dell'utente, e ancora il numero di crediti necessari per prenotarlo. Per effettuare la prenotazione è richiesto all'utente di specificare il periodo di tempo nel quale esso intende usufruire dell'auto, scandito dalla data e ora d'inizio, e dalla durata espressa in ore. Una volta che l'utente conferma la prenotazione, il sistema esegue una serie di verifiche che interessano sia il periodo richiesto, il quale deve ricadere entro certi limiti, sia la disponibilità del veicolo e il numero di crediti presenti nel portafoglio dell'utente. Se almeno una di queste verifiche fallisce, l'utente non può confermare la prenotazione; nel caso in cui l'errore sia dovuto ad una mancanza di crediti, il sistema suggerirà all'utente di ricaricare il suo portafoglio attraverso il pagamento. A prenotazione confermata l'utente avrà 15 minuti di tempo per raggiungere l'auto e sbloccare le portiere attivando così il noleggio.

Nome	Prenotazione
Attori partecipanti	Utente
Precondizioni	L'utente è registrato
Flusso degli eventi	1. L'utente seleziona sulla mappa il veicolo che intende prenotare 2. Il sistema mostra i dettagli del veicolo 3. L'utente conferma di voler prenotare il veicolo con la configurazione mostrata 4. L'utente indica le informazioni relative alla prenotazione (data e ora d'inizio, durata) 5. Il sistema presenta il totale dei crediti da pagare 6. L'utente conferma e paga la prenotazione
Eccezioni	7. Se l'utente ha indicato una data invalida, al passo 6 non può confermare la prenotazione 7.1 Il sistema permette all'utente di reinserire le informazioni e riprovare 8. Se l'utente non ha abbastanza crediti nel portafoglio, al passo 6 non può confermare la prenotazione 8.1 Il sistema permette all'utente di acquistare nuovi crediti e riprovare 9. Se il veicolo non è più disponibile, al passo 6 non può confermare la prenotazione 9.1 Il sistema informerà l'utente che il veicolo non è più disponibile. Si ritorna al passo 1 Si ritorna allo scenario principale a partire dal passo 4

3.3.5 Ricarica crediti tramite pagamento

Durante la creazione di una nuova prenotazione, il sistema verifica che l'utente abbia un numero di crediti sufficiente per effettuarne il pagamento e completare l'operazione. Se l'utente non ha abbastanza crediti nel suo portafoglio elettronico, ha la possibilità di acquistarne di nuovi tramite pagamento online sfruttando il servizio PayPal; il sistema mostra una lista delle opzioni di ricarica disponibili riportando per ciascuna di esse il numero di crediti ottenibili e l'importo totale da pagare. L'utilizzo di tale funzionalità richiede all'utente l'inserimento dei dati di accesso di un account PayPal esistente. Per gli utenti che non hanno ancora effettuato la registrazione a questo servizio, il sistema permette loro di accedere direttamente alla pagina Web ufficiale per creare un nuovo account.

Nome	Ricarica crediti tramite pagamento
Attori partecipanti	Utente
Precondizioni	L'utente è registrato
Flusso degli eventi	1. L'utente visualizza le opzioni di ricarica e seleziona quella che preferisce 2. Il sistema mostra la schermata per raccogliere le informazioni di pagamento (account PayPal) 3. L'utente fornisce i dati del suo account PayPal personale 4. Il sistema autorizza l'acquisto 5. Il sistema aggiunge i crediti acquistati e aggiorna il portafoglio dell'utente
Eccezioni	6. Se l'utente non possiede un account PayPal, al passo 3 non può inserire i propri dati 6.1 Il sistema permette all'utente di accedere al sito Web ufficiale per effettuarne la registrazione e riprovare 7. Se l'utente ha inserito dei dati invalidi, al passo 4 il sistema non autorizza l'acquisto 7.1 Il sistema permette all'utente di reinserire i dati e riprovare 8. Se l'importo disponibile nell'account PayPal specificato non è sufficiente, al passo 4 il sistema non autorizza l'acquisto 8.1 Il sistema permette all'utente di scegliere un account PayPal alternativo e riprovare Si ritorna allo scenario principale a partire dal passo 3

3.3.6 Trova veicolo

L'utente che ha effettuato una prenotazione per un certo veicolo deve poterlo raggiungere, prima di sbloccare le porte e avviare il noleggio. Il sistema è in grado di localizzare l'utente e conosce la posizione del veicolo prenotato, così calcola il percorso, esclusivamente nella modalità a piedi, tra l'utente e l'auto mostrando in una lista le indicazioni da seguire; inoltre l'utente ha la possibilità di accedere al sistema di navigazione real-time offerto dall'applicazione di Google installata sul dispositivo, *Google Maps*. Tale funzionalità è accessibile anche a noleggio avviato per permettere all'utente di trovare l'auto che aveva temporaneamente parcheggiato.

Nome	Trova veicolo
Attori partecipanti	Utente
Precondizioni	L'utente è registrato e ha effettuato una prenotazione/ha avviato il noleggio
Flusso degli eventi	1. L'utente decide di localizzare il veicolo per raggiungerlo 2. Il sistema calcola il percorso migliore tra l'utente e il veicolo tenendo traccia delle indicazioni 3. Il sistema disegna il percorso sulla mappa 4. L'utente visualizza la lista delle indicazioni costituita da una serie di passi da compiere 5. L'utente seleziona la navigazione real-time 5.1 Il sistema avvia l'applicazione *Google Maps*
Eccezioni	6. Se il sistema non riesce a localizzare l'utente, al passo 2 non calcola il percorso 6.1 Il sistema invita l'utente ad attivare il servizio di localizzazione e riprovare

3.3.7 Apertura veicolo

Nel momento in cui l'utente si trova vicino al veicolo prenotato deve poter sbloccare le porte, mentre le chiavi per l'accensione si troveranno all'interno del veicolo. Tale operazione viene realizzata tramite la scansione di un codice QR diverso per ogni operazione. Quando l'utente richiede l'avvio del noleggio, il sistema invia tale codice QR al veicolo che lo mostrerà su un apposito schermo e inoltre avvierà la fotocamera sul dispositivo dell'utente: quest'ultima operazione richiede che l'utente fornisca i permessi per l'accesso alla fotocamera. Nel caso in cui l'utente abbia concesso i permessi richiesti, il codice verrà scansionato e inviato al sistema per verificare che esso sia corretto: in caso affermativo il sistema invierà un comando al veicolo per forzare l'apertura delle porte e il noleggio risulterà avviato. In caso negativo l'utente verrà invitato a riprovare scansionando un nuovo codice QR.

Nome	Apertura veicolo
Attori partecipanti	Utente, Veicolo
Precondizioni	L'utente è registrato e ha effettuato una prenotazione
Flusso degli eventi	1. L'utente richiede l'invio del codice QR per sbloccare il veicolo 2. Il sistema genera e invia il codice al veicolo mostrandolo sullo schermo 3. Il sistema verifica di avere i permessi per utilizzare la fotocamera del dispositivo e l'avvia 4. L'utente punta la fotocamera sul codice per scansionarlo 5. Il sistema verifica il codice scansionato dall'utente e sblocca le porte 6. Il sistema avvia il noleggio
Eccezioni	7. Se l'utente non autorizza l'accesso alla fotocamera, al passo 3 essa non viene avviata 7.1 Il sistema invita l'utente a concedere il permesso richiesto e riprovare 8. Se il codice scansionato è errato, al passo 5 il sistema non sblocca le porte 8.1 Il sistema permette all'utente di richiedere un nuovo codice QR e riprovare 9. Se è passato troppo tempo, al passo 6 il sistema non avvia il noleggio 9.1. Il sistema informa l'utente che la prenotazione è stata annullata

3.3.8 Ricarica crediti tramite il rifornimento del carburante

Durante il noleggio l'utente può decidere in qualsiasi momento di effettuare il rifornimento del carburante se il livello non è sufficiente alle sue necessità. Tale operazione prevede l'acquisizione di nuovi crediti nel portafoglio in base all'importo rifornito e viene gestita dal sistema tramite una procedura guidata attivabile dall'utente direttamente dall'app, la quale è scandita da due azioni fondamentali: la richiesta d'inizio del rifornimento e la terminazione dello stesso con invio dell'importo effettuato. Il sistema è in grado di calcolare autonomamente il quantitativo di litri riforniti e il prezzo del carburante al litro a partire dall'importo inserito dall'utente. Per confermare l'operazione e ricaricare i crediti, il sistema verifica che il valore del prezzo al litro ottenuto sia in linea con i prezzi praticati dai rifornitori situati nelle vicinanze del veicolo; tali dati vengono recuperati da un servizio esterno e memorizzati nel database del sistema. Durante la richiesta di terminazione del rifornimento, oltre all'invio dell'importo speso, l'utente può fornire una foto dello scontrino per dimostrare il pagamento. Se la verifica fallisce l'utente ha la possibilità di contattare l'assistenza che deciderà se confermare l'operazione e ricaricare i nuovi crediti anche in base alla foto della prova; invece, se l'importo inserito è considerato valido il sistema memorizza in automatico il rifornimento, i cui dettagli potranno essere visualizzati in seguito.

Nome	Ricarica crediti tramite rifornimento carburante
Attori partecipanti	Utente, Veicolo
Precondizioni	L'utente è registrato e ha avviato un noleggio
Flusso degli eventi	1. L'utente visualizza il livello di carburante residuo e richiede di eseguire il rifornimento 2. Il sistema mostra un suggerimento all'utente prima di avviare la procedura 3. L'utente conferma l'avvio del rifornimento cliccando su Inizia 4. Il sistema recupera dal veicolo l'attuale livello del carburante e abilita la procedura 5. L'utente fa partire l'erogazione del carburante 6. L'utente conclude il rifornimento e clicca su Termina 7. Il sistema richiede all'utente se l'operazione è andata a buon fine 8. L'utente invia il valore monetario dell'importo effettuato e (opzionale) la foto della ricevuta 9. Il sistema recupera dal veicolo l'attuale livello di carburante e verifica che l'importo sia corretto 10. Il sistema conferma il rifornimento e aggiunge i crediti al portafoglio dell'utente
Eccezioni	11. Se l'utente riscontra dei problemi, al passo 8 non invia l'importo 11.1. Il sistema permette all'utente di contattare l'assistenza oppure di annullare la procedura 12. Se il livello del carburante non è cambiato, al passo 10 il sistema non conferma il rifornimento 12.1 Il sistema invita l'utente a verificare che il livello di carburante sia cambiato e a riprovare 13. Se l'importo non è corretto, al passo 10 il sistema non conferma il rifornimento 13.1 Il sistema permette all'utente di modificare l'importo e riprovare 14. Se l'utente invia l'importo oltre 10 minuti dall'inizio della procedura, al passo 10 il sistema non conferma il rifornimento 14.1 Il sistema annulla la procedura In tutti i casi, eccetto l'11.1 e il 14.1, si ritorna allo scenario principale a partire dal passo 8

3.3.9 Termina noleggio

L'utente che ha avviato un noleggio può decidere in qualunque momento di parcheggiare l'auto e terminarlo. Il sistema riceve la richiesta e chiede conferma all'utente, dopodiché invia un segnale al veicolo per forzarne la chiusura; per permettere all'utente di recuperare eventuali oggetti personali e uscire dal veicolo il sistema comanda la chiusura dopo un certo tempo dalla conferma di terminazione. Se si verificano dei problemi durante la richiesta di fine noleggio l'utente può contattare l'assistenza.

Nome	Termina noleggio
Attori partecipanti	Utente, Veicolo
Precondizioni	L'utente è registrato e ha avviato un noleggio
Flusso degli eventi	1. L'utente invia la richiesta di terminazione 2. Il sistema chiede conferma all'utente 3. L'utente conferma 4. Il sistema aggiorna lo stato del noleggio in 'terminato'
Eccezioni	5. Se l'utente non conferma, al passo 4 il noleggio non viene terminato

3.3.10 Visualizza rifornimenti

L'applicazione permette di visualizzare tutti i rifornimenti di carburante effettuati dall'utente organizzati sotto forma di lista; il sistema mostra solo i rifornimenti andati a buon fine. Per ciascuno di essi è riportata l'indicazione sulla data e l'ora in cui il rifornimento è stato eseguito, il quantitativo di litri rifornito e l'importo pagato.

Nome	Visualizza rifornimenti
Attori partecipanti	Utente
Precondizioni	L'utente è registrato
Flusso degli eventi	1. L'utente richiede di visualizzare tutti i rifornimenti di carburante effettuati 2. Il sistema recupera i rifornimenti 'confermati' e li mostra in una lista
Eccezioni	Nessuna

3.3.11 Visualizza dettagli rifornimento

L'utente può accedere in qualunque momento ai dettagli di un singolo rifornimento presente nella lista. Oltre alle informazioni già presenti, il sistema mostra il numero di crediti ottenuti, alcune informazioni riguardo il veicolo interessato dal rifornimento e se presente la foto della ricevuta.

Nome	Visualizza dettagli rifornimento
Attori partecipanti	Utente
Precondizioni	L'utente è registrato
Flusso degli eventi	1. L'utente richiede di visualizzare un rifornimento presente in lista 2. Il sistema mostra i dettagli relativi al rifornimento
Eccezioni	Nessuna

3.4 Progettazione dei dati

L'interazione tra l'utente e il server della piattaforma, durante lo svolgimento delle attività evidenziate nella sezione precedente, sarà basata sullo scambio di una serie di informazioni che dovranno essere organizzate in apposite strutture dati sia sul server che sulle varie applicazioni client. Come verrà mostrato più avanti, l'applicazione mobile non gestirà un vero e proprio database locale, ma utilizzerà delle classi nelle quali verranno incapsulate le informazioni presenti sul server. Dunque, prima di procedere con l'implementazione delle attività descritte, è stato necessario studiare il modello dei dati su cui è basato il database gestito dal server centrale, al fine di comprendere la struttura dei singoli dati e analizzare le relazioni tra di essi. La Figura 3.5 mostra il modello entità-relazione contenente i dati principali trattati dal sistema: vengono riportati gli attributi più significativi e le relazioni tra i dati qualora presenti.

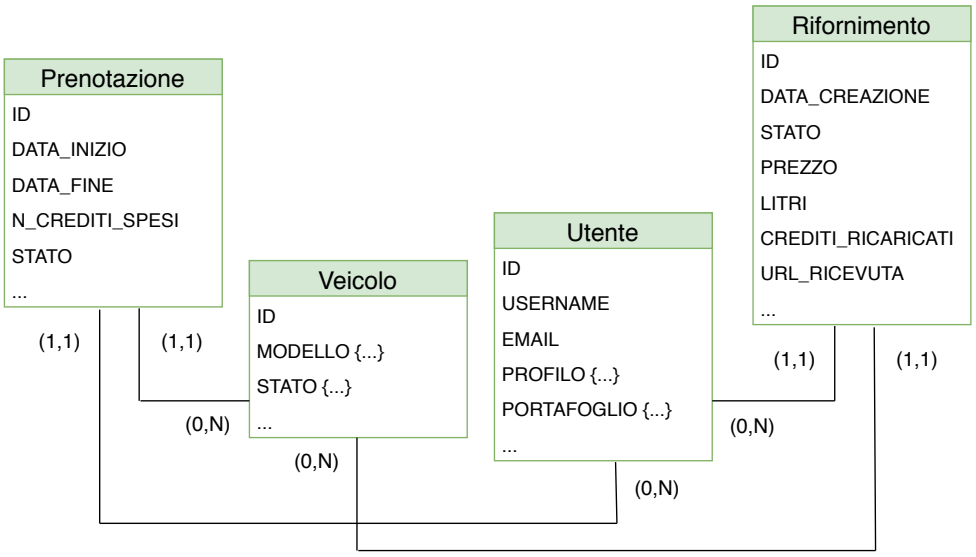


Figura 3.5: Dati del sistema e relazioni principali

Per quanto riguarda l'entità 'veicolo' esso è costituito dall'attributo 'modello' che descrive le caratteristiche del veicolo, tra cui il nome, il tipo di motore, il numero di porte e la capacità del serbatoio. L'attributo 'stato' raccoglie una serie di informazioni riguardanti lo stato del veicolo in un determinato momento: troviamo il livello in percentuale di carburante nel serbatoio, lo stato delle portiere (chiuse o aperte), lo stato di prenotazione (disponibile, prenotato, ecc.) e i dati sulla posizione geografica in cui si trova. Per il singolo veicolo è disponibile anche l'indicazione sul numero di crediti al minuto richiesti per la prenotazione. Considerando l'entità 'utente' viene indicato l'attributo 'profilo', il quale include tutte le informazioni personali dell'utente come il nome, il cognome, la data e il luogo di nascita, il numero di telefono e altri dati. L'attributo 'portafoglio' descrive il contenuto del portafoglio virtuale dello specifico utente, con l'indicazione della data di creazione e di modifica dello stesso e il numero di crediti disponibili.

Capitolo 4

Implementazione della soluzione

4.1 Architettura applicazione mobile

Per supportare le attività descritte nella sezione 3.3, è importante progettare e organizzare una buona architettura software identificando i diversi componenti in base al ruolo che essi devono ricoprire e definendo le interazioni tra gli stessi all'interno del sistema. L'applicazione Splash aderisce in parte al *pattern* architetturale MVC, acronimo di *Model-View-Controller*, il quale prevede la suddivisione della struttura software di un'applicazione in tre macro componenti:

- Model: rappresenta i dati dell'applicazione e la logica che si occupa di manipolarli in lettura e scrittura
- View: definisce tutto ciò che compone l'interfaccia grafica verso l'utente per presentare i dati del modello
- Controller: implementa la logica di controllo dell'applicazione e svolge la funzione di intermediario tra la vista e il modello; traduce i comandi ricevuti dall'utente tramite la vista in azioni eseguite sul modello e viceversa, notifica la vista nel caso di aggiornamenti ai dati nel modello.

