ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

REALIZZAZIONE DI UN'APPLICAZIONE MOBILE ANDROID BASATA SU INTERAZIONI CON INTERFACCE API DI DIVERSI SERVIZI.

Relazione finale in
PROGRAMMAZIONE DEI SISTEMI MOBILE

Relatore
Dott. MIRKO RAVAIOLI

Presentata da
ALEX COLLINI

Seconda Sessione di Laurea
Anno Accademico 2014 – 2015

PAROLE CHIAVE

Android

Application Programming Interfaces

Fansubbing

Android Services

App Developing

If you step back and take a holistic look, I think any reasonable person would say Android is innovating at a pretty fast pace and getting it to users.

- Sundar Pichai

Indice

Introduzione

La grande crescita e l'enorme distribuzione che hanno avuto negli ultimi tempi i moderni devices mobile (smartphones, tablet, dispositivi wearable, etc ...) ha dato l'avvio ad un massiccio sviluppo di applicazioni mobile di qualunque genere, dall'health-care all'AR (*Augmented Reality*, realtà aumentata), dalle applicazioni social alle applicazioni che offrono servizi all'utente.

Alla data di stesura di questo scritto [1], si attesta che siano presenti nei principali sistemi di distribuzione di applicazioni mobile quasi 4 milioni di applicazioni: a farla da padrone, il *Google Play Store* con 1.600.000 applicazioni disponibili sulla piattaforma Google, il secondo posto va al diretto concorrente della casa del robottino verde, la Apple, che, nel suo *Apple Store*, vanta la bellezza di 1.500.000 applicazioni, mentre l'ultimo gradino del podio spetta all'*Amazon Appstore* con solo 400.000 applicazioni; seguono lo store per Windows Phone (340.000) e quello del BlackBerry (130.000). Sebbene lo store di Amazon si limiti a rendere disponibili applicazioni già presenti sugli store Google e Apple, ha alcune applicazioni specificatamente sviluppate per i devices targati Amazon, come i Kindle; da questa stima si può notare come sia davvero attivo il mercato delle applicazioni per devices mobile, il cui utilizzo, in termini di tempo, ormai sta surclassando l'utilizzo che le persone fanno di dispositivi fissi, come il computer. Questo è anche agevolato dal fatto che i moderni dispositivi mobile hanno specifiche hardware che si avvicinano molto alle specifiche dei moderni computer (naturalmente, un dispositivo mobile non avrà mai la stessa capacità di calcolo di un computer, *ndr*), permettendo fluidità e facilità di utilizzo, superando quei limiti che sembravano invalicabili dalle precedenti generazioni di devices.

Nel primo capitolo di questo scritto verrà spiegata per sommi capi la storia e lo sviluppo del sistema operativo di casa Google, Android.

Successivamente, verranno discusse in dettaglio tutte le fasi di sviluppo di un'applicazione Android: *ItaSAMobile*. *ItaSAMobile* nasce dalla partecipazione dell'autore di questo scritto ad una community di fansubbing italiana, per

l'appunto *Italiansubs.net*, e dalla necessità dello stesso e di molti utenti della community di avere un'applicazione che renda possibile la consultazione del portale e del forum su dispositivi mobile in maniera intuitiva e *user-friendly*.

I primi capitoli dello scritto andranno a trattare le prime fasi dello sviluppo e della progettazione dell'applicazione, come l'introduzione dei concetti di API utilizzate dall'applicazione. Le **API** (**A**pplication **P**rogramming **I**nterface) sono entità che forniscono al programmatore degli strumenti standardizzati per svolgere determinate operazioni all'interno di un certo programma; sono molto diffuse nell'ambiente informatico e fanno da "interprete" tra due enti che "parlano" linguaggi differenti. In questo caso, però, si trattano di API di web services, che facilitano l'interfacciarsi di un'applicazione ad un server web per ottenere delle informazioni: *ItaSAMobile* sfrutta le API messe a disposizione sia dal portale *Italiansubs.net*, sia da un gestore di forum per dispositivi mobili, *Tapatalk*. Tutte le comunicazioni che concernono le API sono gestite tramite client HTTP, per le API di *Italiansubs.net*, e tramite client XML-RPC, per le API di *Tapatalk*.

Inoltre, verranno descritti gli aspetti elementari che compongono le applicazioni Android, come le *Activity* e il *Manifest*. Nella stessa sezione saranno oggetto di discussione anche alcune delle classi chiave dell'applicazione, introducendo anche l'argomento della classe di libreria *AsyncTask*, insieme ad una piccola infarinatura sulle varie procedure (gestione dei cookie) e chiamate online (gestione client HTTP) che verranno effettuate dall'applicazione.

Successivamente verranno descritte le fasi della progettazione iniziale dell'applicazione, come la divisione in package e la progettazione del model dell'applicazione, e anche delle scelte grafiche. Le linee guida seguite dall'applicazione, per quanto riguarda la grafica, fanno parte di uno stile grafico introdotto da Google con le nuove release di Android, il *Material design*.

Nel capitolo 5, verranno descritte in dettaglio le implementazioni delle classi chiave dell'applicazione, sia per quanto riguarda le operazioni base dell'applicazione (login), sia per quanto riguarda il meccanismo di parsing delle risposte delle API.

Successivamente, verranno riassunte brevemente le tecnologie utilizzate (IDE, macchine fisiche e devices mobile), le librerie esterne e i linguaggi di programmazione utilizzati.

Infine verranno tratte delle conclusioni sulla natura didattica dello sviluppo

dell'applicazione in questione e dei suoi sviluppi futuri.

Capitolo 1

Stato dell'arte

1.1 Introduzione e cenni storici

Android è un sistema operativo mobile basato sul kernel Linux e attualmente sviluppato e mantenuto da Google. Viene principalmente utilizzato su dispositivi touch, quali tablets, smartphones e, recentemente, anche dispositivi weareable.

Attualmente, Android è il sistema operativo mobile più diffuso al mondo (Android detiene il 78% del mercato mobile); questa diffusione viene facilitata dalla sua natura "*free and open-source*", dando quindi la possibilità ai developer più esperti non solo di sviluppare proprie versioni del sistema nativo Android, le cosidette *custom ROM*, ma anche di sviluppare *custom kernels* in grado di migliorare o addirittura aggiungere features che prima non erano disponibili nel sistema nativo Android.

	About OS				
Feature	**iOS**	**Android**	**Firefox OS**	**Windows Phone**	**BlackBerry 10**
Company	Apple Inc.	Open Handset Alliance/Google	Mozilla Foundation	Microsoft	BlackBerry Ltd.
Market share[1]	18.3%	78.0%	N/A	2.7%	0.3%
Current version	8.4.1	5.1.1	2.1.0	Windows Phone 8.1 Update (8.10.14219.341)[2]	10.3.2.670

Figura 1.1: Divisione del mercato dei principali SO mobile

La storia di Android ha inizio nel 2003, quando Andy Rubin, Rich Miner, Nick Sears e Chris White crearono a Palo Alto, California, la *Android, Inc.* Rubin aveva intenzione di sviluppare "*smarter mobile devices that are more aware of its owner's location and preferences.*", dispositivi mobile intelligenti più consapevoli delle preferenze e della posizione dei loro proprietari; per que-

sto decisero di iniziare a sviluppare sistemi operativi avanzati per fotocamere digitali. Ma all'epoca, quel campo non era così sviluppato come quello dei telefonini; decisero così di cambiare campo di competenza, decidendo di creare un sistema operativo mobile che potesse competere con i pilastri del mobile dell'epoca, Symbian e Windows Mobile. La *Android, Inc.* ha lavorato sempre in segreto, rilasciando al pubblico solo l'annuncio che stesse lavorando ad un software per telefonini. Questo fin quando la Google, nel 2005, decide di comprare la *Android, Inc.* per circa 50 milioni di dollari. Tutto il team iniziale della *Android, Inc.* venne assunto dalla Google e venne avviato lo sviluppo di quello che poi diventerà l'attuale sistema operativo Android, un sistema operativo mobile basato su kernel Linux, scritto in C (parte kernel), C++ e Java (parte UI).

Dal 2008, Android ha ricevuto numerosi aggiornamenti, che hanno migliorato il sistema operativo, andando ad aggiungere features e a fixare numerosi bug presenti nelle precedenti release. Ogni major update prende il nome da famosissimi dolci americani, seguendo l'ordine alfabetico: si parte con la versione 1.5, chiamata "*Cupcake*", fino ad arrivare alle più moderne e conosciute (dalla 4.0 fino alla 4.0.4 "*Ice Cream Sandwich*", dalla 4.1 alla 4.3.1 "*Jelly Bean*", dalla 4.4 alla 4.4.4, comprese le versioni wearable, "*KitKat*")

Figura 1.2: Piccolo giocattolo raffigurante il logo di Android

L'attuale versione di Android è la 5.1.1, denominata *Lollipop*. Google ha recentemente annunciato l'uscita di un nuovo major update, la versione 6.x, denominata *Marshmallow*. [2][3]

1.2 Android releases e milestones

Le prime versioni di Android erano orientate al semplice test su emulatori e non su dispositivi fisici; si tratta della versione Android 0.5, Milestone 3. Questi emulatori avevano una tastiera QWERTY ed un pad fisico a 4 direzioni, simile al layout dei *Blackberry*.

Figura 1.3: Android 0.5 su emulatore

Non era possibile personalizzare la home screen con dei widgets, anche se erano presenti animazioni primitive. Non esisteva nessun tipo di pannello per le notifiche: quest'ultime venivano mostrate sulla status bar come icone e l'unico modo per accedervi era premere "*Su*" sul pad mentre si era sulla home screen. L'intero sistema operativo era molto primitivo, con ancora poche funzionalità.

Con l'avvento delle versioni 1.0 e 1.5 (*Cupcake*), ottenne una veste grafica molto più familiare e curata rispetto alla 0.5. Google ha già acquisito la *Android, Inc.*, così decide inserire proprie applicazioni, quali Gmail, YouTube e il predecessore dell'attuale Google Play Store, l'Android Market. Tutte le applicazioni avevano funzioni elementari e GUI minimale.

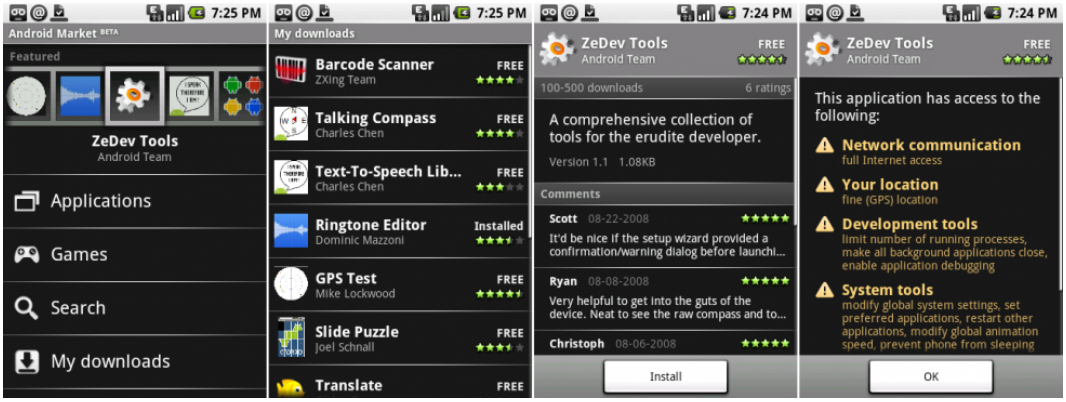

Figura 1.4: Il predecessore del Play Store, Android Market

Con la versione 1.5, viene introdotta la tastiera virtuale, dando la possibilità allo sviluppo di dispositivi mobili senza tastiera fisica, ma con ancora il vincolo di tasti fisici. Essendo Google una società americana e non avendo ancora Android una diffusione a scala mondiale, c'erano problemi di compatibilità con le diverse compagnie telefoniche non-americane; Google decise quindi di aggiungere il supporto CDMA alla versione 1.6 (*Donut*), aumentando quindi le possibilità di questo pionieristico sistema operativo mobile di espandersi velocemente all'estero.

Con *Eclair*, versione 2.0, viene migliorata leggermente la UI e Google decide di aggiungere una nuova feature al suo sistema operativo: la navigazione GPS. Questa feature ha destabilizzato il mercato dei navigatori GPS classici, visto che ore gli utenti Android avevano la possibilità di usare un navigatore GPS sul proprio dispositivo mobile, tramite l'applicazione Google *Google Maps*. Poco dopo l'uscita di Eclair, Google mette sul mercato il suo primo smartphone: il Nexus One. Quest'ultimo fu un passo importante per l'azienda, entrando così di forza all'interno del mercato degli smartphone. Il Nexus One veniva venduto online, sbloccato, senza contratto, a 529.99$. Questa decisione, però, non ebbe molto successo: lo shopping online, all'epoca, non era molto diffuso.

La versione 2.2, denominata *Froyo*, era migliorata in velocità grazie all'aggiunta della **JIT compilation**, *compilazione just-in-time*: questo tipo di compilazione permette la conversione automatica a runtime del bytecode Java in codice nativo, aumentando quindi le performance del sistema operativo. Un'altra novità in *Froyo* è stata la ri-progettazione della dock in basso, che prima occupava l'intera parte inferiore della home screen.

Figura 1.5: Confronto home screen Froyo-Eclair

Dopo *Froyo* troviamo *Gingerbread*, versione 2.3, con un rinnovamento totale della UI: ogni schermata del sistema operativo ottiene una nuova veste grafica. Viene lanciato anche il Nexus S, primo smartphone Nexus creato dalla Samsung.

