Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in Ingegneria dell'Informazione

Tesina di laurea triennale

Sviluppo di App per sistema operativo Android

Caso di studio: gestione di viaggi

Candidato:
Trevisiol Angelo
Matricola 1008797

Relatore:
Andrea Pietracaprina

Anno Accademico 2012–2013

Indice

1 Introduzione **1**

2 Struttura di un'App **3**
2.1 Struttura logica 3
2.1.1 Activity 4
2.2 Struttura del progetto 6
2.2.1 Android Manifest 7
2.2.2 La cartella res 8

3 Grafica e basi di programmazione **11**
3.1 Java e XML 11
3.1.1 La classe R 12
3.2 Creazione di un'Activity 12
3.3 Grafica di base 13
3.3.1 View e Layout 13
3.3.2 TextView 15
3.3.3 EditText 16
3.3.4 Button 18
3.3.5 ListView 19
3.4 Grafica complessa 21
3.4.1 ExpandableListview personalizzata 22
3.4.2 I Fragment 30

4 Android API **33**
4.1 Scattare foto con Android: le API per la fotocamera 34
4.1.1 Scattare foto utilizzando l'Intent Fotocamera 34
4.2 Il salvataggio di dati nella memoria del dispositivo 38
4.2.1 Il File System di Android 38
4.2.2 Salvataggio e organizzazione file multimediali nella SD card . 39

5 Gestione Database in Android **43**
5.1 SQLite 44
5.2 Implementare un database nell'App 45
5.2.1 La classe Helper 45

5.2.2 La classe Adapter 46
5.2.3 Utilizzo del database nella propria Activity 52

6 L'applicazione Viaggio **55**
6.1 Come è nata l'idea 55
6.2 Obbiettivo dell'applicazione 56
6.3 Funzionalità dell'applicazione 56

7 Conclusioni **63**

Bibliografia **65**

Elenco delle figure

2.1 Ciclo di vita di un'Activity [6] 5
2.2 Struttura di un'App: AndroidManifest, src, resources 6
2.3 Le cartelle values, values-en, values-fr 10

3.1 View e ViewGroup 14
3.2 Esempio di TextView e AutoCompleteTextView 16
3.3 Esempio di EditText 18
3.4 Vari tipi di Button 18
3.5 Esempio di Button e ImageButton 20
3.6 Esempio di ListView 21
3.7 Expandable Listview di base 22
3.8 Expandable ListView con layout personalizzato 22
3.9 Layout dell'oggetto Child dell'ExpandableListView 23
3.10 Layout dei Groups dell'ExpandableListView 26
3.11 Esempio di due moduli dell'interfaccia definiti da Fragments, combinati in un'Activity per interfaccia tablet, ma separati per quella smartphone [8]. 31

4.1 Richiesta di un Intent di categoria Video 35
4.2 Intent dell'App fotocamera di Android 37
4.3 Organizzazione File System Android [19] 38

6.1 Riepilogo Viaggi 57
6.2 Inserimento nuovo viaggio 57
6.3 Tappe di viaggio 58
6.4 Spese 59
6.5 Appunti 60
6.6 PhotoGallery 61
6.7 Esempio di Tab [20] 61

Capitolo 1

Introduzione

Il termine smartphone è oggi sulla bocca di tutti. In pochi anni si è passati da telefoni cellulari con semplici funzioni telefoniche e di messaggistica a veri e propri mini computer dotati di fotocamera, navigatore satellitare, connessione internet e lettore multimediale. É cambiato radicalmente anche il modo con cui interagiamo con questi dispositivi. Attraverso di loro infatti è possibile in qualunque momento e in qualunque luogo avere accesso alle informazioni che più desideriamo. Con la recente invenzione dei tablet, dispositivi touch con le stesse funzionalità degli smartphone ma dotati di display più ampi, anche i tradizionali personal computer e notebook stanno avendo una forte crisi. É sempre meno diffusa infatti l'abitudine di mettersi alla scrivania per ricercare informazioni su internet, consultare le mail, interagire con i social network, tutto può essere fatto in tempo reale e dove più desideriamo. Le parole chiave in questi anni sono diventate mobilità e portabilità.

Nel mercato è presente una fortissima domanda di questi prodotti che sta coinvolgendo sempre più in maniera indiscriminata tutte le classi di consumatori. Anche i meno giovani infatti si stanno avvicinando a questo mondo attratti dall'immediatezza e la comodità che questi dispositivi riescono ad offrire. In questo contesto sviluppare applicazioni per dispositivi mobili può essere un'importante opportunità in quanto permette di proiettarsi in un mercato in piena espansione. Un esempio sotto gli occhi di tutti è Whatsapp, una piccola realtà, arrivata oggi a soli 45 dipendenti, che da un'idea brillante, cioè quella di sostituire i tradizionali SMS con messaggi multimediali inviati attraverso la rete, vanta oggi un fatturato medio di 100 milioni di Euro annui e 27 miliardi di messaggi scambiati ogni giorno.

L'evoluzione dei dispositivi mobili è stata caratterizzata da grandi successi ma anche grandi fallimenti (su tutti quello di Symbian) e oggi il panorama mondiale è sostanzialmente diviso tra iOS, sistema operativo mobile di Apple, Windows Phone di Microsoft e Android di Google. Ogni sistema operativo ha i propri punti di forza e le proprie peculiarità. In particolare però la natura "open source", il supporto per i molti dispositivi e la facilità di personalizzazione proprie di Android lo hanno portato rapidamente ad essere la piattaforma mobile più diffusa e apprezzata. La parola open ruota intorno a tutto il mondo Android, inclusa la sua potente piattaforma di sviluppo. I programmatori hanno infatti libero accesso a tutti gli strumenti usati dagli stessi creatori, quindi il potenziale delle applicazioni non ha alcuna limitazione, visto che non è presente discriminazione tra software nativo e

di terze parti. Queste caratteristiche, oltre al supporto degli strumenti di sviluppo per tutte le piattaforme e alla natura gratuita, permettono a chiunque di potersi cimentare nella sfida di creazione di una propria App.

Questa è proprio l'idea alla base della tesina che viene presentata. In risposta ad un concorso indetto da Samsung per gli sviluppatori Android, io e il collega Fabio Forcolin abbiamo deciso di metterci in gioco e sviluppare un'applicazione per questa piattaforma. L'App in questione svolge la funzione di diario di viaggio, nello specifico permette di annotare viaggi, spese, appunti e memorizzare foto, per organizzare e riguardare quando se ne ha la necessità tutti i ricordi accumulati. La tesi è composta da due parti, la prima sviluppata dal collega e la seconda da me. Il primo pezzo tratta gli aspetti generali e i motivi per cui abbiamo deciso di optare per il sistema operativo Android, per poi approfondirne l'architettura e illustrare le linee guida per la creazione di un App di successo. L'obiettivo della seconda parte della tesi è invece quello di fornire delle solide basi per la programmazione delle App, soffermandosi su alcuni aspetti chiave come l'interfaccia grafica, l'interazione con le API e l'utilizzo dei database. La trattazione si concluderà con la presentazione di un caso di studio specifico, l'applicazione che abbiamo sviluppato, illustrandone problematiche e funzionalità.

Nel primo Capitolo illustreremo i componenti fondamentali di un'applicazione, concentrandoci in particolare sull'Activity e sul suo ciclo di vita. Vedremo poi com'è costituito un progetto di un App e la funzione di tutti i file che sono presenti al suo interno.

Nel secondo entreremo proprio nel merito della programmazione, illustrando dapprima il diverso ruolo di Java e XML, per poi proseguire nella creazione della nostra prima Activity e nel posizionamento degli elementi grafici di base. Il Capitolo si concluderà presentando diversi concetti per la creazione di un'interfaccia più complessa, presentandone anche un esempio.

Nel Capitolo successivo introdurremo il concetto di API, mostrandone poi due casi di utilizzo. Il primo sarà allo scopo di interagire con la fotocamera dei dispositivi attraverso l'oggetto Intent, mentre il secondo per memorizzare i file all'interno del FileSystem, di cui verrà fornita una breve descrizione.

Nel quarto Capitolo affronteremo il problema della memorizzazione dei dati su Android. Valuteremo vantaggi e svantaggi di affidarsi ad un DBMS, in particolare SQLite ovvero la soluzione integrata nel sistema, per poi finire mostrando come creare ed utilizzare un database all'interno della nostra applicazione.

Nell'ultimo Capitolo vedremo come tutti i concetti e le conoscenze presentati nei capitoli precedenti sono confluiti nella costruzione dell'applicazione che abbiamo realizzato. Analizzeremo inoltre le funzionalità che essa mette a disposizione e quali sono i motivi che ci hanno spinto ad optare per un programma di questo tipo.

Capitolo 2

Struttura di un'App

Costruire un'App è un lavoro che richiede un'ingente quantità di tempo, specialmente se è la prima volta che se ne progetta una. Scriverne il codice presuppone infatti una buona conoscenza di come essa deve essere strutturata, di come è possibile interagirvi e di tutti i componenti che la costituiscono. In questo Capitolo vedremo in primo luogo come è organizzata un'applicazione a livello logico e come vengono gestiti i vari eventi che occorrono durante l'esecuzione. Nella seconda parte invece cominceremo a "toccare con mano" come sono organizzati i file all'interno di un progetto e i primi aspetti della programmazione, argomento che approfondiremo nei capitoli successivi [21].

2.1 Struttura logica

É possibile identificare quattro componenti principali che caratterizzano un'applicazione per sistema operativo Android. Ciascun componente è un diverso punto attraverso cui il sistema può interagire con l'applicazione. Non tutti però sono punti di accesso reali per l'utente, ma ciascuno di essi possiede una propria entità e svolge un determinato ruolo, contribuendo a definire il comportamento generale dell'applicazione [14].

- Activity: è l'elemento principale di cui sono costituite la maggior parte delle applicazioni. In un primo approccio puo' essere vista come la rappresentazione di una singola schermata (screen) di un progetto. La maggior parte delle App risulta quindi formata da un certo numero di Activity, formando un'esperienza utente coerente mantenendo però l'indipendenza l'une dalle altre. Rimandiamo una trattazione più approfondita alla sezione successiva.

- Services: componenti privi di interfaccia grafica (posseduta invece da un'Activity), solitamente eseguiti in background. Un esempio di loro utilizzo può essere la riproduzione musicale eseguita in sottofondo, mentre l'utente interagisce con gli altri applicativi.

- Broadcast Receiver: è un componente che non svolge alcuna operazione prestabilita, ma bensì rimane in ascolto e reagisce in conseguenza di un annuncio. Gli annunci sono trasmessi dal sistema operativo per notificare

degli eventi alle applicazioni, come per esempio che è stato terminato il download di un file, che lo schermo è stato disattivato, che la batteria è scarica, ecc. Ovviamente nella nostra applicazione possiamo istruire i nostri Receiver per reagire solamente ad alcuni tipi di annunci.

- Content Provider: rende disponibili un insieme di dati di una certa applicazione alle altre applicazioni, per esempio video, immagini, audio, ma anche i dati gestiti dalle applicazioni native (ad esempio i contatti della nostra rubrica). Solitamente i dati che vogliamo rendere, per così dire, "pubblici" vengono salvati all'interno del file system oppure in un database SQLite.

Come si può notare dalla loro descrizione, **Services**, **Broadcast Receiver** e **Content Provider** sono componenti adatti a particolari tipi di applicazioni, rispettivamente per svolgere servizi in background, per interagire le notifiche o per condividere i dati. L'elemento che è invece alla base della quasi totalità delle App presenti sullo store è l'Activity. Se si rende necessario infatti visualizzare delle informazioni, configurare delle impostazioni o richiedere dei dati all'utente tramite interfaccia grafica, non si può prescindere dall'utilizzare questo componente.

2.1.1 Activity

Il materiale per questa sezione è stato preso da [3], [16] e [13].

Cerchiamo quindi di capire quali sono le caratteristiche principali di un'Activity. In modo più preciso può essere definita come una finestra contenente l'interfaccia grafica dell'applicazione con lo scopo di gestirne l'interazione con l'utente[2]. Possono essere definite un qualsiasi numero di Activity in ogni App allo scopo di trattare diversi parti del software: ognuna svolge una particolare funzione e deve essere in grado di salvare il proprio stato in modo da poterlo ristabilire successivamente come parte del ciclo di vita dell'applicazione. Per esempio, in una applicazione per la gestione dei viaggi, un'Activity potrebbe rappresentare la schermata che permette di aggiungere le tappe, un'altra quella per visualizzarle, un'altra magari potrebbe tenere conto delle spese.
Tipicamente viene specificata l'Activity "principale", cioè quella che viene presentata all'utente quando l'applicazione viene avviata per la prima volta. Da ogni Activity è possibile avviarne delle altre per svolgere diverse funzioni, ma ogni volta che una nuova parte, quella precedente viene arrestata e il sistema la conserva in una pila (la "back stack"). L'Activity in esecuzione è detta in *foreground*, mentre quelle che sono in pausa e conservate nello stack si dicono in *background* e potrebbero essere terminate dal sistema operativo qualora la memoria fosse insufficiente. La "back stack" funziona con il principio "last in, first out", quindi quando l'utente ha terminato l'Activity corrente e preme il pulsante indietro, quella precedente viene riattivata.

Ciclo di vita di un'Activity

Dal momento in cui lanciamo l'applicazione fino al momento in cui essa viene messa in background, l'Activity passa attraverso varie fasi che costituiscono il suo **ciclo**

di vita. Per assicurare un corretto funzionamento dell'App, capire quali sono le sue fasi di cui è composto è di importanza fondamentale. Vedremo nel prossimo Capitolo che, per poter definire un'Activity, la nostra classe dovrà estendere la super classe Activity, che fornisce una serie di metodi per gestire tutti gli eventi che ne governano il ciclo di vita. In questa parte li prendiamo come riferimento per la spiegazione, il procedimento per la creazione di un'Activity verrà illustrato successivamente. La Figura 2.1 li illustra e ne mostra gli eventi associati:

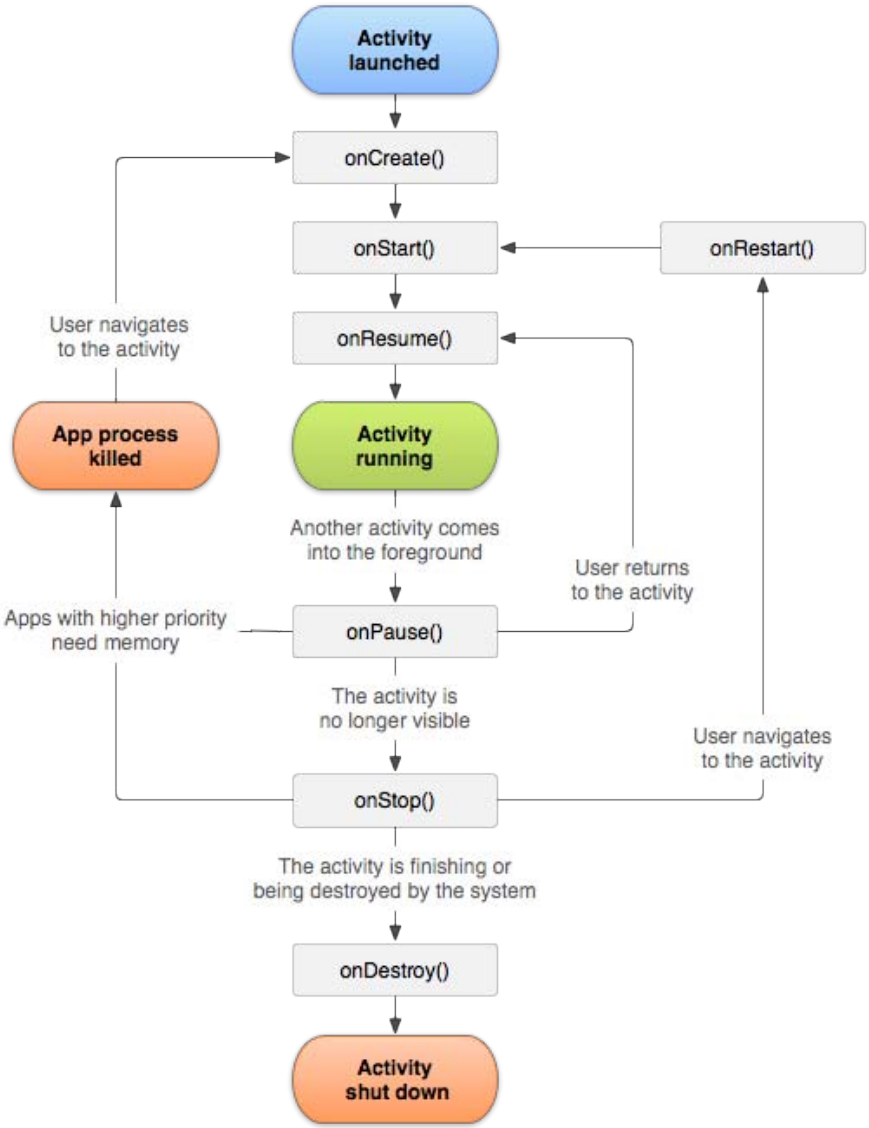

Figura 2.1: Ciclo di vita di un'Activity [6]

Il primo metodo che viene eseguito quando l'Activity viene invocata è **onCreate()**, che ha lo scopo di definire gli elementi di base per il funzionamento. Se al suo interno si fa utilizzo anche del metodo setContentView() è possibile definire un layout personalizzato. Una volta che ne viene portata a termine l'esecuzione, viene eseguito il metodo **onStart()**, che corrisponde al momento in cui l'Activity diventa visibile all'utente. Quando egli inzierà l'interazione con l'applicazione verrà richiamato il metodo **onResume()**. Da questo momento in poi verrà eseguito

normalmente il codice contenuto al suo interno (visualizzazione di file informazioni o file multimediali, pressione di pulsanti,ecc.), fino a quando non sopraggiungerà un altro evento come la chiamata ad un'altra Activity. In questo caso verrà invocato il metodo **onPause()**, dove solitamente si salvano le modifiche ai dati e si bloccano le animazioni per ridurre il carico di lavoro della CPU e il contesto dell'Activity verrà salvato sul prima introdotto "back stack". Se essa poi dovesse venire richiamata, riprenderà la sua esecuzione dal metodo onResume(), mentre se diventerà non più visibile all'utente verrà chiamato il metodo **onStop()**, il quale provvede, in caso di mancanza di memoria, ad eliminare le Activity in sospeso non necessarie. Nel caso l'applicazione venga riaperta dopo l'onStop(), questa dovrà ricominciare la sua esecuzione, tramite il metodo **onRestart()**, dal metodo onStart(). Infine è presente il metodo **onDestroy()**, che permette di svolgere le ultime operazioni prima che l'App venga terminata (dall'utente o per mancanza di memoria).