L'obiettivo principale del MVC è quello di disaccoppiare i dati del sistema dalla loro rappresentazione, così da renderli riutilizzabili in diversi contesti. La struttura dell'applicazione Splash segue tale *pattern* in maniera non rigida, sfruttando dunque i benefici della separazione dei dati dalle viste in grado di presentarli, ma allo stesso tempo affidando ad un unico componente la responsabilità di visualizzarli e interagire con il modello per modificarli. Considerando il sistema Android, lo stesso SDK permette di separare i diversi strati di un'applicazione: infatti è possibile definire la struttura delle viste, detti *layout*, attraverso l'utilizzo di risorse `XML` memorizzate in cartelle separate dagli altri componenti responsabili di gestire la logica dell'applicazione, come ad esempio le `Activity`. In questo senso si può dire che si realizza una separazione tra la visualizzazione dei dati e la loro manipolazione. Tuttavia c'è da dire che spesso ogni `Activity` è associata ad una di queste risorse che ne rappresentano appunto l'interfaccia con la quale gli utenti potranno interagire; in questo caso l'`Activity` possiede un riferimento ai componenti definiti nel *layout*, può modificarne direttamente il loro stato e rispondere all'interazione dell'utente con ognuno di essi, registrando degli appositi `Listener`. Come è stato detto in precedenza questo è il tipico ruolo della componente *View* nell'architettura MVC. Dall'altro lato

invece, proprio come un *Controller*, l'`Activity` si occupa generalmente di eseguire azioni che vanno oltre l'aggiornamento della *UI* tra cui la modifica dei dati presenti nel *Model*. Nell'applicazione Splash dunque si è scelto di seguire il pattern architetturale MVC tenendo in considerazione il sistema Android. La figura 4.1 mostra in dettaglio come avviene l'interazione tra i diversi componenti *View*, *Controller* e *Model* nell'applicazione Splash a seguito di un'azione eseguita dall'utente. Si nota come il *Controller* è l'unico elemento che si interfaccia con il *Model*: quando si verifica un evento su uno degli oggetti grafici collocati nella *View* (ad esempio il click su un bottone), il *Controller* decide come interagire con il *Model*, modificandolo se necessario, e successivamente aggiorna lo stato della *View* per riflettere i cambiamenti avvenuti sul *Model*; lo schema mostra un tipo di *Model* completamente passivo.

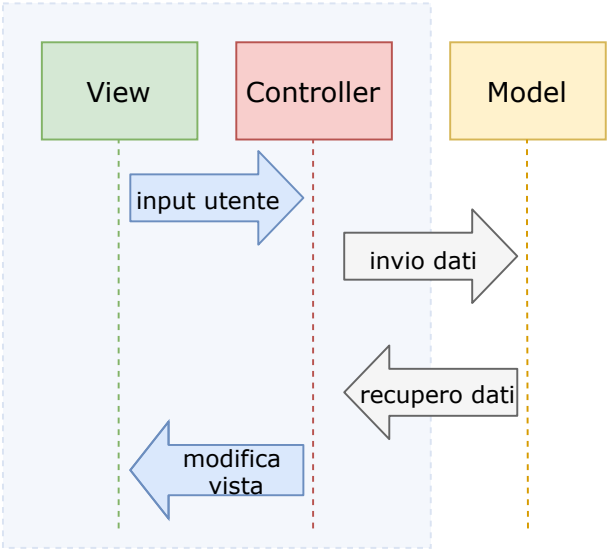


Figura 4.1: Interazione MVC nell'applicazione Splash

Recupero dati dal Model

Nell'architettura MVC il *Model* è la componente responsabile di fornire l'accesso ai dati utili all'applicazione: si occupa di effettuare la traduzione tra il formato in cui i dati sono memorizzati alla sorgente (generalmente un file, un database o un Web Service) e il formato in cui il *Controller* e la *View* si aspettano di riceverli. Come si è detto in precedenza a proposito dell'architettura generale del sistema Splash, la sorgente dei dati è mantenuta e gestita nel server centrale remoto e l'accesso ai vari client è offerto tramite il Web Service REST. L'applicazione Splash, dunque, non gestisce un proprio database locale e, come sarà mostrato più avanti, solo alcune informazioni utili al suo funzionamento verranno memorizzate in maniera persistente nel dispositivo dell'utente. Al modello dei dati remoto corrisponde un modello dei dati in locale: il *Model* dell'app è costituito da una serie di classi che incapsulano una rappresentazione dei dati presenti sul server; tale rappresentazione è ottenuta grazie all'interazione del *Controller* con il Web Service per mezzo di richieste HTTP. Gran parte della logica dell'applicazione è basata su questa interazione,

schematizzata in figura 4.2, in cui il *Controller* è incaricato di interrogare lo stato del *Model* in qualsiasi momento, a seguito delle azioni dell'utente. Dato che la sorgente dei dati è esterna, l'applicazione necessita di utilizzare continuamente una connessione ad Internet per accedervi e questo può introdurre una certa latenza tra la richiesta dei dati e la loro visualizzazione nell'interfaccia utente. Il *Controller* affida l'esecuzione di tutte le richieste HTTP a dei `Task` in grado di processarle in background, senza bloccare l'interfaccia. Questi ultimi attendono il completamento dell'operazione e restituiscono un risultato al *Controller* utilizzando una modalità di comunicazione realizzata tramite le interfacce: la classe che rappresenta il `Task` definisce un'interfaccia con una serie di metodi da chiamare all'occorrenza di un evento, ad esempio la presenza di una risposta contenente i dati o un errore, mentre il *Controller* interessato ne fornisce l'implementazione. Alla ricezione della risposta, prima che venga restituito un risultato al Controller, il *Model* locale effettua la mappatura dei dati in essa contenuti all'interno delle classi che rappresentato i dati dell'applicazione; infine tali dati passano al *Controller* che li elabora per poi aggiornare lo stato della *View*.

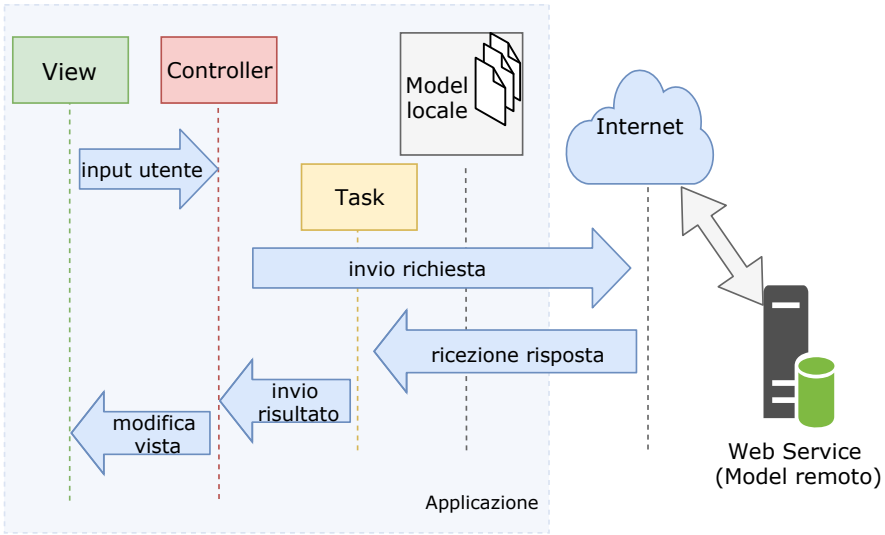


Figura 4.2: Recupero dati dal Model remoto

4.2 Le librerie

4.2.1 Libreria OkHttp

Come già discusso in precedenza, l'applicazione mobile necessita di interagire con il Web Service per reperire i contenuti e fornire all'utente l'accesso al servizio di car-sharing dal proprio dispositivo: la comunicazione tra l'applicazione e il resto del sistema si basa sull'utilizzo delle API REST mediante richieste HTTP. Diventa dunque indispensabile progettare un meccanismo in grado di instaurare una connessione alla rete Internet, che semplifichi la gestione della connettività, del protocollo HTTP e delle relative richieste e risposte evitando di dover agire sui dettagli di basso livello. Esistono diverse librerie per la piattaforma Android che consentono di realizzare un client HTTP; in questo caso si è scelto

di utilizzare la libreria open-source *OkHttp* che fornisce il supporto per HTTP e HTTP/2 sia per applicazioni Android, sia per Java. Essa permette di costruire le richieste HTTP specificando l'URL della risorsa a cui si vuole accedere, il tipo di operazione da eseguire (GET, POST, PUT, DELETE), una lista di *headers* ed eventualmente un corpo contenente una serie di informazioni organizzate secondo uno specifico formato adatto allo scambio dei dati (come JSON e XML). Per utilizzare la libreria è necessario creare un nuovo client di tipo `OkHttpClient` responsabile di inoltrare le richieste e ricevere le relative risposte: tali operazioni vengono gestite attraverso la creazione di un oggetto `Call` che rappresenta una richiesta pronta ad essere eseguita in un qualsiasi momento. `OkHttpClient` supporta l'esecuzione di chiamate in due modalità differenti: quella sincrona attivata tramite il metodo `execute()` che causa il blocco del thread chiamante in attesa della risposta e quella asincrona, realizzata con il metodo `enqueue()` che inserisce la richiesta in una coda e programma la sua esecuzione nel futuro; in quest'ultimo caso la risposta viene restituita non appena è disponibile invocando una specifica `Callback`. Ciascun client gestisce le richieste a lui commissionate riutilizzando una tra le connessioni libere presenti nel proprio `ConnectionPool` al fine di ridurre la latenza di rete. Inoltre, le richieste HTTP verso lo stesso indirizzo possono condividere la stessa connessione sottostante evitando di dover eseguire per ogni richiesta alcune operazioni di configurazione che richiedono un certo tempo, come il *three-way handshake* di TCP, processo che porta a stabilire una nuova connessione TCP; questo porta ad un aumento delle prestazioni in termini di maggiore velocità, minore latenza e consumi della batteria ridotti. Alla ricezione di una risposta la connessione viene restituita al `ConnectionPool` così da essere pronta per le richieste future. Un altro modo per migliorare le prestazioni, evitando lo spreco di risorse, è quello di creare un'unica istanza della classe `OkHttpClient` con una certa configurazione, così da poterla condividere tra tutte le chiamate HTTP, in modo che tutte le richieste verranno eseguite dallo stesso client mediante il proprio `ConnectionPool`. Altre caratteristiche avanzate della libreria sono: la capacità di memorizzazione (*caching*) delle risposte HTTP da rendere disponibili per successive richieste sulla stessa risorsa, evitando così un'ulteriore accesso alla rete, e la funzionalità di recupero dai comuni problemi di connessione. Infine, la libreria offre un modo per garantire la sicurezza di una connessione tramite l'utilizzo di HTTPS, con la possibilità di specificare la versione TLS e la *cipher suite* desiderata, ovvero l'insieme di algoritmi crittografici scelti per la protezione della comunicazione in rete tra client e server.

4.2.2 Libreria Gson

Per consentire al dispositivo mobile di comunicare con il server centrale è stato doveroso definire un formato per lo scambio dei dati, i quali andranno interpretati e memorizzati prescindendo dalle tecnologie informatiche impiegate nello sviluppo dei due componenti: è richiesto dunque un formato adatto all'incapsulamento di dati e indipendente dai linguaggi di programmazione utilizzati lato client e lato server. JSON (RFC-5280 [7]), acronimo di *JavaScript Object Notation*, è un formato testuale per lo scambio di dati la cui sintassi deriva dalla notazione degli oggetti di JavaScript, caratterizzato dall'essere indipendente dalla piattaforma e dal linguaggio utilizzato; sebbene discenda da JavaScript, esso ne è totalmente slegato, in quanto può essere interpretato correttamente da altri linguaggi come ad esempio PHP, Java, C, Python e tanti altri. Uno dei suoi punti di forza è quello di possedere una struttura semplice da comprendere per gli sviluppatori ma anche facile da elaborare per i dispositivi: permette di immagazzinare varie tipologie di informazioni come coppie *chiave-valore* e spesso viene paragonato al formato XML (nonostante quest'ultimo sia un linguaggio di *markup* che descrive i dati attraverso dei marcatori detti *tag*) poiché si

presta all'archiviazione dei dati secondo un modello gerarchico, adattandosi a situazioni in cui è necessario rappresentare relazioni complesse. Tuttavia i documenti in formato JSON tendono ad avere dimensioni più piccole rispetto a quelli in formato XML e questo è un aspetto rilevante sia per la memorizzazione che per lo scambio, soprattutto quando si ha a che fare con dispositivi, come quelli mobili, caratterizzati da risorse limitate. Per quanto riguarda la formattazione del testo, un documento JSON è costituito da una sequenza di *token* che definiscono la struttura e le informazioni contenute, le quali possono essere di vario tipo:

- stringhe
- numeri
- valori booleani (*true*, *false*) e nulli
- oggetti (collezioni di coppie *chiave-valore* dove la chiave è una stringa, mentre il valore può essere di qualsiasi tipo tra quelli standard)
- array (lista ordinata di valori)

Lo standard, dunque, specifica sia i tipi di dato ammessi, sia i caratteri che delimitano e identificano le informazioni all'interno della struttura, rendendo JSON estremamente versatile e facilmente adattabile alla rappresentazione di qualsiasi informazione: questo spiega la sua rapida diffusione in svariati ambienti, dalle comunicazioni browser-server ai dispositivi mobili fino all'impiego nei sistemi IoT dove si presta particolare attenzione alle prestazioni.

Oggigiorno i più moderni linguaggi di programmazione supportano nativamente meccanismi per la manipolazione di documenti JSON, lo stesso Android SDK include fin dalle origini (livello API 1) il package *org.json* contenente alcune classi capaci di eseguire l'analisi direttamente da un file, oppure da una stringa; tuttavia, esse seguono una strategia di *parsing* simile a DOM (Document Object Model)[1], una convenzione per la rappresentazione di documenti in maniera gerarchica sotto forma di albero il cui funzionamento prevede il salvataggio dell'intero documento in memoria, prima che possa essere interpretato, senza tener conto di quante informazioni verranno effettivamente utilizzate dall'applicazione. Questo aspetto rappresenta un grosso problema per le applicazioni mobile, essendo intrinsecamente vincolate dalle ridotte capacità dei dispositivi su cui eseguono, per cui, avvalendosi di una tale strategia, l'analisi di una grande quantità di dati potrebbe addirittura causare l'esaurimento della memoria. Per superare tali debolezze è stata introdotta, a partire dal livello 11 delle API, la classe `JsonReader` caratterizzata da una tecnica di *parsing* completamente diversa, basata su un modello ad eventi derivato da SAX (Simple API for XML): essa lavora direttamente sul flusso JSON in ingresso, navigando attraverso le sue parti e generando eventi man mano che si presenta un elemento significativo. Rispetto al meccanismo precedente è più veloce e meno oneroso in termini di consumo di risorse, perché memorizza solamente la riga del documento da processare, scartandola nel passaggio alla successiva; nonostante sia un modello ottimizzato, si tratta comunque di un supporto di basso livello e in aggiunta ha lo svantaggio di non essere compatibile con le versioni di Android precedenti all'API 11, perciò, diventa indispensabile l'integrazione di

[1] fonte:
https://it.wikipedia.org/w/index.php?title=Document_Object_Model&oldid=99393377

una libreria esterna per colmare questa mancanza. Esistono diverse librerie per la piattaforma Android, in grado sia di realizzare le due tecniche di *parsing* appena descritte, sia di semplificare e automatizzare l'interpretazione di un testo JSON, offrendo funzionalità ancora più avanzate.

Nel caso dell'applicazione Splash, per eseguire il *parsing* dei dati trasferiti durante l'interazione con il Web Service, si è scelta la libreria GSON che oltre ad essere molto diffusa, rappresenta il giusto compromesso in termini di flessibilità e prestazioni. Si tratta di un progetto open-source sviluppato da Google originariamente per l'utilizzo nell'ambito di progetti interni, che è stato in seguito reso pubblico e attualmente è impiegato in svariati progetti sviluppati in linguaggio Java. GSON ha la capacità di trasformare in modo veloce ed efficiente un oggetto Java nel formato JSON e viceversa, analizzandone il tipo o le proprietà e manipolandone i valori a tempo di esecuzione: si contraddistingue dalle altre librerie disponibili perché, per effettuare il *parsing*, non richiede la modifica delle classi per l'aggiunta di annotazioni particolari, così è possibile convertire anche gli oggetti di cui non si possiede il codice sorgente; la mappatura da/verso JSON funziona automaticamente basandosi sui nomi delle variabili d'istanza dell'oggetto e non sui metodi `get()`, come avviene in altre librerie, quindi ogni variabile membro dell'istanza viene mappata sul corrispondente campo JSON che abbia esattamente lo stesso nome, altrimenti il campo viene posto a `null`. Tuttavia, è possibile modificare tale comportamento utilizzando l'annotazione `@SerializedName` su ciascuna istanza, specificando in maniera esplicita il nome alternativo da utilizzare nella conversione; tramite le annotazioni è addirittura possibile escludere alcuni campi dal processo di serializzazione/deserializzazione aggiungendo l'attributo `@Transient`.

La creazione dell'istanza `Gson` da utilizzare durante il processo di conversione può avvenire in due modalità differenti:

- invocando `Gson()`, che richiama il costruttore della classe `Gson` senza nessun argomento, creando un'istanza che sfrutta le opzioni di *parsing* di default
- utilizzando la classe `GsonBuilder`, basata sul *build-pattern*, che permette allo sviluppatore di configurare le impostazioni base di `Gson`

Per serializzare un oggetto partendo dall'istanza `Gson` ottenuta mediante uno dei due metodi precedenti, bisogna invocare il metodo `toJson()` passando come argomento l'oggetto da trasformare, il cui risultato è rappresentato da una stringa in formato JSON; invece, l'operazione inversa avviene invocando il metodo `fromJson()`, che riceve in ingresso sia la stringa JSON contenente i dati, sia la classe nella quale essi andranno trasferiti. Entrambe le trasformazioni vengono eseguite in automatico dalla libreria GSON, sfruttando i serializzatori e i deserializzatori di default per tutti i campi presenti: qualora la conversione automatica non fosse appropriata per alcuni di questi campi, è possibile creare un convertitore personalizzato per l'oggetto generico `<T>`, estendendo la classe astratta `TypeAdapter<T>`. Per far questo è necessario fornire l'implementazione dei metodi `write()` e `read()`, invocati rispettivamente in fase di serializzazione e deserializzazione, all'interno dei quali si specifica come dovranno essere convertiti i campi dell'oggetto Java nel documento JSON e viceversa; in questo caso, affinché la libreria possa utilizzare il nuovo *adapter* in fase di conversione, è necessario costruire l'istanza `Gson` a partire dalla classe `GsonBuilder` che, come detto in precedenza, permette di specificare alcune proprietà, tra cui la registrazione di un *adapter* alternativo per la manipolazione degli oggetti di tipo `<T>`. Spesso però la scrittura di un'intera classe che serializza e deserializza tutti i campi non è la soluzione migliore, soprattutto se si è interessati a modificare la conversione

di alcuni di essi, mentre per i rimanenti si intende sfruttare quella di default offerta da GSON: infatti, questo richiederebbe la scrittura di codice ulteriore per il trattamento di tutti i campi, non solo per quelli di cui si vuole fornire una conversione personalizzata; in più, qualsiasi modifica alla classe `<T>` (aggiunta o rimozione di un campo) richiederebbe la modifica del `TypeAdapter` corrispondente. Inoltre, esistono altre situazioni più complesse in cui è conveniente intervenire al termine della serializzazione/deserializzazione base per apportare le modifiche desiderate, ad esempio se l'attribuzione del valore di un campo dipende da altri presenti all'interno della stessa struttura JSON. Fortunatamente la libreria è abbastanza flessibile e per la gestione dei casi particolari, in cui si vuole evitare la creazione di un intero *adapter*, mette a disposizione l'interfaccia `TypeAdapterFactory`, la quale funge da ponte tra la classe interessata dalle modifiche e il `TypeAdapter` predefinito utilizzato dalla libreria durante la conversione dei suoi campi. In questo modo, registrando presso l'istanza `Gson` la classe che implementa `TypeAdapterFactory`, la libreria consentirà di catturare direttamente i dati in uscita dall'*adapter* per poterli modificare prima che venga prodotto il risultato finale, ovvero l'oggetto Java nella fase di deserializzazione, oppure la stringa JSON nella fase di serializzazione.