Con l'arrivo di *Honeycomb*, versione 3.0, abbiamo l'arrivo dei tablets Android. Questa nuova versione Android cambia completamente la UI, cercando di arrivare ai livelli della Apple, principale concorrente della Google in campo mobile. Abbiamo anche l'introduzione dei tasti touch, presenti sotto la dock; quest'ultimi vanno a rimpiazzare i tasti fisici dei vari dispositivi, iniziando a far intravvedere i più moderni design dei devices. Per i tablet, è stata introdotta la *system bar*, simile alla barra delle applicazioni presente sui moderni sistemi desktop.

A partire dal rilascio di *Jelly Bean*, versione 4.1, successore di *Ice Cream Sandwich*, versione 4.0, abbiamo la volontà di Google di riunire tutte le sue applicazioni sotto l'effigie di Google Play: si vengono così a creare, nel 2012, le moderne applicazioni di Google Play Music, Google Play Store (che va a sostituire completamente l'Android Market) e molte altre. Inoltre, abbiamo l'introduzione di Google Now, un'assistente creato da Google in grado di riuscire a capire, dalle attività del cliente su vari dispositivi Google, le esigenze del consumatore, suggerendogli posti nelle vicinanze, articoli on-line da leggere, meteo e tutta una serie di informazioni utili alla vita quotidiana. *Jelly Bean* è inoltre portatore di novità a livello di sistema operativo. Infatti vengono introdotti i famosissimi *Google Play Services*, che constano in diverse subroutine

ognuna in grado di gestire una diversa parte del sistema operativo, dando così innumerevoli features in più al device.

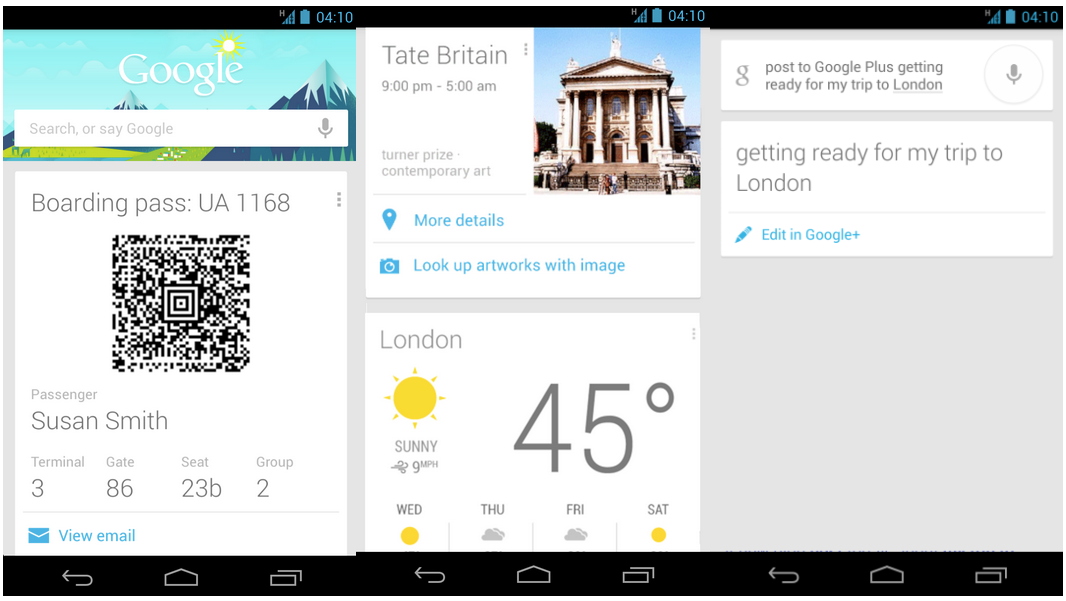

Figura 1.6: Alcune schede esempio di Google Now

Le ultime due major release di Google sono *KitKat*, versione 4.4, e *Lollipop*, versione 5.0. Con KitKat abbiamo l'introduzione di un nuovo tipo di compilazione, per quanto riguarda il codice sorgente delle app: si chiama **ART**, *Android runtime*; con questo tipo di compilazione, il codice viene compilato interamente durante l'installazione dell'applicazione sul device, si parla di compilazione **AOT**, *ahead-of-time*. [4] In KitKat, l'ART poteva essere abilitato tramite il *Menù Sviluppatore*, dove l'utente poteva scegliere la classica compilazione *JIT* tramite Dalvik VM oppure la nuovissima **AOT** tramite ART; invece con *Lollipop*, la Dalvik VM è stata soppiantata dall'ART.

La versione 6.0 è stata mostrata in anteprima al *Google I/O 2015*. Al momento della conferenza, questa versione è stata denominata come *M*, ma , a seguito di un recente annuncio [3], Google ha infine scelto il vero nome della release, *Marshmallow*.

Capitolo 2

Caso di studio

2.1 Caso di studio

L'applicazione di cui andremo a disquisire, *ItaSAMobile*, rappresenta una versione mobile di un forum di fansubbing italiano, *Italiansubs.net*. L'applicazione nasce dall'esigenza di poter accedere al portale da dispositivi mobile in maniera veloce ed efficiente, senza dover aprire il browser e aspettare lunghi caricamenti, data la mole di dati del portale.

L'applicazione è stata inizialmente sviluppata per uso personale, ma dopo aver reclutato una stretta cerchia di betatesters, viene rilasciarla sul Google Play Store, in modo che tutti gli utenti del suddetto forum possano usufruirne.

L'app fornisce tutte le funzionalità del portale versione desktop, quali:

- Possibilità di visualizzare gli ultimi sottotitoli rilasciati in home.
- Possibilità di visualizzare tutte le serie tradotte e attualmente in traduzione e leggerne la scheda informativa.
- Cercare specifici sottotitoli (per qualità video, per stagione, per episodio, etc. . .).
- Visualizzare in maniera ottimizzata il forum, con completa interazione con topics e posts.
- Visualizzare i sottotitoli delle proprie serie preferite.

L'utente, una volta installata l'applicazione, dovrà loggarsi con le credenziali del portale. Se non fosse registrato ad *Italiansubs.net*, in fondo alla schermata di login trova un link che rimanda alla pagina di registrazione del sito. Una

volta loggato, l'utente si troverà davanti la prima schermata dell'applicazione, ovvero quella relativa agli ultimi sottotitoli rilasciati. Finito il caricamento delle news, l'utente è libero di selezionare le altre schede, semplicemente con lo swipe verso destra o verso sinistra.

La seconda scheda dà la possibilità all'utente di visualizzare l'intera lista di serie TV e film che sono stati tradotti o ancora in traduzione, potendo inoltre visualizzare la loro scheda informativa, contenente tutte le informazioni del caso (trama, cast, messa in onda, canale, data ultima puntata, data prossima puntata, stato corrente, etc...).

Come terza scheda abbiamo la ricerca dei sottotitoli all'interno del database dei *Italiansubs.net*: la ricerca ha un unico campo obbligatorio, il titolo della serie TV in questione; fornito quello, è possibile poi raffinare la ricerca dei sottotitoli, potendo specificare la versione video desiderata, la puntata, la stagione, o addirittura la possibilità di scaricare l'archivio completo di una stagione, ove presente.

Arriviamo poi alla scheda del forum; questa scheda offre la possibilità a tutti gli utenti registrati di visualizzare il forum in versione mobile, senza l'ausilio di browser o di applicazioni esterne. L'utente può visualizzare tutte le board del forum, creare topics, rispondere a topics già esistenti, quotare posts, etc...

L'ultima scheda permette la visualizzazione del proprio MyItaSA, contenente tutte le serie seguite sul portale.

Dal menù è possibile effettuare il logout e accedere alle impostazioni dell'applicazione, dove è possibile cambiare la cartella di download dei sottotitoli (impostata di default su *ItaSAMobileSubs*), visualizzare l'intero changelog dell'applicazione, visitare la pagina ufficiale su Facebook e visualizzare l'attuale versione dell'applicazione.

Capitolo 3

Analisi e modellazione

3.1 Analisi del problema

L'applicazione si basa principalmente sulla comunicazione HTTP verso server API remoti. Usando due servizi API completamente differenti, *ItaSAMobile* ha bisogno di ottenere informazioni da risposte HTTP diverse. Per questo si avvale di diversi **parsers**, in grado di estrapolare informazioni dalle risposte HTTP provenienti dai server. Scendiamo più nel dettaglio.

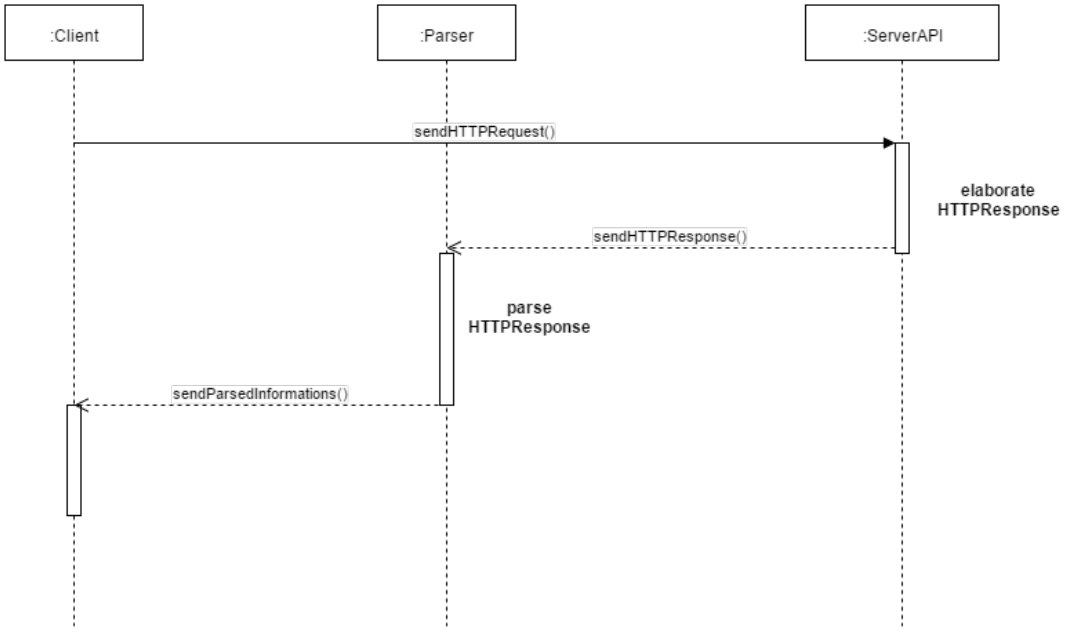

Figura 3.1: Diagramma di sequenza raffigurante il ciclo di chiamate ai server API

3.1.1 API REST di Italiansubs.net

Le API di *Italiansubs.net* sono di tipo **REST**, *REpresentational State Transfer*: questo tipo di API rispondono ai normalissimi comandi HTTP, come GET, POST, etc... Le API di *Italiansubs.net* forniscono risposte in XML ai client che effettuano richieste HTTP specifiche.

Lo schema della risposta delle API in questione rispetta quello classico di un qualunque documento XML: all'inizio del documento troviamo l'elemento *root*, che apre e chiude il documento; subito dopo questo elemento, troviamo poi l'elemento *data*, contenente tutti i dati richiesti tramite *HTTPRequest*. Un documento XML può essere considerato un *albero* a tutti gli effetti, potendolo dividere in *elemento padre* ed *elemento figlio*, o più semplicemente a livelli. Ed è proprio questa visione a livelli che ha permesso sviluppo di un parser molto efficace, che prende come parametri la risposta nativa in XML ricevuta dal server e il livello in cui si trovano le informazioni necessarie, visto che, in base al tipo di chiamata e/o di informazione cercata, si possono trovare su diversi livelli.


```
▼<root>
  ▼<data>
    ▼<news>
      ▼<news>
          <id>23282</id>
          <show_id>3419</show_id>
          <show_name>Baby Daddy</show_name>
          <special/>
          <category>16</category>
        ▼<image>
            http://img.italiansubs.net/news2/data/Baby%20Daddy/Stagione%204/Baby.Daddy.4x17.png
          </image>
          <image_by>TheDany</image_by>
          <team>itasa</team>
          <submit_date>2015-08-19T15:17:19+02:00</submit_date>
          <submitted_by>feloreena</submitted_by>
        ▼<thumb>
            http://img.italiansubs.net/news2/data/Baby%20Daddy/Stagione%204/Baby.Daddy.4x17_thumb.png
          </thumb>
        ▼<episodes>
            <episode>4x17</episode>
          </episodes>
        </news>
      ▼<news>
          <id>23281</id>
          <show_id>5789</show_id>
          <show_name>The Whispers</show_name>
```


Figura 3.2: Estratto di una chiamata al server API di *Italiansubs.net*

Ogni chiamata alle API segue la classica sintassi della *GET request* HTTP; la risorsa desiderata può essere ottenuta tramite l'URI corrispondente, caratteristica propria delle API **REST**. Ad esempio: *http://api.italiansubs.net/api/rest/news?* richiede al server API tutti gli ultimi sottotitoli rilasciati (Figura 3.2); se, invece, si volesse cercare qualcosa di specifico, è possibile aggiungere parametri

di ricerca tramite l'operatore "&", seguito da un identificatore specifico e dalla query di ricerca. È naturalmente possibile concatenare più identificatori per restringere ancora di più la ricerca.

Ad ogni applicazione che si interfaccia a queste API viene assegnata un'univoca *API key*, che deve essere utilizzata ad ogni chiamata per autenticarla.

3.1.2 API di Tapatalk

Le API di *Tapatalk* hanno un tipo di interfacciamento diverso rispetto alle API di *Italiansubs.net*.

Partiamo dicendo che le *HTTPRequest* da parte del client non devono essere di tipo GET, ma **POST**: le API di *Tapatalk* non si basano sull'architettura REST, tutte le richieste devono essere effettuate ad un'unica pagina *.php*, cambiando tipo di funzione ad ogni chiamata ed inserendo in *piggyback* i vari parametri necessari per la funzione in questione. Le risposte a queste chiamate variano in base al tipo della funzione chiamata.