Il materiale per questa sezione è stato preso da [2]e da [3].

2.2 Struttura del progetto

Ora che abbiamo analizzato a livello teorico come si sviluppa nel tempo l'esecuzione di un'applicazione Android e quali sono i componenti principali che la costituiscono, iniziamo a vedere come sono organizzati i file per lo sviluppo del codice. Negli argomenti della prima tesi abbiamo già visto quali sono gli strumenti necessari allo sviluppo e come configurarli al meglio, quindi concentriamoci subito negli aspetti pratici. Quando creiamo un nuovo progetto, viene generata una struttura a cartelle e sottocartelle, che si mantiene anche per applicazioni più complesse e di maggiori dimensioni. Le componenti fondamentali sono quelle visibili anche in Figura 2.2:

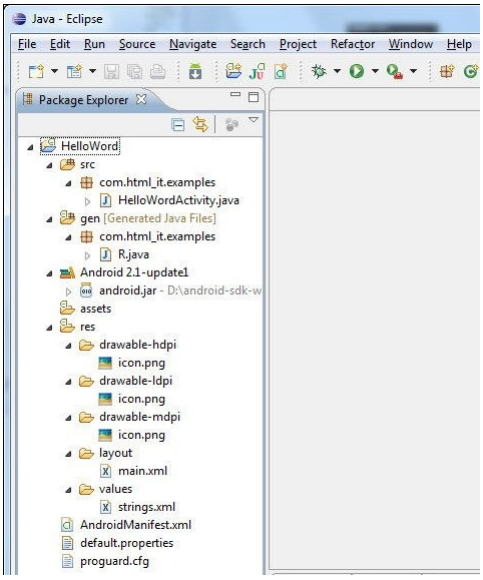

Figura 2.2: Struttura di un'App: AndroidManifest, src, resources

- Un file *AndroidManifest.xml*, situato nella cartella principale, nel quale vengono descritti il comportamento e la configurazione dell'applicazione, le risorse

di cui richiede l'utilizzo (fotocamera, SD,...) e la sua identificazione (nome, versione, logo).

- La cartella *src*, che contiene il codice sorgente dell'applicazione, il codice per l'interfaccia grafica, le classi etc.

- La cartella *res*, che contiene tutti i file e le risorse necessarie del progetto: immagini, video, layout xml etc. Può inoltre essere strutturata in sottocartelle per una migliore organizzazione delle risorse, come ad esempio *drawable*, *layout*, *anim* e così via.

- La cartella *gen*, con all'interno una serie di elementi di codice generati automaticamente al momento della compilazione del progetto, necessari al controllo delle risorse dell'applicazione.

- Infine la cartella *assets*, che contiene tutti gli altri file ausiliari necessari all'applicazione, ad esempio file di configurazione, di dati, etc.

2.2.1 Android Manifest

Vediamo più in dettaglio com'è costituito il file **AndroidManifest.xml**, di importanza fondamentale dato che in sua assenza o in caso di sua errata compilazione, può compromettere l'utilizzo dell'intera applicazione. Esso viene generato automaticamente dal plugin ADT una volta creato il progetto ed è memorizzato nella directory principale. Si distingue dagli altri file XML per la presenza nell'intestazione del tag $< manifest >$, che serve per includere i nodi che definiscono i componenti dell'applicazione, l'ambiente di sicurezza e tutto ciò che fa parte dell'applicazione. Nella sua interezza contiene tutte le informazioni specificate durante la creazione del progetto, l'icona, la versione dell'applicazione e tutte le **Activity** e i **Services** che compaiono al suo interno. Un esempio di dichiarazione di un'Activity può essere:

```
<activity android:name=".ExampleActivity" />
```

Un'altra funzione importante dell'AndroidManifest è la dichiarazione dei permessi di cui necessità l'applicazione. Di default infatti, l'accesso a particolari funzioni del dispositivo quali la fotocamera, il Bluetooth, la memoria o del sistema operativo quali i messaggi e le chiamate, non sarebbe permesso. Pertanto queste risorse, per poter essere utilizzate devono essere dichiarate proprio all'interno di questo file. In questo modo sia il gestore dei pacchetti può avere sotto controllo i privilegi necessari all'applicativo, sia l'utente è sempre avvisato prima dell'installazione su quali risorse l'applicazione potrà andare ad accedere.
I principali permessi che si possono richiedere sono:

- CALL_PHONE: controllare lo stato delle chiamate

- READ_CONTACTS e WRITE_CONTACTS: rispettivamente leggere e scrivere i dati dei contatti dell'utente

- RECEIVE_SMS: monitorare l'arrivo di messaggi SMS

- ACCESS_FINE_LOCATION e ACCESS_GPS: accesso alla posizione e utilizzo del GPS
- CAMERA: utilizzo della fotocamera
- WRITE_EXTERNAL_STORAGE: scrittura della microSD esterna
- INTERNET, ACCESS_NETWORK_STATE e ACCESS_WIFI_STATE: accedere e utilizzare la connessione Internet ed avere accesso allo stato della connessione WiFi.

Per poter dichiarare la necessità di utilizzare un determinato permesso si utilizza il codice XML:

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

Si fa notare che tramite questo codice si intende che l'applicazione necessita obbligatoriamente che questa caratteristica sia presente perché possa funzionare, ad esempio un programma per scattare foto con la fotocamera. Se invece la feature specificata non è strettamente necessaria al funzionamento è possibile utilizzare l'istruzione $< uses - feature >$, specificando la clausola $required = false$.

2.2.2 La cartella res

Approfondiamo ora alcuni aspetti riguardo la gestione delle risorse. Android presenta un'organizzazione lineare per la cartella *res*. Ciò significa che non vengono riconosciute tutte le sottocartelle oltre il secondo livello di gerarchia. Per esempio è raggiungibile la directory *layout* ma non un'eventuale cartella al suo interno. Chiarito questo aspetto vediamo quali funzioni hanno le directory presenti di default all'interno di res.

/layout

Layout contiene i file XML che definiscono i vari componenti della GUI (Graphic User Interface) utilizzati dall'applicazione. Essi possono essere sia definizioni di layout completi (LinearLayout, RelativeLayout,...), che di soli controlli grafici (Button, EditText, ListView,...). Ricordiamo che i vari elementi dell'interfaccia grafica possono anche essere definiti direttamente da codice senza l'utilizzo delle risorse XML.

/drawable

Le tre cartelle *drawable-**, rispettivamente **hdpi** per schermi ad alta densità di pixel, **mdpi** per quelli a media densità e **ldpi** per quelli a bassa densità, sono indicate per contenere le risorse grafiche (icone, sfondi,...) in formato GIF, PNG o JPEG utilizzate nell'applicazione. In questo modo il sistema potrà scegliere in automatico quelle più adatte alle differenti risoluzioni e densità dei display, prelevandole dalla cartella apposita. Recentemente, per ampliare il supporto alle nuove risoluzioni HD e FullHD e ai nuovi dispositivi di più ampie dimensioni come i tablet, è stata introdotta una nuova directory **xhdpi** per supportare i display ad altissima densità di pixel.

/menu

La cartella *menu*, come intuibile dal nome stesso, contiene i file XML che definiscono la struttura dei vari menù (come ad esempio quello delle opzioni).

/values

L'ultima cartella, *values*, contiene file XML che definiscono risorse di diversa natura, per esempio stili, stringhe utilizzate dall'applicazione, colori e tante altre tipologie di dati. Di particolare importanza è il file strings.xml, di cui esploreremo le funzionalità nella sezione successiva.

strings.xml e multilingua

Di grande importanza, sia per chiarezza nel codice, sia per facilità di implementazione del Multilingua, è il file strings.xml, situato nella cartella res/values. Si tratta ovviamente di una risorsa di tipo XML, con la funzione di "collezionare" le stringhe che si intende visualizzare nell'applicazione. La sintassi è un po' particolare, tuttavia Eclipse mette a disposizione una comoda interfaccia grafica per inserire più agevolmente stringhe e array. Di seguito riportiamo un esempio di loro dichiarazione all'interno del file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello World, ExampleActivity!</string>
    <string name="app_name">Example</string>

    <string-array name="planets_array">
        <item>Mercury</item>
        <item>Venus</item>
        <item>Earth</item>
        <item>Mars</item>
    </string-array>
</resources>
```

Nel codice Java possono essere richiamati agevolmente nel modo seguente:

```
    String helloWorld = getString(R.values.hello);
    String [] planets = getResources().getStringArray(R.array.
       planets_array);
}
```

Risulta quindi essere un comodo e ordinato raccoglitore di stringhe da utilizzare all'interno del codice, ma ancora non se ne intravede una reale utilità. Nella prima tesi abbiamo analizzato però che un buon modo per aumentare il possibile bacino di utenti può essere quello di tradurre l'applicazione in più lingue. Ed è qui che si manifesta l'utilità di questa soluzione: la maggior parte del lavoro sarà costituita dal tradurre i testi e le descrizioni presenti, nelle lingue che si vuol rendere disponibili. La cartella values identifica la lingua di default dell'applicazione, per aggiungere altre lingue, diverse da quella con cui si sta sviluppando l'App, è sufficiente creare delle ulteriori cartelle con nome values-xx, dove xx è il suffisso della nuova lingua. Quando l'applicazione verrà avviata, il sistema in base alla regione impostata nel

dispositivo caricherà la cartella values-xx appropriata. Se ad esempio la nostra lingua di default è l'Italiano, abbiamo definito una serie di stringhe all'interno di res/values/strings.xml e vogliamo ottenere il supporto per la lingua Inglese e Francese, è sufficiente tradurre le stringhe del file strings.xml nelle due lingue ed inserire i due nuovi file rispettivamente in values-en e values-fr. Se il sistema non trova una lingua adatta al dispositivo, caricherà quella di default. In Figura 2.3 si possono vedere la struttura all'interno del progetto e le tre cartelle sopra elencate contenenti i file strings.xml tradotti nelle varie lingue. Quindi, senza dover scrivere

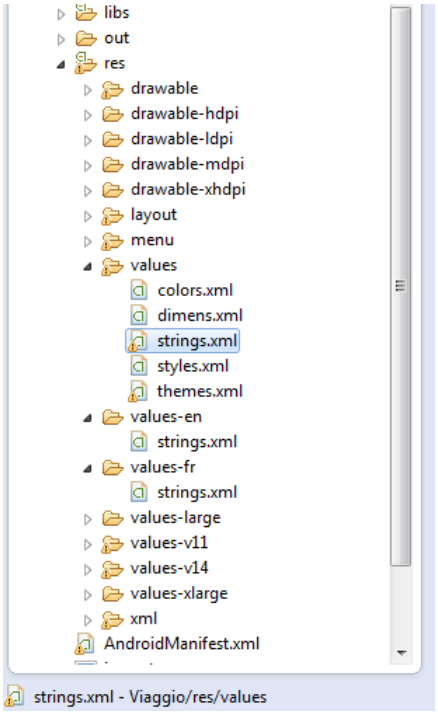

Figura 2.3: Le cartelle values, values-en, values-fr

alcuna riga di codice ulteriore, è possibile aggiungere qualsiasi lingua solamente traducendo il file strings.xml. I suffissi internazionali per le rispettive lingue (-en, -fr, ecc.) possono essere reperiti ad esempio nella colonna "ISO 639-1" della pagina web http://en.wikipedia.org/wiki/ISO_639-1_language_matrix. Sebbene sia meno frequente, è inoltre possibile associare ad ogni lingua, non solo le traduzioni delle stringhe, ma anche delle immagini o delle risorse in generale. Un esempio potrebbe essere una bandierina indicante la lingua, oppure un diverso sfondo; in questi sarebbe sufficiente inserire l'immagine nelle cartelle drawable-en e drawable-fr, lasciando la completa gestione al sistema.

Capitolo 3

Grafica e basi di programmazione

Nel secondo Capitolo abbiamo visto quali sono i componenti delle App a livello logico e come sono strutturati i file all'interno di un progetto, fornendo già qualche assaggio di programmazione. In questa parte chiariremo dapprima il ruolo di Java e di XML nella stesura del codice e, successivamente, come creare piccoli controlli grafici elementari, per poi avventurarci nella creazione di layout più complessi.

3.1 Java e XML

Cominciamo quindi col chiarire bene il ruolo dei due linguaggi all'interno della programmazione di applicazioni [21].

- Java: è un linguaggio di programmazione orientato agli oggetti, creato da James Gosling e altri ingegneri di Sun Microsystems nel 1992. Dal momento della sua creazione si è fatto largo dapprima nella creazione di applicazioni per piccoli dispositivi elettronici e nel 1993 ha cominciato a diffondersi per la realizzazione di pagine web interattive a livello client (ovvero direttamente sulla macchina dell'utente e non su un server remoto). Da quando Google nel 2005 ha preso possesso di Android ha avuto un ruolo fondamentale nello sviluppo di App per questo sistema operativo. Sebbene quest'ultimo non supporti la JVM (Java Virtual Machine) originale, ma ne possieda una versione proprietaria modificata, la DVM (Dalvick Virtual Machine), a livello di sviluppo del codice non si notano particolari differenze dato che sono presenti la maggior parte delle librerie. La principale differenza col classico sviluppo di codice Java è però il fatto che non è presente un entry point (il classico metodo "main") da dove normalmente un programma comincia a caricare le sue parti software e avviarsi: tutto è pensato per essere un "componente" pilotato dagli eventi ("Event Driven"). Un vantaggio è che il sistema operativo potrà ottimizzare le risorse, ad esempio rinunciando a caricare risorse e hardware non supportati o non prioritari perché inutilizzati. Il ruolo di Java, in questo contesto, è quello dello sviluppo delle parti dinamiche del programma come per esempio la gestione degli eventi in seguito agli input dell'utente e al suo interagire con l'interfaccia grafica, la modifica in real time del layout e così via. Per questo

una buona conoscenza di questo linguaggio può essere un ottimo punto di partenza [24].

- XML: eXtensible Markup Language è un linguaggio di markup o linguaggio marcatore, basato su un meccanismo sintattico che consente di definire e controllare il significato degli elementi contenuti in un documento o in un testo. Il suo ruolo nella programmazione Android è complementare a quello di Java. Mentre quest'ultimo viene utilizzato per le parti dinamiche, XML descrive le parti statiche. Esse possono essere quelle caratteristiche che non cambiano durante l'esecuzione dell'applicazione, come per esempio il colore dello sfondo della schermata, la posizione dei bottoni, delle caselle di testo, ecc [27]. In questo caso una buona conoscenza preliminare non è strettamente necessaria, in quanto la padronanza si acquisisce di più con l'utilizzo che non con la teoria.

3.1.1 La classe R

Riassumendo quindi, da una parte abbiamo delle risorse XML che definiscono le parti statiche dell'applicazione, come la GUI, i parametri, gli stili (contenute nella cartella res), mentre dall'altra il codice Java che permette di gestire eventi e interazioni con l'utente (nella cartella src). Il ponte fra questi due mondi viene gestito attraverso la classe R, contenuta nella cartella **gen** e generata in modo dinamico ad ogni compilazione del codice. Essa permette appunto di identificare in modo dinamico tutte le risorse (da stringhe a layout, fino agli stili) dichiarate via XML, per poterle modificare ed utilizzare all'interno di classi Java [11]. Ad esempio per importare il layout di un componente grafico per la visualizzazione di un testo come una TextView è possibile utilizzare l'istruzione:

```
TextView txt1 = (TextView)this.findViewById(R.id.txt1);
```

In cui txt1 è l'ID assegnato ad una TextView dichiarata all'interno di un file XML nella cartella res/layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
   android"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:orientation="vertical" >
<TextView android:id="@+id/txt1
android:layout_width="fill_parent"
android:layout_height="wrap_content" />
```

3.2 Creazione di un'Activity

Ora che abbiamo capito come far interagire i due linguaggi tra di loro e sfruttarne le rispettive potenzialità, vediamo come creare la nostra prima Activity e come riuscire ad associarci un layout. Per fare ciò è necessario creare una nuova classe che dovrà essere sottoclasse di Activity (o di una sua sottoclasse esistente). Qui è

necessario implementare metodi di callback che il sistema chiama quando avvengono le transizioni tra i vari stati del suo ciclo di vita, quando cioè viene creata, fermata, ripresa, o distrutta. Abbiamo già introdotto le varie fasi nel Capitolo precedente, tra di queste i metodi più importanti sono onCreate (), per specificare le azioni da fare dopo la creazione dell'Activity, onResume() durante l'interazione con l'utente e onPause () per salvare le modfiche che devono essere mantenute al di là della sessione utente corrente. Un esempio di un'Activity basilare potrebbe essere questo:

```
public class ExampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // L'Activity e' stata creata
    }
    @Override
    protected void onResume() {
        super.onResume();
        // L'Activity e' diventata visibile
    }
    @Override
    protected void onPause() {
        super.onPause();
        // L'Activity sta per essere distrutta
    }
}
```

Ora che abbiamo creato la struttura della nostra Activity supponiamo di voler assorciarci un layout chiamato main.xml, situato all'interno di res/layout. Ciò è possibile attraverso il metodo setContentView() della classe Activity che è consigliabile utilizzare all'interno di onCreate(). L'istruzione corrispondente sarebbe:

```
    setContentView(R.layout.main)
```

3.3 Grafica di base

Abbiamo capito come creare la nostra Activity, ma non siamo ancora in grado di assorciarci un layout corretto e capire come strutturarlo. A questo scopo è utile introdurre il concetto di View e Layout. La sezione che segue è stata tratta da [4], [16] e [15].