GSON rappresenta dunque un valido strumento per il *parsing* di documenti JSON, che permette di eseguire trasformazioni in maniera semplice e pressoché automatica con i metodi `toJson()` e `fromJson()`. Esso offre in più allo sviluppatore una configurazione precisa di come i JSON debbano essere convertiti in oggetti Java; inoltre, supporta ampiamente i tipi generici e facilita la gestione di oggetti più complessi, come nel caso di quelli polimorfici.

4.2.3 Google Play Services API

I *Google Play Services* rivestono un ruolo sempre maggiore nella realizzazione delle applicazioni mobile. Si tratta di una libreria composta da una collezione di API attraverso cui è possibile accedere alle più importanti funzionalità offerte dalla piattaforma Google, con il vantaggio di ottenere sempre gli ultimi aggiornamenti. L'utilizzo di tali servizi richiede l'inclusione di un elemento `<meta-data>` nel file *AndroidManifest.xml* che indica la versione della libreria *Google Play Services* con cui l'applicazione è stata compilata:

```
<meta-data
android:name="com.google.android.gms.version"
android:value="@integer/google_play_services_version"/>
```

Dopodiché è necessario aggiungere al progetto le dipendenze dalle API che si intende utilizzare. Tra le API offerte da Google, analizzeremo quelle che hanno permesso lo sviluppo dell'applicazione Splash e in modo particolare le API per la gestione delle mappe, per la localizzazione geografica e il calcolo di percorsi tra due posizioni; in ultimo presenteremo l'API per il rilevamento dei codici QR dalle immagini.

Google Maps API e Location API

Una delle funzionalità fondamentali che un'applicazione per il car-sharing deve offrire ai propri utenti consiste nella localizzazione dei veicoli della flotta. L'utente deve conoscere la posizione del veicolo prima di decidere se prenotarlo o meno; inoltre, esso può avere la necessità di raggiungerlo nel minor tempo possibile, per cui è importante conoscerne la distanza. Per questo motivo, è necessario che l'applicazione interagisca con un servizio di localizzazione in grado di rilevare la posizione degli utenti e di fornire loro le informazioni

dettagliate sui percorsi che conducono verso i veicoli prenotati. In aggiunta, è preferibile disporre di una mappa in cui poter evidenziare, per mezzo di opportuni segnaposto, la posizione dei veicoli e quella degli utenti, rispetto alla semplice visualizzazione di informazioni testuali. Questo può contribuire a ridurre i tempi di ritrovamento dei veicoli e a migliorare l'esperienza d'uso dell'app. Tramite la libreria *Google Play Services* è possibile implementare le funzionalità appena descritte, utilizzando l'API di *Google Maps* per interagire con le mappe e la *Location* API per rilevare la posizione degli utenti in tempo reale.

Oltre alle informazioni di configurazione descritte in precedenza, per accedere a queste API è necessario registrare il proprio progetto Android tramite la Google API console, accessibile online, per l'ottenimento di una chiave di autenticazione, detta API *key*, da inserire anch'essa nel file *AndroidManifest.xml*:

```
<meta-data
  android:name="com.google.android.geo.API_KEY"
  android:value="API_KEY"/>
```

In questo modo l'applicazione potrà visualizzare correttamente le mappe e accedere agli altri servizi offerti da Google.

Per aggiungere una mappa all'interno di un'applicazione è possibile utilizzare la classe `MapFragment` oppure `MapView` a seconda che il contenitore a gestirla sia un'`Activity` o un `Fragment`. `MapFragment` è il modo più semplice per posizionare una mappa direttamente nel *layout* di un'`Activity` specificando il tag XML `<fragment>` e l'attributo `android:name` con il valore `com.google.android.gms.maps.MapFragment`; altrimenti si può procedere in maniera programmatica aggiungendo dinamicamente all'`Activity` un oggetto instanza di `MapFragment` tramite una `FragmentTransaction`. In entrambi i casi il ciclo di vita della mappa verrà gestito in automatico dall'oggetto `MapFragment` stesso. Per i `Fragment` invece è disponibile la classe `MapView`, una `View` che gestisce sia la visualizzazione di una mappa, sia gli eventi di tocco al suo interno (zoom, rotazione, altro), in modo da renderla interattiva e poterla esplorare più in dettaglio. A differenza di `MapFragment`, l'utilizzo di `MapView` richiede la chiamata ai metodi del proprio ciclo di vita, tra cui il metodo `onCreate()`, `onResume()` ed altri, in corrispondenza degli stessi metodi di `callback` del `Fragment` che la contiene. Una volta posizionata in uno dei due modi appena descritti, l'accesso alla mappa vera e propria si ottiene tramite il metodo `getMapAsync()`, da invocare sul `MapFragment` o sulla `MapView`, che restituisce un'istanza della classe `GoogleMap` su cui potranno essere effettuate le manipolazioni desiderate. Infatti, attraverso questa classe è possibile personalizzare la mappa aggiungendo degli indicatori di posizione, detti `Marker`, utili nell'evidenziare luoghi d'interesse, oppure tracciare percorsi che passano per una serie di coordinate e disegnare poligoni o inserire immagini.

Come si è già detto all'inizio, un importante servizio offerto dalle *Google Play Services* API è la localizzazione geografica dei dispositivi mobili, che permette alle applicazioni di conoscere la posizione dei propri utenti; l'accesso ai dati di localizzazione implica la richiesta dei dovuti permessi da parte dell'applicazione, secondo la modalità prevista dal sistema Android che verrà mostrata nella sezione successiva. Per utilizzare questo servizio bisogna innanzitutto creare un `FusedLocationProviderClient` in grado di determinare la posizione del dispositivo con una certa precisione sfruttando i segnali forniti da più sensori come il GPS e il Wi-Fi, con la possibilità di indicare la qualità dei dati desiderata. L'accuratezza della posizione dipende non solo dal *provider* e dai sensori, ma anche dal tipo di permesso specificato nel file *AndroidManifest.xml* (`ACCESS_COARSE_LOCATION` per dati approssimativi o `ACCESS_FINE_LOCATION` per dati precisi) e dalle opzioni configurate

tramite la classe `LocationRequest` in fase di creazione delle richiesta al *provider*; quest'ultima espone i metodi `setPriority()` e `setInterval()` che consentono rispettivamente di impostare la priorità nell'ottenimento della posizione (influenzando eventualmente la scelta del sensore di localizzazione da parte del *provider*) e l'intervallo degli aggiornamenti, tenendo conto persino del consumo della batteria.

Considerato che il *provider* utilizza i sensori del dispositivo per recuperare i dati sulla posizione, è necessario che essi siano attivi affinché la richiesta possa essere completata. A tal proposito, le API forniscono dei metodi di utilità che permettono all'applicazione di interagire direttamente con le impostazioni di localizzazione del dispositivo per conoscerne lo stato; esiste, infatti, la classe `SettingsClient` che permette, in base alla configurazione della `LocationRequest` fatta al *provider*, di verificare lo stato del GPS o del Wi-Fi . Tale verifica viene eseguita in maniera asincrona, richiamando sull'istanza `SettingsClient` il metodo `checkLocationSettings()`, il quale restituisce un `Task<LocationSettingsResponse>`; collegando a quest'ultimo un *listener*, è possibile conoscere l'esito della verifica non appena il *task* sarà terminato. Nel caso di esito negativo, la stessa classe offre la possibilità di visualizzare una finestra di dialogo come quella mostrata in Figura 4.3, per chiedere all'utente il consenso ad abilitare le impostazioni di localizzazione.

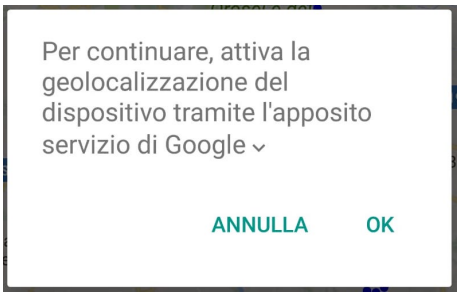


Figura 4.3: Richiesta attivazione impostazioni di localizzazione

Google Directions API

La piattaforma *Google Maps* permette alle applicazioni di ottenere le indicazioni dettagliate di un percorso attraverso un'altra API, detta *Directions* API, che a differenza delle precedenti si basa sull'esecuzione di semplici richieste HTTP GET ad un URL specifico, in cui i dati vengono inviati come *query parameter* separati dal carattere '&' e le risposte formattate in JSON o XML. Il formato della risposta deve essere specificato dal client nell'URL all'atto della richiesta, la quale deve contenere anche la chiave di autenticazione all'interno del parametro `API`; quest'ultima garantisce l'accesso gratuito a tutte le informazioni contenute nei database di *Google Maps* fino ad un certo numero di richieste giornaliere, oltre il quale è richiesto l'acquisto di un apposito abbonamento. Inoltre l'utilizzo dei dati forniti dalla presente API è consentito ad eccezione di quanto espresso nei termini di servizio definiti da Google, che impongono una serie di restrizioni, tra cui l'impossibilità di sfruttare i dati con mappe diverse da quelle di *Google Maps* e la realizzazione di un sistema di navigazione *real-time* [8, sezione 10.4, paragrafo c]. Tra i parametri obbligatori di una richiesta troviamo `origin` e `destination` che indicano l'indirizzo di partenza e di arrivo del percorso, entrambi espressi tramite coordinate terrestri (latitudine e longitudine), in forma testuale (ad esempio `Torino+Via+Roma+18`) oppure utilizzando un identificativo detto *Place ID* che identifica il luogo nei database di Google. Oltre ai

parametri obbligatori è possibile indicare opzionalmente la modalità di attraversamento del percorso, di base impostata su 'driving' ovvero 'alla guida' che sfrutta dunque la rete stradale per il calcolo del tragitto; sono disponibili anche le modalità 'walking' , 'bicycling' e 'transit' per richiedere rispettivamente le indicazioni 'a piedi', 'in bici' e tramite il servizio di trasporto pubblico (autobus, treni e altro). In aggiunta, le API sono in grado di calcolare percorsi che passano attraverso delle posizioni d'interesse specifiche, indicate con il parametro `waypoints`, oppure di rispettare delle restrizioni specificate per mezzo del parametro `avoid`, per evitare le tratte che prevedono ad esempio autostrade o ponti e prediligerne altre. La risposta contiene sempre le informazioni sul miglior percorso trovato, che può essere il più breve in termini di tempo o di distanza, anche se l'algoritmo usato da Google può tener conto di ulteriori fattori. Al suo interno si trovano diversi elementi, tra cui uno `status` che indica appunto lo stato della richiesta e l'elemento `routes` che rappresenta l'insieme di tutte le rotte trovate tra l'indirizzo sorgente e destinazione specificati; tale insieme contiene sempre un solo risultato eccetto nel caso in cui si aggiunga alla richiesta il parametro `alternatives=true`, così che il servizio possa restituire più di un cammino alternativo per la stessa coppia di posizioni. Infine ogni rotta riporta le informazioni sulla durata e sulla distanza di percorrenza totale del percorso e la lista dei passi che lo compongono.

Mobile Vision API

La Mobile Vision API è un altro dei servizi offerti dai *Google Play Services*, che include una serie di strumenti in grado di rilevare volti di persone, codici a barre e testo da immagini e video. Il funzionamento di questa libreria è basato su degli oggetti di tipo `Detector`, i quali ricevono in ingresso uno o più `Frame` di un'immagine e producono come risultato un array contenente gli elementi rilevati. È possibile utilizzare un `Detector` per estrarre gli oggetti di interesse da una singola immagine, oppure da un flusso continuo di frame prodotto da una sorgente, quale ad esempio la fotocamera di un dispositivo mobile. In quest'ultimo caso, è necessario associare l'istanza di un oggetto di tipo `Detector.Processor` al `Detector` per ottenere gli elementi da lui rilevati all'interno dei frame ricevuti dalla sorgente. La libreria è costituita da tre *package*, ciascuno progettato per rilevare un elemento preciso tra quelli elencati sopra. Servendosi delle classi e dei metodi presenti nei diversi *package*, è possibile integrare nelle applicazioni mobile le seguenti funzionalità:

- rilevamento facciale: per individuare delle zone di un'immagine che corrispondono a dei volti umani. Ad ogni volto viene associata una posizione, una dimensione e un orientamento, i cui valori sono relativi all'immagine elaborata da un `FaceDetector`. Permette di identificare la posizione dei punti di riferimento di un volto come gli occhi, la bocca e il naso e inoltre, fornisce informazioni sulle espressioni facciali, tra cui la presenza di occhi aperti o chiusi e di una bocca sorridente o meno. Questi dati possono essere sfruttati per personalizzare i volti aggiungendo effetti e decorazioni, ma anche per eseguire delle azioni all'occorrenza di una certa espressione.

- riconoscimento di testo: per estrarre e riconoscere un qualsiasi testo scritto in una tra le diverse lingue di derivazione latina. Il *detector*, di tipo `TextRecognizer`, rileva un testo organizzandolo secondo una struttura costituita da blocchi, linee e parole, che riflette quella originale. Tale funzionalità è basata sulla tecnologia OCR (*Optical*

Character Recognition)[2], dedicata al rilevamento di caratteri presenti in documenti e alla loro conversione in formato digitale.

- scansione di codici a barre: per il rilevamento e la decodifica di codici a barre di vario tipo (unidimensionali e bidimensionali) attraverso l'utilizzo di un `BarcodeDetector`. Ciascun codice a barre è identificato dal tipo (EAN-13, EAN-8, Data Matrix, Code-128, PDF-417, QR code, ecc.) e dal valore in esso contenuto.

In merito all'applicazione Splash si utilizza la presente libreria per la realizzazione di un lettore di codici QR da utilizzare durante lo sblocco delle portiere di un veicolo. Si tratta di un sistema versatile, ormai diffuso in svariati settori, da quello automobilistico a quello commerciale, che si distingue dagli altri codici a barre per la capacità di memorizzazione e la facilità di utilizzo. Infatti, basta inquadrarli con la fotocamera di un qualunque dispositivo in cui è installato un lettore di codici QR, per aver accesso ad una serie di servizi e informazioni. I codici QR permettono di memorizzare caratteri numerici e alfanumerici, così da poter codificare indirizzi Internet, testi, numeri di telefono, SMS, email e immagini. Inoltre, si distinguono da altri tipi di codice a barre per la rapidità offerta in fase di decodifica.

4.3 Git - Controllo di Versione

Nello sviluppo di un progetto, sia esso piccolo o grande, è indispensabile tenere traccia della continua evoluzione del codice sorgente nel tempo, dell'aggiunta e/o della rimozione di file, delle modifiche apportate agli stessi e delle versioni rilasciate. Inoltre, è consigliabile tenere al sicuro una o più copie dello stato del progetto in un certo istante, in modo da evitare una perdita permanente dell'intero lavoro svolto, a seguito di malfunzionamenti e danni improvvisi. Quando più persone partecipano allo sviluppo dello stesso progetto, si ha la necessità di suddividere il lavoro in più parti, ciascuna portata a termine dal singolo, da integrare successivamente nel progetto condiviso risolvendo eventuali conflitti dovuti a modifiche apportate agli stessi file; talvolta risulta utile addirittura annullare tali modifiche per ritornare ad una condizione funzionante precedente. Nasce così l'esigenza di affiancare al progetto un potente strumento di versionamento del codice, che consente agli sviluppatori di lavorare in piena autonomia simultaneamente, di tracciare la cronologia e l'attribuzione dei file di progetto nel tempo, e ancora di scambiarsi le modifiche e sincronizzare il lavoro. Uno dei più diffusi è Git[3], un sistema di controllo di versione distribuito e open-source, ideato per gestire progetti anche di grandi dimensioni in modo veloce ed efficiente; il controllo di versione non è altro che un modo per registrare nel tempo i cambiamenti apportati ad un file o ad una serie di file da un certo autore, revisionare tutte le modifiche ed, eventualmente, ripristinare lo stato di un file ad una specifica versione nel caso di problemi.

Git prevede la creazione di molteplici copie dei dati del progetto, a partire da un *repository* centralizzato, attraverso un'operazione detta di clonazione; in questo modo, tutti coloro che partecipano al progetto possono lavorare in parallelo sulla propria versione

[2] fonte:
https://it.wikipedia.org/w/index.php?title=Riconoscimento_ottico_dei_caratteri&oldid=97161656

[3] fonte:
https://git-scm.com/

dei dati. Il *repository* centrale è necessario per garantire che tutte le copie locali siano aggiornate e coerenti tra loro. Esistono numerosi servizi, gratuiti e non, per la creazione di un *repository* centralizzato da condividere tra sviluppatori: per il progetto Splash si utilizza Gitlab[4]. Si tratta di una piattaforma preposta all'archiviazione (*hosting*) e al controllo di progetti software, che permette dunque agli sviluppatori di ospitare i propri progetti in modo pubblico o privato e di partecipare in maniera collaborativa e distribuita allo sviluppo degli stessi. Come si è detto in precedenza, è richiesta la creazione di un *repository* attraverso cui organizzare un singolo progetto: quest'ultimo dovrà essere clonato da tutti gli sviluppatori partecipanti, in modo da avere sul proprio disco, in locale, l'intera copia dei dati presenti sul server di Gitlab; ciascuno lavorerà sulla propria versione locale del *repository* e successivamente dovrà sincronizzarsi con quello centrale nel caso di modifiche al progetto.

Accanto al concetto di *repository* occorre menzionare quello dei suoi elementi costitutivi, i *commit*: ogni *repository* non è altro che una sequenza di *commit*, considerati come una sorta di contenitori di modifiche. Ogni volta che si modifica un file, Git si accorge che esso è cambiato rispetto all'ultimo *commit* di cui tiene traccia: per salvare tali modifiche all'interno del *repository* occorre crearne uno nuovo e così ancora per le successive modifiche. Ad ogni *commit* è possibile assegnare una descrizione che riassume la ragione dei cambiamenti apportati; per ognuno di essi Git produce un'istantanea (snapshot) dei file modificati del progetto, memorizzandone il loro stato dopo il cambiamento. Prima di creare un *commit* è necessario indicare quali dei file modificati debbano essere inclusi tramite il comando `git add <nome_file>`; tale operazione aggiunge il file in un'area particolare, detta area di stage, in cui si trovano tutte le modifiche in attesa di essere salvate nel prossimo *commit*. È opportuno sottolineare che Git prevede tre possibili stati per un file: *unmodified* è lo stato base, ossia nessun cambiamento rispetto allo stato precedente, *modified* quando sono presenti modifiche sul file non ancora aggiunte all'area di stage e infine *staged* per le modifiche pronte al *commit*; durante il processo di sviluppo i file aggiunti al progetto passano continuamente da uno stato all'altro seguendo il ciclo di vita mostrato in figura 4.4, fino all'ottenimento di una versione definitiva. Git consente inoltre di escludere alcuni file dalla gestione del versionamento mantenendoli in uno stato detto *untracked*.