Nel dettaglio:

- Le API di *Tapatalk* sono basate sul linguaggio XML-RPC, per questo il client HTTP di base di Android si rivela inefficace ed incompatibile con i requisiti delle API. Esempio di *XML-RPC Request*:

  $POST/connect/mobiquo/mobiquo.phpHTTP/1.0$
  $User-Agent : Mozilla/5.0(WindowsNT6.1; WOW64; rv : 14.0)$
  $Gecko/20100101Firefox/14.0.1$
  $Host : sightfirstcommunity.in : 80$
  $Mobiquo_id : 2$
  $Mobiquo-Id : 4$
  $Accept-Charset : UTF-8, ISO-8859-1, US-ASCII$
  $Cookie : mybb[lastvisit] = 1404279459; mybb[lastactive] = 1404279459;$
  $sid = aeba40744ca95796aa5cc4a064c1d260f$
  $Content-Type : text/xml$
  $Debug : 0$
  $Content-Length : 156$
  $<?xmlversion = "1.0"?>$
  $<methodCall>$
  $<methodName>get\_raw\_post</methodName>$
  $<params><param><value><string>87212</string></value>$
  $</param></params>$

$</methodCall>$

Analizzando il codice, notiamo un elemento chiamato "*methodName*" ed è proprio quell'elemento che va ad identificare e distinguere il tipo di informazione richiesta dalla chiamata del client. Sotto a quest'ultimo elemento troviamo i "*params*", parametri richiesti in ingresso dal metodo da chiamare.

Input Parameters:

Name	Type	Required?	Description	Level
forum_id	String	yes		3
start_num	Int		For pagination. If start_num = 0 & last_num = 9, it returns first 10 topics from the forum, sorted by date (last-in-first-out). If both are not presented, return first 20 topics. if start_num = 0 and last_num = 0, return the first topic only. If last_num - start_num > 50, returns only first 50 topics starting from start_num	3
last_num	Int		See above.	3
mode	String		if mode = "TOP", returns sticky topics, if mode = "ANN", returns "Announcement" topics only, otherwise returns standard topics.	3

Output Parameters:

Name	Type	Required?	Description	Level
total_topic_num	Int	yes	Total number of topics in this forum. If this forum has no topic, it returns 0, and the "topics" key returns return null	3
forum_id	String	yes		3
forum_name	byte[]	yes		3
can_post	Boolean		return false if user cannot create new topic in this forum	3
unread_sticky_count	Int		Add a "unread_sticky_count" and "unread_announce_count" integer value in get_topic first level Hash-table to indicate users there are unread stickies topics and announcement in this sub-forum. This allow the app to show a unread badge when entering a sub-forum.	4
unread_announce_count	Int		Add a "unread_sticky_count" and "unread_announce_count" integer value in get_topic first level Hash-table to indicate users there are unread stickies topics and announcement in this sub-forum. This allow the app to show a unread badge when entering a sub-forum.	4

Figura 3.3: Estratto dalla documentazione delle API di Tapatalk

- Le risposte alle varie chiamate alle funzioni messe a disposizione dalle

API di *Tapatalk* variano di formato in base alla funzione invocata. Ad esempio, la funzione "get_forum" ritorna un oggetto di tipo *Object[]*, al cui interno troviamo la root del forum in questione, insieme ad altri elementi utili alla navigazione del forum; invece il resto delle funzioni hanno come valore di ritorno una *Map<String, Object>*. Il vero problema di questo tipo di risposta è che i veri dati richiesti dal client attraverso quella chiamata si trovano in un'altra *Map<String, Object>*, figlia della *Map<String, Object>* ricevuta come risposta da parte delle API, diventando a tutti gli effetti un oggetto di tipo *Map.Entry<String, Object>*. Ed è proprio su quella *Map* che deve andare ad interagire il parser: la struttura gerarchica è simile a quella delle API di *Italiansubs.net* ma il tipo di struttura dati da elaborare è completamente diversa; da qui deriva la necessità di sviluppare un parser completamente diverso da quello per le API di *Italiansubs.net*.

3.2 Possibili soluzioni al problema

3.2.1 XML Parser per le API di Italiansubs.net

Per riuscire ad estrapolare le informazioni necessarie dalla risposta XML da parte delle API di *Italiansubs.net*, è possibile implementare un semplice parser, avente come parametro in ingresso il documento XML e il livello dell'albero in cui si trovano le informazioni. Per gestire al meglio la manipolazione dei documenti XML, è possibile utilizzare una libreria esterna, **JDOM**, in grado di facilitare le operazioni con qualunque documento XML.

3.2.2 ResultMapper per le API di Tapatalk

Per quanto riguarda le API di *Tapatalk*, molte delle informazioni necessarie sono, come detto prima, all'interno di una *Map<String, Object>* figlia di una *Map* dello stesso tipo. L'approccio più efficace è quello di recuperare la lista figlia e, tramite iteratore, leggere ogni coppia *chiave-valore*, in modo poi da inserirli in una lista di oggetti, di cui verranno specificati i dettagli nella prossima sezione.

Per l'implementazione di entrambi i parser, si rimanda la lettura al capitolo 5 di questo scritto.

3.3 Elementi di base del sistema operativo Android

In questa sezione si andrà a parlare sommariamente di alcuni elementi base del sistema operativo Android che sono di vitale importanza per tutte le applicazione sviluppate per questo sistema operativo: le *Activity* e il *Manifest*. Per le *Activity* verrà data una spiegazione per sommi capi, per quanto riguarda il *Manifest* invece si scenderà un po' più nel dettaglio, spiegando brevemente anche i permessi di Android.

3.3.1 Activity

Le *Activity* sono oggetti della architettura Android molto particolari; infatti, l'utente interagisce con quello che le *Activity* mettono a disposizione graficamente allo stesso, si possono considerare come il *front-end* di qualsiasi applicazione Android. Ogni classe Java può essere convertita ad *Activity* semplicemente facendo estendere alla classe la classe *ActionBarActivity*, come in questo codice d'esempio:

```
public class className extends ActionBarActivity
```

Così facendo, si ha la necessità di implementare il metodo *onCreate()*, che andrà ad effettuare il codice al suo interno non appena l'*Activity* verrà avviata. È possibile controllare anche tutto il ciclo di vita dell*Activity* implementando anche altri metodi, come *onDestroy*, *onPause*, *onResume*: questi metodi vengono usati quando qualche *Activity* utilizza risorse in maniera esclusiva, come la fotocamera, e che quindi deve liberare non appena l'*Activity* cambia stato.

Ogni *Activity*, per essere lanciata dall'applicazione, deve essere inserita all'interno del *Manifest*: l'*Activity* che viene lanciata premendo l'icona dell'applicazione nel launcher ha un*intent filter*(una specie di filtro che cattura solo determinate azioni inviate in broadcast sia dal sistema operativo che da diverse applicazioni) particolare ed è il seguente:

```
<intent-filter>
        <action android:name="android.intent.action.MAIN" />

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

Tutte le *Activity* che sono inserite nel *Manifest* possono essere avviate tramite l'invocazione del metodo *startActivity(intent)*, dove *intent* corrisponde ad

un *Intent* specifico che lancia una determinata *Activity*. È possibile usare anche il metodo*startActivityForResult(intent)*, così facendo, una volta che l'*Activity* verrà terminata tramite la chiamata del metodo *finish()*, l'*Activity* che aveva avviato l'*Activity* appena terminata andrà a leggere il codice di *result* e agirà di conseguenza.

Ogni *Activity* ha il proprio layout, che può essere gestito in maniera completamente autonoma dallo sviluppatore, scritto in un file XML, al cui interno possiamo trovare tutti gli elementi grafici che andranno a comporre la schermata dell'*Activity*. I layout Android sono altamente personalizzabili, con molti elementi che possono andare a comporre la view finale dell'*Activity*, dando anche la possibilità allo sviluppatore di scegliere il metodo di composizione del layout stesso: se si vuole un risultato a mo' di tabella, si può utilizzare il *LinearLayout*, scegliendo poi l'orientamento degli elementi, se si vuole un layout più "permissivo" è possibile usare il *RelativeLayout*.

3.3.2 AndroidManifest

L'*AndroidManifest*, o semplicemente *Manifest*, è un documento XML contenente le principali informazioni dell'applicazione, come il nome, la versione, il codice della versione.

Il *Manifest* contiene anche la lista di tutte quelle classi che devono essere considerate come *Activity*; la definizione di un'*Activity* all'interno del *Manifest* viene effettuata in questo modo:

```
<activity
    android:name="activityName"
    android:theme="activityStyle"
    android:screenOrientation="portrait">
</activity>
```

Spieghiamo brevemente questi campi (è possibile aggiungerne degli altri, ma questi sono quelli fondamentali):

- *android:name*: il nome dell'*Activity* in questione.
- *android:theme*: il tema grafico dell'*Activity*; in assenza di questo campo, l'*Activity* erediterà il tema base dell'applicazione.
- *android:screenOrientation*: specifica l'orientamento dello schermo del dispositivo al momento della visualizzazione dell'*Activity*, può essere *por-*

trait, quindi visualizzazione in verticale, o *landscape*, cioè con visualizzazione in orizzontale.

Un'altra parte importante del *Manifest* è la dichiarazione dei premessi garantiti all'applicazione. Android è un sistema operativo molto restrittivo per quanto riguarda l'ambiente della sicurezza del device: infatti, proprio tramite questi permessi, il sistema operativo sa a quali risorse ed aree del device l'applicazione può accedere (connessione internet, messaggi, rubrica, lettura/-scrittura su disco, etc...); in questo modo l'utente si assicura che l'applicazione non possa interagire con aree che lui stesso non vuole condividere con l'applicazione. Esempio di permesso nel *Manifest*:

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

Con questo permesso, l'applicazione può utilizzare la connessione internet del device e conoscerne lo stato (connessa o disconnessa). I permessi che Android mette a disposizione dello sviluppatore riguardano tutte le aree del device.

3.4 AsyncTask, cuore della comunicazione online dell'applicazione

Dalla versione 3.0 di Android, ogni volta che qualsiasi operazione di rete viene effettuata sul thread principale di un'applicazione, viene lanciata un'eccezione del tipo *android.os.NetworkOnMainThreadException*; questa restrizione viene introdotta da Google per aumentare la sicurezza del proprio sistema operativo.

Per facilitare l'implementazione di operazioni su più thread, in questo caso specifico quelle di rete, viene introdotta la classe *AsyncTask*. Questa classe permette, con i suoi metodi, di effettuare operazioni di rete in maniera facile ed efficiente. Vediamo il suo schema classico:

```
@Override
 protected void onPreExecute() {
     super.onPreExecute();
 }

 @Override
 protected String doInBackground(String... params) {
     super.doInBackground();
     return null;
```

```
    }

    @Override
    protected void onPostExecute(String result) {
        super.onPostExecute();
    }
```

Come possiamo vedere da questo codice d'esempio, la classe *AsyncTask* offre allo sviluppatore la possibilità di effettuare operazioni prima (*onPreExecute()*) e dopo (*onPostExecute()*) l'avvio delle operazioni in un altro thread (*doInBackground()*). Le operazioni effettuate nei metodi *onPreExecute()* e *onPostExecute()* vengono eseguite al livello del thread principale, quindi è buona norma effettuare le ultime interazioni con la parte UI nel metodo *onPreExecute()*, per poi completarle, magari integrando i dati ricevuti dal metodo *doInBackground()*, nel metodo *onPostExecute()*.

Ogni classe che estende *AsyncTask* deve implementare necessariamente il metodo *doInBackground()*, in modo da rendere possibile l'avvio delle operazioni in un secondo thread, chiamando il metodo *.execute()* della classe che estende *AsyncTask*.