3.3.1 View e Layout

View è la classe di base per la definizione dell'interfaccia grafica e la gestione degli eventi nati dall'interazione dell'utente con la UI. L'interfaccia grafica di un'Activity viene in genere definita da una o più View, organizzate in una struttura ad albero e rappresentate sullo schermo tramite il **Layout**. Questi gruppi di View vengono appunto chiamati **ViewGroup**. Una View può quindi essere considerata come una porzione dell'interfaccia dell'Activity e spesso consiste in controlli grafici come Button, TextView, EditText, ecc., che sono delle sue sottoclassi con particolari funzionalità che verranno analizzate in seguito.

I Layout sono quindi estensioni della classe ViewGroup usati per posizionare le

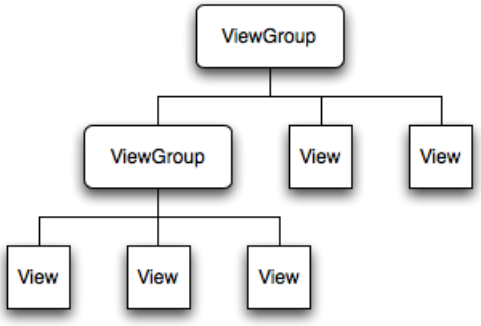

Figura 3.1: View e ViewGroup

View all'interno dell'interfaccia. Essi possono essere di quattro tipi:

- LinearLayout: è probabilmente il tipo di layout più diffuso. Dispone ogni elemento contenuto al suo interno lungo una riga se l'attributo orientation è impostato su vertical e su colonna se l'attributo è impostato su horizontal. In pratica il layout viene considerato come un "campo" con tanti rettangoli ordinati per riga oppure per colonna in cui disporre i vari componenti. Questa caratteristica ci permette di rendere il nostro form dinamico, astraendolo dalla dimensione dello schermo.

- TableLayout: ci permette di disporre gli oggetti View usando una griglia composta da righe e colonne, proprio come in una tabella.

- RelativeLayout: è il layout più flessibile tra quelli nativi perché permette di definire le posizioni di ogni oggetto View in relazione agli altri oggetti presenti. Il primo viene posizionato in alto a sinistra e poi per ogni nuovo elemento aggiunto deve essere definita la posizione relativa da questo.

- FrameLayout: è la tipologia più semplice di layout, perché non fa altro che posizionare in ordine di inserimento tutti gli oggetti View partendo dall'angolo in alto a sinistra dello schermo. È molto comodo quando implementiamo degli switcher di immagini o delle interfacce a tab.

Non si può dire che esista un layout migliore di un altro: in base alla situazione e al tipo di oggetto da posizionare, deve essere abilità del programmatore scegliere quello che più si adatta. L'obiettivo da perseguire è che le informazioni si adattino bene alla superficie dello schermo e che lo facciano sui dispositivi più disparati, aumentando la possibilità che gli utenti apprezzino l'ottimo lavoro di progettazione. Ovviamente per creare layout complessi è possibile inserire più layout, anche di diverso tipo, uno dentro l'altro. Questo aspetto verrà approfondito nella sezione di Grafica complessa.

Ora che abbiamo capito come posizionare gli elementi all'interno della schermata cominciamo a vedere come gestire gli elementi grafici di base, per poi avventurarci nella prossima sezione nello studio di layout più complessi. Per queste sezioni è stato preso spunto dagli esempi forniti all'interno dell'ebook [1] e dal sito [9].

3.3.2 TextView

Nella prima parte ci concentreremo su tre componenti per la visualizzazione del testo: TextView, AutoCompletetextview, MultiAutoCompleteTextView. La **TextView** non è altro che il classico componente etichetta, la cui funzione è quella di visualizzare del testo sul display. L'**AutoCompleteTextView** è invece una casella di testo con delle parole di default che possono essere inserite: quando iniziamo a scrivere nella casella, questa mostrerà le possibili alternative che possono essere scelte da una lista. Ciò comunque non è limitante, infatti non è obbligatorio inserire solamente le parole presenti, ma è possibile scegliere qualsiasi parola. La particolarità dell'AutoCompleteTextView è che l'intellinsense, ovvero la lista di possibili parole che ci viene proposta, visualizza una sola parola. La parola che viene proposta è quella che inizia con le stesse lettere della parola da noi scritta. Il **MultiAutoCompleteTextView** invece è simile al precedente componente ma può mostrare più parole alla volta, separate da un carattere speciale. Con un esempio di codice andiamo a vedere come si utilizzano i tre componenti.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:orientation="vertical" >
<TextView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/hello" />
<AutoCompleteTextView android:id="@+id/txtauto"
android:layout_width="fill_parent"
android:layout_height="wrap_content" />
<MultiAutoCompleteTextView android:id="@+id/txtmulti"
android:layout_width="fill_parent"
android:layout_height="wrap_content" />
<CheckedTextView android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/check"
android:gravity="center_horizontal"
android:textColor="#0000FF"
android:checked="false"
android:checkMark="@drawable/ic_launcher" />
</LinearLayout>
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
ArrayAdapter<String> aa = new ArrayAdapter<String>(this,
android.R.layout.simple_dropdown_item_1line, new String[] {"
    Inglese", "Spagnolo", "Tedesco",
"Italiano"});
AutoCompleteTextView actv = (AutoCompleteTextView) findViewById(R.
    id.txtauto);
actv.setAdapter(aa);
MultiAutoCompleteTextView mactv =
```

```
(MultiAutoCompleteTextView)this.findViewById(R.id.txtmulti);
mactv.setAdapter(aa);
mactv.setTokenizer(new MultiAutoCompleteTextView.CommaTokenizer())
    ;
}
```

Dal codice si può notare che, per prima cosa vengono ricavati i layout dei due componenti tramite l'ID definito nel rispettivo file XML e tramite lo stesso ArrayAdapter vengono caricare le parole all'interno dei componenti. L' **ArrayAdapter** è una sottospecie di vettore, nel nostro caso contenente oggetti di tipo String, con lo scopo di adattare le stringhe passate come parametri ad elementi visualizzabili dal componente grafico indicato: in questo caso simple_dropdown_item_1line, cioè una semplice lista a discesa. Per il MultiAutoCompleteTextView è necessario anche specificare il metodo setTokenizer, dove viene specificato che il carattere che farà da specificatore sarà una virgola. Un esempio dei controlli appena spiegati è mostrato nella Figura 3.2.

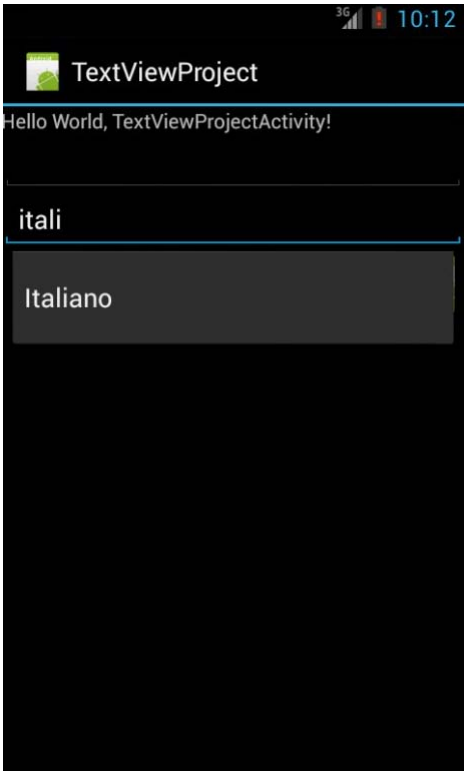

Figura 3.2: Esempio di TextView e AutoCompleteTextView

3.3.3 EditText

In molte applicazioni spesso è presente la necessità di inserire del testo, ad esempio la località in cui ci si trova o altre informazioni. Per fare ciò è presente l'**EditText**, che non è altro che la TextBox di altri linguaggi di programmazione come il Visual Basic. Tramite questo componente grafico è possibile inserire del testo e gestire gli eventi relativi al suo contenuto.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
   android"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:orientation="vertical" >
<EditText android:id="@+id/edit1"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:hint="Scrivi qualcosa..." />
<TextView android:id="@+id/testo"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="" />
</LinearLayout>
```

La proprietà **hint** è il messaggio iniziale presente nella casella di testo, solitamente un piccolo suggerimento riguardo cosa inserire. Viene visualizzato in grigio chiaro e scompare quando si comincia a scrivere il contenuto. É possibile ricavare il testo inserito nell'EditText, come nella TextView e molti altri controlli simili, tramite il metodo getText(), che restituisce la sequenza di caratteri contenuta nel controllo. Gli altri due metodi beforeTextChanged e afterTextChange restituiscono, rispettivamente, il testo prima e dopo la sua modifica.

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
text = (TextView)this.findViewById(R.id.testo);
EditText edit1 = (EditText)this.findViewById(R.id.edit1);
edit1.addTextChangedListener(new TextWatcher() {
@Override
public void beforeTextChanged(CharSequence arg0, int arg1, int
   arg2, int arg3) {
}
@Override
public void onTextChanged(CharSequence arg0, int arg1, int arg2,
   int arg3) {
text.setText(arg0);
}
@Override
public void afterTextChanged(Editable s) {
}
});
}
```

Nell'esempio di codice, tramite l'evento onTextChanged, si controlla se il testo nell'EditText è cambiato e, in caso affermativo, il testo viene assegnato alla TextView tramite il metodo setText(). Possiamo vedere un esempio di EditText in Figura 3.3.

Figura 3.3: Esempio di EditText

3.3.4 Button

Il **Button** (pulsante) è un altro dei componenti grafici classici presente in molti programmi. Può essere costituito da testo o un'icona (o da entrambi) che comunica quale azione si verifica quando l'utente andrà ad interagire con esso tramite un tocco. L'**ImageButton** è appunto la classe che permette di realizzare un bottone con un'immagine di sfondo ed è in comportamento e in struttura simile alla classe Button.

Figura 3.4: Vari tipi di Button

Vediamo ora con un esempio come è possibile includere un Button e un ImageButton all'interno del nostro programma e come gestire gli eventi a loro associati:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
   android"
 android:layout_width="fill_parent"
 android:layout_height="fill_parent"
 android:orientation="vertical" >
<Button android:id="@+id/btn1"
 android:text="@string/pulsante"
 android:layout_width="wrap_content"
```

```
 android:layout_height="wrap_content" />
<ImageButton android:id="@+id/btnimg1"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:contentDescription="@string/contentimg"
 android:text="@string/immagine"
 android:src="@drawable/ic_launcher"
 android:onClick="TestClick" />
</LinearLayout>
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.main);
 Button btn1 = (Button) findViewById(R.id.btn1);
 btn1.setOnClickListener(new OnClickListener() {
 @Override
 public void onClick(View arg0) {
  Toast.makeText(getApplicationContext(), "Click di Test",
  Toast.LENGTH_LONG).show();
  }
 });
}
public void TestClick(View view) {
 Toast.makeText(getApplicationContext(), "Click di Test Immagine",
      Toast.LENGTH_LONG).show();
}
```

Nel codice abbiamo gestito l'evento legato al click sui componenti con due diversi approcci. Il primo attraverso un **Listener** (gestore eventi), che permette di intercettare la pressione avvenuta del tasto (evento onClick), settando il bottone nello stato OnClickListener (in attesa di essere premuto). Il secondo approccio è invece quello di associare un metodo direttamente nella gestione del layout nel file .xml, nel nostro caso nel layout del componente ImageButton è stato configurato l'attributo onClick sul metodo TestClick attraverso l'istruzione android:onClick=TestClick. In entrambi i casi l'evento legato alla pressione del tasto è la visualizzazione di un Toast (riquadro di avviso) con scritto Click di Test per il Button e Click di Test Immagine per l'ImageButton. Il risultato è mostrato nella Figura 3.5.

3.3.5 ListView

La **ListView** è un componente che permette di visualizzare, come dice il nome stesso, una lista di elementi. Le voci contenute vengono caricate in automatico da un **Adapter**, un oggetto che si occupa sia dell'accesso ai dati, sia della loro visualizzazione.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
   android"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:orientation="vertical" >
<ListView android:id="@+id/listview1"
```

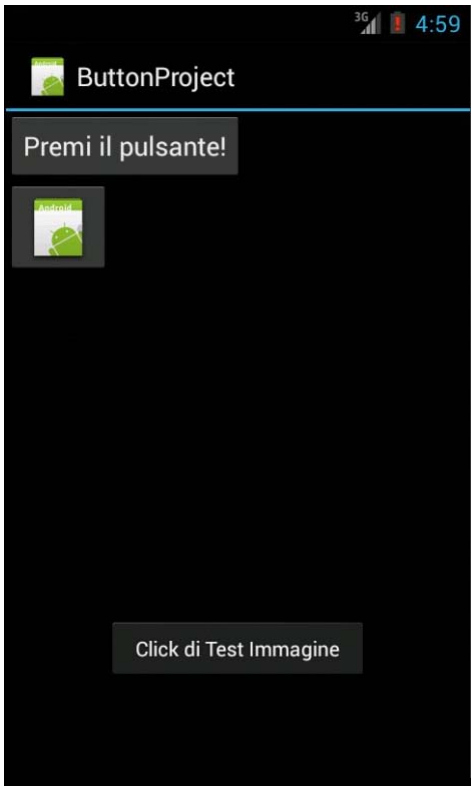

Figura 3.5: Esempio di Button e ImageButton

```
android:layout_width="fill_parent"
android:layout_height="wrap_content" />
</LinearLayout>
```

Invece di utilizzare il generico ArrayAdapter però, come abbiamo fatto ad esempio nella TextView, è più conveniente utilizzare il ListAdapter, componente pensato per adattarsi meglio proprio a questo tipo di componente grafico. Più precisamente si occupa di associare ad ogni elemento della lista un layout predefinito, in questo caso simple_list_item_1, e di caricare nel componente grafico l'array di stringhe cols che abbiamo definito.

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
String[] cols = new String[]{"Elemento 1", "Elemento 2", "Elemento
    3", "Elemento 4"};
ListView list1 = (ListView)this.findViewById(R.id.listview1);
ListAdapter adapter = new ArrayAdapter<String>(this, android.R.
   layout.simple_list_item_1,
cols);
list1.setAdapter(adapter);
}
```

In Figura 3.6 si trova un esempio di ListView.

Naturalmente è possibile definire Adapter personalizzati in modo da poter inserire nei componenti non solo delle semplici stringhe, ma anche pulsanti e qualsiasi

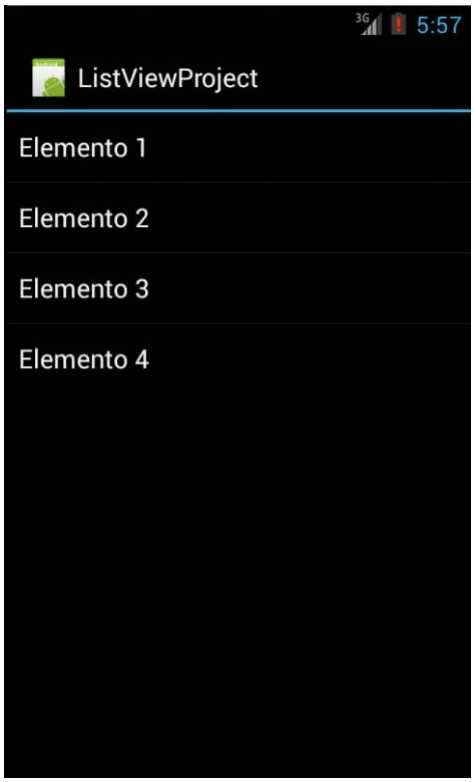

Figura 3.6: Esempio di ListView

controllo grafico desideriamo. Più precisamente sarà possibile gestire un vero e proprio layout completo all'interno per esempio degli elementi della ListView. Questo è proprio l'aspetto che viene approfondito nella prossima sezione.