Tutte le modifiche apportate tramite *commit* rimarranno nella versione locale del repository, sconosciute agli altri sviluppatori, fintanto che non si effettua l'operazione di `push`, che trasferisce le modifiche locali sul server nel *repository* remoto condiviso. Per aggiornare le altre copie locali con i nuovi commit, invece, è necessario eseguire l'operazione inversa, detta `pull`, che trasferisce le modifiche remote in locale.

Un'altra particolarità di Git è il *branching*, un potente sistema di ramificazioni adatto ad uno sviluppo non lineare, che permette di strutturare il progetto in vari rami, detti *branch*, nei quali si ha la possibilità di apportare nuove modifiche prima che siano pronte per far parte del *branch* definitivo, il cosiddetto *master*. Creando un nuovo *branch* a partire dal *master* si crea di fatto una copia di quest'ultimo; le successive modifiche effettuate al *branch* principale non verranno propagate a quello secondario e viceversa. Una volta che le funzionalità sviluppate sono pronte per essere integrate nel *master* ha luogo l'operazione di *merge*: le modifiche contenute nel *branch* secondario andranno trasferite con un *commit* su quello principale, dove saranno unite con quelle eventualmente aggiunte in seguito alla creazione del *branch*. Infine, nel caso in cui siano state effettuate modifiche agli stessi file

[4] fonte:
https://gitlab.com/

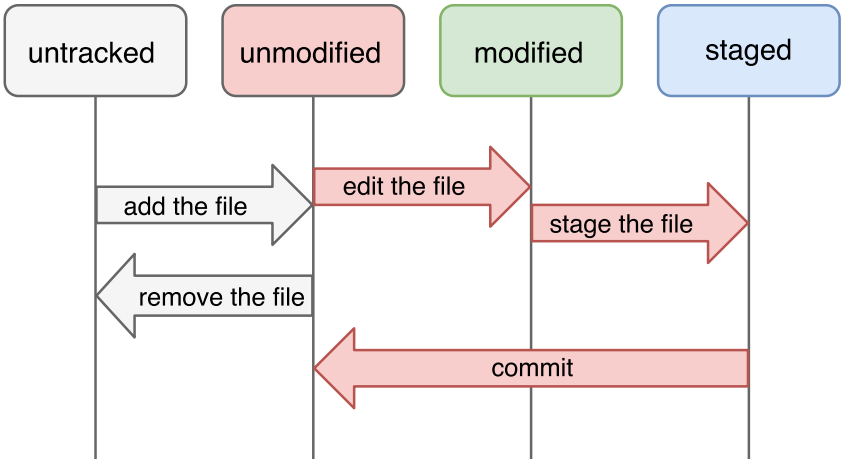


Figura 4.4: Ciclo di vita dello stato di un file Git

su entrambi i *branch*, possono verificarsi dei conflitti che devono essere risolti affinché il *merge* venga completato.

4.4 Gestione dei Permessi

Nel sistema Android ogni applicazione esegue all'interno del proprio ambiente, in completo isolamento rispetto alle altre e possiede una cartella dedicata nella memoria del dispositivo in cui leggere o scrivere i propri dati. L'utilizzo di informazioni e funzionalità esterne a questo ambiente (offerte dal sistema stesso o da altre applicazioni) è protetto da una serie di autorizzazioni: le applicazioni possono accedere unicamente alle risorse per le quali hanno ottenuto un'esplicita autorizzazione all'uso. La gestione dei permessi necessari al funzionamento di un'applicazione varia a seconda del tipo di permesso richiesto e della versione di Android in esecuzione nel dispositivo. I permessi sono classificati tenendo conto del livello di protezione richiesto dalle funzionalità e dai dati per cui ne autorizzano l'accesso; ciò determina la procedura seguita dal sistema nel concedere o meno l'autorizzazione in esame. È possibile distinguere tra i seguenti tipi di permessi:

- *normali*: sono considerati tali i permessi che autorizzano un'applicazione ad accedere a risorse esterne al proprio ambiente senza compromettere la privacy dell'utente, oppure nel caso in cui il rischio sia minimo; essi vengono concessi in automatico dal sistema al momento dell'installazione, senza chiedere il consenso all'utente.

- *pericolosi*: appartengono a questa categoria i permessi che forniscono l'accesso ai dati privati dell'utente (ad esempio i contatti della rubrica) oppure a risorse che ne mettono a rischio la sua privacy (come l'utilizzo della fotocamera o dei servizi di localizzazione). A differenza dei precedenti non vengono concessi in automatico ma necessitano il consenso da parte dell'utente.

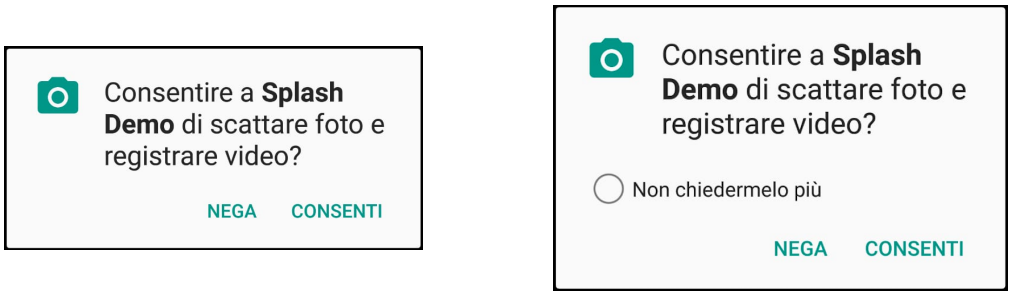
- di *firma* o *signature*: sono quei permessi concessi automaticamente dal sistema all'installazione, solamente se l'applicazione richiedente il permesso è firmata con lo stesso certificato dell'applicazione che lo ha dichiarato. È il caso delle applicazioni di

sistema che essendo firmate con il codice di firma della piattaforma Android possono ottenere i permessi di sistema con un livello di protezione di tipo *signature*. Tuttavia, anche le applicazioni di terze parti possono sfruttare il livello di protezione *signature* definendo dei permessi personalizzati; in questo modo un'applicazione può limitare l'accesso ai suoi componenti (`Activity` e `Service`) ad altre app create dallo stesso autore.

Tutti i permessi di cui un'applicazione ha bisogno, indipendentemente dal loro tipo, vanno dichiarati nel file *AndroidManifest.xml* aggiungendo il tag XML `<uses-permission>` e specificando il nome del permesso richiesto (ad esempio per ottenere l'accesso ad Internet si utilizza il permesso con nome `android.permission.INTERNET`). Qualora siano presenti permessi *pericolosi* il sistema deve poter chiedere l'autorizzazione all'utente in maniera esplicita. La modalità di richiesta differisce in base alla versione Android del dispositivo in cui esegue l'applicazione. Infatti, sino alla versione Android 5.1.1 (livello API 22) tutti i permessi classificati *pericolosi* vengono richiesti una sola volta al momento dell'installazione dell'app e non potranno più essere revocati; questo implica un certo livello di fiducia nell'applicazione ancor prima del suo utilizzo. Invece, a partire dalla versione Android 6.0 (livello API 23) il sistema chiede i permessi durante l'esecuzione dell'applicazione attraverso una finestra di dialogo, nel momento in cui sono necessari; ciascun permesso può essere negato al momento della richiesta, oppure revocato una volta concesso accedendo alle impostazioni dell'applicazione. Questo consente all'utente di avere un controllo maggiore sulle attività svolte dall'app e sulle informazioni sensibili ad essa fornite.

La gestione dei permessi a tempo di esecuzione avviene attraverso una serie di metodi offerti dalla libreria di supporto di Android. Il metodo `checkSelfPermission()` permette di verificare se l'applicazione è già in possesso di un certo permesso oppure no. Tale verifica deve essere fatta prima di eseguire l'azione protetta dal permesso considerato; in questo modo si evitano le interruzioni improvvise dell'applicazione causate da eccezioni di tipo `SecurityException`, dovute proprio ad una violazione di sicurezza nel tentativo di accesso non autorizzato ad una risorsa. Se la verifica fallisce si procede con la richiesta del permesso all'utente tramite il metodo `requestPermissions()`: è possibile richiedere più permessi contemporaneamente passandoli tra i parametri all'interno di un oggetto di tipo `Array<String>`. Il metodo, una volta invocato, visualizza nel contesto dell'`Activity` chiamante una finestra di dialogo; quest'ultima non è personalizzabile e il suo aspetto dipende dalla versione di Android del dispositivo in cui è visualizzata. La finestra di dialogo mostrata in figura 4.5a è relativa alla prima richiesta riguardante un unico permesso, quello per l'accesso alla fotocamera. In caso esso venga negato, nelle successive richieste il sistema aggiungerà l'opzione 'non chiedermelo più', come mostrato in figura 4.5b, offrendo all'utente la possibilità di negarlo in modo permanente; ciò significa che la richiesta non verrà più effettuata e sarà possibile abilitare il permesso solamente dalle impostazioni dell'app.

La richiesta eseguita utilizzando il metodo `requestPermissions()` è asincrona. L'utente, dunque, può decidere se autorizzare o negare il permesso e il risultato di questa scelta viene restituito al chiamante tramite il metodo `onRequestPermissionsResult()`. Per garantire una buona esperienza d'uso è necessario limitare il numero di richieste di un permesso, considerato che esse avvengono durante l'utilizzo dell'applicazione, interrompendone il normale flusso e obbligando l'utente a prendere una decisione. Inoltre, se è necessario ottenere un permesso all'interno di un contesto in cui esso risulti poco chiaro, è importante fornire all'utente informazioni dettagliate, che spieghino il motivo per cui esso viene richiesto. L'applicazione può mostrare una spiegazione prima di effettuare la richiesta oppure nel momento in cui l'utente ha negato il permesso. Per gestire quest'ultimo



(a) Finestra di dialogo prima richiesta (b) Finestra di dialogo dopo la negazione

Figura 4.5: Esempio richiesta del permesso Camera (versione Android 7.0)

caso esiste il metodo `shouldShowRequestPermissionRationale()` che restituisce *true* nel caso di permesso negato: in questa condizione è possibile fornire le spiegazioni aggiuntive generalmente all'interno di una finestra di dialogo personalizzata. Invece, lo stesso metodo restituisce *false* se l'utente ha negato il permesso selezionando l'opzione 'non chiedermelo più'.

4.5 Funzionalità principali realizzate

Prendendo in considerazione quanto detto riguardo la struttura software dell'applicazione e avendo analizzato le principali librerie impiegate nello sviluppo, vedremo come esse sono state utilizzate e adattate alle diverse esigenze del prodotto software in esame. Per ciascuna funzionalità verrà presentata l'interfaccia grafica con la quale interagiranno gli utenti del servizio, focalizzando l'attenzione sugli elementi grafici e le strutture di visualizzazione dati utilizzate, evidenziando, se necessario, le ragioni di alcune scelte fatte. In questo modo sarà possibile descrivere in dettaglio l'implementazione della logica interna, identificando le classi e i dati coinvolti nel processo che traduce i comandi dell'utente nelle corrispondenti azioni verso il Web Service REST. Tutte le richieste HTTP utilizzate per realizzare le funzionalità seguenti includono un *token*, il quale è stato previsto per l'accesso autorizzato alle API esposte dal servizio. Questo *token* viene inviato dal server al termine della fase di *login* per poi essere salvato localmente nell'applicazione tramite le `SharedPreferences`, una modalità offerta da Android per il salvataggio di dati in maniera persistente, che di default rende le informazioni accessibili esclusivamente ai componenti dell'app in cui vengono memorizzate.

Schermata principale

La prima schermata che si presenta all'utente, una volta effettuato il *login*, mostra una mappa con l'indicazione di tutti i veicoli della flotta (Figura 4.6). Questa schermata rappresenta l'`Activity` principale dell'applicazione, che permette di accedere a tutte le sue funzionalità. La scelta di visualizzare una mappa a pieno schermo è stata adottata al fine di porre la massima attenzione sui veicoli e facilitarne l'individuazione da parte degli utenti. È importante sottolineare che le informazioni sulla posizione dei veicoli vengono recuperate dal server remoto tramite richieste HTTP; infatti, esso è l'unico in grado di costruire l'intera rete di veicoli della flotta, grazie ai dati rilevati dai sensori GPS installati a bordo degli stessi. Nell'attuale configurazione dell'applicazione, la richiesta dei veicoli

viene eseguita senza l'aggiunta di filtri particolari, in modo da ottenere la lista completa dei veicoli presenti.

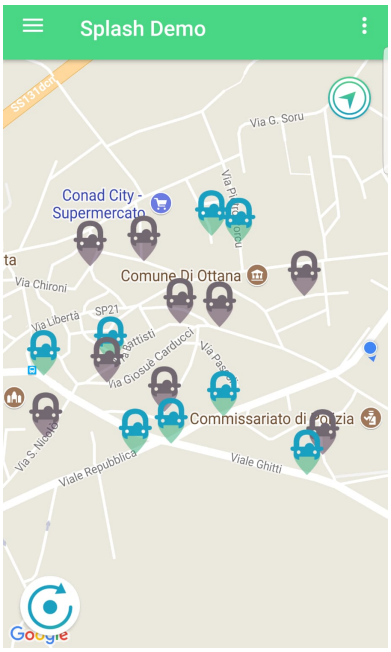


Figura 4.6: Schermata principale

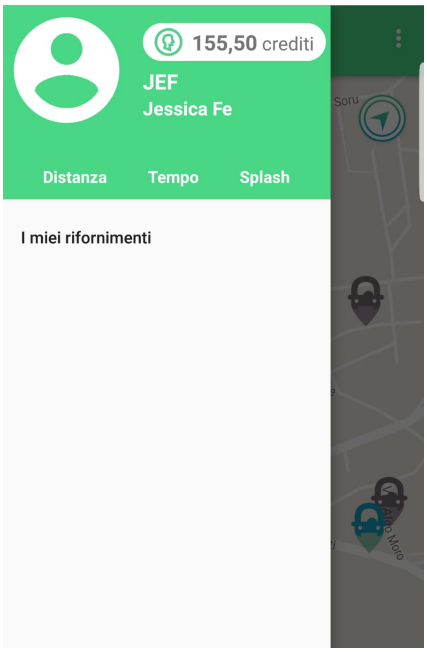


Figura 4.7: Navigation Drawer

Prima di posizionare i veicoli sulla mappa, si procede verificando per ciascuno di essi lo stato di prenotazione, che può essere impostato su 'disponibile' oppure 'occupato', così da poterli distinguere all'interno della mappa. Per tale motivo, si è scelto di utilizzare un'icona a forma di auto diversa per ogni stato: le icone di colore verde indicano i veicoli disponibili, mentre quelle di colore marrone indicano i veicoli già prenotati. Invece, il veicolo prenotato dall'utente che visualizza la mappa avrà un'icona di colore blu. All'interno della schermata principale, inoltre, sono presenti due elementi di tipo `FloatingActionButton`, posti in rilievo sulla mappa: quello in alto a destra aggiorna la posizione dell'utente, identificandola con il puntino blu tipico delle mappe di Google, invece quello in basso a sinistra permette di effettuare la ricerca dei veicoli per ottenere eventuali aggiornamenti sullo stato.

L'`Activity` principale gestisce un pannello a scorrimento laterale realizzato tramite un `NavigationDrawer` (Figura 4.7). Esso è composto da una vista superiore che riassume le informazioni dell'utente e mostra lo stato del suo portafoglio virtuale (credito residuo), mentre la vista inferiore rappresenta il menu dell'applicazione. Quest'ultimo è stato progettato per accogliere eventuali funzionalità di gestione dell'applicazione (notifiche, personalizzazione dell'aspetto, ecc.) e altre funzionalità richieste dallo specifico gestore del servizio. Nella configurazione base dell'app l'unica voce presente è quella che permette di visitare la sezione contenente i rifornimenti di carburante effettuati dall'utente.

4.5.1 Trova veicolo

Dopo aver effettuato una prenotazione, l'utente accede alla sezione dedicata al ritrovamento del veicolo tramite un oggetto `Button` posizionato all'interno della schermata principale; quest'ultimo viene visualizzato e attivato solo al termine della richiesta di prenotazione. A seguito del *click* sul bottone viene aperta una seconda `Activity` che esegue il caricamento del `Fragment` mostrato in Figura 4.8: si è scelto di creare un'`Activity` diversa per la

gestione di questa funzionalità in modo da renderla totalmente indipendente dalle altre presenti nell'applicazione. L'interfaccia grafica del `Fragment` è costituita da una mappa che mostra il tragitto a piedi migliore tra la posizione dell'utente e quella del veicolo da raggiungere. Tutte le informazioni riguardanti il percorso, invece, possono essere accedute tramite il pannello scorrevole (di colore bianco) presente nella parte inferiore della schermata; inoltre, al di sotto del pannello, nella sezione di colore grigio, è stato posizionato il bottone 'Avvia' per avviare l'applicazione *Google Maps* e utilizzare il servizio di navigazione passo-passo. Il pannello scorrevole è stato realizzato utilizzando la libreria *Android Sliding Up Panel*, basata sul componente `SlidingPaneLayout` già presente nella libreria di supporto di Android. *Android Sliding Up Panel* permette di creare un *layout* verticale costituito da due riquadri: un riquadro principale fisso e un riquadro secondario scorrevole dedicato ad informazioni di dettaglio; tipicamente solo uno di essi è visibile alla volta. Di base il riquadro scorrevole è nascosto e il suo contenuto viene visualizzato a seguito di un'operazione di trascinamento, che avviene al di sopra del riquadro principale causandone la copertura completa o parziale.