```
public class className extends AsyncTask<String, String, String>
```

Questa è l'intestazione di una classe che estende *AsyncTsk*. Vediamo come la classe *AsyncTask* richieda tre tipi di dato come parametro, in ordine:

- Tipo di dato da inserire come parametro al momento della chiamata del metodo *.execute()*
- Tipo di dato da inserire come parametro per il metodo *onProgressUpdate()*, implementabile se si ha a che fare con una barra che indica in tempo reale il progresso delle operazioni in background
- Tipo di dato da ricevere come risultato dal metodo *doInBackground()*, in modo da poterlo facilmente riutilizzare nel metodo *onPostExecute()*.

Le sottosezioni seguenti andranno a spiegare diverse funzioni chiave dell'applicazione, tutte da implementare tramite *AsyncTask*.

3.4.1 Comunicazione HTTPS

La comunicazione con il server API di *Italiansubs.net* è gestita da un client HTTPS, per garantire maggiore sicurezza ed affidabilità. Viene anche gestito il controllo della *Certificate chain* del protocollo SSL.

3.4.2 Persistenza dei cookie

Trattandosi di un portale che necessita di una procedura di registrazione/login, è necessario recuperare e memorizzare i cookies identificativi della propria sessione su *Italiansubs.net*. Android mette a disposizione il *CookieManager*, che recupera e gestisce i cookies ricevuti al momento del login. Questi cookies verranno poi riutilizzati per riuscire a scaricare i vari sottotitoli dall'applicazione, visto che la sezione download del portale è raggiungibile ed usufruibile solo dagli utenti registrati e correttamente loggati. Di seguito, verrà descritto dettagliatamente le operazioni effettuate dall'applicazione durante le procedure di login e di logout.

3.4.3 Login

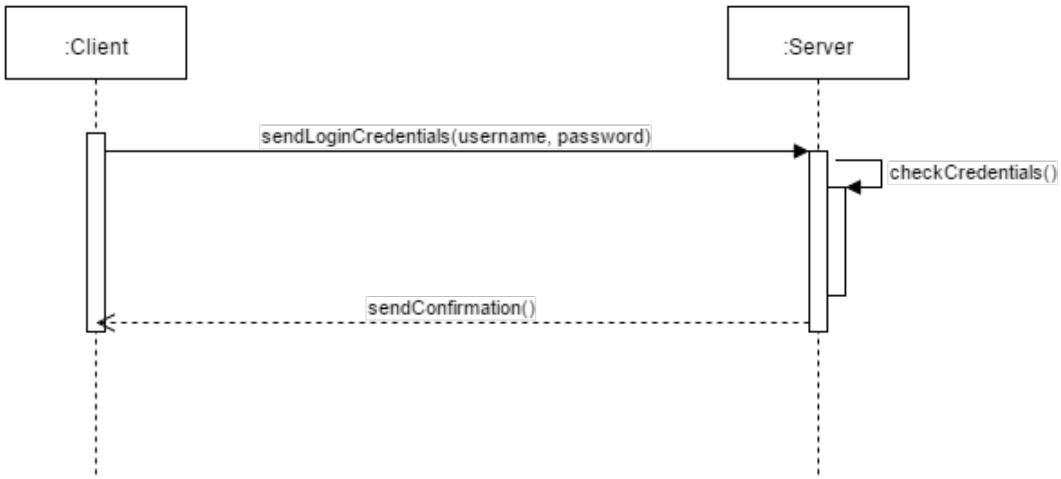

Figura 3.4: Diagramma di sequenza raffigurante la chiamata di login al server di *Italiansubs.net*

La procedura di login, che avviene a livello del server di *Italiansubs.net*, è uno dei pilastri dell'applicazione. La schermata di login viene mostrata all'avvio dell'applicazione, chiedendo all'utente username e password per autenticarsi al portale. Una volta forniti questi dati, il client HTTPS si collega all'homepage di *Italiansubs.net* e avvia il parsing del codice sorgente della pagina. Questo perché l'applicazione deve effettuare una chiamata HTTP **POST** interfacciandosi al server come se la chiamata di login fosse stata effettuata dalla homepage del portale: il parsing della homepage va a cercare il form di login presente nella pagina, imposta una lista di coppie *chiave-valore* con username e password inserite dall'utente nell'applicazione e inviarla al server

con la chiamata **POST**. Non appena la richiesta riceve una risposta, se il login è corretto, l'applicazione riceve e memorizza i cookies della sessione di login.

Un'altra informazione importante che viene presa al momento del login è l'*authcode* dell'utente. Questa volta, però, si tratta di un'informazione ottenibile soltanto dalle API di *Italiansubs.net*, ma comunque necessaria per riuscire a comunicare con le suddette API. Anche l'*authcode* viene salvato una volta ottenuto, così da poterlo riutilizzare ad ogni chiamata.

Nella schermata di login viene anche data la possibilità di effettuare il login automatico ad ogni avvio dell'applicazione; quando viene effettuato con successo un primo login e la checkbox dell'autologin è selezionata, l'applicazione salva le credenziali in locale, nelle *SharedPreferences*, una modalità di salvataggio dati messa a disposizione da Android: tutti i dati salvati nelle *SharedPrefereces* sono visibili e modificabili solamente dall'applicazione, in modo tale da ridurre al minimo interazioni maligne da parte di altre applicazioni.

3.4.4 Logout

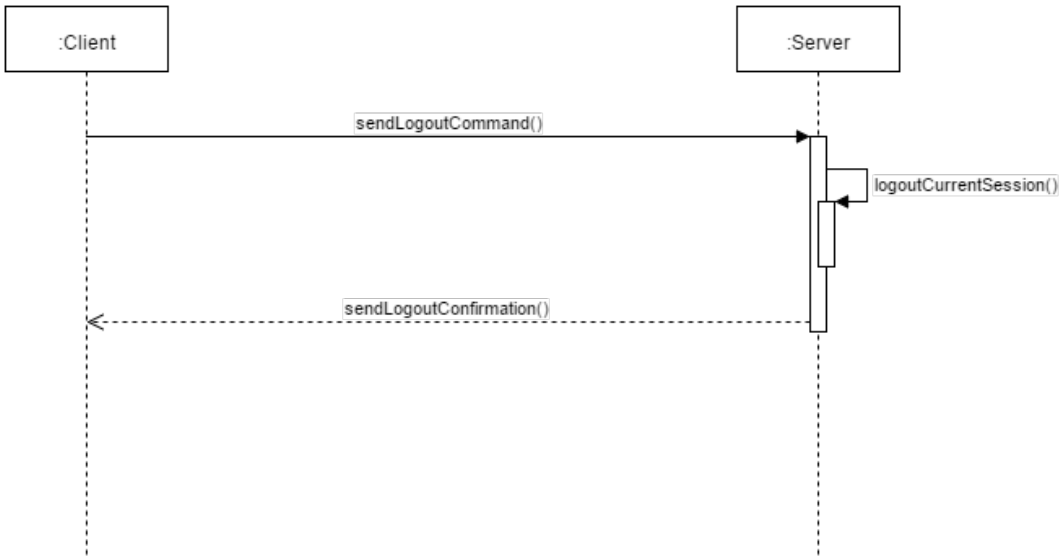

Figura 3.5: Diagramma di sequenza raffigurante la chiamata di logout al server di *Italiansubs.net*

La procedura di logout non è complicata come quella di login. Basta semplicemente inviare una chiamata HTTP **GET** ad un URL specifico per effet-

tuare il logout dal portale. Una volta effettuata la chiamata, tutti i dati salvati dall'applicazione sul dispositivo vengono azzerati.

3.4.5 Download dei sottotitoli

La gestione del download dei sottotitoli viene presa in carico da un *Service* messo a disposizione dal sistema operativo, il *DownloadManager Service*. I *Services* sono entità che compiono operazioni in background, completamente invisibili all'utente finale e possono svolgere diversi compiti. È possibile utilizzare sia i *Services* offerti da Android, che implementarne di nuovi, facendo uso anche dei *BroadcastReceiver*, entità anch'esse messe a disposizione dal sistema operativo di Google; i *BroadcastReceiver* sono "ricevitori" che ricevono determinati messaggi inviati in broadcast dal sistema o da altre applicazioni e hanno il compito di effettuare una determinata azione, ad esempio avviare un *Service*, quando ricevono un determinato messaggio.

Questo *Service* facilita il download di file da internet: va a creare un file temporaneo all'interno della cartella di download, che verrà poi "riempito" con i dati che verranno scaricati. Alla richiesta che il *Service* andrà ad inviare al server di *Italiansubs.net* verranno aggiunti i cookies precedentemente ricevuti al momento del login, in modo da poter scaricare in maniera corretta i sottotitoli. È possibile controllare il progresso del download tramite l'apposita notifica.

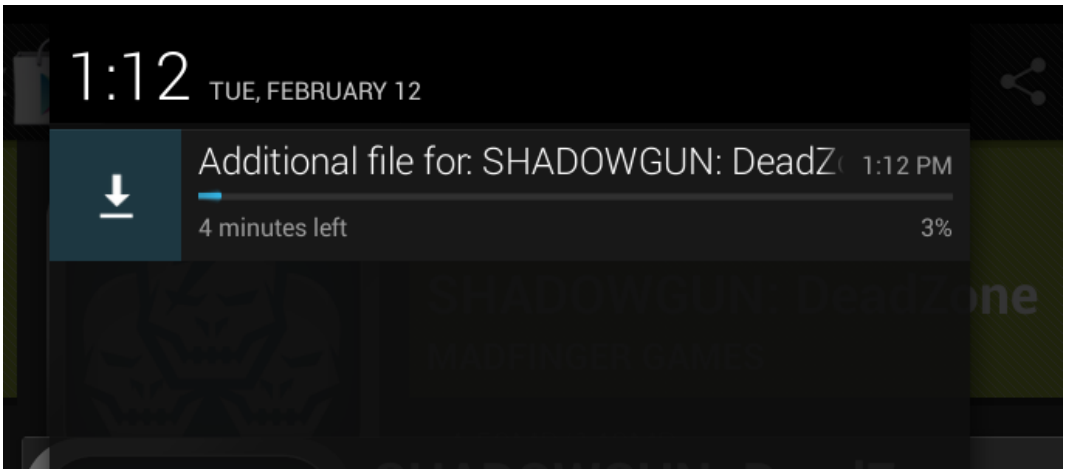

Figura 3.6: Esempio di notifica del DownloadManager

3.5 Selezione cartella download

L'utente può cambiare la cartella di download di default tramite l'apposita sezione nel menu *Opzioni*. Questa feature viene implementata nell'applicazione tramite l'utilizzo di una libreria esterna, leggermente modificata per rispecchiare lo stile grafico dell'applicazione.

Con questa funzionalità, l'utente può scegliere la cartella di destinazione che più gli aggrada e, una volta selezionata, tutti i sottotitoli verranno scaricati in quella cartella. Quando l'utente effettua la sua scelta, l'applicazione va a immettere una variabile nelle *SharedPreferences*, contenente il percorso scelto dall'utente. Ad ogni cambio di cartella, anche la variabile assume valori differenti.

3.6 Modellazione dell'applicazione

Per riuscire a gestire al meglio le informazioni ottenute dalle API, viene creato un "oggetto" contenente le informazioni ricavato dalle varie risposte da parte dei server API, ognuno con i propri metodi *getter* e *setter*; ogni classe che va a comporre il *model* dell'applicazione implementa la specifica interfaccia.

Sommariamente, è possibile dividere il *model* dell'applicazione in due macroaree: quella relativa al portale *Italiansubs.net* e quella relativa alla parte forum gestita dalle API di *Tapatalk*. Le specifiche della progettazione e la divisione in package dell'applicazione verranno discusse nel prossimo capitolo.

Capitolo 4

Progettazione

4.1 Progettazione generale

4.1.1 Divisione in package

Come accennato del capitolo precedente, in questa sezione andremo a descrivere la divisione in package dell'applicazione *ItaSAMobile*.

Partiamo con il dire che, essendo l'applicazione "divisa" in due parti, i package sono strutturati per rispettare questa divisione: il package principale contiene tutti i sotto-package relativi alla parte che si interfaccia con le API di *Italiansubs.net*, le activity principale e il model della parte riguardante le API di *Italiansubs.net*, mentre la parte che interagisce con le API di *Tapatalk*, le activity e il model riguardanti il forum si trovano nel sotto-package "*forum*". Ma procediamo con ordine.

Il package principale ha delle divisioni per tipo di oggetto o per grado di utilità all'interno dell'applicazione; possiamo trovare i seguenti packages:

- *activities*: all'interno di questo package possiamo trovare tutte le *Activity* che vanno a comporre la parte dell'applicazione che interagisce con *Italiansubs.com*.
- *connections*: all'interno di questo package sono presenti le diverse classi che andranno a gestire tutte le connessioni necessarie all'applicazione.
- *forum*: in questo package, che verrà descritto successivamente, si collocano tutti gli elementi che vanno a formare la parte riguardante il forum, cioè tutte le interazioni con le API di *Tapatalk*, activities, e model della parte forum.

- *model*: è il package che contiene classi che vanno a rappresentare i vari oggetti creati appositamente per gestire al meglio le informazioni ottenute dalle API di *Italiansubs.net*. Contiene un sotto-package, *interfaces*, che contiene tutte le interfacce dei vari oggetti presenti nel *model*.

- *parsers*: come suggerisce il nome, in questo package troviamo tutti i parser che vanno a ricavare dalle pagine XML delle API di *Italiansubs.com* tutte le informazioni necessarie. L'implementazione di alcuni parsers verrà descritta nel capitolo 5.

- *utils*: package principalmente composto da classi che portano utilità all'applicazione, la maggior parte delle quali sono gli *Adapter* delle varie liste di oggetti usate durante lo sviluppo. Un *Adapter* fa da ponte tra una *AdapterView*, come ad esempio una *ListView*, elemento Android che viene utilizzato per creare una lista scorrevole, e i dati sottostanti. Questo concetto verrà poi ripreso nelle sottosezioni successive.

Il package *forum* contiene, come appena descritto, tutti gli elementi che compongono la parte dell'applicazione che interagisce con le API di *Taptalk*; la sua struttura interna è del tutto identica a quella del package principale: troviamo infatti i package *activities*, *connections*, *model* e *utils* come nel package principale, tutti con lo stesso tipo di contenuto.

4.1.2 Progettazione del model

Il *model*, parte fondamentale del pattern di sviluppo **MVC**, ***M**odel **V**iew **C**ontroller*, uno dei più diffusi pattern di sviluppo usato per la buona programmazione in Java, di un applicazione può essere considerato come il suo "nucleo ad oggetti". Questo perché il *model* è composto da classi che vanno a rappresentare i vari oggetti che l'applicazione gestisce; per oggetto si intende propriamente anche l'astrazione di oggetti reali: è possibile, quindi, ricrearli all'interno dell'applicazione, assegnandogli le stesse caratteristiche dell'oggetto reale.

In questo caso, il *model* dell'applicazione è composto da classi che vanno a rappresentare gli oggetti di cui si vanno ad ottenere le informazioni attraverso l'interazione con entrambe le API. Di seguito verranno analizzata una delle classi che compongono il *model* dell'applicazione, visto che l'implementazione delle altre è molto simile.

La classe *News* modella le informazioni ricevute dalle API di *Italiansubs.net* per quanto riguarda le news del portale: ogni oggetto *News* è composto da

diversi elementi che caratterizzano le news anche sul portale, come l'ID della news, l'immagine corrispondente, il titolo della news e la data di uscita della news. Tutte queste informazioni sono campi della classe *News* che, tramite specifici metodi *getter* e *setter*, possono essere ottenuti e modificati a runtime.

Tutte le classi del *model* dell'applicazione implementano la loro corrispettiva interfaccia, contenente tutti i metodi che caratterizzano un determinato oggetto.

4.2 Scelte implementative

Di seguito verranno descritte le varie scelte implementative prese in considerazione durante la fase di progettazione dell'applicazione, focalizzando l'attenzione sui vari tipi di strutture di visualizzazione dati e sulle scelte grafiche.

4.2.1 Strutture di visualizzazione dati

Per quanto riguarda le strutture di visualizzazione dati, la *ListView* fa da padrona all'interno dell'applicazione. La *ListView* è il modo più semplice ed efficace per ottenere una lista scorrevole di elementi e viene usata in maniera intensiva all'interno dell'applicazione, visto che la maggior parte delle informazioni anche sul portale vengono visualizzate come liste di elementi. I dati delle varie *ListView* sono contenuti in oggetti di tipo *List¡¿* che vengono poi manipolati da *Adapter* di diverso tipo, uno per ogni tipo di oggetto presente nel *model*. Gli *Adapter* gestiscono la generazione della *View* di ogni oggetto presente all'interno della lista, dandogli il layout e le proprietà che caratterizzano quell'oggetto.

Per permettere una visualizzazione ottimale dei dati e garantire più facilità di utilizzo, è stato impiegato il *ViewPager*, che dà la possibilità di creare una specie di visualizzazione a schede, schede a cui l'utente può accedere tramite swipe verso destra o verso sinistra. L'utilizzo del *ViewPager* implica anche l'utilizzo dei *Fragment*, oggetti che racchiudono porzioni dell'interfaccia utente delle *Activity* e che vengono utilizzati proprio per creare *Activity* multi-pannello.

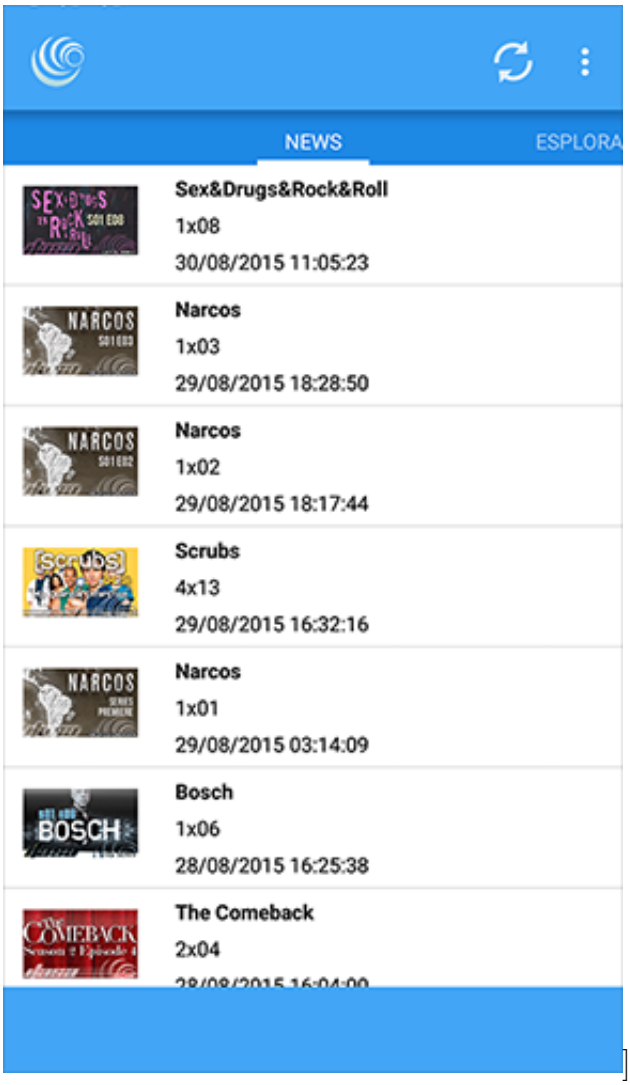

Figura 4.1: Screenshot dell'*Activity* principale: utilizzo di *ListView* e *ViewPager*

4.2.2 Scelte grafiche

Per quanto riguarda la parte grafica, l'applicazione è stata sviluppata in stretto contatto con gli amministratori del portale, in modo da rendere la grafica dell'applicazione simile a quella del portale.

Essendo principalmente sviluppata su Android 5.0+, l'applicazione segue le linee guida fornite da Google per quanto riguarda il **Material Design**, stile grafico introdotto dalla stessa Google con il rilascio della versione Android 5.0:

i colori sono stati presi dalla palette di colori *Material* fornita da Google, i bottoni dell'applicazione sono anch'essi di tipo *Material* (implementati con una libreria esterna che forniva già uno stile *Material* ai bottoni) e l'interfaccia del device si adatta anche ai colori dell'applicazione (la *StatusBar* del device viene colorata dello stesso colore dell'applicazione, questo però solo su dispositivi con Android 5.0+).

Le schermate di login e di logout si ispirano principalmente alle rispettive schermate dell'applicazione mobile di Facebook. Il logo dell'applicazione è stato fornito dagli amministratori stessi.

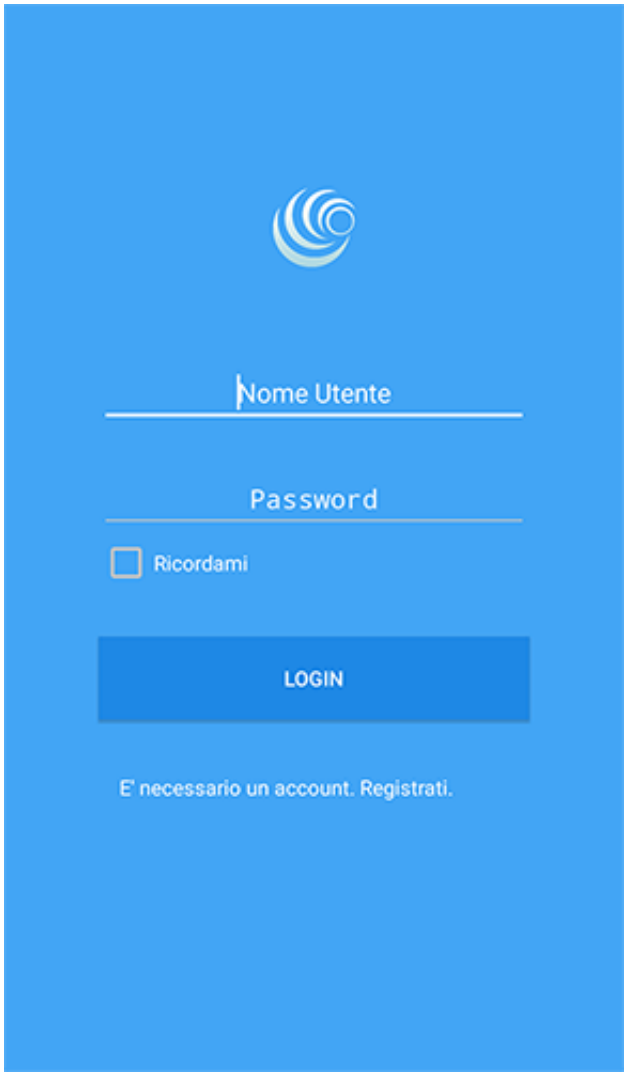

Figura 4.2: Screenshot della schermata di login

La parte forum, invece, è stata organizzata in modo da rendere il forum leggibile e facilmente consultabile: tutti gli elementi grafici che compongono la scheda del forum sono stati creati a mano dallo sviluppatore.

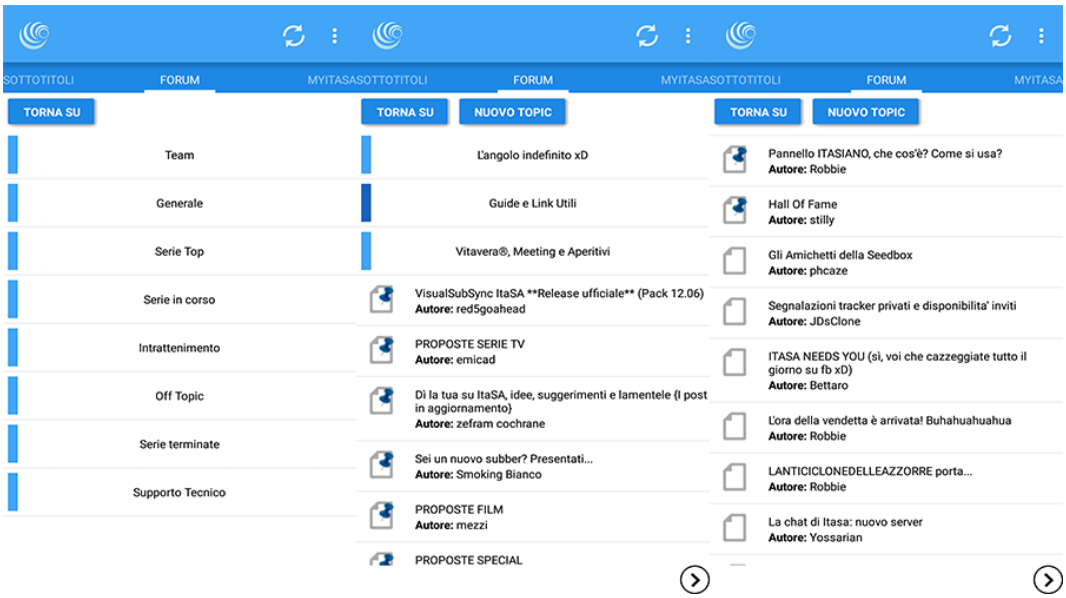

Figura 4.3: Screenshot delle varie schermate della parte forum

Sono stati usati principalmente due tipi di layout forniti da Android, il *LinearLayout* e il *RelativeLayout*; nello screenshot qui sotto sono stati usati entrambi i layout: in questo caso, il form di ricerca è formato da un *RelativeLayout* padre, con diversi *LinearLayout* con attributo *orientation:horizontal*, che permettono agli oggetti di rimanere affiancati.

Sono state inoltre modificate le dialog base del sistema operativo, per renderle fedele allo stile dell'applicazione e al *Material Design*.

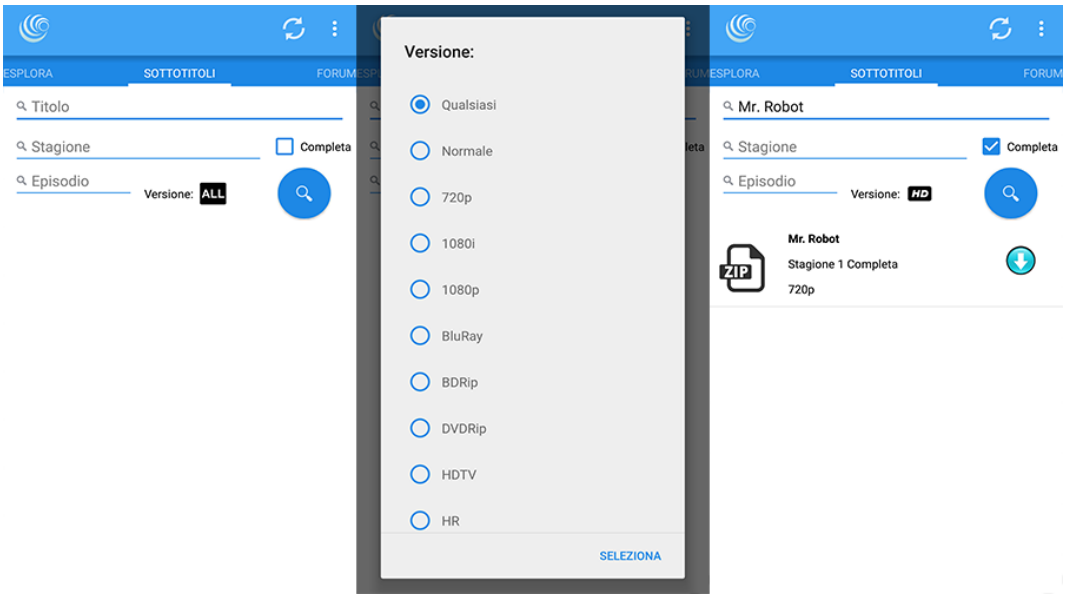

Figura 4.4: Screenshot dell'utilizzo dei diversi layout e di dialog custom

La parte del menù Opzioni dell'applicazione è formata da un'unica *ListView*, modificata in modo tale da avere dei titoli per ogni sezione: nell'adapter viene differenziato il tipo dell'oggetto presente nella lista (la lista viene popolata manualmente tramite codice) e in base al tipo dell'oggetto (che può essere *header*, *folder* o *option*) vengono assegnate alla cella di competenza attributi e caratteristiche differenti.

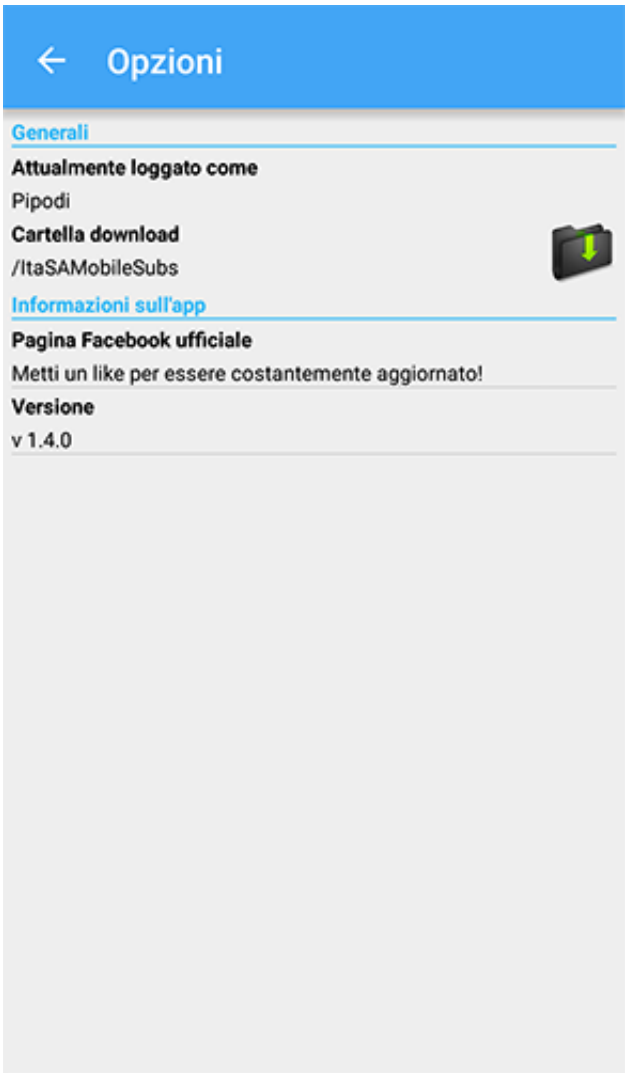

Figura 4.5: Screenshot della schermata del menù Opzioni

Capitolo 5

Implementazione

5.1 Scelte implementative

In questa sezione verranno descritte le implementazioni, inserendo anche spezzoni di codice per aiutare la comprensione, delle principali classi dell'applicazione.

5.1.1 MainActivity

La *MainActivity* è l'*activity* principale dell'applicazione ed è quella che permette l'utilizzo di tutte le funzionalità dell'applicazione. È una classe ricca di righe di codice, dovendo controllare diverse *tab*, visto che l'*activity* è stata progettata per essere utilizzata con un *ViewPager* diviso in 5 tab, ognuna delle quali può essere considerata come una specie di "*Activity* a sé stante"; questo particolare oggetto è il nome di *Fragment*.

La *MainActivity*, come detto in precedenza, contiene moltissimi controlli, sia per quanto riguarda le azioni effettuate dall'utente (swipe, tap su determinati campi, etc...), sia per quanto riguarda il controllo delle risorse del dispositivo (controllo disponibilità connessione internet): tutti questi controlli agiscono sul comportamento dell'applicazione con l'utente, fornendo il più possibile un'esperienza che ne facilita l'utilizzo e che la renda il più intuitiva possibile.

All'interno della classe *MainActivity* sono presenti alcune sottoclassi, necessarie per utilizzare al meglio il *PagerAdapter*, incaricato alla creazione e gestione delle tab del *ViewPager*; una di queste, chiamata *MainActivityFragment*, gestisce la corretta visualizzazione del contenuto delle varie tab, basandosi sul loro numero progressivo, e le loro interazioni con le altre classi dell'applicazione. Essendoci 5 tab, di seguito verrà descritto il funzionamento di una tab,

in questo caso quella riguardante le news. Di seguito, il codice della parte interessata:

```
switch (position) {
    case 0:
      view = inflater.inflate(R.layout.news_layout, container,
         false);
        final ListView newsList = (ListView)
            view.findViewById(R.id.news_list);
        newsFragment = this;
        final ButtonRectangle downloadMore = (ButtonRectangle)
            view.findViewById(R.id.loadmoreButton);
        if (!Constants.newsDownloaded) {
            newsParser = new NewsParser(new
                ArrayList<NewsInterface>(), newsList, getActivity(),
                inflater, downloadMore, newsFragment);
            newsParser
                .execute("https://api.italiansubs.net/api/rest/
            news?&apikey=ce0a811540244ad5267785731a5c37a0");
        } else {
            newsParser = new NewsParser(TempStorage.downloadedNews,
                newsList, MainActivity.context, inflater,
                    downloadMore, newsFragment);
                newsParser.getDownloadedNews();
            }
            downloadMore.setOnClickListener(new
                View.OnClickListener() {
            @Override
                public void onClick(View v) {
                    newsParser = new NewsParser(
                        TempStorage.downloadedNews, newsList,
                            getActivity(), inflater, downloadMore,
                                newsFragment);
                        newsParser
                            .execute("https://api.italiansubs.net/api
                     /rest/news/direct?%2Fapi%2Frest%2Fnews=
                     &apikey=2513ef3ea5eed856b879f8fa960e2a26&page="
                            + TempStorage.nextPage);
                        TempStorage.nextPage++;
                    }
                });
        return view;
```

La gestione delle varie tab viene effettuata tramite *switch-case*: ogni tab

ha un numero intero progressivo, in questo caso lo 0, e ogni *case* rappresenta una tab; quindi ogni operazione effettuata all'interno di un *case* riguarda solo ed esclusivamente una tab. La variabile *view*, passata come parametro dal metodo che contiene questo estratto di codice, rappresenta la veste grafica del tab ed è proprio da questa variabile che vengono recuperati tutti gli elementi grafici e di visualizzazione della tab, come una *ListView*, dando la possibilità alla tab di ottenere dati dalle diverse connessioni presenti nell'applicazione. Con l'istruzione finale, *return view*, viene correttamente visualizzata la tab con tutti i dati aggiornati e i vari *listener* inseriti sui vari elementi.

5.1.2 Login

La classe del login, chiamata *ItaSALogin*, è una classe che estende *AsyncTask*; possiamo, infatti, trovarne i 3 principali metodi: *onPreExecute*, *doInBackground* e *onPostExecute*. Il metodo più importante di questa classe è proprio *doInBackground*: in questo metodo la classe va a cercare, come prima cosa, l*authcode* dell'utente, un codice univoco per ogni utente che viene utilizzato in molte chiamate alle API di *Italiansubs.net*, tramite parsing XML di una risposta delle API.

Una volta ottenuto l'*authcode*, la classe procede con la preparazione del client HTTP usando le classi di libreria *DefaultHttpClient* e *HttpPost*; infatti, per autenticarsi correttamente al portale tramite login, è necessario effettuare una chiamata POST, contenente i diversi dati dell'utente. Viene settato l'header alla connessione *HttpPost* e subito l'applicazione va ad interfacciarsi con la homepage del portale: connettendosi alla pagina iniziale di *Italiansubs.net*, l'applicazione va a ricercare nel codice sorgente della pagina la parte riguardante il form di login; una volta trovata, vengono assegnate le credenziali d'accesso che vengono fornite dall'utente alle corrispettive chiavi ottenute tramite questo processo di parsing della pagina. Qui sotto, l'estratto di codice riguardante la parte appena discussa:

```
httpPost = new HttpPost("http://www.italiansubs.net");
httpPost.setHeader("User-Agent", "ItaSA Mobile Application test by
    Pipodi");
httpPost.setHeader("Content-Type",
    "application/x-www-form-urlencoded");
List<NameValuePair> pairs = new ArrayList<>();
Document doc = null;
    try {
      doc = Jsoup.connect("http://www.italiansubs.net").get();
        } catch (IOException e1) {
```

```
            e1.printStackTrace();
        }
        Element loginform = null;
        try {
            loginform = doc.getElementById("form-login");
        } catch (NullPointerException e) {
            e.printStackTrace();
        }
        Elements inputElements = loginform.getElementsByTag("input");
        for (Element inputElement : inputElements) {
            String key = inputElement.attr("name");
            String value = inputElement.attr("value");
            if (key.equals("username"))
                value = username;
            else if (key.equals("passwd"))
                value = password;
            pairs.add(new BasicNameValuePair(key, value));
        }
```

Dopo aver effettuato queste operazioni, il client avvia la procedura di login effettuando la chiamata POST ai server di *Italiansubs.net* e recupera i cookies di sessione non appena il login va a buon fine.

Nel *postExecute* abbiamo diverse operazioni, in base sia al risultato del login, sia alla condizione della rete, ma si tratta prettamente di operazioni di visualizzazione di *Toasts*, piccole notifiche che compaiono nella parte bassa del display per informare l'utente del risultato delle operazioni. Inoltre viene impostata anche la variabile booleana *remember* nelle *SharedPreferences*: se l'utente avrà scelto, al momento dell'inserimento dei dati di login, di autenticarsi automaticamente ad ogni avvio dell'applicazione, la variabile verrà impostata a *true*, altrimenti a *false*.

Per il logout, le operazioni da svolgere sono molto più semplici: viene creato un oggetto di tipo HttpGet e viene effettuata una chiamata ai server di *Italiansubs.net* tramite uno specifico URL che va ad effettuare il logout dell'attuale sessione dal server. Nel *postExecute* verranno poi svolte le operazioni di azzeramento dei dati dell'utente, come *authcode*, cookies, username e password.

5.1.3 Parsers

I *parsers* svolgono le principali operazioni di *information retrieving*, ossia recuperare da documenti, in questo caso XML, informazioni che poi verranno

utilizzate per creare gli oggetti che verranno poi gestiti dall'applicazione.

I *parsers* sviluppati per questa applicazione sono numerosi, ma il meccanismo di fondo è condiviso da tutti quanti; quindi, di seguito, verrà descritto il funzionamento di uno dei tanti *parsers*, quello addetto al recupero delle informazioni delle varie news del portale *Italiansubs.net*.

La classe in questione è chiamata *NewsParser* e si trova nel package *parsers*; la classe in questione estende anch'essa *AsyncTask* e svolge tutte le operazioni di parsing del documento XML nel metodo *doInBackground*. Gli altri due metodi, *onPreExecute* e *onPostExecute* svolgono operazioni sulla parte UI dell'*Activity*.

Le operazioni di parsing si avviano con una connessione da parte dell'applicazione alle API di *Italiansubs.net*, chiedendo una lista di tutte le news postate sul portale; una volta ottenuto, il documento XML viene salvato come oggetto Document, tipo di oggetto reso disponibile dalla libreria esterna **JDOM**, e gli viene subito assegnato un iteratore, che andrà a scansionare il documento per livelli.

È stata creata una classe che si occupa di questa operazione, chiamata **XMLIterator**, che ha un metodo statico pubblico, *parseXMLTree*, che prende come parametri in ingresso un intero che indica il livello di profondità dell'albero XML da dover scansionare alla ricerca delle informazioni necessarie, e un oggetto di tipo Document che rappresenta il documento XML da scansionare. Il funzionamento del metodo è molto semplice ed è facilmente intuibile dal codice sottostante:

```
public class XMLIterator {

    public static Iterator parseXMLTree(int level, Document
        document) {
        Element doc = document.getRootElement();
        for (int i = 0; i < level; i++) {
            doc = doc.getChildren().get(0);
        }
        return doc.getChildren().iterator();
    }
}
```

Questa utility vene usata ogni qual volta viene richiesto un documento dalle API di *Italiansubs.net*.

Una volta ottenuto, il contenuto dell'iteratore sarà composto da tutti gli elementi XML che componevano l'albero iniziale ed è qui che inizia il vero compito del parser che andrà a creare, in questo caso, un oggetto di tipo NewsInterface, interfaccia dell'oggetto di tipo News: vengono prese tutte le informazioni che servono per visualizzare poi tutte le news in una lista presente nell'*Activity* tramite il metodo "*.getChild(childName).getText()*", dove *childName* sta ad indicare il nome della chiave che contiene il valore dell'informazione cercata. Viene anche scaricata l'immagine della news tramite il metodo statico *getPhotoFromURL(url)* della classe *ConnectionForImg* e viene salvata come Bitmap. Non appena vengono recuperate tutte le informazioni necessarie, viene creato l'oggetto *NewsInterface* con tutte le informazioni appena trovate e viene inserito all'interno della lista che poi verrà inserita, nel metodo *onPostExecute()*, all'interno di un *Adapter* creato appositamente per le news, chiamato *NewsAdapter*, che verrà poi assegnato alla *ListView* di competenza per essere visualizzato correttamente all'interno dell'*Activity*. Nel metodo *onPostExecute* vengono inoltre registrati i diversi listener della lista.

Qui sotto, la parte principale del parsing, effettuata nel metodo *doInBackground()*:

```
Document doc = Connection.connectToAPIURL(params[0]);
        Iterator iter = XMLIterator.parseXMLTree(2, doc);
        while (iter.hasNext()) {
            Element item = (Element) iter.next();
            int showId =
                Integer.parseInt(item.getChild("show_id").getText());
            int showCategory =
                Integer.parseInt(item.getChild("category").getText());
            if (showId != 0 && showCategory == 16) {
                String title = item.getChild("show_name").getText();
                Iterator temp =
                    item.getChild("episodes").getChildren().iterator();
                String episode = "";
                if (!item.getChild("special").getText().equals("")) {
                    episode += (item.getChild("special").getText() + "
                        ").toUpperCase();
                }
                while (temp.hasNext()) {
                    episode += ((Element) temp.next()).getText() + " ";
                }
                String release =
                    item.getChild("submit_date").getText();
                String photo_url = item.getChild("image").getText();
```