3.4 Grafica complessa

Nei precedenti paragrafi abbiamo illustrato come gestire gli elementi fondamentali per comporre un'interfaccia grafica basilare. Quello che distingue però un'ottima applicazione da una ordinaria è non solo riuscire a svolgere ciò per cui è stata progettata, ma possedere un'interfaccia utente originale e piacevole da utilizzare. E' inoltre molto importante che rispetti le linee guida fornite da Google, come illustrato nei capitoli precedenti, che si propongono come aiuto in tal senso. In questo Capitolo vedremo appunto come sviluppare dei layout più complessi all'interno della nostra applicazione, in modo da mostrare le informazioni in modo più efficace ed elegante. In particolare ci concentreremo nella creazione di una grafica complessa per un oggetto di tipo ExpandableListview e nello studio dei Fragment, dei componenti introdotti recentemente per organizzare l'interfaccia in modo più flessibile e poter supportare con la stessa applicazione differenti dispositivi, dagli smartphone ai tablet.

3.4.1 ExpandableListview personalizzata

Un'ExpandableListView è un tipo di elemento grafico molto simile ad un ListView ma che a differenza di quest'ultima permette una suddivisione a due livelli. Nel primo sono presenti i principali argomenti da rappresentare detti anche gruppi (Groups), nel secondo i sotto-elementi relativi alla voce principale chiamati figli (Child). Ogni gruppo può essere espanso o compresso per visualizzare o nascondere i suoi elementi. Nel componente di base è possibile definire solamente una TextView per l'oggetto padre (primo livello) e una per i figli (secondo livello). Il risultato che vogliamo ottenere è a partire da questa configurazione riuscire ad aggiungere altri componenti per poter visualizzare altre informazioni. Presenteremo un esempio tratto dalla nostra applicazione in cui nell'elenco delle spese vogliamo visualizzare anche la data e l'ora in cui sono state effettuate e l'importo speso. Il concetto è illustrato nelle figure 3.7 e 3.8:

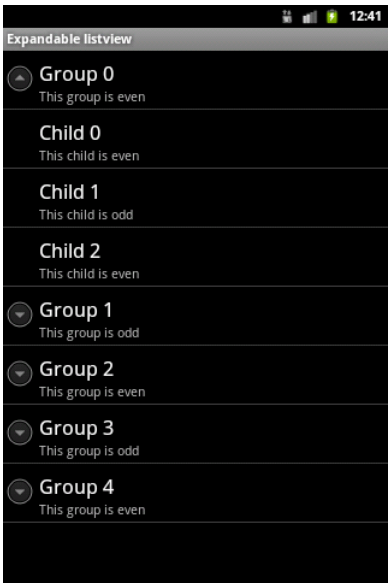

Figura 3.7: Expandable Listview di base

Figura 3.8: Expandable ListView con layout personalizzato

Per prima cosa è necessario definire una classe Child, che definisce il tipo di dato e i metodi associati agli elementi presenti nel secondo livello dell'Expandable ListView.

```
package utilities.pack;

import java.util.List;

public class SpesaChild {
    public String spesaChiledId;
    public ContenutiSpesa spesa;
    public SpesaChild(String chiledId, ContenutiSpesa mspesa) {
        super();
        this.spesaChiledId = chiledId;
        this.spesa=mspesa;
    }
    public String getChiledId() {
```

```
        return spesaChiledId;
    }
    public void setChiledId(String chiledId) {
        this.spesaChiledId = chiledId;
    }
    public void setChildSpese(ContenutiSpesa mspesa) {
        spesa=mspesa;
    }
    public ContenutiSpesa getChildSpesa() {
        return spesa;
    }
}
```

I metodi implementati sono molto semplici e sono solo quelli per impostare o estrarre i dati contenuti all'interno dell'elemento figlio. L'oggetto associato nel nostro esempio è di tipo ContenutiSpesa che serve a memorizzare tutti i dati relativi a una spesa che abbiamo definito nel database, cioè id, descrizione, importo, data, ora e categoria associata. Questo un piccolo estratto della classe:

```
package utilities.pack;

public class ContenutiSpesa {
    private String id_spesa="";
    private String descrizione="";
    private String importo="";
    private String data="";
    private String ora="";
    private String idCategoria;
    public ContenutiSpesa(String mid_spesa, String mdescrizione,
       String mimporto,String mdata,String mora,String
       midCategoria){
        .
        .
        .
    }
}
```

Una volta creato il contenitore dei dati, in un file XML definiamo come dev'essere costituito il layout di ogni elemento figlio dell'ExpandableListView. Nel nostro caso sarà presente nella parte superiore la descrizione della spesa, allineata sulla sinistra l'importo allineato sulla destra. Nello spazio immediatamente sottostante saranno invece presenti data e ora automaticamente salvate dal sistema. Il risultato è illustrato in Figura 3.9:

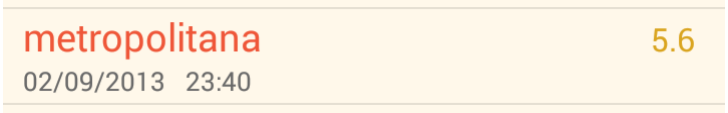

Figura 3.9: Layout dell'oggetto Child dell'ExpandableListView

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
   android"
    android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content"
    android:orientation="vertical" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:weightSum="100" >

        <TextView
            android:id="@+id/descr_vista"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="80"
            android:paddingLeft="10dp"
            android:paddingTop="1dp"
            android:textColor="#FF4500"
            android:textSize="19sp" >
        </TextView>

        <TextView
            android:id="@+id/importo_vista"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="20"
            android:gravity="right"
            android:paddingRight="15dip"
            android:paddingTop="1dip"
            android:textColor="#DAA520"
            android:textSize="16sp" >
        </TextView>
    </LinearLayout>

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <TextView
            android:id="@+id/data_vista"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:paddingLeft="10dp"
            android:textColor="#696969"
            android:textSize="13sp" >
        </TextView>

        <TextView
            android:id="@+id/ora_vista"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:paddingBottom="2dip"
            android:paddingLeft="10dp"
            android:textColor="#696969"
            android:textSize="13sp" >
        </TextView>
```

```
    </LinearLayout>

</LinearLayout>
```

Una volta impostato correttamente l'oggetto contenuto nei vari figli, vediamo come definire quelli contenuti nelle categorie principali:

```
package utilities.pack;

import java.util.ArrayList;

public class SpesaGroup {
    public String groupId;
    public ContenutiCategoria categoria;
    public ArrayList<SpesaChild> childrens;
    public SpesaGroup(String groupId, ContenutiCategoria categoria
        , ArrayList<SpesaChild> childrens) {
        super();
        this.groupId = groupId;
        this.categoria = categoria;
        this.childrens = childrens;
    }
    .
    .
    public ArrayList<SpesaChild> getChildrens() {
        return childrens;
    }
    public void setChildrens(ArrayList<SpesaChild> childrens) {
        this.childrens = childrens;
    }
}
```

La classe Group contiene, oltre ai metodi della classe Child come getGroupId(), setGroupId() ecc., omessi per evitare ridondanza, i metodi getChildrens() e setChildrens() allo scopo di rispettivamente estrarre e salvare gli elementi per ogni categoria. Essi verranno memorizzati in un ArrayList in modo da essere facilmente ricercati e selezionati. Anche per questa classe dobbiamo definire in un file XML come organizzare la visualizzazione delle categorie:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
   android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:weightSum="100" >

    <TableLayout
        android:layout_width="0dp"
        android:layout_height="fill_parent"
        android:layout_weight="78"
        android:orientation="vertical"
        android:paddingLeft="30dip" >

        <TableRow
            android:id="@+id/tableRow1"
```

```
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:paddingBottom="5sp"
            android:paddingLeft="5sp"
            android:paddingTop="5sp" >

            <View
                android:id="@+id/colorCategoria"
                android:layout_width="6dip"
                android:layout_height="28dip" />

            <TextView
                android:id="@+id/nomeCategoria"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:paddingLeft="5sp"
                android:textColor="#4A3F37"
                android:textSize="20sp" />
        </TableRow>
    </TableLayout>

    <TextView
        android:id="@+id/totaleCategoria"
        android:layout_width="0dp"
        android:layout_height="fill_parent"
        android:layout_weight="22"
        android:gravity="right"
        android:paddingLeft="5sp"
        android:paddingRight="10dip"
        android:paddingTop="5sp"
        android:textColor="#4A3F37"
        android:textSize="18sp"
        android:textStyle="bold" />

</LinearLayout>
```

Nel layout abbiamo definito tre elementi: a sinistra un'etichetta verticale del colore della categoria, il nome e allineato a destra il totale della spese finora accumulate. Anche in questo caso chiariamo meglio il concetto in Figura 3.10:

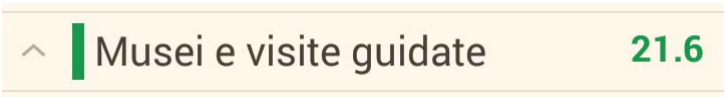

Figura 3.10: Layout dei Groups dell'ExpandableListView

Veniamo ora alla classe fondamentale di definizione dell'**Adapter**. Esso, come già anticipato, è un oggetto che si occupa di gestire l'interazione con un set dati e della rappresentazione grafica di ogni elemento che lo compone. Per poter creare un layout complesso non è più possibile utilizzare, come abbiamo fatto con la ListView, un Adapter predefinito, ma dobbiamo con una nuova classe ridefinirne il comportamento per adattarsi alla diversa natura dei dati.

```
package utilities.pack;
```

```
import java.util.ArrayList;
import viaggio.pack.R;
import android.content.Context;
.
.
public class SpeseExpandableAdapter extends
    BaseExpandableListAdapter {
     LayoutInflater inflater;

     /*list of group */
     private ArrayList<SpesaGroup> groups;
     public SpeseExpandableAdapter(Context context,ArrayList<
        SpesaGroup> groups) {
          super();
          this.groups=groups;
          inflater= (LayoutInflater) context.getSystemService(
             Context.LAYOUT_INFLATER_SERVICE);
     }
     /**
      * @param child
      * @param group
      *  use for adding item to list view
      */
     public void addItem(SpesaChild child,SpesaGroup group) {
          .
          .
     }
     public SpesaChild getChild(int groupPosition, int
        childPosition) {
          .
          .
     }
     public long getChildId(int groupPosition, int childPosition) {
          .
          .
     }
     @Override
     public int getChildrenCount(int groupPosition) {
          .
          .
     }
     public View getChildView(int groupPosition, int childPosition,
          boolean isLastChild,
               View convertView, ViewGroup parent) {
          SpesaChild child= (SpesaChild) getChild(groupPosition,
             childPosition);
          TextView descrizione=null;
          TextView importo=null;
          TextView data=null;
          TextView ora=null;
          if(convertView==null) {
               convertView=inflater.inflate(R.layout.riga_lista, null
                  );
          }
          descrizione=(TextView) convertView.findViewById(R.id.
             descr_vista);
```

```
        descrizione.setText(child.getChildSpesa().getDescrizione()
            );
        descrizione.setTextColor(Color.parseColor(groups.get(
            groupPosition).getGroupCategoria().getColore()));
        importo=(TextView) convertView.findViewById(R.id.
            importo_vista);
        importo.setText(child.getChildSpesa().getImporto());
        data=(TextView) convertView.findViewById(R.id.data_vista);
        data.setText(child.getChildSpesa().getData());
        ora=(TextView) convertView.findViewById(R.id.ora_vista);
        ora.setText(child.getChildSpesa().getOra());

        return convertView;
    }

    public SpesaGroup getGroup(int groupPosition) {

    }
    public int getGroupCount() {
        return groups.size();
    }
    public long getGroupId(int groupPosition) {
        return groupPosition;
    }
    public View getGroupView(int groupPosition, boolean isExpanded
        , View convertView,
            ViewGroup parent) {
        TextView categoriaName = null;
        TextView totaleCategoria=null;
        View coloreCat;
        SpesaGroup group=(SpesaGroup) getGroup(groupPosition);
        if(convertView==null) {
            convertView=inflater.inflate(R.layout.group_spesa_row,
                null);
        }
        categoriaName=(TextView) convertView.findViewById(R.id.
            nomeCategoria);
        categoriaName.setText(group.getGroupCategoria().getNome())
            ;
        coloreCat= (View) convertView.findViewById(R.id.
            colorCategoria);
        coloreCat.setBackgroundColor(Color.parseColor(group.
            getGroupCategoria().getColore()));
        totaleCategoria=(TextView) convertView.findViewById(R.id.
            totaleCategoria);
        totaleCategoria.setText(group.getGroupCategoria().
            getTotCategoria());
        totaleCategoria.setTextColor(Color.parseColor(groups.get(
            groupPosition).getGroupCategoria().getColore()));
        return convertView;
    }
    public boolean isChildSelectable(int groupPosition, int
        childPosition) {
        return true;
    }
    public boolean hasStableIds() {
```

```
        return true;
    }
}
```

Di particolare importanza sono i metodi getChildView() e getGroupView() che si occupano di associare alle viste dei gruppi e dei figli i file XML che abbiamo creato in precedenza e definiscono da dove prelevare i dati che poi verranno visualizzati. Il primo compito viene svolto attraverso l'operazione di inflating. Essa ci permette di istanziare un'oggetto da una risorsa XML, ad esempio per l'oggetto Group attraverso l'istruzione:

```
convertView=inflater.inflate(R.layout.group_spesa_row, null);
```

In questo modo all'interno dell'oggetto convertView vengono richiamati tutti gli elementi che abbiamo definito precedentemente in group_spesa_row.xml. Poi per ogni elemento presente andiamo ad associarci il layout e, attraverso il metodo setText() andiamo a impostarne il contenuto. Ad esempio per la textView associata al totale della categoria le operazione vengono svolte dalle istruzioni:

```
totaleCategoria=(TextView) convertView.findViewById(R.id.
    totaleCategoria);
totaleCategoria.setText(group.getGroupCategoria().getTotCategoria
    ());
```

Una volta completate tutte queste operazioni è molto facile utilizzare l'ExpandableListView personalizzata nella propria Activity, seguendo gli stessi passaggi che abbiamo utilizzato per gli altri componenti grafici:

```
import android.widget.ExpandableListView;
class ProvaActivity extends Activity{
    private SpeseExpandableAdapter expandableListAdapter;
    private ExpandableListView expandableListView;

    public void onCreate(){
    //inizializzo l'ExpandableListView
    expandableListView=(ExpandableListView) view.findViewById(R.id
        .listSpese);
    }

    public void onResume(){
        //associo l'Adapter all'ExpandableListView
        expandableListAdapter=new SpeseExpandableAdapter(
            getActivity().getApplicationContext(), groups);

        expandableListView.setAdapter(expandableListAdapter);
        }
}
```

Ovviamente sarà necessario anche dichiarare l'ExpandableListView in un opportuno file XML:

```
<ExpandableListView
        android:id="@+id/listSpese"
        android:layout_width="match_parent"
        android:layout_height="0dip"
        android:layout_weight="99" />
```

3.4.2 I Fragment

Per poter gestire elementi grafici molto complessi, dobbiamo introdurre il concetto di Fragment. Esso rappresenta un comportamento o una porzione di interfaccia grafica all'interno di un' Activity [8]. È possibile combinarne molti all'interno di una stessa Activity per costruire una interfaccia utente multi-riquadro ed inoltre essi sono riutilizzabili più volte all'interno di diverse schermate dell'applicazione. Una caratteristica importante dei Fragment è però quella di possedere un proprio ciclo di vita e di ricevere i propri eventi di input, con la possibilità di aggiungerli o rimuoverli mentre l'Activity è in esecuzione (un po' come un'Activity secondaria). Tuttavia essi devono sempre essere incorporati in un'Activity principale e il loro ciclo di vita è sempre direttamente influenzato dal quello dell'Activity ospite. Ad esempio, quando l'Activity è in pausa, lo sono anche tutti i suoi Fragment e lo stesso accade quando viene distrutta. In fase di esecuzione tuttavia, è possibile manipolare ogni Fragment in modo indipendente. Quando un Fragment viene aggiunto come parte del layout di un'Activity, esso vive in un ViewGroup all'interno della gerarchia del layout ed è possibile definirne uno specifico layout di visualizzazione. Per inserirlo nel layout di un'Activity è sufficiente dichiararlo nel file di layout con il tag fragment, oppure dal codice dell'applicazione. Tuttavia si possono anche utilizzare Fragment privi di interfaccia utente, come operatori invisibili per l'Activity.

Android ha introdotto i Fragment nella versione 3.0 (API livello 11), con lo scopo principale di supportare il design dell'interfaccia utente in modo più dinamico e flessibile su schermi di grandi dimensioni, come i tablet. Dato che lo schermo di questi ultimi è più ampio rispetto a quello degli smartphone, c'è più spazio per combinare e scambiare i componenti dell'interfaccia utente. Dividendo infatti il layout di un'Activity in Fragment, è possibile modificarne l'aspetto in fase di esecuzione, preservandone i cambiamenti in base al dispositivo utilizzato dall'utente.