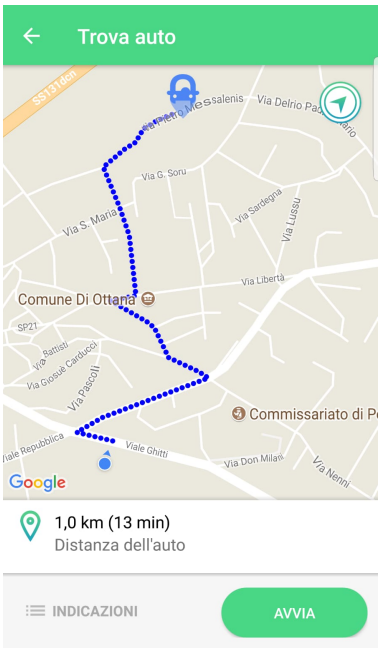


Figura 4.8: Visualizzazione percorso su mappa

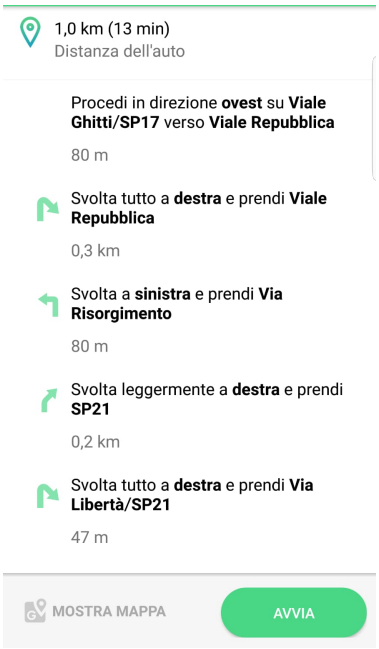


Figura 4.9: Lista delle indicazioni sul percorso

In riferimento alla Figura 4.8, il riquadro fisso è rappresentato dalla mappa, mentre il riquadro scorrevole è costituito da due parti: un'anteprima del contenuto, sempre visibile, che riporta le informazioni sulla distanza del veicolo e sulla durata del percorso, e l'altra (non ancora visualizzata in Figura) che mostra la lista delle indicazioni stradali. In questo caso, dunque, il pannello scorrevole si trova nello stato base (chiuso), per cui il contenuto addizionale è nascosto. Trascinando il pannello dal basso verso l'alto, oppure tramite un semplice *click* sull'anteprima, l'utente può visualizzare la lista delle indicazioni. La Figura 4.9 mostra il risultato dello scorrimento in cui il pannello è completamente aperto sopra la mappa.

Il *layout* del `Fragment` è composto, dunque, da due *layout* figli: il primo di tipo `SlidingUpPanelLayout` che racchiude la mappa e il pannello scorrevole e il secondo di tipo `LinearLayout` contenente il bottone 'Avvia' e una `TextView` 'Indicazioni' (o 'Mostra

mappa'). Quest'ultima è stata aggiunta per migliorare l'esperienza d'uso e facilitare l'utente nell'accesso alla lista qualora il pannello fosse chiuso; infatti, se cliccata permette di aprire e/o chiudere il pannello, mentre il testo al suo interno cambia in base allo stato della vista (indicazioni o mappa nascoste).

Dalla Figura 4.9 si nota che il pannello scorrevole arriva sino alla parte superiore della schermata nascondendo completamente la `Toolbar`. Normalmente questo effetto si ottiene in automatico se essa è posizionata all'interno dell'elemento `SlidingUpPanelLayout`, come parte del contenuto principale. Nel nostro caso, invece, la `Toolbar` è definita nel *layout* dell'`Activity` in cui viene aggiunto quello del `Fragment` appena esaminato. Dunque, mantenendo i due *layout* inalterati, si è deciso di realizzare lo spostamento della `Toolbar` direttamente da codice, modificando la sua posizione in relazione a quella del pannello all'interno della schermata. Per monitorare gli eventi riguardanti lo spostamento del pannello ed effettuare di conseguenza la traslazione della `Toolbar`, è stata implementata l'interfaccia `PanelSlideListener`, in particolare il metodo `onPanelSlide()`:

```
override fun onPanelSlide(panel: View, slideOffset: Float) {
  val position = ((1 - slideOffset) * panel.height).toInt()
  if(position < toolBar.height){
    appBar.translationY = (position - toolBar.height).toFloat()
  }else {
    appBar.translationY = 0f
  }
}
```

L'argomento `panel` rappresenta il pannello scorrevole, mentre il secondo identifica la sua posizione nell'intervallo 0-1; in particolare, quando il pannello è chiuso `slideOffset` vale 0, invece quando è aperto il suo valore è 1, così che il pannello raggiunge la sua massima altezza. Quest'ultima si ottiene accedendo alla proprietà `height`. Una volta recuperato il riferimento alla `Toolbar`, tramite il suo identificativo, è possibile accedere alle proprietà `toolBar.height` e `toolBar.translationY`, necessarie per effettuarne lo spostamento; la prima indica la sua l'altezza e la seconda indica la posizione verticale del suo margine superiore all'interno della schermata, entrambe espresse in *pixel*. Si noti che, in questo caso, la proprietà `translationY` vale 0 quando il margine superiore sinistro della `Toolbar` e del *layout* coincidono. Agendo su questa proprietà è possibile traslare la `Toolbar` in base alla posizione del pannello, il cui valore è indicato dalla variabile `position`. Tale valore rappresenta la posizione raggiunta dal pannello, in *pixel*, rispetto al margine superiore della schermata; ciò significa che `position` vale 0 quando il pannello è completamente aperto e raggiunge il margine superiore della `Toolbar`. Per effettuare la traslazione si confronta il valore di `position` con l'altezza della `Toolbar`: se è minore significa che il pannello si trova all'interno dello spazio occupato dalla `Toolbar` e di conseguenza essa deve traslare verso l'alto; il valore dello spostamento corrisponde alla porzione di `Toolbar` coperta dal pannello. Invece, in tutti gli altri casi essa viene spostata verso il basso, nella sua posizione originaria, risultando completamente visibile. Così facendo si realizza uno scorrimento regolare.

Per quanto riguarda i dati relativi al percorso, essi sono stati ottenuti per mezzo di una richiesta HTTP GET alla Google *Directions* API, effettuata in background da un `AsyncTask`. La richiesta viene eseguita in automatico all'apertura del `Fragment`, una volta che l'utente ha concesso i permessi di localizzazione; in questo caso si è scelto di richiedere il permesso `ACCESS_FINE_LOCATION` per rilevare una posizione quanto più precisa possibile, sfruttando il GPS del dispositivo. La posizione del veicolo, invece, è già nota all'applicazione nel momento in cui viene creata la prenotazione. Il seguente URL mostra

un esempio di richiesta per il calcolo del percorso tra la posizione dell'utente e il veicolo prenotato, la cui risposta dovrà essere formatta in JSON:

```
https://maps.googleapis.com/maps/api/directions/json?
origin=40.233188,9.043832&destination=40.237933,9.047108
&mode=walking&language=it&key=API_key
```

I parametri `origin` e `destination` contengono, nell'ordine, le coordinate geografiche relative all'utente e al veicolo. Il parametro `mode` permette di ottenere le informazioni del solo percorso a piedi e `language` consente di richiederle in lingua italiana. Per effettuare il *parsing* della risposta è stata utilizzata la classe `JSONObject` del package *org.json*, che permette di estrarre i dati strettamente necessari, in maniera semplice, senza il bisogno di creare delle classi apposite per ogni tipo di informazione presente. Dunque, per la realizzazione del `Fragment`, gli elementi principali estratti dal JSON sono: l'oggetto `overview_polyline`, che contiene una rappresentazione codificata dell'insieme dei punti del percorso e l'elemento `routes`, nel quale vengono riportate le informazioni sul tragitto migliore trovato tra l'utente e il veicolo. In questo caso, non avendo effettuato una richiesta per rotte alternative, il servizio *Google Maps* restituisce una singola rotta all'interno dell'elemento `routes`, che dovrebbe essere ottimizzata rispetto al tempo di percorrenza.

Come si è già detto in precedenza, le API dei *Google Play Services* offrono la possibilità di personalizzare le mappe in base alle esigenze e allo stile complessivo delle applicazioni che le utilizzano. Nel caso della funzionalità in questione, prima di tutto, è stato aggiunto un oggetto `Marker` per evidenziare la posizione del veicolo prenotato. Il marcatore viene creato tramite la classe `MarkerOptions` specificando le coordinate (`LatLng`) del punto sulla mappa in cui esso verrà posizionato. Viene aggiunto anche un titolo per indicare l'indirizzo corrispondente alla coppia di coordinate, il quale compare in automatico all'interno di una finestra informativa a seguito di un *click* sul marcatore. Inoltre è stata sostituita l'icona che Google assegna di default a tutti i marcatori, con una personalizzata a forma di veicolo. Per evidenziare la posizione dell'utente sulla mappa, invece, si utilizza sempre il puntino blu presente di base nelle `GoogleMap`.

Il percorso tra l'utente e il veicolo viene disegnato sulla mappa mediante un oggetto `Polyline`, che consiste di un elenco di punti collegati tra loro per formare un tracciato. Nel nostro caso l'elenco viene ottenuto decodificando l'elemento `overview_polyline` estratto dal JSON, di tipo `String`, tramite il metodo `decodePoly()`; questo restituisce un oggetto `List<LatLng>`, in cui ogni punto è rappresentato con i valori latitudine e longitudine. Dopodiché si utilizza la classe `PolylineOptions()` per costruire la `Polyline` a partire dall'elenco ottenuto, con la possibilità di personalizzarne il colore, lo spessore e il modello di tratto della linea disegnata tra i vari punti (in questo caso si è scelto il modello punteggiato di colore blu). Si noti che il tracciato viene disegnato una sola volta, non appena si riceve la risposta da parte dell'API *Directions* e non cambia in base allo spostamento dell'utente. L'aggiornamento continuo del percorso richiederebbe l'esecuzione di un numero significativo di richieste verso il servizio e questo risulterebbe oneroso per l'applicazione che dovrà processare tutte le risposte. Inoltre, l'utilizzo dell'API *Directions* prevede un limite massimo di richieste giornaliere, oltre il quale viene applicata una tariffa in base al tipo di piano scelto (standard o premium). Nel nostro caso, trattandosi di un prototipo, è stato scelto il piano standard che prevede fino a 2500 richieste gratuite al giorno, un valore ritenuto sufficiente a soddisfare le esigenze dell'applicazione nella sua configurazione base. L'eventuale passaggio ad un piano premium verrà valutato in seguito, a seconda di quanto concordato con i potenziali clienti.

In merito ai dati fruibili dall'interfaccia esaminata, come già detto, è disponibile una lista di indicazioni stradali da seguire per raggiungere il veicolo prenotato. Tale lista è stata

realizzata utilizzando una `RecyclerView` e le informazioni in essa contenute sono quelle fornite dalle API sotto forma di un array di oggetti JSON, identificato all'interno della risposta dal nome `steps`; ogni oggetto dell'array rappresenta un passo da compiere lungo il percorso. Per ciascun elemento della lista vengono mostrati solo alcuni dei dettagli che lo caratterizzano; infatti, il *layout* presenta una struttura abbastanza semplice contenente due `TextView`, una con le istruzioni da seguire per il passo considerato e l'altra per indicare la lunghezza. Vi è anche un'`ImageView` che illustra la direzione da seguire per entrare nel tratto di strada relativo al passo a cui l'immagine si riferisce. È bene sottolineare che i file immagine di tutte le possibili direzioni non vengono forniti direttamente dalle API di Google; nelle risposte JSON, per ogni passo, è riportata solamente un'informazione testuale dell'azione da compiere, mediante l'elemento `maneuver`, il quale può assumere uno tra una serie di valori possibili (ad esempio 'turn-left', 'turn-right', 'straight', ecc.). Per migliorare la struttura della lista e differenziare a livello grafico ciascun passo, si è scelto di creare un set di icone personalizzate, una per ogni singolo valore previsto dall'elemento `maneuver`. Durante la costruzione della lista, tale valore viene utilizzato dall'*adapter* per decidere quale immagine caricare all'interno dell'oggetto `ImageView`.

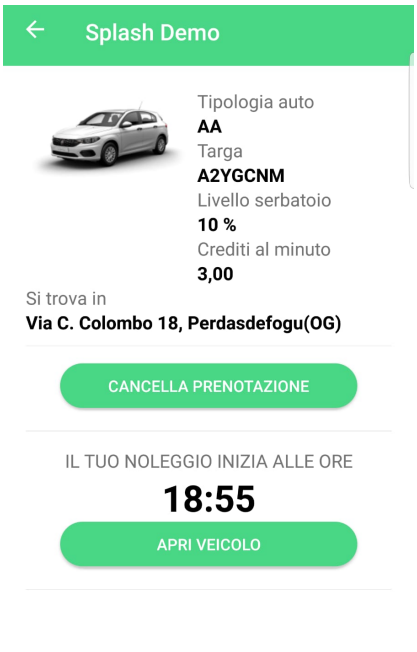
Per quanto si è detto all'inizio, oltre a visualizzare le informazioni dettagliate sul percorso, l'utente può scegliere di avviare facilmente l'applicazione *Google Maps* installata sul proprio dispositivo per mezzo del bottone 'Avvia', qualora desiderasse utilizzare la navigazione passo-passo. A tal proposito, l'app *Google Maps* espone una serie di `Intent` da utilizzare per richiedere all'applicazione stessa diversi servizi, tra cui il servizio di navigazione. Un `Intent` è una struttura dati che descrive un'azione da eseguire. Viene utilizzato principalmente per lanciare le `Activity` o altri componenti di Android; se l'`Intent` specifica il nome del componente o dell'app che dovrà eseguire l'azione richiesta è detto esplicito, altrimenti si parla di `Intent` implicito. Nel nostro caso per avviare la navigazione di *Google Maps* si utilizza un `Intent` esplicito; infatti, è necessario specificare il package *com.google.android.apps.maps*, affinché sia proprio l'app Google Maps a gestirlo. Inoltre, bisogna indicare un URI contenente l'indirizzo della destinazione da raggiungere e infine il tipo di azione da eseguire sui dati passati tramite l'URI; per tutti gli `Intent` di *Google Maps* si utilizza l'azione `ACTION_VIEW`. Si noti che prima di avviare l'app *Google Maps*, bisogna verificare che essa sia installata nel dispositivo, per evitare una possibile interruzione della nostra applicazione. L'`Intent` di base da usare per avviare la navigazione verso la destinazione scelta presenta la seguente forma: `google.navigation:q=destination`; il parametro `destination` può contenere un indirizzo oppure una coppia di coordinate. A questa stringa è possibile aggiungere dei parametri per mezzo del carattere &: nel nostro caso viene aggiunto il parametro `mode` per richiedere la modalità di navigazione a piedi. La posizione dell'utente, invece, verrà rilevata autonomamente dall'app *Google maps*.

4.5.2 Apri veicolo e inizia noleggio

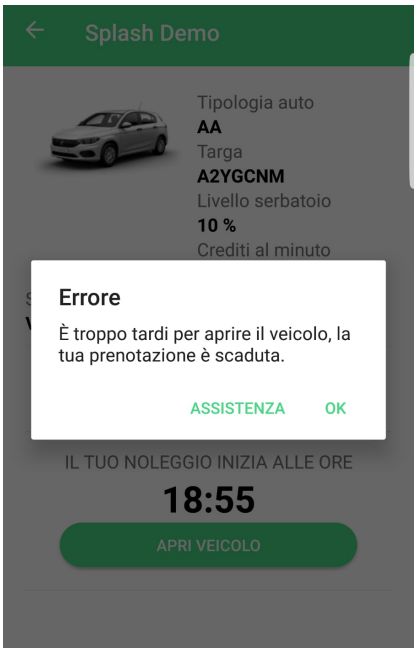
L'apertura di un veicolo può avvenire a prenotazione effettuata e consiste nella scansione di un codice QR univoco mostrato in uno schermo collocato sul parabrezza dello stesso. Si è scelto di utilizzare questa tecnologia per facilitare l'utente e automatizzare la procedura, piuttosto che richiedere l'inserimento manuale di un PIN o altre informazioni. Dunque, il sistema prevede che la chiave di sblocco delle portiere venga codificata all'interno del codice QR; inquadrando tale codice, l'utente ottiene in un certo senso la chiave che dovrà essere inviata al server per la conseguente verifica e conferma dell'apertura. L'interazione tra l'utente e il server è articolata in due fasi: nella prima fase l'utente comunica al server l'intenzione di aprire il veicolo, mentre nella seconda si invia il valore ottenuto dalla scansione del codice QR. Entrambe le interazioni vengono realizzate mediante richieste

HTTP eseguite da appositi `AsyncTask`. Lo sblocco delle portiere permette all'utente di attivare il noleggio e di utilizzare il veicolo prenotato fino a che non viene inviata una richiesta di terminazione.

L'utente fa partire la procedura di sblocco cliccando il bottone 'Apri veicolo' posizionato all'interno del *layout* mostrato in Figura 4.10a. Tale schermata è relativa ad un `Fragment` contenente i dati e le azioni disponibili per una certa prenotazione e accessibile dall'`Activity` principale dell'applicazione. All'azione di *click* sul bottone, segue l'esecuzione di una richiesta HTTP GET verso il server per la generazione di un codice QR. Il server verifica lo stato della prenotazione fatta dall'utente prima di generare e inviare il codice al veicolo. Si noti che la richiesta di apertura deve avvenire entro un certo limite di tempo, attualmente impostato su 15 minuti, oltre il quale la prenotazione è dichiarata scaduta e l'utente non può più accedere al veicolo. In questa situazione l'utente viene notificato tramite una finestra di dialogo, come quella in Figura 4.10b, e ha la possibilità di contattare l'assistenza. Il *click* sul bottone non fa altro che avviare il compositore di numeri telefonici installato sul dispositivo, mostrando il numero da chiamare; questo comportamento si ottiene lanciando un `Intent` implicito contenente l'azione `ACTION_DIAL` e il numero da comporre. Tutte le procedure attivabili dall'assistenza clienti nella condizione di errore appena esaminata o in altre, dovranno essere definite dal gestore del servizio.



(a) Schermata gestione prenotazione



(b) Errore: prenotazione scaduta

Figura 4.10: Richiesta apertura veicolo prima fase

Se la richiesta va a buon fine si entra nella seconda fase, quella relativa alla scansione del codice QR, effettuata tramite un'`Activity` la cui schermata somiglia a quella in Figura 4.11. L'interfaccia di scansione è molto semplice: mostra a pieno schermo l'anteprima della fotocamera del dispositivo, proiettata all'interno di un elemento `SurfaceView`; inoltre, nella parte superiore sono presenti due oggetti di tipo `Button`, i quali permettono all'utente di terminare la scansione chiudendo l'`Activity`, e di attivare o disattivare il flash. La `TextView` in basso fornisce alcune indicazioni riguardanti la scansione. Questi elementi vengono sovrapposti alle immagini in arrivo dalla fotocamera grazie all'utilizzo di un `FrameLayout`; esso rappresenta il contenitore principale dell'intera vista e include tutti gli elementi descritti.

Figura 4.11: Scansione codice QR

La funzionalità di scansione è stata implementata utilizzando gli strumenti offerti dalla libreria Mobile Vision API: in particolare le classi `CameraSource` e `BarcodeDetector`. La classe `CameraSource` si occupa di inviare i frame catturati dalla fotocamera ad un `BarcodeDetector`, il quale deve essere opportunamente configurato per rilevare la presenza di soli codici QR. Questo viene fatto, in fase di creazione del *detector*, mediante il metodo `setBarcodeFormat()`, a cui viene passata la costante `QR_CODE`; così facendo, si restringe l'insieme di codici rilevabili nelle immagini inquadrate con la fotocamera e si ottimizzano le prestazioni. La classe `CameraSource` ha l'ulteriore responsabilità di mostrare l'anteprima della fotocamera all'utente sfruttando proprio la `SurfaceView` collocata nel *layout*, la quale fornisce una superficie di disegno e ne gestisce la sua presentazione all'interno della vista del `Fragment`. L'accesso a questa superficie è controllato da un oggetto `SurfaceHolder`, che viene passato a `CameraSource` in fase di avvio della scansione affinché l'utente possa visualizzare ciò che viene catturato dalla fotocamera.