```
int news_id =
    Integer.parseInt(item.getChild("id").getText());
release = DateFormatParser.changeDateFormat(release);
Bitmap photo =
    ConnectionForImg.getPhotoFromURL(photo_url);
NewsInterface news = new News(showId, news_id, title,
    episode, release, photo);
this.items.add(news);
```

Gli altri parsers, come detto in precedenza, condividono tutti gli stessi meccanismi di fondo: infatti tutti i parsers vanno a recuperare i file XML dalle risposte delle API e poi, tramite iteratore, vanno a recuperare le informazioni necessarie usando il metodo "*.getChild(childName).getText()*".

5.1.4 Connessioni per informazioni

Concettualmente, anche queste classi fanno parte della famiglia dei parsers, visto che anche loro vanno a recuperare delle informazioni dai documenti XML delle API di *Italiansubs.net*, però vi è una leggera differenza: infatti queste classi hanno il compito di far visualizzare a schermo determinate informazioni, senza andare a creare oggetti per poi inserirli in qualche lista. Anche in questo caso, essendocene diverse di classi di questo tipo all'interno dell'applicazione, verrà analizzata una classe di esempio, la classe addetta al download della descrizione dettagliata delle news, chiamata *ConnectionForNewsDescription*.

La classe *ConnectionForNewsDescription* estende *AsyncTask* e, come anche le altre, svolge le sue principali funzione nel metodo *doInBackground()*. All'interno di questa classe si fa uso anche dell'HTML, usato per dare diversi stili al testo a runtime: vengono preparati diversi oggetti di tipo *String*, tanti quante sono le informazioni da visualizzare a schermo, vengono inizializzati con del testo che deve essere visualizzato in grassetto (ad esempio "*Prova *" che avrà come risultato finale **Prova**), a cui poi verranno aggiunte altre stringhe di testo ottenute dal parsing del documento XML. Una volta ottenuta la stringa completa, nel metodo *onPostExecute()*, verrà poi assegnata alla corrispettiva *TextView*, cella di testo di Android. La stringa viene convertita dall'HTML tramite il metodo statico della classe di libreria *Html Html.fromHtml(string)*, dove *string* sta per la stringa che deve essere visualizzata in quella *TextView*. Questo stratagemma implementativo è stato usato per limitare al minimo le *TextView* da gestire: infatti, le *TextView* sarebbero dovute essere 2 per ogni informazione necessaria (ad esempio, **Titolo:** Prova), in questo modo il numero viene ridotto ad 1, alleggerendo di molto l'applicazione e facilitando la gestione del codice allo sviluppatore.

Qui sotto, il codice semplificato della parte di gestione delle stringhe in HTML:

```
@Override
    protected String doInBackground(String... params) {
        Document doc = Connection.connectToAPIURL(
      "https://api.italiansubs.net/api/rest/news/"
      + this.newsID + "?apikey=ce0a811540244ad5267785731a5c37a0");
        this.sTranslators = "<b>Traduttori: </b>";
        this.sTeam = "<b>Team: </b>";
        this.sInfo = "<b>Info sulla puntata: </b>";
        this.sResync = "<b>Resync: </b>";
        this.sImage = "<b>Immagine by: </b>";
        this.sReviewer = "<b>Revisione: </b>";

      /**
      Altro codice qui sotto...
      **/

      this.sTranslators += item.getChild("translation").getText();
      this.sTeam += item.getChild("sync").getText();
      this.sInfo += item.getChild("info").getText();
        this.sReviewer += item.getChild("submitted_by").getText();
      this.sResync += item.getChild("resync").getText();
      this.sImage += item.getChild("image_by").getText();

  }

  @Override
   protected void onPostExecute(String result) {
     this.translators.setText(Html.fromHtml(this.sTranslators));
       this.team.setText(Html.fromHtml(this.sTeam));
       this.info.setText(Html.fromHtml(this.sInfo));
       this.image.setText(Html.fromHtml(this.sImage));
       this.resync.setText(Html.fromHtml(this.sResync));
       this.reviewer.setText(Html.fromHtml(this.sReviewer));
  }
```

Da notare come, anche in questo caso, gli elementi testuali vengano ricavati dal documento XML usando il metodo "*.getChild(childName).getText()*".

5.1.5 Download dei sottotitoli

Il download dei sottotitoli viene gestito da questa classe, *ConnectionForSubtitle*, che contatta le API di *Italiansubs.net* richiedendo il sottotitolo da scaricare. Per ottimizzare al meglio il download, è stato usato un service messo a disposizione da Android, il *DownloadManager* service: il service in questione dà la possibilità all'applicazione di interfacciarsi con il download manager del sistema operativo, fornendo diverse funzionalità all'applicazione.

La classe *ConnectionForSubtitle* estende *AsyncTask* e all'interno del metodo *doInBackground()* viene implementato il service *DownloadManager*. Più in dettaglio: come prima cosa, viene letto dalle *SharedPreferences* il percorso della cartella di download, quello di default oppure quello impostato dall'utente tramite l'apposito menù, viene *parsato*, in modo poi da poter creare un oggetto di tipo *File* che corrisponde alla cartella di destinazione del file (se la cartella selezionata non esiste, viene creata sul momento); dopo di che viene istanziata la richiesta che verrà indirizzata al *DownloadManager*, alla quale vengono aggiunti i cookies di sessione ottenuti al momento del login iniziale, e vengono aggiunte anche altre informazioni sul download, ad esempio la descrizione del file, visibile nelle notifiche. Il file viene scaricato sotto forma di file *.zip*, visto che, nativamente, le API di *Italiansubs.net* forniscono file di tipo *.zip*. Di seguito, la parte di codice riguardante l'implementazione del service *DownloadManager* e l'aggiunta di alcuni parametri alla richiesta:

```
DownloadManager mgr = (DownloadManager) MainActivity.context
                .getSystemService(Context.DOWNLOAD_SERVICE);
        Uri downloadUri = Uri
                .parse(
            "https://api.italiansubs.net/api/rest
            /subtitles/download?subtitle_id="
                        + this.id
                        + "&authcode="
                        + "0a7623231022fde8b519d5f6d3084700"
                        + "&apikey="
                        + Constants.APIKey);
        DownloadManager.Request request = new DownloadManager.Request(
                downloadUri);
        List<Cookie> loginCookies = LoginVariables.loginCookies;
        request.setAllowedNetworkTypes(
                DownloadManager.Request.NETWORK_WIFI
                        | DownloadManager.Request.NETWORK_MOBILE)
                .setAllowedOverRoaming(false)
                .setTitle("Download sottotitoli")
```

```
            .setDescription(fileName)
            .setDestinationInExternalPublicDir(directory,
                    fileName + ".zip");
      request.allowScanningByMediaScanner();
      request.setNotificationVisibility(
    DownloadManager.Request.VISIBILITY_VISIBLE_NOTIFY_COMPLETED);
      String cookieString = "";
      for (Cookie cookie : loginCookies) {
          cookieString += cookie.getName() + "=" +
              cookie.getValue() + "; path=" + cookie.getPath()
                  + "; domain=" + cookie.getDomain() + "; expiry=" +
                      cookie.getExpiryDate() + "; ";
      }
      request.addRequestHeader("Cookie", cookieString);
      mgr.enqueue(request);
```

5.1.6 Adapters

Gli *adapters* sono classi create appositamente per gestire l'inserimento di liste di oggetti all'interno delle *ListView* e per gestirne la corretta visualizzazione. Anche gli *adapters* sono numerosi all'interno dell'applicazione, quindi in questa sezione si andrà a spiegare l'implementazione di una classe d'esempio, la classe *ShowAdapter* che contiene anche un filtro istantaneo per la ricerca nella lista.

La classe *ShowAdapter* estende la classe *BaseAdapter*, ereditando alcuni metodi che devono essere implementati secondo le necessità dell'applicazione, ma i più importanti sono: *getView()* e *getItem()*. Il primo è il metodo che viene invocato in automatico non appena un elemento dell'adapter (e quindi un elemento presente sulla *ListView* collegata all'adapter) viene visualizzato a schermo, andando quindi a gestire tutta la parte grafica dell'elemento della lista, mentre il secondo consente di recuperare un elemento dalla lista associata all'adapter, data in ingresso la posizione in lista dell'elemento desiderato.

Per quanto riguarda la parte del filtro istantaneo per le ricerche, la situazione è leggermente più complicata. Innanzitutto, è stato necessario far implementare alla classe l'interfaccia *Filterable* e, quindi, implementare il metodo pubblico *getFilter()*: in *getFilter()* viene effettuato l'*override* manuale di due metodi specifici degli oggetti Filter, *publishResults()*, metodo invocato dopo l'operazione di filtraggio eseguita nel secondo metodo, cioè *performFiltering()*: questo metodo prende in ingresso una *CharSequence*, corrispondente alla stringa da

ricercare all'interno della lista; se questa stringa è nulla o uguale a 0, verrà ritornata la lista originale, quando invece l'utente immette qualcosa nel campo di ricerca il metodo va ad individuare tutti i riscontri tra la stringa immessa e gli oggetti presenti nella lista. Tutti gli oggetti che avranno una corrispondenza con la stringa verranno aggiunti all'interno di una nuova lista, lasciando quindi invariata la lista originale, così da permettere l'azzeramento della ricerca ed evitare problemi tra le due liste. Di seguito, la parte di codice riguardante il filtro:

```
@Override
    public Filter getFilter() {
        return new Filter() {
            @Override
            protected void publishResults(CharSequence constraint,
                FilterResults results) {
                if (results.count == 0)
                    notifyDataSetInvalidated();
                else {
                    list = (List<ShowInterface>) results.values;
                    notifyDataSetChanged();
                }
            }

            @Override
            protected FilterResults performFiltering(CharSequence
                constraint) {
                FilterResults results = new FilterResults();
                if (constraint == null || constraint.length() == 0) {
                    results.values = list;
                    results.count = list.size();
                } else {
                    List<ShowInterface> filteredList = new
                        ArrayList<ShowInterface>();
                    for (ShowInterface show : list) {
                        if (show.getShowName().toLowerCase()
                     .contains(constraint.toString()
                  .toLowerCase())) {
                            filteredList.add(show);
                        }
                    }

                    results.values = filteredList;
                    results.count = filteredList.size();
```

```
            }
            return results;
        }
    };
}
```

5.1.7 TapatalkService

Questa classe è la classe addetta alla gestione delle chiamate da effettuare verso le API di *Tapatalk*. In questa classe è possibile osservare l'implementazione del client XML-RPC, di cui abbiamo discusso in precedenza nel capitolo 3 di questo scritto, e le varie strutture delle chiamate di tipo XML-RPC.

Nel dettaglio: nel costruttore della classe vengono effettuate le operazioni preliminari pre-chiamata (assegnamento del client XML-RPC e creazione del *CookieStore*, oggetto dove verranno salvati i cookies identificativi della sessione sulle API di *Tapatalk*); la classe contiene poi tutti i metodi per effettuare le chiamate alle API necessari per il corretto funzionamento della parte forum dell'applicazione. Come succede per la maggior parte delle classi che si interfacciano con le API, tutti questi metodi mantengono una struttura comune, differenziandosi l'uno dall'altro da piccole istruzioni, per questo si andrà a descrivere un metodo d'esempio, il metodo per ottenere tutti i topic di una determinata sezione, il metodo *getTopic()*. Di seguito, il codice sorgente:

```
public Map<String, Object> getTopic(String forumID) throws
    XMLRPCException {
        this.loginForum(this.username, this.password);
        Object[] params = {forumID, TempStorage.startPageForum,
            TempStorage.endPageForum};
        Map<String, Object> topics = (Map<String, Object>)
            this.getClient().call("get_topic", params);
        return topics;
    }
```

Come possiamo notare, il metodo, come prima cosa, effettua il login, andando ogni volta ad aggiornare i cookies di sessione, mantenendola costantemente attiva, subito dopo prepara un oggetto di tipo *Object[]*, un array di Object, che contiene i parametri da inviare con la chiamata alle API. Una volta riempito l'array con i parametri necessari alla chiamata (in questo caso sono l'ID del forum di cui si vogliono ottenere i topic, sotto forma di stringa, e due interi, il primo che indica da quale pagina iniziare e il secondo a quale pagina fini-

re: questi ultimi parametri sono molto importanti per l'applicazione, dandole la possibilità di sfogliare ogni forum e topic in maniera efficiente. L'array in questione viene poi inserito all'interno della chiamata XML-RPC, che viene inviata tramite metodo *this.getClient().call("get$_t$opic", params)*, dove *get$_t$opic* corrisponde alla funzione delle API di *Tapatalk* che permette il recupero dei topic e *params* è l'array di Object. L'oggetto che verrà restituito dalla chiamata verrà poi gestito da un'altra classe, *ResultMapper*, di cui si discuterà a breve.

5.1.8 ForumConnection

La classe *ForumConnection* è il cuore della parte forum dell'applicazione: infatti gestisce qualunque interazione tra l'utente e connessione con le API sia necessaria per garantirne il corretto funzionamento. La classe anch'essa estende *AsyncTask* però questa volta non svolge le proprie operazioni solo ed esclusivamente del metodo *doInBackground*, ma anche nel metodo *onPostExecute*, dovendo effettuare numerose operazioni in base alle molteplici possibilità di utilizzo da parte dell'utente. Verranno ora descritte in maniera dettagliata tutte le operazioni di controllo e connessione che la classe *ForumConnection* mette in atto.

Nel metodo *doInBackground* troviamo la solita gestione delle connessioni e alle chiamate alle API di *Tapatalk*, però questa volta si differenzia dalle altre esclusivamente perché le chiamate alle API si differenziano tra di loro dal tipo di oggetto che ha invocato tale chiamata. Codice d'esempio:

```
switch (type) {
    case FORUM:
        if (this.childField == null) {
            this.items =
                ResultMapper.getResultFromArray(tapatalkService
         .getForum());
        } else {
            this.items = ResultMapper.parseChild(this.childField);
            if (TempStorage.startPageForum == 0) {
                this.items.addAll(ResultMapper.getResultFromTopicMap(
            tapatalkService.getStickyTopic(params[0]), true));
            }
            this.items.addAll(ResultMapper.getResultFromTopicMap(
          tapatalkService.getTopic(params[0]), false));
        }
        break;
```