Ad esempio un'App di notizie potrebbe utilizzare un Fragment per mostrare un elenco di articoli sulla sinistra e un altro per visualizzare l'articolo selezionato sulla destra. Così, invece di utilizzare un'Activity per selezionare un articolo e un'altra per leggerlo, l'utente può compiere entrambe le operazioni all'interno della stessa. Su uno smartphone invece, basterebbe separare i Fragment e fornire una interfaccia utente con un unico riquadro, che visualizza prima il Fragment con la lista articoli e poi quello con l'articolo selezionato. In questo modo l'applicazione potrebbe supportare sia tablet che telefoni mediante il riutilizzo dei Fragment in diverse combinazioni. Il concetto è illustrato in Figura 3.11
.

Vediamo più in dettaglio come si procede alla realizzazione di un Fragment. Per la sua creazione, è necessario creare una sottoclasse di Fragment (o di una sottoclasse esistente di esso). Questa classe avrà un codice molto simile a quello di un'Activity. Contiene infatti circa gli stessi metodi, come ad esempio onCreate(), onStart(), onPause() e onStop(). Di solito, è necessario implementare almeno i seguenti metodi del ciclo di vita, che gestiscono gli aspetti principali:

- onCreate(): evocato durante la creazione del Fragment allo scopo di inizializzare i componenti essenziali che si desidera mantenere, quando esso verrà sospeso o interrotto e poi ripreso.

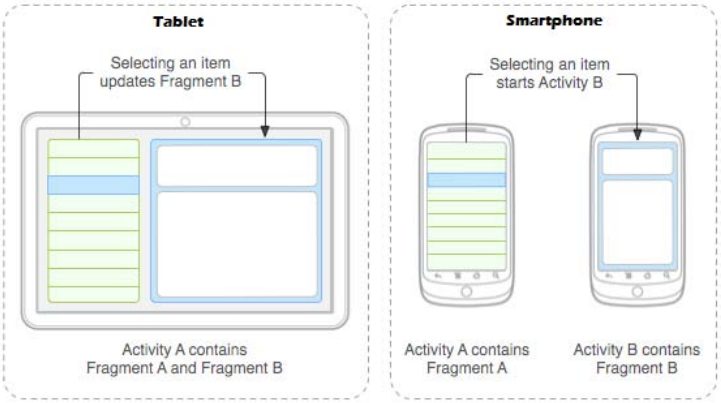

Figura 3.11: Esempio di due moduli dell'interfaccia definiti da Fragments, combinati in un'Activity per interfaccia tablet, ma separati per quella smartphone [8].

- onCreateView(): evocato quando è il momento di disegnare l'interfaccia utente del Fragment per la prima volta. A questo scopo è necessario che il metodo restituisca un oggetto di tipo View, oppure null se non è necessaria nessuna interfaccia utente.

- onPause(): evocato come prima indicazione del fatto che l'utente abbia lasciato il Fragment (anche se non sempre si traduce in una sua distruzione). Qui si dovrebbe salvare tutte le modifiche che devono essere mantenute al di là della sessione corrente (perché l'utente potrebbe non tornare).

La maggior parte delle applicazioni necessita di implementare almeno questi tre metodi, ma molti altri metodi del ciclo di vita si possono utilizzare per gestire al meglio le varie fasi. Un esempio di realizzazione della classe può essere il seguente:

```
public static class ExampleFragment extends Fragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
       container,
                             Bundle savedInstanceState) {
        // Associamo il layout contenuto in example_fragment.xml
        return inflater.inflate(R.layout.example_fragment,
           container, false);
    }
    @Override
    public void onPause() {
        //gestiamo l'evento del fragment in pausa
    }
}
```

Anziché la classe Fragment di base sono inoltre presenti alcune sottoclassi che si potrebbe voler estendere a seconda delle varie esigenze. Ad esempio è presente la classe DialogFragment utile per creare finestre di dialogo, ListFragment per

visualizzare un elenco di elementi e PreferenceFragment, utile quando si crea una schermata di “Impostazioni“ per l’applicazione.

Capitolo 4

Android API

In informatica con il termine**Application Programming Interface API** (Interfaccia di Programmazione di un'Applicazione) si è soliti indicare un insieme di procedure disponibili allo sviluppatore, raggruppate per formare un set di strumenti, adatti all'espletamento di un determinato compito all'interno di un programma. Spesso indica le librerie software messe a disposizione per un determinato linguaggio di programmazione, nel nostro caso specifico per il Sistema Operativo Android. Lo scopo è ottenere un'astrazione a più alto livello, di solito tra l'hardware e il programmatore o tra software a basso e quello ad alto livello, semplificando così il lavoro di sviluppo del codice. Le API permettono infatti di evitare ai programmatori di riscrivere ogni volta tutte le funzioni necessarie al programma dal nulla (dal basso livello) e rientrano quindi nel più ampio concetto di riutilizzo del codice . Il software che fornisce una certa API è detto implementazione dell'API.

Nello specifico il Sistema Operativo Android mette a disposizione un gran numero di API per far interagire l'applicazione con le funzionalità hardware dello smartphone e il software di sistema. Quelle principali sono:

- leggere/scrivere SMS, gestione delle chiamate in entrata e uscita
- modificare impostazioni di sistema
- leggere dati della rubrica, del calendario, dai social network, ecc.
- leggere scrivere sulla memoria interna o sulla memoria SD
- registrare file audio
- scattare fotografie tramite la fotocamera del dispositivo
- uso dell'NFC, della geolocalizzazione GPS, del WiFi e del Bluetooth

Vi sono inoltre una serie di API che sono state messe a disposizione dalle varie applicazioni che si sono sviluppate nel tempo per la piattaforma. Ad esempio quelle di Facebook e Google+ per la condivisione sui social network, quelle di Dropbox, Google Drive e Skydrive per il salvataggio dei dati in cloud, quelle di Gmail per l'invio di mail, ecc. Nei paragrafi successivi ci concentreremo in particolare sull'utilizzo delle API per lo scatto di fotografie tramite la fotocamera e

la loro memorizzazione all'interno del dispositivo, quelle da noi utilizzate all'interno dell'applicazione.

4.1 Scattare foto con Android: le API per la fotocamera

Le API per l'utilizzo della fotocamera sono di sicura utilità, poiché praticamente ogni smartphone equipaggiato con Android ne possiede almeno una, se non più di una per quelli dotati di quella frontale. Vi sono due approcci per l'utilizzo della fotocamera nella propria applicazione. Nel primo caso può essere fatto tramite l'utilizzo dell'applicazione presente nel software del telefono. In questo caso si avvia l'applicazione Fotocamera attraverso un**Intent** e si utilizzano i dati restituiti nella propria applicazione per il salvataggio, la modifica o la visualizzazione della foto. In alternativa si può decidere di integrare direttamente l'applicazione fotografica nella propria, tramite l'API Camera, per crearne una versione personalizzata, ad esempio con particolari filtri o modalità di scatto. Nella trattazione ci concentreremo nella prima soluzione, dato che si tratta di quella da noi utilizzata.

4.1.1 Scattare foto utilizzando l'Intent Fotocamera

Come premessa chiariamo il concetto di Intent, aspetto basilare della struttura di Android per permettere la condivisione delle funzionalità tra le varie applicazioni. Le informazioni per questa sezione sono state ricavate da una delle numerose guide fornite da Google [5].

Le azioni: Intent

Il materiale per questo paragrafo è stato preso da [17] e [10].

Un'**Intent** è un messaggio asincrono che consente, all'interno di un'applicazione, di richiedere delle funzionalità fornite da altre componenti di Android o da altri applicativi. Ad esempio, un'Activity può invocarne una esterna per scattare delle foto, inviare delle mail, effettuare una chiamata, ecc., senza preoccuparsi di dover implementare da zero tutto il codice. In particolare al suo interno riporta tutta una serie di parametri quali: **Component name** e **Category**, rispettivamente il nome e la categoria del componente che dovrà gestire la richiesta, **Action**, ovvero l'azione che dovrà avere luogo e **Data**, cioè la localizzazione della risorsa che vogliamo elaborare. Un esempio di quest'ultimo parametro potrebbe essere quando si sta avviando l'Intent per effettuare una chiamata, il numero di telefono della persona richiesta. Il Component name tuttavia non è sempre presente, dato che si tratta di un campo opzionale. Nei casi in cui viene specificato si parla di **Intent esplicita** e viene utilizzata principalmente per richiamare altre Activity, di cui si conosce precisamente il nome, all'interno della stessa applicazione. In caso contrario si parla di **Intent implicita**, dove assume un ruolo fondamentale il campo Category. In questo caso sarà compito del sistema operativo trovare il

componente più appropriato per svolgere il compito richiesto. Per esempio un'Intent potrebbe richiedere l'apertura di una determinata pagina web ad un componente di categoria browser. Il sistema valuterà tutte le soluzioni installate che corrispondono alla descrizione e proporrà la scelta all'utente, se non è già stata impostata un preferenza predefinita. Un esempio di questa situazione è rappresentata in Figura 4.1.

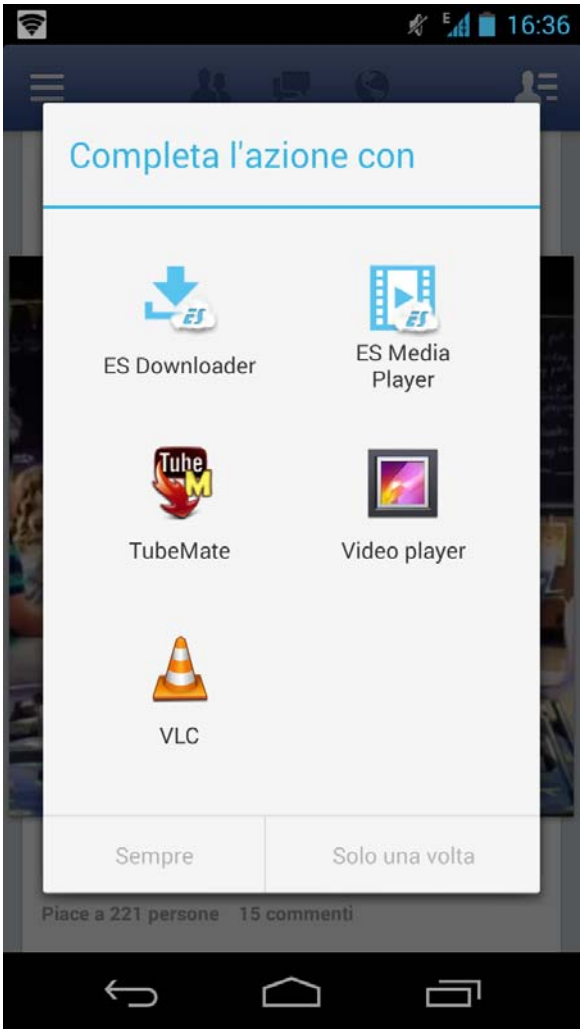

Figura 4.1: Richiesta di un Intent di categoria Video

Praticamente ogni funzionalità presente in Android è gestita tramite gli Intent ed essi sono a disposizione dello sviluppatore per utilizzarli o sostituire diversi componenti software. Come riesce però il sistema operativo a gestire un'Intent implicta se non viene specificato il Component name? Ciò avviene mediante gli **Intent Filters**, cioè delle direttive che hanno lo scopo di informare su quali funzionalità una certa applicazione è in grado di fornire. Per poter associare i filtri alla nostra applicazione è necessario agire nel già menzionato AndroidManifest.xml. Ad ogni Activity presente nel progetto sarà possibile associare una serie di Intent Filters per dichiarare ciò che essa è in grado di gestire:

```
<intent-filter . . . >
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    . . .
</intent-filter>
```

Dopo questa premessa introduttiva, vediamo appunto come gestire la Api per la fotocamera. Il primo passo da eseguire è quello di aggiungere tra i permessi della nostra applicazione quello di poter utilizzare la fotocamera:

```
    <uses-feature android:name="android.hardware.camera" />
```

Se questo permesso non è strettamente vincolante al funzionamento dell'intero programma, ma ne costituisce invece solo una delle funzionalità, si può comunicare al sistema che questo non è un requisito fondamentale:

```
    <uses-feature android:name="android.hardware.camera"
       android:required="false" />
```

In questo modo inoltre Google Play eviterà di nascondere l'applicazione ai dispositivi non dotati di fotocamera, che invece potrebbero usufruire delle altre funzionalità messe a disposizione. La procedura per invocare l'Intent segue i seguenti passaggi:

1. Creare l'Intent che richiede un'immagine o un video, utilizzando uno dei tipi seguenti:
   - MediaStore.ACTION_IMAGE_CAPTURE: richiede un'immagine da un'applicazione fotocamera esistente;
   - MediaStore.ACTION_VIDEO_CAPTURE: richiede un video da una applicazione fotocamera esistente;
2. Avviare l'Intent Camera. Utilizzando il metodo startSubActivity() avviamo l'Intent fotocamera, in questo modo dopo l'avvio comparirà sul display l'interfaccia dell'applicazione Fotocamera del dispositivo e l'utente potrà scattare una foto o un video.
3. Ricevere il Risultato dell'Intent. Impostare un metodo nell'applicazione per ricevere i dati dall'Intent tramite il metodo onActivityResult(). Questo verrà invocato quando l'utente avrà terminato di scattare una foto o un video (o avrà annullato l'operazione). Ecco un esempio di codice per poter eseguire tutte queste operazioni:

```
private static final int CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE =
    100;
private Uri fileUri;

public void onCreate(Bundle savedInstanceState) {
// creo l'Intent per scattare la foto e restituisco il controllo
    all'applicazione chiamante
Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);

fileUri = getOutputMediaFileUri(MEDIA_TYPE_IMAGE); // creo un file
    per salvare l'immagine
```

```
intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri); // imposto il
    nome per il file immagine

// avvio l'Intent per la cattura dell'immagine
startActivityForResult(intent, CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE
    );
}
```

Vediamo ora come poter gestire i dati al ritorno dall'Intent:

```
private static final int CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE =
    100;

@Override
protected void onActivityResult(int requestCode, int resultCode,
    Intent data) {
    if (requestCode == CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE) {
        if (resultCode == RESULT_OK) {
            // Immagine catturata e salvata nel fileUri
               specificato nell'Intent
            Toast.makeText(this, "Immagine salvata in:\n" +
                    data.getData(), Toast.LENGTH_LONG).show();
        } else if (resultCode == RESULT_CANCELED) {
            // L'utente ha cancellato l'immagine catturata
        } else {
            // Cattura dell'immagine fallita, avviso all'utente
        }
    }
}
```

Figura 4.2: Intent dell'App fotocamera di Android

Attraverso il metodo onActivityResult() è quindi possibile gestire gli eventi nel momento del ritorno dall'Intent della Fotocamera, sia quando l'utente ha deciso di salvare l'immagine, sia quando ha deciso di scartarla o c'è stato un errore improvviso.

4.2 Il salvataggio di dati nella memoria del dispositivo

Vediamo ora come è possibile, una volta prodotto il file immagine o in generale un qualunque file multimediale o di testo, salvare il file e organizzare al meglio le risorse all'interno della memoria del dispositivo. A tale scopo è utile fare una piccola introduzione su come è organizzato il File System di Android. [19]

4.2.1 Il File System di Android

Android utilizza diverse partizioni (per esempio boot, system, recovery, data, ecc) per organizzare i file e le cartelle all'interno dispositivo, con la stessa filosofia del sistema operativo Windows. Ad ogni partizione è associata una ben precisa funzionalità:

Figura 4.3: Organizzazione File System Android [19]

- /boot: come suggerisce il nome, questa è la partizione di avvio del dispositivo. Include il kernel Android e il Ramdisk e deve essere necessariamente presente perchè il dispositivo acceda al sistema operativo.
- /system: contiene l'intero sistema operativo Android, ovviamente escluse le due componenti presenti nella partizione di boot. Include cioè l'interfaccia grafica di Android e tutte le applicazioni di sistema preinstallate sul dispositivo.
- /recovery: è appositamente progettata per il backup. La partizione recovery può essere considerato come una partizione di boot alternativa, che consente al dispositivo di avviarsi in una console di ripristino che permette l'esecuzione di operazioni avanzate di manutenzione e di recupero.
- /data: è indicata come la partizione dei dati dell'utente. Infatti essa contiene ad esempio i contatti della rubrica, gli sms, le impostazioni e tutte le applicazioni Android installate.
- /cache: in questa partizione, Android memorizza i dati ad accesso più frequente ed i componenti delle App.

- /misc: contiene varie impostazioni del sistema che includono il CID (Carrier ID o Regione), la configurazione USB e alcune impostazioni hardware.

- /sdcard: non è una partizione della memoria interna del dispositivo, ma piuttosto della scheda SD. In termini di utilizzo, questo è lo spazio di archiviazione da utilizzare, ad esempio, per memorizzare i file multimediali, i documenti e i file personali. Nei dispositivi dotati sia di SD interna sia di quella esterna, la partizione /sdcard viene comunque utilizzata sempre per fare riferimento alla partizione interna. Per quella esterna è invece utilizzata un nome alternativo, che però differisce da dispositivo a dispositivo, ad esempio /sdcard2. Non ci sono tuttavia dati di sistema o di applicazioni memorizzati automaticamente in questa partizione, ma tutto il contenuto presente deve essere aggiunto manualmente dall'utente.