Per far partire la scansione dei codici QR bisogna invocare il metodo `start()` sull'oggetto `CameraSource` passando, dunque, come argomento l'*holder* della `SurfaceView`. Si è scelto di attivare la scansione in corrispondenza della `callback onResume()` e di disattivarla nella `onPause()`: in questo modo, la cattura delle immagini dalla fotocamera può avvenire soltanto se l'`Activity` si trova in primo piano, completamente visibile all'utente. Inoltre, dal momento che l'attività di scansione prevede il controllo completo del sensore della fotocamera, l'applicazione deve ottenere l'autorizzazione dell'utente per il suo utilizzo, considerando che si tratta di un permesso 'pericoloso' che può essere revocato in qualunque momento una volta concesso. Quindi, per le versioni Android 6.0 e superiori, si verifica e, se necessario, si richiede all'utente tale permesso, prima di avviare la scansione; se il permesso viene negato non si visualizza la schermata di scansione ma un *layout* diverso, creato appositamente per segnalare l'errore. Se, in aggiunta, è stata selezionata l'opzione 'non chiedermelo più', trattandosi questa di una funzionalità importante, viene mostrata una finestra di dialogo (Figura 4.12) per invitare l'utente ad accedere alle impostazioni dell'app e a concedere l'autorizzazione richiesta.

Quando la scansione viene avviata, il *detector* inizia ad esaminare il flusso di frame ricevuto, fintanto che non rileva un codice QR, oppure l'utente decide di terminare l'operazione e chiude l'`Activity`. L'area di scansione utilizzata dal *detector* corrisponde

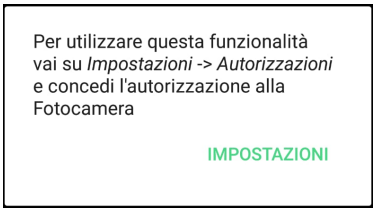


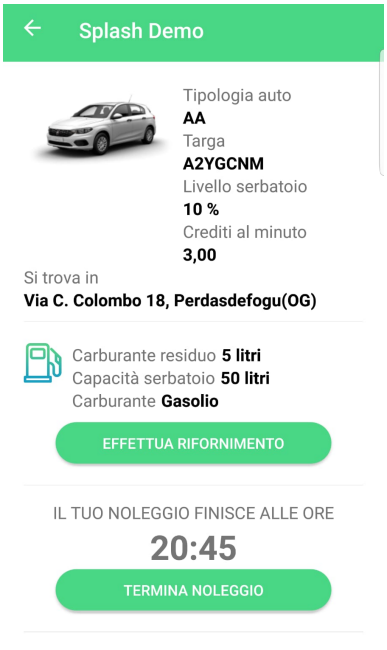
Figura 4.12: Finestra di accesso diretto alle autorizzazioni dell'app

all'intera anteprima mostrata all'utente, per cui il codice QR può essere inquadrato in qualsiasi punto della schermata mostrata. Appena il *detector* rileva un codice, restituisce il risultato all'`Activity` tramite il metodo `receiveDetections()`. Si tratta di una `callback` relativa all'interfaccia `Detector.Processor`, la cui implementazione deve essere fornita al *detector* mediante il metodo `setProcessor()`, in modo da ricevere la notifica. Alla ricezione del risultato si estrae il valore codificato all'interno del codice QR e viene fatto vibrare il dispositivo dell'utente, così da segnalarne il corretto riconoscimento. Per utilizzare la funzionalità di vibrazione è sufficiente dichiarare il permesso `VIBRATE` nel file *AndroidManifest.xml*, considerando che si tratta di un permesso 'normale' concesso in automatico dal sistema. A questo punto, non è più necessario tenere il lettore attivo, quindi, a seguito della vibrazione, la scansione viene interrotta in modo che il *detector* non riceva più i frame da analizzare. Per fare questo, si utilizza il metodo `stop()` della classe `CameraSource`, il quale deve essere invocato necessariamente al di fuori della `callback` `receiveDetections()`. Tale limite deriva dall'implementazione della funzionalità di scansione fornita dalla libreria, in cui è previsto che il metodo `receiveDetections()` venga eseguito in un thread secondario, dedicato al rilevamento dei codici QR, e che il metodo `stop()` attenda la sua terminazione prima di bloccare la scansione. Di conseguenza, l'esecuzione del metodo `stop()` all'interno della `receiveDetections()` provocherebbe un blocco di questo thread e la scansione non verrebbe mai terminata. Per evitare questo problema è necessario invocare il metodo `stop()` in un thread diverso da quello di scansione. Per semplicità, si è scelto di sfruttare il metodo `post()` della `TextView` definita nel *layout*, il quale accetta come argomento un oggetto `Runnable` contenente le azioni da eseguire e fa si che il `Runnable` venga eseguito all'interno dell'*UI Thread* dell'applicazione, diverso appunto dal thread precedente.

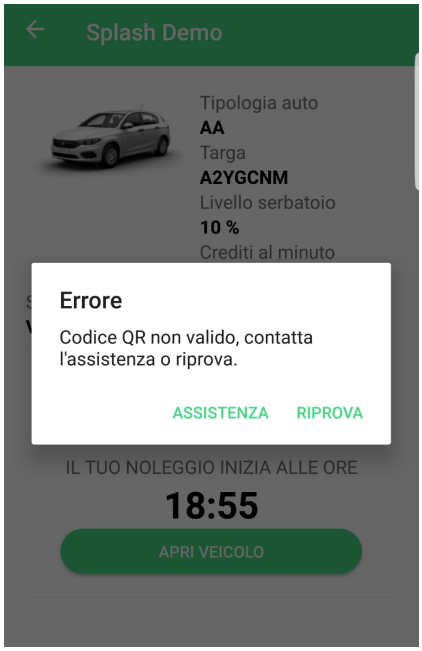
Terminata la scansione, si invia il risultato estratto dal codice QR al server per richiedere l'apertura del veicolo; tale valore viene inviato attraverso una richiesta HTTP POST. Il server effettua nuovamente alcuni controlli relativi alla prenotazione e confronta il valore ricevuto dall'app con quello precedentemente generato. Se non vengono riscontrati dei problemi e i due codici coincidono, viene inviato un comando al veicolo per indicare l'apertura delle portiere. Se invece il codice non è corretto, oppure si verificano dei problemi tecnici che impediscono lo sblocco del veicolo, l'utente verrà notificato e avrà la possibilità di riprovare scansionando un nuovo codice o, in alternativa, di contattare l'assistenza che provvederà ad effettuare l'apertura oppure al rimborso dei crediti spesi dall'utente per la prenotazione.

In tutti i casi appena descritti, l'applicazione riceve una risposta HTTP dal server contenente l'esito della richiesta. Se quest'ultima è andata a buon fine, la risposta contiene la rappresentazione JSON dello stato del veicolo. Per effettuare la deserializzazione delle risposte HTTP si utilizza la libreria GSON, in modo da eseguire la mappatura dei dati all'interno delle classi del *Model* locale. Avendo eseguito la richiesta di sblocco dall'`Activity`

di scansione, tramite un `AsyncTask`, tutte le risposte vengono ricevute all'interno di questa classe, già deserializzate, per poi essere passate al `Fragment` chiamante per la gestione del risultato. Infatti, alla ricezione di una risposta, l'`Activity` viene terminata e si restituisce il risultato al `Fragment` tramite un `Intent`. A questo punto, se il risultato è positivo e il veicolo è stato sbloccato, si aggiorna la schermata del `Fragment` (Figura 4.13a) in modo da mostrare le operazioni disponibili per il noleggio. La Figura 4.13b, invece, mostra il risultato della scansione di un codice QR errato. Cliccando sul bottone 'Riprova' l'utente può riavviare la procedura di sblocco richiedendo al server un nuovo codice da scansionare.



(a) Schermata noleggio avviato



(b) Errore: codice QR errato

Figura 4.13: Risultato apertura veicolo seconda fase

4.5.3 Termina noleggio e chiudi veicolo

Una volta che il noleggio è stato avviato, l'utente può utilizzare il veicolo per tutto il tempo richiesto in fase di prenotazione e ha l'obbligo di rilasciarlo entro e non oltre questo tempo. Il sistema, infatti, considera conclusi in maniera corretta tutti i noleggi per i quali è stata ricevuta una richiesta di terminazione. Nella presente configurazione del sistema, non è stata implementata la gestione del mancato rilascio dei veicoli, quindi, affinché sia garantita una massima efficienza del servizio, gli utenti sono tenuti a rispettare tale vincolo; qualora l'utente riscontri dei problemi ad effettuare il rilascio nei limiti di tempo previsti, deve necessariamente contattare l'assistenza.

Per effettuare la richiesta di terminazione, l'utente può cliccare sul bottone 'Termina noleggio' presente nella schermata in Figura 4.13a. L'applicazione mostra una finestra di dialogo per richiedere conferma all'utente prima di avviare la procedura. A questo punto, se l'utente conferma, viene eseguita una richiesta HTTP POST per comunicare al server l'intenzione di concludere il noleggio. In questo caso, l'API esposta dal server non prevede l'invio di alcuna informazione, ma i dati sull'ultimo noleggio relativo all'utente vengono recuperati grazie al *token* presente nella richiesta HTTP. Tale richiesta viene gestita in background mediante una `Coroutine`, la quale viene creata a partire dal `Fragment` in esame e rimane sospesa su un thread secondario, in attesa di una risposta dal server.

Se la richiesta viene accettata, l'applicazione riceve la rappresentazione JSON dello stato del veicolo, così come avviene nella fase di apertura descritta in precedenza. Da questo oggetto, una volta che è stato deserializzato con GSON, è possibile verificare se il server ha avviato o meno il processo di chiusura del veicolo, estraendo il valore identificato dal campo `doorsOpened`. Nel caso in cui la richiesta sia stata accettata, il campo contiene il valore *false*, indicando appunto che il server sta eseguendo la procedura di chiusura del veicolo. Si noti che, all'atto dell'accettazione della richiesta, le portiere del veicolo risultano ancora aperte così da permette all'utente di recuperare i suoi oggetti personali e di abbandonare il veicolo. Infatti, il blocco delle portiere viene effettuato successivamente, dopo un certo tempo a partire dalla conferma di terminazione. Prima di allontanarsi dal veicolo, l'utente deve attendere la conferma di fine noleggio comunicata mediante una finestra di dialogo. L'applicazione, a questo punto, aggiorna l'interfaccia dell'app chiudendo il `Fragment` e riportando alla schermata principale nella quale è visualizzata la mappa. Nell'eventualità che si verifichino dei problemi durante l'esecuzione della procedura e il server non sia in grado di terminare il noleggio, viene comunicato l'errore all'applicazione che permetterà all'utente di riprovare, oppure di contattare l'assistenza.

4.5.4 Ricarica crediti tramite il rifornimento del carburante

Dopo aver sbloccato il veicolo e avviato il noleggio, l'utente può eseguire il rifornimento del carburante seguendo la procedura guidata proposta dall'applicazione. Nella configurazione base dell'app tale funzionalità è sempre disponibile se il livello del carburante è sotto il 100%. Essa può essere utilizzata come modalità di ricarica dei crediti nel portafoglio, in alternativa all'acquisto degli stessi tramite il servizio PayPal. Il valore dei crediti ricaricati dipende dall'importo effettivamente speso per il rifornimento e dal valore monetario attribuito al singolo credito nel servizio Splash. Nel caso in esame questo valore è fisso e in seguito potrà essere modificato dal potenziale gestore attraverso il pannello Web a seconda delle proprie esigenze. La procedura di ricarica dei crediti attraverso il rifornimento del carburante si svolge in tre fasi: nella prima si richiede l'avvio della procedura così da permettere al server di memorizzare il livello iniziale del carburante nel serbatoio, nella seconda vengono gestite, se necessario, eventuali problematiche riscontrate dall'utente durante il rifornimento, mentre nell'ultima è previsto l'invio dell'importo pagato dall'utente per il conseguente ottenimento dei crediti. In questa fase il server recupera il nuovo livello di carburante nel serbatoio per risalire alla quantità di litri realmente rifornita a seguito dell'operazione.

Prima di effettuare la ricarica dei crediti, il server verifica la veridicità dell'importo dichiarato dall'utente basandosi sia sulle informazioni recuperate dal veicolo, sia sui dati forniti dal Ministero dello Sviluppo Economico e in particolare da una delle banche dati del progetto *OpenData*[5] che pubblica i prezzi di vendita dei carburanti praticati presso gli impianti di distribuzione italiani. Il server dunque può calcolare il prezzo al litro del carburante per il rifornimento effettuato dall'utente, in base all'importo pagato e alla quantità ricaricata, così da poterlo confrontare con i valori ottenuti dagli *OpenData*. Per la verifica vengono prelevati i prezzi al litro di tutti i distributori situati nelle vicinanze del veicolo entro un raggio di 10 Km: se il prezzo calcolato dal server è compreso tra il minimo e il massimo di questi valori, il rifornimento viene confermato con successo e l'utente

[5] fonte:
http://www.mise.gov.it/index.php/it/open-data/elenco-dataset/2032336-carburanti-prezzi-praticati-e-anagrafica-degli-impianti

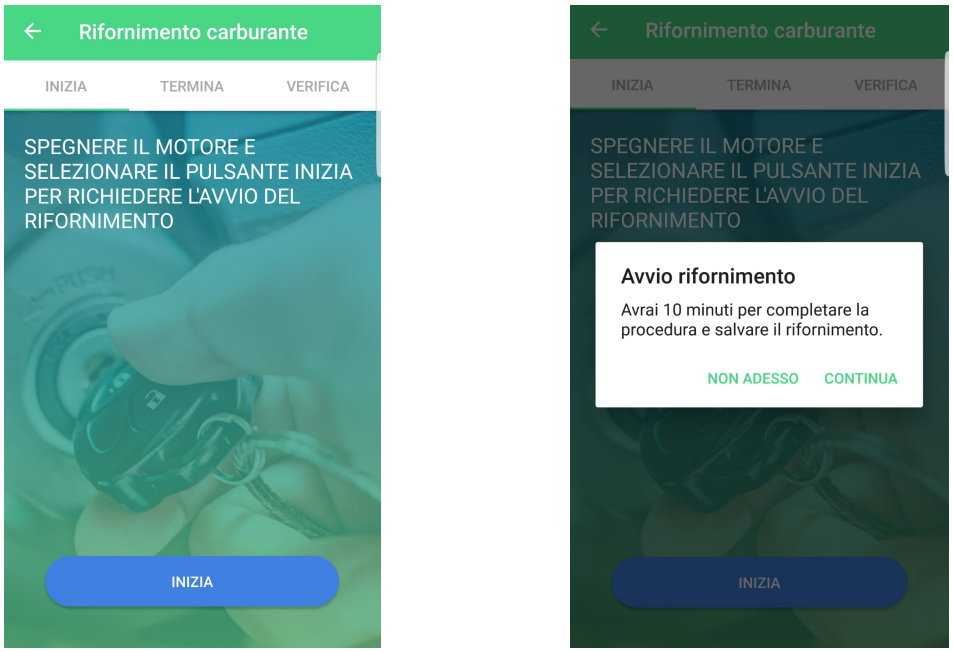
può ricevere i crediti corrispondenti. Considerato che la richiesta di avvio e di conferma del rifornimento sono separate, per evitare di avere rifornimenti pendenti non ancora completati, è stato implementato lato server un timeout di conferma dei rifornimenti che è fissato a 10 minuti. Se la richiesta di conferma, unita all'invio dell'importo, non avviene entro questo tempo, il rifornimento viene annullato e non vengono ricaricati i dovuti crediti. L'esistenza del timeout di conferma viene comunicata all'utente mediante una finestra di dialogo visualizzata prima di eseguire la richiesta di avvio del rifornimento.

L'utente può accedere alla funzionalità di rifornimento del carburante attraverso il *click* sul bottone 'Effettua rifornimento' visualizzato nella schermata in Figura 4.13a. L'interfaccia utente della procedura guidata è stata implementata utilizzando un `TabLayout` collocato nella parte superiore del *layout* di un'`Activity` e costituito da tre `Tab` corrispondenti alle tre fasi previste 'Inizia', 'Termina' e 'Verifica', come mostrato in Figura 4.14a. Dunque, per ogni `Tab` è stato creato un `Fragment` diverso in grado di gestire l'interazione dell'utente nella specifica fase; il passaggio da uno all'altro viene effettuato esclusivamente dall'`Activity` a seguito di una notifica ricevuta da parte dei singoli `Fragment` per indicare lo stato di avanzamento della procedura. Per realizzare questo meccanismo di notifica è stata utilizzata un' interfaccia definita all'interno di una classe da cui ereditano i `Fragment`, che viene implementata dall'`Activity`. L'interfaccia contiene la dichiarazione del metodo `setCurrentTab()` il quale accetta tra gli argomenti la posizione della `Tab` da visualizzare. I `Fragment` invocano il metodo `setCurrentTab()` in base all'esito della risposta ricevuta dal server: in caso di esito positivo è possibile passare alla fase successiva, per cui l'`Activity` viene notificata così da effettuare il caricamento del nuovo `Fragment` e selezionare la `Tab` opportuna. In questo modo l'utente accede alle varie fasi nell'ordine proposto dall'applicazione.

Tuttavia, è necessario eseguire le transazioni dei `Fragment` durante il tempo di visibilità dell'`Activity` affinché possano essere mostrate all'utente. In particolare, esse devono avvenire prima del salvataggio dello stato dell'`Activity`, che si verifica in corrispondenza del metodo `onSaveInstanceState()` così che vengano memorizzate e ripristinate nel caso in cui l'`Activity` venga distrutta e ricreata dal sistema per mancanza di memoria. Qualsiasi cambiamento applicato alla UI dopo l'esecuzione di questo metodo non può essere registrato e di conseguenza non è possibile riprodurlo per l'utente; per impedire il verificarsi di questa condizione il sistema genera un'eccezione di tipo `IllegalStateException()` che indica una perdita di stato dell'interfaccia utente definita con il termine *state loss*. Nel nostro caso, non potendo prevedere il momento esatto in cui si verifica una richiesta di sostituzione per un `Fragment`, considerando che dipende dall'esito della risposta del server ricevuta per mezzo di una `callback` asincrona, è stato aggiunto un controllo che permette di eseguire le transazioni dei `Fragment` solo se l'`Activity` è in esecuzione e completamente visibile all'utente (tra il metodo `onResume()` e `onPause()`). Eventuali transazioni richieste dopo `onPause()` (richiamato dal sistema sempre prima di `onSaveInstanceState()`) vengono temporaneamente sospese in attesa di essere eseguite non appena l'`Activity` diventa nuovamente visibile: così facendo si evita l'eccezione e l'interfaccia utente della procedura risulta comunque aggiornata. Si noti invece che le transazioni richieste dopo la chiamata al metodo `onSaveInstanceState()` non possono essere salvate in alcun modo se l'`Activity` viene distrutta: in questo caso si va a ripristinare uno stato precedente della procedura. Tuttavia il server conosce a che punto era rimasto l'utente (ad esempio restituisce uno specifico errore se il rifornimento era già stato confermato), pertanto è possibile concludere ugualmente l'operazione con successo.

Per poter creare delle transazioni posticipate si è scelto di definire la classe astratta `PendingFragmentTransaction` dentro la classe base da cui eredita l'`Activity`; in questo

modo può essere condivisa anche da altre classi. Al suo interno sono stati definiti due attributi comuni a tutte le transazioni, ovvero l'istanza del `Fragment` per la quale vengono eseguite e l'elemento nel *layout* dell'`Activity` in cui tale `Fragment` deve essere mostrato: entrambi gli attributi vengono inizializzati tramite il costruttore della classe astratta in fase di creazione delle sottoclassi concrete, così da poter essere utilizzati internamente per l'esecuzione della transazione. Inoltre, è presente il metodo astratto `performTransaction()` il quale deve essere implementato fornendo il codice che realizza la specifica transazione secondo le esigenze delle sottoclassi. Quando l'`Activity` necessita di registrare una transazione posticipata, si procede creando un oggetto di una sottoclasse concreta derivata dalla classe `PendingFragmentTransaction`, il cui metodo `performTransaction()` effettua una transazione dinamica utilizzando il `FragmentManager` dell'`Activity`; quest'ultimo aggiunge il `Fragment` al contenitore specificato nella sottoclasse e aggiorna la posizione della `Tab` nel *layout*. Infine, l'oggetto viene utilizzato all'interno della `callback onPostResume()` per eseguire la sostituzione del `Fragment` invocando il metodo `performTransaction()`.



(a) Schermata iniziale procedura guidata (b) Notifica timeout di conferma del rifornimento

Figura 4.14: Rifornimento carburante prima fase

Nella Figura 4.14a viene mostrata l'interfaccia del primo `Fragment` nel quale vengono indicate le istruzioni da seguire nella fase iniziale del rifornimento. Prima di procedere con l'erogazione del carburante, l'utente deve premere il bottone 'Inizia' per far partire una richiesta HTTP POST (utilizzando una `Coroutine`) in cui si comunica al server l'*id* del veicolo per il quale si vuole effettuare il rifornimento. Come si è detto in precedenza, prima di eseguire tale richiesta l'utente viene notificato della presenza di un tempo limite entro cui è necessario terminare l'intera procedura (Figura 4.14b). Presa visione del vincolo, se l'utente decide di continuare viene effettuata la richiesta e si attende la risposta del server, il quale verifica che l'utente abbia attivato un noleggio per il veicolo specificato. Se il server conferma l'avvio del rifornimento restituisce all'applicazione la rappresentazione dello stato del portafoglio con il valore attuale dei crediti presenti. Tale valore viene comunicato all'`Activity` che dovrà memorizzarlo in modo che sia disponibile per la fase finale, in cui si riceve la nuova rappresentazione del portafoglio a seguito della conferma del rifornimento. Così facendo è possibile calcolare il numero di crediti aggiuntivi ricevuti

dall'utente al termine dell'operazione. Per comunicare il valore dei crediti dal `Fragment` all'`Activity` si utilizza il metodo `setCurrentTab()` dell'interfaccia descritta in precedenza che accetta anche questo parametro.

Avviato il rifornimento si passa alla seconda fase e l'interfaccia viene aggiornata per mostrare la `Tab` corrispondente, visualizzata in Figura 4.15. A questo punto l'utente può eseguire il rifornimento vero e proprio facendo partire l'erogazione del carburante. Si noti che da questa fase non è possibile tornare alla precedente attraverso il *click* sul pulsante 'Indietro' del dispositivo o dell'`Activity`. Infatti, l'azione 'Indietro' è stata modificata implementando la `callback onBackPressed()` in modo da permettere solamente la chiusura dell'`Activity` e di conseguenza l'annullamento dell'intera operazione. Quindi, se l'utente desidera abbandonare la procedura prima di arrivare all'ultima fase può farlo: alla pressione del pulsante 'Indietro' viene mostrata una finestra di dialogo in cui si richiede all'utente di confermare la cancellazione del rifornimento e in questo caso il server non effettuerà nessuna ricarica di crediti.

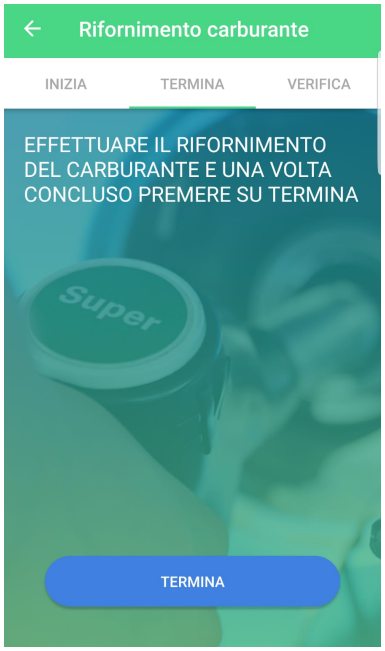


Figura 4.15: Procedura rifornimento seconda fase

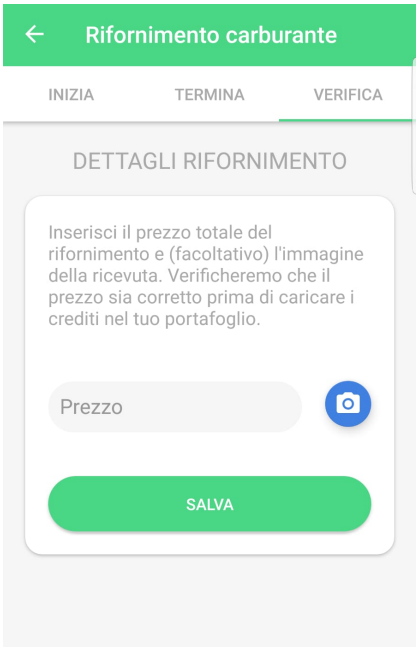


Figura 4.16: Procedura rifornimento terza fase

Per portare a termine la procedura di rifornimento e accedere alla terza fase, l'utente deve cliccare sul pulsante 'Termina' una volta che è stata completata l'erogazione del carburante. Prima di continuare, l'applicazione permette all'utente di contattare l'assistenza per notificare eventuali problematiche riscontrate durante questa fase. Se l'utente, invece, dichiara che l'erogazione è andata a buon fine si entra nell'ultima fase, la cui schermata è mostrata in Figura 4.16, dedicata alla verifica dell'importo pagato e all'eventuale invio della foto dello scontrino. L'interfaccia è costituita da una `ScrollView` che contiene tutti gli elementi mostrati; in questo caso non è stata selezionata ancora una foto per cui l'elemento `ImageView` non è visibile. Per confermare un rifornimento è sufficiente inviare l'importo inserito dall'utente tramite una richiesta HTTP POST avviata al *click* del bottone 'Salva'; prima di inoltrare la richiesta l'applicazione verifica che l'importo digitato sia un numero valido sfruttando un'espressione regolare che accetta solo numeri e il carattere '.' per l'inserimento di cifre decimali.

Alla ricezione della richiesta di conferma il server prima di tutto verifica se il rifornimento è ancora attivo, oppure se sono passati più di 10 minuti da quando la procedura è stata avviata. Se il timeout non è scaduto procede recuperando dal veicolo il nuovo livello di carburante nel serbatoio e calcola la quantità di litri riforniti. Dopodiché viene verificato il prezzo sostenuto dall'utente sfruttando le informazioni ottenute dagli *OpenData* come descritto all'inizio: se tale valore viene accettato, il server calcola i crediti da aggiungere al portafoglio virtuale dell'utente e conferma il rifornimento, restituendo anche in questo caso la rappresentazione aggiornata del portafoglio così che l'applicazione possa risalire al numero di crediti effettivamente ricaricati. A questo punto la procedura si conclude con successo: l'applicazione aggiorna l'interfaccia mostrando nuovamente il `Fragment` relativo al noleggio in cui è indicato il nuovo livello del carburante nel serbatoio. Infatti, alla chiusura della procedura, l'`Activity` del rifornimento effettua una richiesta HTTP per ottenere lo stato aggiornato del veicolo e restituisce il risultato al `Fragment` attraverso un'`Intent`.

Nel caso in cui il prezzo non venga accettato, l'utente ha la possibilità di riprovare inviando un nuovo importo, oppure può contattare l'assistenza in modo che possano essere effettuati ulteriori controlli. In questi casi risulta utile la foto dello scontrino, se disponibile, che attesta l'esecuzione del rifornimento e del relativo importo pagato. L'assistenza, infatti, visionando lo scontrino può decidere se confermare il rifornimento o rifiutarlo. Il caricamento dello scontrino è opzionale e viene gestito tramite un'API diversa da quella prevista per la conferma dell'importo. Se l'utente ha selezionato una foto, all'atto della richiesta di conferma viene eseguita in parallelo anche quella per il caricamento del file immagine. Questo viene realizzato per mezzo di due `Coroutine` che si sospendono in attesa del completamento di entrambe le richieste. Una volta che l'applicazione ha ricevuto il risultato delle due operazioni si procede innanzitutto verificando l'esito della conferma dell'importo: se quest'ultimo è stato accettato significa che il server ha provveduto a ricaricare i dovuti crediti all'utente e quindi la procedura può essere conclusa. Tuttavia, prima di chiudere l'`Activity` è necessario controllare se la foto è stata salvata oppure se è stato riscontrato qualche problema nell'esecuzione della richiesta di caricamento: in quest'ultimo caso l'utente può scegliere di caricarla nuovamente in modo da poterla visionare in seguito nella sezione dedicata ai dettagli del rifornimento.

La funzionalità di aggiunta degli scontrini è stata realizzata tramite la libreria open-source *ImagePickerUtil* che è stata integrata nel progetto e successivamente modificata per soddisfare le esigenze dell'applicazione. L'utente può aggiungere una foto sia dalla galleria del dispositivo, sia dalla fotocamera cliccando l'elemento `FloatingActionButton` presente nella schermata in Figura 4.16, che visualizza una finestra di dialogo in cui sono presenti le due opzioni di selezione (Figura 4.17); tale finestra è stata ottenuta creando un `Chooser`, ovvero un menu nel quale sono stati aggiunti due `Intent` impliciti corrispondenti alle azioni di apertura della fotocamera e della galleria. Dal menu è possibile selezionare anche l'azione per la rimozione di una foto già caricata, che ha effetti diversi a seconda che sia stato effettuato o meno il caricamento dell'immagine sul server. Infatti la cancellazione avviene soltanto in locale se la foto non è stata ancora caricata, altrimenti si effettua la rimozione direttamente sul server tramite una richiesta HTTP. In quest'ultimo caso l'applicazione chiede la conferma da parte dell'utente prima di eseguire la chiamata. A differenza delle due azioni per l'acquisizione delle foto, la classe `Intent` non offre una `Action` standard per eseguire la rimozione. Tale funzionalità è stata implementata creando una nuova `Activity` la quale è stata dichiarata nel file *AndroidManifest.xml* con un'icona e un'etichetta personalizzate in modo da visualizzare l'opzione di rimozione nella finestra del `Chooser` (Figura 4.18). L'`Activity` non possiede un'interfaccia utente e viene utilizzata per collegare l'azione di *click* sull'icona al codice della libreria che effettua la rimozione.

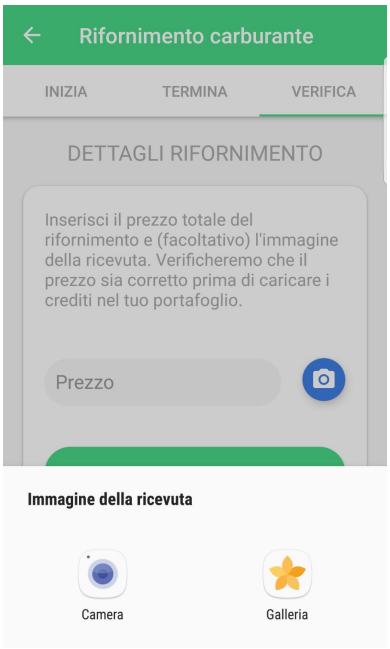


Figura 4.17: `Chooser` con le opzioni per l'acquisizione di una foto

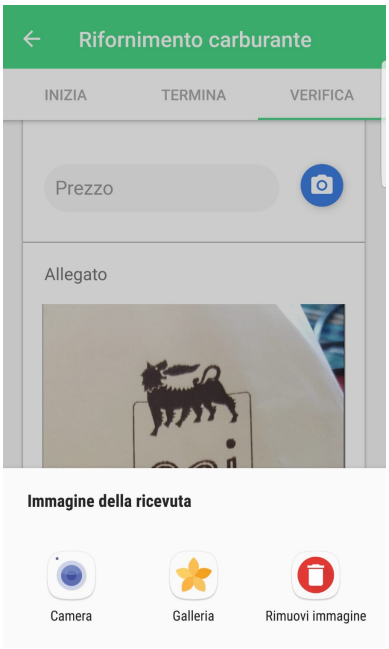


Figura 4.18: `Chooser` con l'opzione di rimozione foto abilitata

Per utilizzare la fotocamera e accedere alla galleria del dispositivo viene chiesto all'utente di concedere i dovuti permessi. Considerando che l'applicazione offre l'accesso alla galleria in sola lettura è sufficiente richiedere il permesso `READ_EXTERNAL_STORAGE`. L'utente deve concederli entrambi, altrimenti il `Chooser` non viene visualizzato e non è possibile allegare nessuno scontrino. Tutti i file acquisiti dalla fotocamera e quelli selezionati dalla galleria vengono memorizzati in una cartella ad uso esclusivo dell'applicazione, creata tramite il metodo `getExternalFilesDir()`, che restituisce il percorso assoluto dell'area di archiviazione esterna nel dispositivo dell'utente dedicata all'applicazione in esame.

4.5.5 Visualizzazione dei rifornimenti di carburante

L'utente può visualizzare le informazioni sui rifornimenti di carburante effettuati durante tutti i noleggi, accedendo al menu laterale collocato nella schermata principale dell'app (Figura 4.7). Toccando la voce 'I miei rifornimenti' si apre un `Fragment` contenente la lista dei soli rifornimenti confermati, ossia quelli per cui è stata ricevuta la ricarica dei crediti nel portafoglio. Si noti che il `Fragment` viene posizionato all'interno del *layout* della schermata principale a seguito di una transazione dinamica, la quale viene salvata nel *back stack* per offrire all'utente una navigazione all'indietro utilizzando il pulsante 'Indietro' del dispositivo, oppure quello collocato nella `Toolbar`.

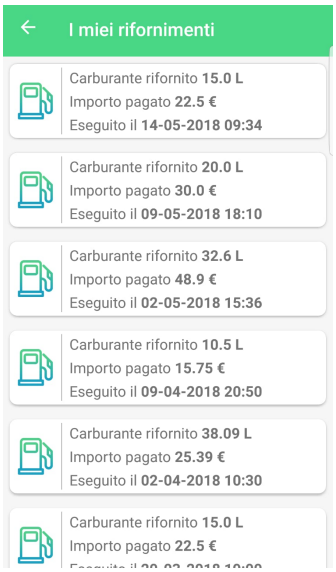


Figura 4.19: Lista dei rifornimenti di carburante

L'interfaccia in Figura 4.19 è costituita da una `RecyclerView` costruita a partire dalle informazioni ottenute mediante una richiesta HTTP GET. Tale richiesta viene eseguita per mezzo di una `Coroutine` avviata in automatico dall'applicazione non appena l'utente esegue l'apertura del `Fragment`. Dal momento che i dati sui rifornimenti non variano in base all'interazione dell'utente con la vista del `Fragment` in esame, è sufficiente recuperarli all'atto della sua creazione e quindi la richiesta viene avviata in corrispondenza della `callback onCreateView()`. Il server, in assenza di errori, restituisce una risposta contenente un array di oggetti JSON dove ciascuno identifica un singolo rifornimento. A questo punto, eseguita la deserializzazione tramite GSON, viene notificato l'*adapter* collegato alla lista perché possa aggiornarne il contenuto. Se l'utente non ha ancora eseguito nessun rifornimento, non viene mostrata la lista ma un *layout* con una sola `TextView` al centro che indica l'assenza di informazioni a riguardo.