4.2.2 Salvataggio e organizzazione file multimediali nella SD card

E' proprio quest'ultima partizione quella su cui porre la maggior attenzione per le nostre finalità. I file multimediali creati dagli utenti, devono infatti essere salvati proprio in questa posizione per risparmiare spazio per il sistema e consentire agli utenti di accedere ai file anche senza il loro dispositivo. All'interno delle varie directory della partizione ci sono varie posizioni dove è possibile salvare i file multimediali, tuttavia Google propone due posizioni standard che è bene considerare come sviluppatore [5]:

- Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES): questo metodo restituisce il percorso standard consigliato per il salvataggio di foto e video. La directory è condivisa in modo che altre applicazioni possano facilmente trovare, leggere, modificare e cancellare i file salvati in questa posizione. Se l'applicazione viene disinstallata i file multimediali salvati in questo percorso non saranno rimossi. Per evitare di interferire con le foto e i video esistenti, è utile creare una sottodirectory per quelli relativi alla nostra applicazione.

- Context.getExternalFilesDir(Environment.DIRECTORY_PICTURES): questo metodo invece restituisce un percorso standard per la memorizzazione delle immagini dei video associati all'applicazione. Se essa viene disinstallata, i file salvati in questa posizione verranno però rimossi.

É possibile considerare una terza soluzione, come ad esempio fatto nella nostra App, cioè quello di creare al primo avvio dell'applicazione una directory all'interno di quella principale della memoria SD col nome dell'App. In questo modo, attraverso l'opportuna creazione di sottodirectory, sarà possibile organizzare le informazioni in spazi riservati e facilmente consultabili. In particolare noi abbiamo creato una cartella "Viaggio" con all'interno varie sottocartelle in base ai vari viaggi salvati. Ecco il codice che mostra il concetto appena descritto nel nostro esempio:

```
String path="/"+destinazione+"_"+partmod+"_"+ritmod; //creo l'
    ultima parte del path della cartella
```

```
File mainFolder = new File(Environment.
   getExternalStorageDirectory() + "/Viaggio"); //creo un file
    con il percorso della directory viaggio in quella
   principale della SD
File thisFolder = new File(Environment.
   getExternalStorageDirectory() + "/Viaggio"+path); //creo la
    cartella specifica del viaggio
if (!mainFolder.exists()) {//se la cartella principale non
   esiste
                mainFolder.mkdir();             //la creo
                thisFolder.mkdir();             //creo quella
                   per il viaggio
        }
        else{ //se esiste gia
                thisFolder.mkdir();  //creo solo quella per il
                    viaggio
        }
```

Ora che abbiamo organizzato dove poter salvare le immagini catturate, vediamo come è possibile modificare il nome dei file per la memorizzazione. Notiamo che all'interno del codice per l'avvio dell'Intent della fotocamera avevamo utilizzato la seguente istruzione:

```
fileUri = getOutputMediaFileUri(MEDIA_TYPE_IMAGE); // creo un
   file per salvare l'immagine
```

Essa richiama il metodo getOutputMediaFileUri() che si occupa di creare l'URI(Uniform Resource Identifier) per la memorizzazione dell'immagine. Essa non è altro che una stringa che identifica univocamente una risorsa generica che può essere un indirizzo Web (in questo caso viene chiamata URL), un documento, un'immagine o un file [26]. Nel nostro caso consisterà nel percoso in cui dovrà essere salvato il file.

```
/** Create a file Uri for saving an image or video */
private static Uri getOutputMediaFileUri(int type){
      return Uri.fromFile(getOutputMediaFile(type));
}
```

A sua volta questo metodo però, per restituire l'URI della risorsa, ne richiama un altro: getOutputMediaFile(). Quest'ultimo si occupa di creare il file con un nome appropriato, attraverso il quale con il metodo Uri.fromFile() sarà ricavato l'URI corretto.

```
/** Create a File for saving an image or video */
private static File getOutputMediaFile(int type){
         File mediaStorageDir = new File(Environment.
            getExternalStorageDirectory() +"/Viaggio"+percorso
            );

 // Creiamo il nome per il file multimediale
    String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss")
       .format(new Date()); //ricaviamo l'ora di sistema
    File mediaFile;
    if (type == MEDIA_TYPE_IMAGE){
        mediaFile = new File(mediaStorageDir.getPath() + File.
           separator +"IMG_"+ timeStamp  + ".jpg");
```

```
        } else if(type == MEDIA_TYPE_VIDEO) {
            mediaFile = new File(mediaStorageDir.getPath() + File.
                separator +"VID_"+ timeStamp + ".mp4");
        } else {
            return null;
        }
        return mediaFile;
    }
```

Per i nomi dei file abbiamo deciso, per uniformità e familiarità dell'utente, di applicare gli stessi che il sistema operativo applica abitualmente. Quello dei file immagine sarà infatti del tipo: IMG_data dello scatto.jpg. In generale tuttavia è possibile definire un nome personalizzato in base alle esigenze.

Capitolo 5

Gestione Database in Android

Il materiale per questo capitolo è stato preso da [18]
La piattaforma Android fornisce diversi metodi e strumenti per salvare i dati delle applicazioni in modo permanente. Abbiamo già visto in un Capitolo precedente come gestire la memorizzazione dei file all'interno della memoria condivisa (SD card). Se volessimo invece equipaggiare la nostra applicazione con l'attivazione/disattivazione del suono di sottofondo, oppure con la scelta della lingua da utilizzare potremmo implementare un menu di preferenze ed utilizzare le **Shared Preferences** per salvare le scelte dell'utente, funzionalità disponibile in modo nativo. Un'alternativa potrebbe essere quella di memorizzare i dati sul web attraverso una connessione di rete, in modo da renderli accessibili anche ad altri dispositivi. Un'ultima possibilità per salvare i nostri dati può essere quella di organizzarli in file di testo o in un database. Relativamente a questa soluzione, i vantaggi del primo approccio sono la semplicità e il poco sforzo a livello di codice, ma se emerge la necessità di trattare dati più numerosi e complessi la potenza e la versatilità della seconda soluzione sono da preferire. Nella nostra applicazione abbiamo deciso di utilizzare la seconda soluzione, non tanto per la mole di dati che ci siamo trovati a dover trattare, ma per la loro grande diversità e varietà. In nostro supporto è venuto il **DBMS** (Database Management System) integrato all'interno del sistema operativo Android. Esso è un sistema software progettato per consentire la creazione, la manipolazione e l'interrogazione (da parte di uno o più utenti) di database [23]. Quali sono allora i vantaggi pratici di usare un DBMS? Per prima cosa in un database i dati possono essere strutturati e tipizzati. Immaginiamo ad esempio di dover realizzare un'applicazione che gestisca una lista di contatti, come un'agenda. Ogni contatto sarà caratterizzato da un identificativo (ID), un nome, un cognome e un indirizzo. Se dovessimo salvare queste informazioni in un file di testo, dovremmo inventare da capo un nuovo formato e programmare poi le classi in modo da gestirlo. La soluzione più classica consisterebbe nell'organizzare ogni contatto su ciascuna riga, separando poi i campi con una virgola. Ad esempio:

- 1,Mario,Rossi,061299178
- 2,Antonio,Verdi,028667661

I dati di un contatto potrebbero inoltre non essere solo di natura testuale, ad esempio delle date o dei numeri interi o decimali. Ciò richiederebbe un sacco di

lavoro di codifica e decodifica, sia per gestire la struttura del file, sia per una corretta interpretazione dei tipi di dato coinvolti. Con un database, invece, è tutto più semplice: si crea una tabella con tanti campi quanti sono quelli necessari, ognuno abbinato già al tipo più consono. Le operazioni di lettura e scrittura, servendosi della libreria di accesso al DBMS, necessitano poi di pochissime righe di codice. Un altro settore nel quale è difficile competere con un DBMS è quello delle ricerche: nell'agenda basata su file di testo bisognerebbe implementare degli algoritmi per la ricerca dei contatti ideati su misura. Fare però un buon algoritmo di ricerca, allo stesso tempo veloce quanto poco pretenzioso di risorse, non è una cosa semplice. Un DBMS contiene il risultato di anni di sviluppo nel settore delle ricerche ottimizzate, ecco che quindi l'ago della bilancia si sposta ulteriormente dalla parte dei database.

5.1 SQLite

Il DBMS integrato in Android proviene dal mondo dell'Open Source. Si tratta di SQLite, che si distingue per leggerezza e facilità di integrazione. Per poterlo utilizzare non è necessario compilare né installare alcun software, è tutto compreso nel sistema operativo stesso. Gli sviluppatori di Google hanno anche realizzato delle interfacce di programmazione Java utili per richiamare ed utilizzare database dall'interno di una qualsiasi applicazione. SQLite si distingue dagli altri DBMS per la sua leggerezza, infatti lo spazio occupato in memoria e sul disco varia da appena 4KB a circa 350KB. É pertanto la tecnologia perfetta per creare e gestire database in un ambiente come quello degli applicativi mobile, dove le risorse sono molto limitate e dunque è molto importante ottimizzarne l'utilizzo. Esso possiede inoltre praticamente tutti gli strumenti principali che caratterizzato i più importanti database SQL (tabelle, viste, indici, trigger) ed il codice è distribuito gratuitamente sia per scopi commerciali che privati. La sua diffusione è più ampia di quanto ci si possa aspettare: è infatti utilizzato in moltissimi applicativi e device che utilizziamo quotidianamente, come l'iPhone della Apple, Mozilla Firefox, negli smartphone Symbian, in Skype, in diversi applicativi PHP o Adobe AIR, e in molti altri progetti [25].

Android memorizza i file seguendo uno schema preciso, infatti il database di ogni applicazione viene memorizzato in una directory il cui percorso è: /data/data/packagename/databases dove "packagename" è il nome del package del corrispondente applicativo. Ogni applicazione crea ed utilizza uno o più database in maniera esclusiva, cioè nessun'altra può riuscire ad accedere ai dati in essi contenuti. La condivisione di dati strutturati fra più applicazioni Android, infatti può avviene solo mediante i Content Providers, che abbiamo analizzato nel secondo Capitolo. Per accedere al file del database dalla nostra app abbiamo a disposizione i tipici statements del linguaggio SQL. Grazie alle classi che vedremo nei paragrafi successivi che permettono di implementare diversi Helper e Adapter, Android nasconde allo sviluppatore parte del lavoro a basso livello, ma è in effetti necessario avere una conoscenza almeno basilare di SQL.

5.2 Implementare un database nell'App

Quando si vuole implementare un database in un'applicazione è buona pratica creare una classe **Helper** e una classe **Adapter** in modo da semplificare le interazioni con esso. In questo modo si introduce un livello di astrazione che fornisce metodi intuitivi, flessibili e robusti per inserire, eliminare e modificare i record del database. Compito della classe Helper è quello di memorizzare le costanti statiche come il nome della tabella e dei vari campi, mentre quello della classe Adapter quello di fornire query e metodi per la creazione del database e la gestione delle connessioni (apertura e chiusura) con esso.
Il materiale per le prossime due sezioni è stato preso da [12] e [18].

5.2.1 La classe Helper

La classe Helper, da noi chiamata DatabaseHelper, dovrà estendere android.database.sqlite.SQLiteOpenH
e contiene appunto il nome del database, la versione, il codice di creazione delle varie tabelle e la loro successiva modifica. Facciamo notare che una classe di questo tipo deve essere creata per ciascun database necessario alla propria applicazione. Vediamo in dettaglio il codice utilizzato:

```
package utilities.pack;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class DatabaseHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME = "viaggio.db"; //
       nome del database
    private static final int DATABASE_VERSION = 1;            //
       versione database

// Lo statement SQL di creazione del database
    private static final String VIAGGIO_CREATE = "create table
       viaggio (_idviaggio integer primary key
       autoincrement, destinazione text not null, partenza text
       not null, arrivo text not null,trasporto          text not
        null,tabshown text not null);";
    .
    .
    .
    // Costruttore
    public DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
```

In questa prima parte, nelle due costanti DB_NAME e DB_VERSION vanno indicati appunto rispettivamente, il nome del database (viaggio.db) e il numero di versione (1). Quest'ultimo è un intero che va incrementato ogni volta che, in un nuovo rilascio del software, la base di dati viene modificata nella sua struttura. In questo modo quando l'utente aggiorna la propria applicazione, la classe riesce a capire se il database della versione precedentemente installata deve essere aggiornato oppure

no. Sono presenti poi una serie di stringhe, una per ogni tabella, che verranno poi utilizzate nei metodi onCreate() e onUpgrade() per la loro creazione e modifica all'interno del database.

```
    // Creazione del database
    @Override
    public void onCreate(SQLiteDatabase database) {
        database.execSQL(VIAGGIO_CREATE);
        database.execSQL(TAPPA_CREATE);
        database.execSQL(APPUNTO_CREATE);
        database.execSQL(SPESA_CREATE);
        database.execSQL(VALUTA_CREATE);
        database.execSQL(CATEGORIA_CREATE);
    }

    // Upgrade del database, es. incremento il numero di versione
    @Override
    public void onUpgrade( SQLiteDatabase database, int oldVersion
        , int newVersion ) {
        database.execSQL("DROP TABLE IF EXISTS viaggio");
        database.execSQL("DROP TABLE IF EXISTS tappa");
        database.execSQL("DROP TABLE IF EXISTS appunto");
        database.execSQL("DROP TABLE IF EXISTS spesa");
        database.execSQL("DROP TABLE IF EXISTS valuta");
        database.execSQL("DROP TABLE IF EXISTS categoria");
        onCreate(database);
    }
}
```

In quest'ultima parte di codice sono presenti i metodi onCreate() e onUpdate(), che devono necessariamente essere implementati, dato che si sta estendendo la classe astratta SQLiteOpenHelper. Il primo viene richiamato quando l'applicazione è stata appena installata e non fa altro che eseguire lo script SQL di creazione della base di dati definito dalle costanti di tipo stringa dichiarate nella prima parte di codice. Ciò avviene mediante il metodo execSQL() degli oggetti di tipo SQLiteDatabase, che permette di eseguire del codice SQL arbitrario. OnUpdate(), invece, viene richiamato quando il database è già presente sul sistema, ma stando al numero di versione richiesta, risulta obsoleto. Di solito ciò avviene dopo aver cambiato la versione di un'applicazione già installata. I parametri oldVersion e newVersion, come è facile intuire, indicano rispettivamente la versione già installata del database e quella ancora da installare. In questa versione il metodo onUpdate() effettua semplicemente il DROP (eliminazione) delle tabelle esistenti (viaggio, tappa, spesa, ecc.) e la sostituisce con la nuova definizione delle stesse richiamando il metodo onCreate(). Una versione di questo metodo più complessa ma anche più corretta sarebbe quella che implementa la migrazione dei dati dalle tabelle esistenti a quelle nuove.