Per ogni rifornimento in lista vengono mostrate le informazioni più significative, tra cui il quantitativo di carburante rifornito, l'importo pagato e la data e l'ora in cui è stato effettuato. Tali informazioni vengono ricavate a partire dai dati contenuti nella classe `Refueling` che rappresenta un oggetto rifornimento nel *Model* locale, i cui campi vengono valorizzati in fase di deserializzazione sfruttando proprio il contenuto della risposta HTTP; tuttavia, non tutti i dati ricevuti sono pronti per essere visualizzati all'utente per cui necessitano di una trasformazione. In particolare, il valore della data di creazione del rifornimento viene memorizzato lato server in *UTC*, quindi deve essere convertito tenendo conto del fuso orario impostato nel dispositivo in cui esegue l'applicazione. Inoltre, per

quanto riguarda la quantità di litri riforniti per un certo veicolo, il server fornisce il valore percentuale del livello di carburante nel serbatoio prima e dopo il rifornimento e la capacità totale del serbatoio, così da poter calcolare il quantitativo di litri effettivamente aggiunti. Anziché eseguire queste trasformazioni in fase di creazione della lista, ad esempio dal codice dell'*adapter*, si è scelto di personalizzare il processo di deserializzazione automatico che viene eseguito invocando il metodo `fromJson()` sull'istanza `Gson` utilizzata durante le conversioni. A tal proposito, sono stati aggiunti dei campi nella classe `Refueling` i quali vengono utilizzati per ospitare il risultato delle manipolazioni, oltre ai campi previsti per il *mapping* diretto dei dati JSON generati dal server. Per personalizzare la deserializzazione sono stati usati gli strumenti offerti dalla libreria GSON e in particolare l'interfaccia `TypeAdapterFactory`, tramite la quale è possibile generare un `TypeAdapter` adatto a svolgere le manipolazioni necessarie al termine della fase di deserializzazione. Per utilizzare il *factory* si procede creando una classe che implementa l'interfaccia `TypeAdapterFactory` la quale espone il metodo `create()`, da utilizzare per la creazione di un `TypeAdapter<T>` personalizzato e in grado di trattare qualsiasi oggetto, non solo quelli di tipo `Refueling`:

```
override fun <T> create(gson: com.google.gson.Gson, type:
    TypeToken<T>): TypeAdapter<T> {
  val delegate: TypeAdapter<T> = gson.getDelegateAdapter(this, type)

  return object : TypeAdapter<T>() {

    @Throws(IOException::class)
    override fun write(out: JsonWriter, value: T) {
      delegate.write(out, value)
    }
    @Throws(IOException::class)
    override fun read(input: JsonReader): T {
      val obj: T = delegate.read(input)
      if (obj is PostProcessable)
        (obj as PostProcessable).gsonPostProcess()
      return obj
    }
  }
}
```

Il `TypeAdapter<T>` viene creato definendo i metodi `write()` e `read()` invocati rispettivamente durante il processo di serializzazione e deserializzazione. Tale adattatore si basa su quello predefinito utilizzato da `Gson` per convertire i tipi `<T>`, ottenuto attraverso il metodo `getDelegateAdapter()`. Nel nostro caso, dal momento che le trasformazioni vanno eseguite sui dati in uscita dalla deserializzazione, si interviene all'interno del metodo `read()`: tramite l'adattatore di default si ottiene l'oggetto `<T>` già deserializzato e su di esso si invoca il metodo `gsonPostProcess()` appartenente all'interfaccia `PostProcessable`, la quale permette allo stesso oggetto che la implementa di definire la logica per effettuare le elaborazioni necessarie. In riferimento alla classe `Refueling` il metodo `gsonPostProcess()` viene utilizzato per convertire la data di creazione del rifornimento e per ottenere il quantitativo in litri del carburante rifornito; questi valori vengono memorizzati all'interno dei campi aggiunti all'oggetto `Refueling`, i quali verranno utilizzati durante la creazione della lista da mostrare all'utente. La computazione di questi valori avviene nel momento in cui si effettua la conversione della risposta ottenuta dal server e quindi per ogni rifornimento presente nell'array ricevuto viene invocato il metodo `gsonPostProcess()` il cui codice è

mostrato di seguito:

```
override fun gsonPostProcess() {
  // calcolo della data in base al fuso orario in uso nel dispositivo
  ...
  val calendar = Calendar.getInstance()
  calendar.time = dateFormat.parse(serverDateTime)
  calendar.set(Calendar.ZONE_OFFSET, TimeZone.getDefault().rawOffset)
  refuelingDateTime = calendar.time
  // calcolo della quantità di litri riforniti
  ...
  totalFuelRecharged = (((endLevel - startLevel) * storageVolume)/100)
  ...
}
```

4.5.6 Visualizzazione singolo rifornimento

Una volta che l'utente accede alla lista dei rifornimenti di carburante ha la possibilità di visualizzare ulteriori dettagli effettuando un *click* su ciascun elemento, scatenando l'apertura dell'`Activity` mostrata in Figura 4.20. Oltre alle informazioni già presenti nella lista, vengono visualizzati l'ammontare di crediti accumulati a seguito dell'operazione di rifornimento ed eventualmente la foto dello scontrino se è stata aggiunta dall'utente. Il *layout* contiene un elemento `ScrollView` così che l'utente possa scorrere all'interno della schermata per visualizzare interamente il contenuto. Le informazioni di dettaglio presenti appartengono all'oggetto `Refueling` e sono già disponibili al momento dell'operazione di *click* sugli elementi della lista, per cui vengono passate all'`Activity` attraverso un `Intent`. Per semplicità viene trasferita la rappresentazione JSON dell'oggetto che non è altro che una stringa ottenuta tramite il metodo `toJson()` della libreria GSON. Nel codice dell'`Activity` invece viene fatta l'operazione inversa sfruttando il metodo `fromJson()` per convertire la stringa ricevuta nel corrispondente oggetto `Refueling` da utilizzare per la costruzione della vista. Nella schermata vengono mostrati anche il nome del veicolo che è stato rifornito e il codice della targa, entrambi attributi dell'oggetto `Vehicle` ottenuto per mezzo di una richiesta HTTP GET il cui `URL` contiene l'*id* del veicolo recuperato dall'oggetto `Refueling`; tale richiesta viene affidata ad una `Coroutine` eseguita in automatico all'apertura dell'`Activity` a seguito dell'inizializzazione dell'interfaccia. Nel caso in cui la richiesta dovesse fallire, ad esempio per la mancanza di connessione ad Internet oppure a causa di altri problemi riscontrati sul server, l'applicazione notifica l'utente tramite un `Toast` in cui viene spiegato brevemente il problema. Inoltre, si aggiorna il *layout* per mostrare un `Button` che permette all'utente di eseguire manualmente la richiesta per i dati del veicolo (Figura 4.21).

Per quanto riguarda la visualizzazione dell'immagine dello scontrino è stata utilizzata la libreria open-source *Picasso* che permette di recuperarla da remoto a partire dall'`URL` memorizzato all'interno dell'oggetto `Refueling` e di mostrarla all'utente mediante un elemento `ImageView` adattandola alle sue dimensioni. La libreria gestisce automaticamente il download dell'immagine in maniera asincrona in un `Thread` di background e supporta la memorizzazione in *cache* così che alle successive visualizzazioni essa non venga scaricata nuovamente. Dal momento che il download non è immediato, per dare un riscontro all'utente durante questa operazione, è stata aggiunta una `ProgressBar` circolare in rilievo sull'`ImageView` che viene mostrata all'apertura della schermata e nascosta al termine del caricamento dell'immagine. Se quest'ultima non può essere scaricata a causa di un errore

Figura 4.20: Dettagli di un rifornimento

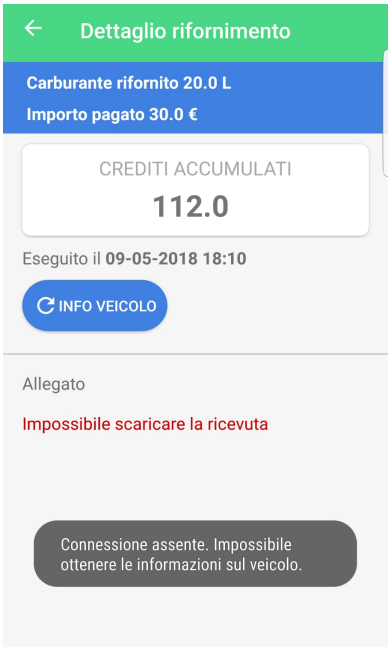


Figura 4.21: Errore: assenza di connessione

di connessione o di eventuali errori sul server, si procede aggiornando il *layout* in modo da nascondere l'`ImageView` e mostrare un messaggio di errore generico per mezzo di una `TextView`, come si può vedere in Figura 4.21. Per conoscere l'esito del caricamento eseguito dalla libreria e gestire di conseguenza l'aggiornamento del *layout* è stata implementata l'interfaccia `Callback`, che permette di ricevere la notifica degli eventi `onSuccess()` e `onError()` all'interno dell'`Activity`. Tuttavia, per i download che prevedono tale meccanismo di notifica viene mantenuto un riferimento forte all'`Activity` in cui è stato richiesto e ciò impedisce al garbage-collector di rimuoverla fintanto che il download non è terminato. Per evitare questo problema è necessario cancellare la richiesta di caricamento dell'immagine nel momento in cui l'`Activity` sta per essere terminata, impedendo così alla libreria di inviare notifiche all'`Activity` quando quest'ultima non è più in esecuzione.

Capitolo 5

Conclusioni

A conclusione di questa tesi, il cui obiettivo è stato quello di partecipare allo sviluppo del prototipo di un'applicazione Android dedicata agli utenti di un servizio di car-sharing, è possibile affermare che, rispetto ai test effettuati sulle funzionalità implementate, l'applicazione rappresenta una buona base di partenza per la realizzazione del prodotto finale che l'azienda fornirà ai propri clienti. Gli aspetti principali che fin da subito sono stati presi in considerazione per le fasi di progettazione e sviluppo riguardavano la possibilità di ottenere un'applicazione intuitiva, chiara nell'organizzazione dei contenuti e semplice da utilizzare per gli utenti finali. Hanno contribuito al raggiungimento di tali obiettivi anche le diverse librerie impiegate nello sviluppo: l'integrazione delle Google Maps ha permesso di organizzare la distribuzione dei veicoli della flotta in base alla loro posizione e di distinguerli secondo alcune caratteristiche, semplificando le fasi di ricerca e prenotazione dei veicoli. Anche la visualizzazione dei dati forniti dalle *Directions* API di Google, riguardanti le indicazioni stradali sui percorsi tra gli utenti e le auto, ha contribuito a migliorare il livello qualitativo del prototipo; infatti, tali informazioni facilitano il ritrovamento delle auto prenotate e ne riducono i tempi.

In questa prima analisi, il sistema di sblocco adottato per le auto, che prevede la scansione di codici QR tramite la stessa applicazione, si è dimostrato abbastanza comodo per l'utente che lo utilizza, in quanto non richiede l'inserimento di codici da tastiera ma è sufficiente usare lo smartphone per inquadrare il codice mostrato dall'apposito schermo. A livello di affidabilità garantita da questa procedura, si può dire che essa risulta utile al gestore del servizio a provare che l'utente abbia raggiunto effettivamente l'auto; tuttavia, l'inoltro al sistema del solo codice QR scansionato non sarebbe sufficiente, da solo, a dimostrare tale condizione. Come si è detto a proposito dell'implementazione di tale funzionalità, alla ricezione del codice QR il sistema effettua ulteriori controlli prima di aprire l'auto, quali l'identificazione del cliente tramite il *token* presente nella richiesta e la presenza di una prenotazione attivata dal cliente per il veicolo in questione. Ciò rende il sistema di sblocco complessivamente affidabile.

Per quanto riguarda la funzionalità di rifornimento del carburante, introdotta come meccanismo di ricarica dei crediti utilizzabili dagli utenti nelle prenotazioni e allo stesso tempo per migliorare la disponibilità del servizio, essa è facilmente fruibile da parte degli utenti grazie alla procedura guidata disponibile sull'app. Infatti, vengono fornite in maniera chiara le informazioni indispensabili affinché l'utente interagisca correttamente con il sistema per portare a termine l'intera operazione con successo; l'inserimento dell'importo pagato, quale unica informazione richiesta a conclusione del rifornimento, permette di velocizzare il processo. Inoltre, i veicoli possono essere riforniti presso una qualsiasi stazione di servizio, offrendo all'utente una maggiore flessibilità nello svolgere tale operazione

secondo le proprie esigenze (scegliendo la stazione più vicina se il livello di carburante non è sufficiente al viaggio da effettuare, oppure quella meno affollata se si ha fretta di partire). La possibilità di fornire lo scontrino direttamente dall'applicazione, per dimostrare l'avvenuto pagamento, permette all'utente di risolvere i casi in cui il sistema rifiuti in automatico l'importo inserito. Grazie all'utilizzo degli strumenti forniti dalle API del sistema Android, si è riusciti ad integrare perfettamente la selezione degli scontrini all'interno della stessa schermata di conferma del rifornimento, senza compromettere la qualità del prototipo in termini di usabilità.

Sebbene la soluzione realizzata soddisfi pienamente gli obiettivi prefissati, di seguito vengono presentate alcune problematiche riscontrate durante l'esecuzione dei test con l'intero sistema. Tali elementi dovranno essere presi in considerazione per migliorare il prototipo e perfezionare il servizio offerto nella sua complessità. L'attuale configurazione del sistema Splash non presenta un meccanismo di comunicazione sicuro tra il server e l'applicazione mobile, il quale permetterebbe di proteggere le informazioni scambiate tra i due componenti: una tra queste è il *token* rilasciato dal server nella fase di *login* che autorizza l'applicazione ad accedere alle funzionalità del servizio. Inoltre, la mancanza di una comunicazione protetta espone il sistema a possibili attacchi anche nell'esecuzione della procedura di sblocco delle portiere, considerando che attualmente il valore del codice QR scansionato viene inviato al server in chiaro. Tali problematiche potrebbero essere superate adottando il protocollo TLS per le comunicazioni app-server e veicolo-server, il quale garantisce un ottimo livello di protezione. Tuttavia, sarebbe opportuno proteggere la stessa applicazione eseguendo la cifratura dei dati sensibili memorizzati nell'area del dispositivo ad essa dedicata; in questo modo è possibile evitare accessi non autorizzati eseguiti in particolari condizioni, come ad esempio, montando il filesystem del dispositivo tramite un computer.

Riguardo la gestione dello stato dell'applicazione, esistono alcune funzionalità per le quali attualmente non viene fornito il supporto dal server; ciò ha richiesto l'inserimento nell'applicazione di controlli e interventi specifici al fine di superare tale limitazione. Tuttavia, si tratta di strategie che non risolvono totalmente i problemi che potrebbero verificarsi in determinate condizioni. Ad esempio, nel caso della funzionalità di rifornimento del carburante, se l'applicazione venisse interrotta a causa di un malfunzionamento del telefono (oppure se il processo dell'app viene terminato da Android per mancanza di memoria, quando l'`Activity` del rifornimento si trova in background) proprio una volta che la richiesta d'inizio della procedura è stata già ricevuta dal server, si potrebbe verificare la perdita della risposta fornita da quest'ultimo, per cui lo stato dell'applicazione e del server risulterebbe disallineato. Al fine di garantire che l'applicazione sia sempre coerente con l'intero sistema, sarebbe opportuno fornire all'app stessa la possibilità di chiedere al server lo stato aggiornato in corrispondenza di eventi per i quali essa potrebbe risultare non sincronizzata. In merito alla terminazione dei noleggi, come si è evidenziato nella parte relativa all'implementazione, è stata riscontrata una gestione non corretta da parte del server dei casi di mancata terminazione dei noleggi da app: allo scadere del periodo di prenotazione richiesto da un utente, il server chiude automaticamente il noleggio, indipendentemente dalla ricezione di una richiesta di terminazione eseguita tramite app, rendendo il veicolo nuovamente disponibile. Tale comportamento rappresenta un problema per il sistema di prenotazione e compromette l'efficienza complessiva del servizio. Dunque, una soluzione potrebbe essere quella di tenere traccia dei noleggi non ancora terminati, in modo da impedire che avvengano eventuali prenotazioni per il veicolo in questione, se quest'ultimo non è stato ancora rilasciato dal precedente noleggio. Inoltre, grazie alla tracciabilità dei noleggi, il gestore può valutare quale strategia attuare per gli utenti che non hanno provveduto al rilascio dei veicoli nei tempi previsti.

Possibili sviluppi futuri

Avendo analizzato i principali punti di forza del prototipo sviluppato, nonché le problematiche legate all'interazione con il resto del sistema, è possibile evidenziare alcuni dei possibili sviluppi futuri mirati ad ampliare le funzionalità già presenti e ad inserirne di nuove, al fine di innalzare il livello qualitativo dell'offerta proposta all'utente. Come già sottolineato, la personalizzazione del design e delle funzionalità dell'app dipenderà maggiormente dalle richieste del potenziale operatore di car-sharing. Tuttavia, si possono individuare già da ora le principali migliorie da apportare al prototipo in modo da renderlo più maturo. In particolare, alcuni aspetti quali il caricamento dei dati personali degli utenti (documento d'identità e patente di guida) e la possibilità di selezionare una modalità di pagamento alternativa al servizio PayPal, non sono stati trattati perché ritenuti non indispensabili al raggiungimento degli obiettivi del prototipo, ma rappresentano requisiti essenziali per un servizio di car-sharing. Per tale motivo, l'applicazione mobile finale dovrebbe integrare nell'operazione di registrazione la richiesta di questi dati, i quali dovranno poter essere modificati all'occorrenza.

Considerando invece la funzionalità di localizzazione dei veicoli su mappa, nonostante al momento essi vengano già differenziati a livello grafico tramite l'utilizzo di icone diverse in base allo stato di prenotazione, sarebbe opportuno offrire all'utente la possibilità di applicare dei filtri per visualizzare i veicoli che soddisfano determinate caratteristiche (nascondendo sulla mappa quelli a cui non si è interessati). Ad esempio si potrebbe prevedere un filtro basato sull'orario di prenotazione, così che l'utente visualizzi solamente i veicoli disponibili per un certo periodo. Ulteriori filtri potrebbero essere relativi sia alla posizione geografica in cui prelevare i veicoli (tramite inserimento dell'indirizzo o selezionando il punto o la zona d'interesse sulla mappa), sia alla tipologia di auto e in generale ad alcune sue caratteristiche (livello carburante, numero di posti, ecc.).

Riguardo il sistema di prenotazione attuale si potrebbe integrare una modalità alternativa a quella esistente, eventualmente più flessibile e basata sul consumo a tempo del mezzo, in cui non è richiesto in anticipo di specificare il periodo di noleggio. Tale modalità risulterebbe più adatta ad effettuare spostamenti brevi. Inoltre, sarebbe opportuno aggiungere al modello di prenotazione esistente, la possibilità di effettuare prenotazioni per dei veicoli i quali, nel momento in cui vengono richiesti, risultano già in uso (o prenotati) da altri utenti. In questo caso la prenotazione dovrebbe essere accettata solamente se non si verifica una sovrapposizione con altre già presenti. Questo contribuirebbe ad aumentare il livello di utilizzo dei veicoli della flotta e a garantire una maggiore disponibilità del servizio agli utenti. Infine, considerato che il sistema tiene traccia dei noleggi effettuati per ciascun veicolo, si potrebbe aggiungere nel menu dell'applicazione una voce dedicata allo storico dei noleggi effettuati dall'utente.

Bibliografia

[1] Android Developers, URL https://developer.android.com

[2] Primo Rapporto Nazionale 2016, "La Sharing Mobility in Italia: numeri, fatti e potenzialità", URL http://osservatoriosharingmobility.it/wp-content/uploads/2016/11/Rapporto-Nazionale-SM_DEF_23_11_2016.pdf

[3] Secondo Rapporto Nazionale sulla Sharing Mobility 2017, URL http://osservatoriosharingmobility.it/wp-content/uploads/2018/04/Rapporto-nazionale-Sharing-mobility-2018.pdf

[4] Claudia Burlando e Marco Mastretta, "Il carsharing: un'analisi economica e organizzativa del settore", Franco Angeli, 2007, URL http://www.icscarsharing.it/wp-content/uploads/2017/01/Libro_Car_Sharing_analisi_economica_e_organizzativa_del_settore.pdf

[5] DECRETO Ministero dell'Ambiente, "Mobilità sostenibile nelle aree urbane", 27 Marzo 1998, URL http://host.uniroma3.it/uffici/mobilitymanager/materiali/normativa/Decreto%20ministero%20ambiente_27%20marzo%201998_previsione%20MM%20aziendali%20e%20PSCL.pdf

[6] "Kotlin Language Documentation", URL https://kotlinlang.org/docs/kotlin-docs.pdf

[7] D.Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)", RFC-4627, July 2006, DOI 10.17487/RFC4627

[8] Google Maps APIs Terms of Service, "Restrictions on Unfair Exploitation of the Service and Content", URL https://developers.google.com/maps/terms