5.2.2 La classe Adapter

Dopo aver definito una classe Helper è utile definirne una, denominata DbAdapter che fornisce l'interfaccia diretta per manipolare il database. L'Adapter è, nella pratica, un livello di astrazione: fornisce tutta una serie di metodi che, una volta

testati opportunamente, permettono al programmatore di concentrare le proprie energie sugli aspetti importanti dell'applicazione e non sulla gestione del database.

```
package utilities.pack;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;

public class DbAdapter {
    private Context context;
    private SQLiteDatabase database;
    private DatabaseHelper dbHelper;

    //      Database fields
    private static final String VIAGGIO_TABLE      = "viaggio";
    public static final String KEY_VIAGGIOID   = "_idviaggio";
    public static final String KEY_DESTINAZIONE    = "destinazione
       ";
    public static final String KEY_PARTENZA = "partenza";
    public static final String KEY_ARRIVO        = "arrivo";
    public static final String KEY_TRASPORTO ="trasporto";
    public static final String KEY_TABSHOWN="tabshown";

    public static final String KEY_EXTVIAGGIO = "idviaggio";
    public static final String KEY_EXTVALUTA="idvaluta";
    public static final String KEY_EXTCATEGORIA="idcategoria";

    public DbAdapter(Context context) {
        this.context = context;
    }
```

Nella prima parte abbiamo definito per comodità alcune proprietà e alcune costanti che utilizzeremo massicciamente all'interno della classe: la prima costante definisce il nome della tabella mentre le altre definiscono il nome di ogni colonna della stessa. In questo modo in tutto il codice utilizzato per implementare l'Adapter verranno utilizzate queste costanti all'interno delle query, in modo da poterne modificare il nome in caso di necessità senza doverle riscrivere. La parte di codice seguente riguarda la definizione del costruttore: l'unico elemento obbligatorio da configurare al suo interno è il **Context**, che dovremo passare ad ogni istanza di un nuovo Adapter per eseguire delle query. In ambiente Android il Context è un elemento fondamentale ed è rappresentato da una classe astratta, la cui implementazione è fornita dal sistema. Esso permette di accedere alle risorse riservate all'applicazione, come ad esempio le Activity, gli Intent, ecc.

```
    public DbAdapter open() throws SQLException {
        dbHelper = new DatabaseHelper(context);
        database = dbHelper.getWritableDatabase();
        return this;
    }

    public void close() {
        dbHelper.close();
```

```
    }
```

Proseguiamo implementando i metodi open() e close(). Ogni volta che ne avremo la necessità sarà sufficiente richiamare questi metodi per poter interagire con il database, senza dover capirne e conoscerne nei minimi dettagli il funzionamento. Nel metodo open() istanziamo un oggetto di tipo DatabaseHelper che, come abbiamo visto nel Paragrafo precedente, fornisce l'interfaccia di creazione/aggiornamento/-gestione del database. A questo punto non ci rimane che richiamare il metodo getWritetableDatabase() definito in SQLiteOpenHelper, il quale restituisce un oggetto database in modalità lettura/scrittura attivo fino alla chiamata del metodo close(). La prima volta che viene richiamato getWritableDatabase() all'interno del codice, il database viene aperto e vengono automaticamente richiamati i metodi onCreate(), onUpgrade() e se necessario anche il metodo onOpen() (di cui non viene fornita un'implementazione). Il metodo close() invece, non fa altro che richiamare il metodo close() della classe SQLiteOpenHelper, il quale chiude ogni oggetto database aperto. Ogni volta che all'interno del nostro codice abbiamo bisogno di interagire con la base di dati è buona norma chiudere le comunicazioni subito dopo aver terminato, in modo da risparmiare le già limitate risorse dei dispositivi mobili. I dati verranno memorizzati nella cache, per cui non sarà un problema chiudere ed eventualmente riaprire il database ogni qualvolta sarà necessario.

```
    private ContentValues createContentValuesViaggio(String
        destinazione, String partenza, String arrivo, String
        trasporto) {

         ContentValues values = new ContentValues();
         values.put( KEY_DESTINAZIONE, destinazione);
         values.put( KEY_PARTENZA, partenza );
         values.put( KEY_ARRIVO, arrivo);
         values.put(KEY_TRASPORTO, trasporto);
         return values;
    }

    private ContentValues createContentValuesViaggio(String
        tabShown) {

         ContentValues values = new ContentValues();
         values.put(KEY_TABSHOWN, tabShown);
         return values;
    }
```

Prima di trattare l'implementazione vera e propria dei metodi che si occupano di interrogare la base di dati, possiamo implementare un altro metodo per migliorare l'efficienza e la comodità di utilizzo dell'Adapter: createContentValues(). Esso ha un compito molto semplice: memorizzare un insieme di valori che lo strumento ContentResolver potrà poi processare per fornirne l'accesso condiviso da parte di altre applicazioni. Quando avremo la necessità di accedere ai dati di un Content Provider potremo quindi utilizzare l'oggetto ContentResolver nel Context della nostra applicazione e comunicare con il Provider del database. Quest'ultimo una volta ricevuta la richiesta, la gestisce e restituisce i risultati prodotti. Per evitare parti di codice ridondanti abbiamo riportato solo i metodi di creazione dei Content

Values per la tabella viaggio. In particolare nel Content Value riguardante i viaggi abbiamo inserito destinazione, data di partenza, data di ritorno, mezzo di trasporto utilizzato e nell'altro metodo le funzionalità dell'applicazione che l'utente desidera attivare, utile per le impostazioni.

Creazione di un record

```
    //create tables
    public long createViaggio(String destinazione, String partenza
        , String arrivo, String trasporto,String tabShown) {

        ContentValues initialValues = createContentValuesViaggio(
            destinazione, partenza, arrivo,trasporto,tabShown);
        return database.insertOrThrow(VIAGGIO_TABLE, null,
            initialValues);
    }
```

Una volta definito quindi l'insieme di dati che possiamo utilizzare nelle nostre query attraverso il metodo createContentValues() vediamo in dettaglio i metodi per creare, aggiornare e cancellare le tabelle. Il primo metodo createViaggio() richiede 5 parametri, ovviamente gli stessi che abbiamo analizzato nel Paragrafo precedente nella descrizione di createContentValues(), ovvero: destinazione, data di partenza, data di ritorno, mezzo di trasporto e quali tab da visualizzare nell'applicazione. Nel codice di definizione della tabella inoltre, abbiamo dichiarato una chiave primaria auto incrementale (_idviaggio), quindi durante la fase di creazione di un nuovo viaggio verrà automaticamente incrementato questo valore, in modo da distinguere ogni record. In dettaglio il metodo, dopo aver richiamato createContactValuesViaggio(), utilizza l'istanza SQLiteDatabse impostata nel metodo open() per richiamare la funzione insertOrThrow(). Essa permette di inserire un record nel database e come parametri prevede il nome della tabella in cui inserire la riga, un valore che indica come comportarsi nel caso in cui i valori iniziali siano vuoti ed infine i valori da inserire. Il parametro restituito è l'id, ovvero la chiave primaria del record appena creato o il valore -1 in caso di errore.

Modifica di un record

```
    //update tables
    public boolean updateViaggio( long viaggioID, String
        destinazione, String partenza, String arrivo, String
        trasporto) {

        ContentValues updateValues = createContentValuesViaggio(
            destinazione, partenza, arrivo,trasporto);
        return database.update(VIAGGIO_TABLE, updateValues,
            KEY_VIAGGIOID + "=" + viaggioID, null) > 0;

    }

    public boolean updateTabView( long viaggioID, String tabShown)
        {
```

```
        ContentValues updateValues = createContentValuesViaggio(
            tabShown);
        return database.update(VIAGGIO_TABLE, updateValues,
            KEY_VIAGGIOID + "=" + viaggioID, null) > 0;

    }
```

Aggiornare un record è un'operazione molto simile alla creazione. Il metodo updateViaggio() richiede gli stessi 5 parametri del metodo createViaggio(), più un parametro che indica l'identificatore del viaggio da aggiornare, viaggioID. Anche in questo caso la prima operazione è richiamare il metodo createContentValues() e, successivamente la funzione update() della classe SQLiteDatabase sull'istanza database. Questo metodo ci fornisce uno strumento molto potente per aggiornare la nostra base di dati e richiede 3 parametri: il nome della tabella, l'oggetto ContentValues che contiene i valori del viaggio da aggiornare e la dichiarazione della clausola where. Quella utilizzata nel metodo è:

KEY_VIAGGIOID + = + viaggioID

che equivale alla stringa _id = identificatore del viaggio passato come parametro. La clausola where è opzionale perché possiamo anche decidere di passare il parametro speciale null, che provocherebbe l'aggiornamento di tutti i record della tabella. Il metodo restituisce il numero di record modificati, altrimenti 0.

Eliminazione di un record

```
    //delete tables
    public boolean deleteViaggio(long viaggioID) {

        return database.delete(VIAGGIO_TABLE, KEY_VIAGGIOID + "="
            + viaggioID, null) > 0;

    }

    public boolean deleteAllInfoViaggio(long viaggioID){
        boolean b=database.delete(SPESA_TABLE, KEY_EXTVIAGGIO + "=
            " + viaggioID, null) > 0;
        b=database.delete(TAPPA_TABLE, KEY_EXTVIAGGIO + "=" +
            viaggioID, null) > 0;
        b=database.delete(APPUNTO_TABLE, KEY_EXTVIAGGIO + "=" +
            viaggioID, null) > 0;
        b=database.delete(CATEGORIA_TABLE, KEY_EXTVIAGGIO + "=" +
            viaggioID, null) > 0;
        return b;
    }
```

Il metodo deleteContact() è ancora più intuitivo. Accetta un solo parametro, ovvero l'id del viaggio da eliminare, e richiama il metodo delete() della classe SQLiteDatabase: questo non fa altro che eliminare il record con identificativo uguale al valore passato come parametro. Passando null invece verrebbero cancellate tutte le righe della tabella. Il metodo restituisce il numero di record cancellati, oppure il valore 0 se non sono stati trovati record corrispondenti. Per cancellare tutte le

righe della tabella e conoscere il loro numero è sufficiente indicare il valore 1 nella clausola where.

Esempi di query

```
    //query che restituisce tutti i viaggi
    public Cursor fetchAllViaggi() {

        return database.query(VIAGGIO_TABLE, null, null, null,
           null, null, null);
    }

    //query che restituisce il viaggio corrispondente ad un certo
       id
    public Cursor fetchViaggioById(String id){
        String[] whereArgs = { id };
        return database.query(VIAGGIO_TABLE, null,"_idviaggio=?" ,
            whereArgs, null, null, null);
    }
}
```

Gli ultimi due metodi sono due esempi di query per l'interrogazione del database:fetchAllViaggi() e fetchViaggioById(). Il primo metodo permette di recuperare tutti i viaggi presenti nel nostro database, mentre il secondo di ricercare un viaggio con un determinato id. La prima funzione è composta da una sola riga di codice. Utilizzando come negli altri casi l'istanza di SQLiteDatabase database richiamiamo la funzione specifica messa a disposizione dalla piattaforma Android utile a questo scopo: query(). Essa, in base ai parametri inseriti, genera le query necessarie per interrogare il database e recuperare i dati che ci interessano. La piattaforma Android mette a disposizione 3 funzioni query(), che accettano rispettivamente 7, 8 e 9 parametri, in base al tipo di query di cui abbiamo bisogno. Nel nostro programma abbiamo utilizzato la versione che richiede 7 parametri perché sufficiente alle nostre necessità. Analizziamo in dettaglio com'è costituito e quali sono i parametri da specificare:

```
public Cursor query(String table, String[] columns, String
   selection, String[] selectionArgs, String groupBy, String
   having, String orderBy)
```

- table è il nome della tabella da interrogare.
- columns è la lista con i nomi delle colonne i cui valori devono essere inclusi nella risposta. Nel caso del valore null vengono restituite tutte le colonne della tabella.
- selection è la clausola WHERE, per stabilire quali righe della tabella restituire, come trattato nei casi precedenti. Se null vengono restituite tutte le righe della tabella.
- selectionArgs è la lista degli argomenti della clausola WHERE del punto precedente.

- groupBy è la clausola SQL per raggruppare i risultati di una query in base al campo specificato. Se null non si applica alcun raggruppamento.
- having è la clausola SQL di restrizione sui dati risultanti dall'operazione di raggruppamento. Il funzionamento è simile a quello della clausola WHERE ma anziché operare sui campi del database opera sui raggruppamenti. Se null non si applicano restrizioni.
- orderBy è la clausola SQL di ordinamento. Consente di ordinare i risultati della query secondo certi criteri specificabili. Se null non si applica un ordinamento alla lista dei risultati restituiti.

Il metodo query() restituisce un oggetto di tipo Cursor, che fornisce l'accesso in modalità di lettura-scrittura all'insieme di dati restituiti dalla query. Sarà quindi sufficiente ciclare sull'oggetto Cursor per avere accesso a tutti i dati restituiti. A questo punto è facile capire come il primo metodo permetta di selezionare tutte le entry della tabella viaggio, dato il valore null presente in tutti i parametri della query. FetchViaggioById() permette invece di ricavare il viaggio corrispondente a un determinato id passato come parametro. Ciò avviene specificando nei parametri selection e selectionArgs, che corrispondono alla clausola where SQL, l'dentificativo desiderato.

5.2.3 Utilizzo del database nella propria Activity

Diventa relativamente semplice quindi implementare un Helper e un Adapter molto potenti e flessibili capace di semplificare notevolmente l'utilizzo dei database nella nostra App. Una volta coperti i casi basilari di costruzione della base di dati, vediamo come poter utilizzare in un'Activity generica quanto progettato finora.

```
public class ViaggioActivity extends Activity {
    .
    .
    private DbAdapter dbHelper;
    private Cursor cursor;
    List listaViaggi = new ArrayList();
    .
    .
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
     }
    .
    .
    public void onResume(){
        listaViaggi = caricaViaggi();
    }
    .
    .
    private List caricaViaggi(){
        dbHelper = new DbAdapter(this);
```

```
        dbHelper.open();
        cursor = dbHelper.fetchAllViaggi();
        while ( cursor.moveToNext() ) {
            listaViaggi.add(cursor.getString( cursor.
                getColumnIndex(DbAdapter.KEY_DESTINAZIONE) ));
        }
        cursor.close();
        dbHelper.close();
        return listaViaggi;
        }
    .
    .
    }
```

La prima cosa da fare è definire un oggetto di tipo DbAdapter per memorizzare l'istanza dell'Helper, uno di tipo Cursor per scorrere il database e una lista per contenere il nome dei viaggi. I Cursor sono degli oggetti che permettono di avere accesso in lettura/scrittura al database [7]. Nel metodo onCreate() impostiamo solamente tramite il metodo setContentView() il layout contenuto all'interno di main.xml. Nel metodo onResume() invece, che viene eseguito nell'istante successivo, richiamiamo il metodo caricaViaggi() che sarà quello in cui andremo ad interagire in modo mirato col database. Nella prima parte creiamo l'istanza dell'Helper e apriamo la connessione al database tramite il metodo open() definito in precedenza. Ora è possibile richiamare tutte le funzioni che abbiamo implementato per interfacciarci col database, ad esempio il metodo fetchAllViaggi(). Esso restituisce un oggetto di tipo Cursor che punta a tutti i viaggi memorizzati, da cui potremo prelevare i campi che ci interessano. Il metodo moveToNext() permette di spostare il cursore sulla riga successiva, da cui possiamo salvare il valore che ci interessa attraverso il metodo getColumnIndex(). Esso restituisce l'indice della colonna richiesta e, tramite il metodo getString(), possiamo memorizzare all'interno della lista la stringa della colonna richiesta. Potremo poi decidere di utilizzare queste informazioni per riempire una Listview, uno Spinner o altri elementi grafici a nostra scelta. Infine come ultima operazione utilizziamo il metodo close() per liberare le risorse (Cursor e DbAdapter) che abbiamo occupato per la query. L'esempio presentato si occupa dell'estrazione di un solo campo del database, ma dall'oggetto Cursor è possibile ricavare tutti i dati della tabella che si desiderano con un solo ciclo.

Implementare delle classi Helper e Adapter permette quindi di semplificare notevolmente la creazione, la modifica e la cancellazione di dati all'interno di un database, consentendo anche ad un programmatore con poche conoscenze di SQL, un utilizzo sufficiente alla maggior parte delle App.

Capitolo 6

L'applicazione Viaggio

6.1 Come è nata l'idea

Avere buone competenze nel campo della programmazione è sicuramente fondamentale per riuscire a creare delle App, anche basilari. Tuttavia nel market ci sono tantissime applicazioni che svolgono ormai quasi tutte le funzioni di cui l'utente medio ha bisogno: calcolatrici, riproduttori musicali, giochi, traduttori linguistici e molto altro. Crearne quindi una molto simile a quelle già presenti non è una scelta molto conveniente: infatti non riscuoterebbe grande successo perché l'utente tenderebbe a preferire quella che usa da più tempo o quella il cui nome è più diffuso. Una scelta migliore potrebbe essere quella di creare un'App con nuove funzioni o migliorando quelle già presenti. Un esempio banale potrebbe essere una calcolatrice: supponiamo che ne sia già presente una ben diffusa nel market che svolge però solo le operazioni fondamentali. Realizzarne un'altra con solo quelle operazioni non sarebbe conveniente in quanto l'utente non avrebbe motivo di scaricarne una nuova senza che abbia nulla in più e con l'onere di doversi adattare ad una grafica ed un modo di utilizzo probabilmente diverso. Sviluppare invece una calcolatrice scientifica, che permette quindi operazioni più complesse, potrebbe attirare l'attenzione dell'utente spingendolo a scaricare l'applicazione per avere delle funzioni nuove che prima non aveva. La prima operazione che un buono sviluppatore deve fare quindi è pensare a quale App potrebbe essere veramente utile agli utenti, controllare se ne sono già presenti di simili sul market e provarle per poter migliorare, aggiungere o prendere spunto per alcune funzioni.

Spinti dalla voglia di imparare qualcosa di nuovo e da un concorso a premi indetto dalla Samsung che metteva in palio degli smartphone a chi fosse riuscito ad ottenere il maggior numero di download per la propria App, insieme al collega di studi Fabio Forcolin abbiamo deciso di partecipare.
Trovare l'idea illuminante non è stato per nulla semplice e ha richiesto infatti dei giorni di tempo. Si è pensato dapprima qualcosa che potesse essere utile a noi, poi a qualcosa che potesse riguardare i social network e poi a qualcosa riguardo la multimedialità, ma le proposte non erano molto interessanti e piuttosto complesse come nostra prima App. Dopodiché, sfogliando le pagine del sito web Androidiani, ci siamo imbattuti nel post di un utente che chiedeva se fosse possibile realizzare un'App per svolgere la funzione di diario di viaggio, con molti commenti positivi.

Visto che l'occasione sembrava allettante, abbiamo cercato sul market se qualche applicazione del genere fosse già presente, ma non ne abbiamo trovate. La scelta dello scopo dell'App quindi non deve necessariamente arrivare da un'intuizione, ma può anche essere il frutto di un'attenta ricerca e analisi delle richieste degli utenti che non trovano delle applicazioni per soddisfare le loro esigenze.

6.2 Obbiettivo dell'applicazione

L'applicazione che abbiamo deciso di realizzare è rivolta praticamente a qualsiasi utente, infatti chiunque si ritrova prima o poi, per lavoro o in vacanza, ad affrontare un viaggio. Il suo obbiettivo, come in genere quello di molte App, è quello di riunire in una unica applicazione tutte quelle cose che avremmo realizzato con molti oggetti: infatti utilizzando esclusivamente il cellulare è possibile tenere traccia di tutti i viaggi (perlopiù culturali) che effettuiamo così da poterli organizzare, confrontare e rivedere in futuro. Più precisamente questo obiettivo lo abbiamo ottenuto rendendo possibile l'esecuzione, con un'unica applicazione, delle operazioni più comuni nell'organizzazione e svolgimento di un viaggio:

- Organizzazione, suddivisa per giorni, di monumenti e luoghi da visitare indicando anche l'orario e il mezzo di trasporto. Questo evita numerosi blocchi note e il tipo di visualizzazione adottata rende facile ed immediata la consultazione e la modifica.

- Annotazione delle spese che vengono effettuate, dividendole anche per categorie e aggiungendoci eventualmente una piccola nota. É possibile così evitare di avere con sé numerosi scontrini ed inoltre si ha subito sott'occhio il totale speso fino a quel momento.

- Annotazione di appunti, come per esempio indicazioni stradali o piccoli promemoria. Anche questo evita numerosi foglietti disordinati.

- Scatto di foto, che vengono poi organizzate automaticamente per giorno. Certamente le foto di un cellulare non sono paragonabili a quelle di una macchina fotografica, ma per chi non ha particolari esigenze o ha un telefono di ultima generazione in grado di catturare foto di ottima qualità, poterle associarle direttamente al viaggio effettuato può essere un'ottima alternativa alla macchina fotografica.

6.3 Funzionalità dell'applicazione

Nel Paragrafo precedente abbiamo descritto a grandi linee le funzionalità presenti nell'applicazione. Per capire meglio, ci possiamo aiutare con degli screen-shot presi dall'app eseguita su un terminale, motivando anche le varie scelte che abbiamo dovuto affrontare durante la creazione del progetto.

All'apertura dell'applicazione viene visualizzata la schermata di Figura 6.1 che mostra la lista dei viaggi effettuati in modo semplice ed efficace. Infatti, se consideriamo la riga corrispondente ad un viaggio, ad esempio Berlino, possiamo

Figura 6.1: Riepilogo Viaggi

Figura 6.2: Inserimento nuovo viaggio

leggere nell'ordine il simbolo che rappresenta il mezzo di trasporto utilizzato, la destinazione del viaggio e le date di partenza e di ritorno. In una sola riga sono state inserite tutte le informazioni necessarie ad una corretta identificazione del viaggio effettuato ed inoltre la loro disposizione permette anche una lettura delle stesse molto intuitiva e veloce. La ListView che contiene i dati è stata realizzata con il procedimento spiegato alla fine del terzo Capitolo, inserendo all'interno di ogni voce un immagine e tre TextView, una per la destinazione e le altre per date di partenza e ritorno. In questo modo si è resa più accattivante la grafica dell'applicazione, riuscendo poi a mostrare all'utente tutte le informazioni più importanti relative al viaggio in un solo componente. Per aggiungere un nuovo viaggio è sufficiente premere il tasto in alto a destra nell'Action Bar, a forma di cartina geografica, mentre per eliminarli o modificarli è sufficiente tenere premuto uno dei viaggi per far comparire la Contextual Action Bar, la quale mostrerà un cestino per l'eliminazione o una penna per la modifica. La schermata adibita all'aggiunta di un nuovo viaggio, mostrata in Figura 6.2, invece è costituita da una EditText nella quale si può inserire la destinazione. A questa è stata aggiunta la funzione di "AutoComplete", attraverso la quale inserendo le prime lettere di una città, questa con buona probabilità verrà mostrata come suggerimento grazie all'Intellinsense. Dal menù a tendina invece si può scegliere il mezzo che si utilizzerà per effettuare il viaggio: auto, aereo, moto, bici, eccetera. Due calendari permettono di selezionare agevolmente le date di partenza e di arrivo.

Premendo su un viaggio, si possono visualizzare, aggiungere o modificare le informazioni ad esso relative. Nella Figura 6.3 vediamo la gestione delle tappe di viaggio. Tramite il menù a tendina si può selezionare il giorno di cui si vogliono visualizzare le tappe che si è deciso di effettuare. Per la visualizzazione delle informazioni relative alle tappe si è adottato un metodo del tutto simile a quello

Figura 6.3: Tappe di viaggio

precedentemente indicato: da sinistra verso destra troviamo l'ora, la destinazione ed opzionalmente la distanza per raggiungerla. Anche questa ListView è stata realizzata con il metodo all'interno della sezione Grafica Complessa del Capitolo 3. Le tappe sono ovviamente ordinate per orario e premendone una si possono visualizzare ulteriori informazioni eventualmente aggiunte durante la creazione. Grazie alla suddivisione oraria, per ogni giorno di villeggiatura è possibile organizzare i propri impegni e le proprie attività. Per aggiungere una tappa è presente, sempre nell'Action Bar, un pulsante apposito che apre una nuova schermata, nella quale troviamo il campo per inserire il nome della tappa, la distanza (opzionale), l'ora, la data e alcune informazioni aggiuntive.

Molto più elaborata è invece la parte relativa alle spese, come si vede dalla Figura 6.4, raggiungibile con uno swipe sullo schermo. Possono infatti essere annotate tutte le spese effettuate, con la particolarità che possono essere divise per categorie. Le categorie sono totalmente personalizzabili, sia nel nome che nel colore; per facilitarne la lettura si può assegnare ad ogni categoria un colore diverso. Per aumentare la facilità di utilizzo e ottimizzare lo spazio a disposizione, premendo su ogni categoria si possono nascondere/visualizzare le spese ad esse relative. Inoltre a lato di ogni categoria è presente il totale delle spese che la compongono ed in alto si trova invece quello di tutte le spese del viaggio. Grazie all'impiego dell'Expandable ListView è possibile quindi avere subito a portata di mano i saldi parziali di ogni categoria e quello totale. In caso di necessità sarà sempre possibile espandere la categoria di appartenenza e visualizzare l'acquisto nel dettaglio. Per aggiungere una nuova spesa, è sufficiente premere l'icona con il carrello in alto a destra, nell'Action Bar. La schermata che si apre è provvista di un campo per l'inserimento del nome della spesa, dell'importo e della categoria. Le categorie sono completamente personalizzabili e se ne possono aggiungere quante se ne vogliono,

Figura 6.4: Spese

ognuna delle quali può essere distinta dalle altre da un colore diverso. Inoltre il sistema salva automaticamente la data e l'ora di inserimento della spesa, dati che vengono visualizzati sotto di essa.

Ci si può spostare sempre con uno swipe o premendo la parte del nome della scheda visibile nella barra in alto, nella sezione relativa agli appunti. I fogliettini che si vedono in Figura 6.5 rendono bene l'effetto dell'appunto: hanno un titolo e una descrizione e si "impilano" uno sull'altro. Quando se ne vuole visualizzare uno, basta premere su di esso e questo scorrerà verso il basso per mostrare il contenuto. I fogliettini sono un componente non presente nativamente in Android, ma importato da delle librerie Open Source. Android è molto flessibile e ricco sotto questo punto di vista: è infatti possibile, oltre a creare propri componenti grafici personalizzati, importarne di realizzati da altri in modo da migliorare l'esperienza utente. L'aggiunta di un appunto è molto semplice: sempre nell'Action Bar si trova un pulsante a forma di appunto, che apre una schermata nella quale si possono inserire il titolo e il contenuto.

Con un ulteriore swipe ci si può spostare nell'ultima scheda, quella relativa alla PhotoGallery, Figura 6.6. Da qui si può attivare la fotocamera e catturare i momenti di cui si vuole tenere un ricordo. Questi vengono suddivisi e visualizzati per giorno. Per questa funzionalità abbiamo appunto dovuto approfondire il concetto di API e di Intent per interfacciarci con la fotocamera dello smartphone e per salvare i file multimediali all'interno di una directory del FileSystem come illustrato nel Capitolo 4. Abbiamo optato per utilizzare il software nativo dello smartphone, dato che non era necessaria una personalizzazione dell'applicazione Fotocamera e per non stravolgere l'esperienza dell'utente. Inoltre per la memorizzazione delle immagini abbiamo deciso alla prima creazione di un viaggio di generare una directory all'interno di quella principale della memoria SD col nome dell'applicazione "Viaggio" che conterrà tutti

Figura 6.5: Appunti

i file. Per ogni viaggio che viene salvato creiamo poi una sottodirectory specifica con la seguente struttura: nomeviaggio_datapartenza_dataritorno. In questo modo ognuno potrà avere i propri dati in una directory riservata, in particolare le foto, che saranno facilmente consultabili e reperibili anche dall'applicazione Galleria.

Inizialmente per dividere le varie schede avevamo pensato di utilizzare le classiche Tab, presenti già dalle prime versioni di Android, come illustrato in Figura 6.7. Successivamente invece, per consentire un passaggio più facile e rapido abbiamo optato per un componente grafico più moderno detto ViewPager. Questo a differenza delle Tab permette il passaggio da una scheda ad un'altra con uno semplice swipe sullo schermo ed inoltre a livello di codice rende anche più facile la stesura della varie classi attraverso l'uso combinato con i Fragment, componente che abbiamo approfondito nel corso del terzo Capitolo. Infatti le classi che descrivono le diverse funzionalità di un viaggio (tappe,spese,appunti,foto) sono state incluse ciascuna in un diverso Fragment all'interno di un'Activity padre. Per passare da un Fragment all'altro sarà poi possibile come descritto con un semplice swipe sullo schermo, in modo da rendere tutto più immediato.

Tutte le informazioni testuali relative a ogni viaggio, come la destinazione e il periodo di soggiorno, le varie tappe, spese e appunti, sono state memorizzare all'interno di un database SQLite come descritto ampiamente nel corso del 5 Capitolo. Abbiamo deciso di optare per questa soluzione, non tanto per la mole di dati che ci siamo trovati a dover trattare, ma per la loro grande diversità e varietà. Grazie alla strutturazione dei dati in tabelle inoltre, abbiamo potuto sfruttare la comodità delle classi Helper e Adapter per interfacciarci con il database e per la ricerca dei dati al suo interno, senza dover sviluppare soluzioni particolari.

In conclusione anche per sviluppare questo primo progetto per la gestione dei

Figura 6.6: PhotoGallery

Figura 6.7: Esempio di Tab [20]

viaggi è stata richiesta una grande mole di lavoro e di studio. Come abbiamo potuto vedere infatti, sono molti i concetti che stanno alla base della programmazione anche di semplici App. Tuttavia una volta acquisita la padronanza necessaria è possibile dare sfogo alla propria fantasia e avventurarsi nei progetti più disparati, seguendo le proprie idee. Con la speranza che esse un giorno possano avere successo.

Capitolo 7

Conclusioni

La tesina tratta gli aspetti fondamentali della programmazione di App per Android mediante lo studio di una soluzione per la gestione dei viaggi.

In particolare, il primo Capitolo illustra quali sono i componenti logici che costituiscono un'applicazione, prestando particolare attenzione all'Activity e al ciclo di vita che ne governa il funzionamento. Viene poi presentata l'organizzazione di un progetto, concentrando l'attenzione sullo scopo dei vari file presenti e sulla loro collocazione all'interno della gerarchia di directory.

Una parte chiave della tesina è quella successiva riguardante le basi per la programmazione e la grafica delle App. L'obiettivo è quello di capire i diversi ruoli di Java e XML nella costruzione del progetto e fornire gli strumenti necessari per la creazione della prima Activity. Viene affrontato inoltre l'argomento dell'interfaccia, presentando i controlli grafici alla base di gran parte delle applicazioni, come TextView, EditText, Button e ListView per poi finire spiegando com'è possibile creare un Layout efficace ed originale per la propria App.

Il Capitolo successivo riguarda l'interazione con le componenti hardware e software dello smartphone, attraverso il concetto di API. Verranno in primo luogo presentati gli aspetti per la gestione della fotocamera, per poi proseguire illustrando come è possibile memorizzare i file multimediali all'interno della memoria del dispositivo, fornendo anche una piccola descrizione del FileSystem che ne regola il funzionamento.

Il penultimo Capitolo affronta il problema della memorizzazione di dati all'interno dell'applicazione. Vengono in primo luogo esplorate tutte le soluzioni possibili, per poi evidenziare pregi e difetti dell'adottare un DBMS. Nella seconda parte viene invece spiegato come poter integrare nel modo più comodo ed efficace un database, ovvero attraverso delle classi intermedie di Helper e Adapter che permettono la creazione, cancellazione e modifica dei dati senza doversi preoccupare in dettaglio di come è strutturata la base di dati.

L'ultima parte è anch'essa di importanza fondamentale. Viene infatti spiegato come tutti i concetti illustrati nei capitoli precedenti ci siano stati utili per la realizzazione della nostra App per la gestione dei viaggi. Contestualmente vengono chiariti i motivi per cui abbiamo deciso di sviluppare un software di questo tipo ed analizzate tutte le varie funzionalità che essa mette a disposizione. Un'interessante punto di partenza per uno sviluppo futuro di questa parte potrebbe essere

l'integrazione nel progetto di un maggior numero di API. Ad esempio la rilevazione automatica della posizione tramite geolocalizzazione GPS nelle tappe che si effettuano, la loro visualizzazione all'interno di Google Maps, oppure la condivisione coi principali Social Network, per esempio per far sapere dove ci si trova ad amici e conoscenti.

L'obiettivo della tesina, ovvero fornire delle basi per la programmazione di App per la piattaforma Android è stato raggiunto prendendo come esempio e caso di studio l'applicazione per la gestione di viaggi ideata insieme al compagno di studi Fabio Forcolin. La prima parte sviluppata dal collega tratta gli aspetti generali e i motivi per cui abbiamo deciso di optare per questo sistema operativo, per poi approfondirne l'architettura e illustrare le linee guida per la creazione di un App di successo. La seconda parte invece si propone di entrare in modo deciso nel merito della programmazione, presentando tutti i concetti che abbiamo dovuto acquisire durante la realizzazione del progetto e mostrando poi come essi sono confluiti al suo interno.

Bibliografia

[1] Rossi Pietro Alberto. *Android by Example v4.2 JellyBean*. 2013. URL: http://www.sprik.it/guida/Android4_2.pdf.

[2] AndroidGeek. *Activity*. URL: http://www.androidgeek.it/tutorials/programmazione-android/capire-le-activity-ed-il-ciclo-di-vita-di-unapplicazione-android/.

[3] Google. *Activities*. URL: http://developer.android.com/guide/components/activities.html.

[4] Google. *Android Developers*. URL: http://developer.android.com/design/index.html.

[5] Google. *Camera*. 2013. URL: http://developer.android.com/guide/topics/media/camera.html.

[6] Google. *Ciclo di vita di una Activity*. URL: http://developer.android.com/guide/components/activities.html.

[7] Google. *Cursor*. URL: http://developer.android.com/reference/android/database/Cursor.html.

[8] Google. *Fragments*. 2013. URL: http://developer.android.com/guide/components/fragments.html#Design.

[9] Google. *Input Controls*. URL: http://developer.android.com/guide/topics/ui/controls.html.

[10] Google. *Intents and Intent Filters*. URL: http://developer.android.com/guide/components/intents-filters.html.

[11] Satya Komatineni. *Understanding R.java*. URL: http://knowledgefolders.com/akc/display?url=displaynoteimpurl&ownerUserId=satya&reportId=2883.

[12] Marco Lecce. *La gestione dei database in Android*. URL: http://www.html.it/articoli/la-gestione-dei-database-in-android-1/.

[13] Mauro Lecce. *Activity*. URL: http://www.html.it/pag/19499/le-attivit-activity/.

[14] Mauro Lecce. *Guida Android*. 2012. URL: http://www.html.it/guide/guida-android/.

[15] Mauro Lecce. *View*. URL: http://www.html.it/pag/19498/la-visualizzazione-view/.

[16] Androidiani Lunard. *Capire e programmare le Activity*. 2009. URL: http://www.androidiani.com/applicazioni/sviluppo/capire-e-programmare-le-activity-869.

[17] Mr.Webmaster. *Intent ed Intent Filter*. URL: http://www.mrwebmaster.it/mobile/intent-intent-filter_10688.html.

[18] Redazione Io Programmo. *Android Programming*. Punto Informatico Libri, Edizioni Master, 2011.

[19] Smruti Ranjan. *Android File System Structure/Architecture/Layout Details*. 2013. URL: http://techblogon.com/android-file-system-structure-architecture-layout-details/#.

[20] Mina Sami. *Tabbed Application in Android*. URL: http://www.codeproject.com/Articles/107693/Tabbed-Applications-in-Android.

[21] Wikipedia. *Android*. URL: http://it.wikipedia.org/wiki/Android.

[22] Wikipedia. *API*. URL: http://it.wikipedia.org/wiki/Application_programming_interface.

[23] Wikipedia. *Database management system*. URL: http://it.wikipedia.org/wiki/Database_management_system.

[24] Wikipedia. *Java (linguaggio di programmazione)*. URL: http://it.wikipedia.org/wiki/Java_(linguaggio_di_programmazione).

[25] Wikipedia. *SQLite*. URL: http://it.wikipedia.org/wiki/SQLite.

[26] Wikipedia. *Uniform Resource Identifier*. URL: http://it.wikipedia.org/wiki/Uniform_Resource_Identifier.

[27] Wikipedia. *XML*. URL: http://it.wikipedia.org/wiki/XML.