```
    case TOPIC:
        if (TempStorage.startPageForum == 0) {
            this.items = ResultMapper.getResultFromTopicMap(
         tapatalkService.getStickyTopic(params[0]), true);
            }
        this.items.addAll(ResultMapper.getResultFromTopicMap(
      tapatalkService.getTopic(params[0]), false));
        break;
    case POST:
        this.items = ResultMapper.getResultFromPostMap(tapatalkService
      .getPosts(params[0]));
        break;
    }
```

Come si può notare, è presente uno *switch/case* che va a confrontare i vari tipi di oggetto che ha invocato la classe *ForumConnection*: questi tipi fanno parte di un oggetto di tipo *Enum*, contenente tutti i tipi di oggetti che possono essere presenti all'interno del forum.

Nel metodo *onPostExecte()* abbiamo invece tutta la gestione della UI della parte forum, che deve reagire ad ogni operazione effettuata dall'utente durante la navigazione nel forum. Come prima cosa, l'applicazione va a controllare se c'è un'operazione di *refresh* in corso da parte dello *SwipeRefreshLayout*, un layout di aggiornamento usato in molte applicazioni Android, che permette di effettuare un aggiornamento dell'attuale lista semplicemente facendo scorrere il dito dall'alto verso il basso, quando si è in cima alla lista: in tal caso, blocca l'utilizzo della parte UI, per evitare richieste multiple. Subito dopo, sono stati implementati numerosi controlli su diverse variabili, che sono di vitale importanza per quanto riguarda la gestione delle pagine del forum da visualizzare. Infatti, le API di *Tapatalk* non forniscono il numero preciso di pagine, a differenza delle API di *Italiansubs.net* che, per ogni documento con più pagine, forniscono il numero totale di pagine di quel documento; per questo è stato necessario usare degli stratagemmi implementativi per ottimizzare la navigazione nel forum. Esempio di controllo pagine:

```
if (TempStorage.startPageForum == 0 &&
    (this.type.equals(ForumTypes.TOPIC) ||
   this.type.equals(ForumTypes.FORUM))) {
    TempStorage.remainingTopics = TempStorage.totalTopicsOnThisForum;
} else if (TempStorage.startPagePost == 0 &&
    this.type.equals(ForumTypes.POST)) {
    TempStorage.remainingPosts = TempStorage.totalPostsOnThisThread;
}
```

Dopo il blocco di controllo delle pagine e dei pulsanti di risposta, vengono implementati i diversi *listener* della *ListView* del forum: anch'essa reagisce alle varie azioni dell'utente dipendentemente dal tipo di oggetto che, in precedenza, ha invocato la classe *ForumConnection*. Ad ogni tipo di oggetto corrisponde una chiamata diversa della classe *ForumConnection* e, di conseguenza, una diversa gestione dei dati risultanti da tale chiamata (vedere esempio di codice presente ad inizio sezione). Ad esempio:

```
case TOPIC:
   TopicInterface currentTopic = (TopicInterface) currentItem;
   TempStorage.forumHistory.push(items);
   TempStorage.paramsHistory.push(currentParam);
   TempStorage.tempChild = childField;
   ForumConnection postConnection = new ForumConnection(context,
       viewList,
     null, upButton, replyButton, nextButton, previousButton,
         refreshLayout,
     ForumTypes.POST, progressBar);
   postConnection.execute(currentTopic.getTopicID());
   break;
```

Notare come la classe *ForumConnection* venga reinstanziata, ogni volta con un diverso parametro corrispondente al tipo di dato del forum: ad ogni *case* dello *switch* corrisponde un oggetto del forum e quando viene cliccato si "entra" in quell'oggetto, quindi è necessario invocare *ForumConnection* con un tipo di oggetto forum diverso, se sono rispettate determinate condizioni. Ed è proprio questa navigazione per livelli che ha spinto alla realizzazione di un oggetto di tipo *Stack*, nel quale, com'è possibile notare nell'esempio di codice precedente, vengono inseriti diversi oggetti, che vengono poi riutilizzati quando viene premuto il tasto "*Torna su*", che permette all'utente di tornare al livello di navigazione precedente, senza dover riottenere tutti i dati dalle API.

Nel metodo *onPostExecute()* sono presenti anche tutti i listener dei vari bottoni che sono presenti nella UI e diversi metodi che riguardano le varie modalità di aggiornamento dei dati.

5.1.9 ResultMapper

La classe *ResultMapper* è la classe addetta al recupero delle informazioni del forum dalle risposte delle API di *Tapatalk*. È una classe di utility, tutti i suoi campi sono pubblici e statici e si differenziano tra di loro dal tipo di

informazioni che devono estrarre: ogni funzione delle API chiamata dall'applicazione ha la propria funzione di parsing nella classe *ResultMapper*. Verrà descritto, quindi, solo un metodo, in modo da descrivere sommariamente l'intera classe.

Qui sotto, l'implementazione del recupero dei topic dalla mappa ricevuta dalle API.

```
public static List<ForumObjInterface>
    getResultFromTopicMap(Map<String, Object> item, Boolean sticky)
   throws UnsupportedEncodingException {
    List<ForumObjInterface> topicList = new ArrayList<>();
    Iterator<Map.Entry<String, Object>> iterator =
        item.entrySet().iterator();
    while (iterator.hasNext()) {
        Map.Entry<String, Object> entry = iterator.next();
        if (entry.getKey().equals("total_topic_num")) {
            if ((int) entry.getValue() == 0) {
                TempStorage.totalTopicsOnThisForum = 0;
                TempStorage.remainingTopics = 0;
                return topicList;
            } else {
                TempStorage.totalTopicsOnThisForum = (int)
                    entry.getValue();
            }
        } else if (entry.getKey().equals("topics")) {
            Object[] topics = (Object[]) entry.getValue();
            for (int i = 0; i < topics.length; i++) {
                Map<String, Object> temp = (Map<String, Object>)
                    topics[i];
                Iterator<Map.Entry<String, Object>> tempIterator =
                    temp.entrySet().iterator();
                String title = "";
                String topicID = "";
                String topicAuthor = "";
                ForumTypes type;
                while (tempIterator.hasNext()) {
                    Map.Entry<String, Object> currentTopic =
                        tempIterator.next();
                    switch (currentTopic.getKey()) {
                        case "topic_title":
                            title = new String((byte[])
                                currentTopic.getValue(), "UTF-8");
                            break;
```

```
                    case "topic_id":
                        topicID = (String) currentTopic.getValue();
                        break;
                    case "topic_author_name":
                        topicAuthor = new String((byte[])
                            currentTopic.getValue(), "UTF-8");
                        break;
                }
            }
            if (sticky){
                type = ForumTypes.STICKY_TOPIC;
            }else {
                type = ForumTypes.TOPIC;
            }
            TopicInterface topic = new Topic(title, topicID,
                topicAuthor, type);
            Log.i("Added topic", topic.toString());
            topicList.add((ForumObjInterface) topic);
        }
    } else if (entry.getKey().equals("can_post")){
        TempStorage.canUserPost = (Boolean) entry.getValue();
    }
}
return topicList;
}
```

L'oggetto che il metodo accetta come parametro corrisponde all'oggetto ottenuto tramite chiamata alle API di *Tapatalk*. Viene subito creata un oggetto *List*, che verrà riempita successivamente con gli oggetti che il metodo andrà ad istanziare, e un iteratore per la mappa: questo iteratore andrà ad analizzare l'*EntrySet* della mappa, andando a confrontare le chiavi con dei valori ben precisi, andando prima ad ottenere informazioni utili per l'impaginazione, quali il numero dei topic presenti in quel forum, per poi passare a trovare l'elemento della mappa identificato dalla chiave *topics*. Questo elemento è una mappa anch'esso, quindi è necessario creare un nuovo iteratore per l'*EntrySet* della mappa dei topic; dopo averlo creato, si avviano le procedure di parsing dei dati necessari per creare un oggetto, in questo caso, di tipo *TopicInterface*, che viene successivamente ri-castato a *ForumObjInterface*, un'interfaccia che viene implementata da tutti gli oggetti del model riguardante il forum, facilitando così alcune operazioni di gestione della lista multi-oggetto del forum.

Piccola parentesi sulla questione *ForumObjInterface*: questa interfaccia na-

sce dalla necessità di dover creare una lista multi-oggetto per alleggerire il codice e la gestione delle risorse da parte del sistema operativo. Così facendo, è possibile creare una lista multi-oggetto (contenente, ad esempio, oggetti di tipo *Forum* e *Topic*) e gestirla semplicemente come fosse una lista composta da un solo oggetto.

Capitolo 6

Sviluppo

6.1 Tecnologie utilizzate

6.1.1 Dispositivi mobile

Come device di testing sono stati utilizzati sia devices fisici che emulati, per rendere visualizzabile al meglio l'applicazione sotto diverse densità di schermo:

- OnePlus One, con *Lollipop* 5.1.1 (dispositivo fisico).
- Samsung SIII Mini, con *Jelly Bean* 4.1 (dispositivo fisico).
- Nexus 5, con *Lollipop* 5.1.1 (dispositivo fisico).
- Nexus One, con *Lollipop* 5.1.1 (dispositivo emulato).

L'applicazione è stata sviluppata a stretto contatto con gli amministratori di *Italiansubs.net* e con una stretta cerchia di betatesters, riuscendo così ad avere feedback importanti per la risoluzione dei major bugs che affliggevano l'applicazione.

6.1.2 Macchine fisiche

Per sviluppare l'applicazione sono state utilizzate due macchine fisiche, entrambe montanti Windows:

- PC fisso con processore Intel i7-4770k, 8 GB di RAM DDR3.
- PC portatile con processore Intel i7-3610QM, 8 GB di RAM DDR3.

6.2 IDE utilizzato

L'applicazione è stata realizzata utilizzando l'ambiente di sviluppo, sviluppato da Google, **Android Studio**.

6.3 SDK Android

L'SDK Android contiene tutti gli strumenti necessari per realizzare applicazioni per smartphone, tablet e altri vari dispositivi aventi Android come sistema operativo nativo. L'SDK Android viene fornito in bundle con l'IDE **Android Studio**, ma deve essere spesso aggiornato, visto che fornisce la possibilità agli sviluppatori di accedere a build di test di versioni del sistema operativo ancora non disponibili al pubblico.

6.4 Librerie esterne

Per avere una visione d'insieme delle già citate librerie esterne, verranno rielencate in questa sezione.

- **JDOM** usata per gestire al meglio i documenti XML ottenuti dalle API di *Italiansubs.net*.
- **aXMLRPC**, client esterno per la comunicazione XML-RPC, utilizzata per la comunicazione con le API di *Tapatalk*.
- **com.afollestad:material-dialogs** usata per ottenere delle dialog in *Material design*.
- **com.regwuxian.materialedittext** usata per cambiare lo stile delle EditText, ossia campi di testo modificabili dall'utente, in *Material design*.
- **com.github.navasmdc** usata per creare dei pulsanti in *Material Design*.
- **com.squareup.picasso** libreria molto efficiente per quanto riguarda la gestione e il cacheing delle immagini.
- **filechooser** non si tratta di una vera e propria libreria esterna, ma è un **modulo**, espansione dell'applicazione che le fornisce nuove funzioni. In questo caso, viene usata per dare la possibilità all'utente di modificare la cartella di destinazione dei download.

6.5 Linguaggi utilizzati

L'applicazione è stata completamente sviluppata in Java, linguaggio principale per lo sviluppo di applicazioni Android. Viene utilizzato anche del linguaggio XML, vista la natura delle risposte da parte delle API di *Italiansubs.net*.

Conclusioni

Lo sviluppo di questa applicazione mi ha permesso di ampliare la mia conoscenza in campo Android, grazie soprattutto alla vastità di campi che ha ricoperto lo sviluppo (client HTTP/XML-RPC, comunicazioni con API, etc. . .) Tutti i bug riscontrati durante l'implementazione dell'applicazione hanno reso anche loro lo sviluppo interessante sotto l'aspetto educativo, spronandomi sempre a cercare la soluzione più efficace al fine di risolvere il problema che l'aveva generato.

Gran parte sia del codice che della parte grafica è stato prima descritto e schematizzato su carta, permettendomi uno sviluppo più pulito e il più possibile esente da errori di gestione. La parte grafica, inizialmente, era, anche su dispositivo, molto minimale, andandosi via via a migliorare a mano a mano che lo sviluppo diventava sempre più intensivo e complicato.

L'applicazione al momento risulta pubblicata sullo Store, nonostante abbia avuto di reclami da Google per violazione di copyright, causati dalla presenza, secondo loro, di immagini coperte da copyright, ma che effettivamente non lo sono. L'applicazione è correttamente installata su più di 100 dispositivi in tutta Italia.

Sviluppi Futuri

Nonostante l'applicazione abbia già parecchie delle funzionalità maggiormente richieste dagli utenti del portale di *Italiansubs.net*, molte features sono rimaste ancora in stato embrionale, come:

- implementazione di notifiche all'aggiornamento di un topic e/o forum a cui si è partecipato o si vuole ricevere notifiche
- implementazione del pannello privato, dove l'utente può vedere tutte le sue statistiche riguardante il forum

- implementazione di specifiche funzionalità in base al grado dell'utente nel forum (traduttore, moderatore, amministratore)

Essendo un portale che frequento assiduamente, non escludo ulteriori funzionalità, via via che il portale si evolverà.

Ringraziamenti

Ringrazio i miei coinquilini, Andrea e Matteo, per avermi sopportato durante le nottate insonni passate a sviluppare l'applicazione e fixare dei bug. Ringrazio Chiara, Francesco, Tommaso, Alice e tutti gli altri membri del gruppo di betatesting dell'applicazione per avermi dato consigli su quali funzionalità implementare. Ringrazio, infine, l'intera community di *Italiansubs.net* che mi sprona a continuare lo sviluppo dell'applicazione.

Bibliografia

[1] Statista, *Number of apps available in leading app stores as of July 2015*, http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[2] Brandon Chester, *Google Announces Android M At Google I/O 2015*, http://www.anandtech.com/show/9291/google-announces-android-m-at-google-io-2015

[3] Sophie Curtis, *Android Marshmallow: Google reveals name of Android 6.0*, http://www.telegraph.co.uk/technology/google/11809163/Android-Marshmallow-Google-reveals-name-of-Android-6.0.html

[4] Wikipedia, *Android Runtime*, https://en.wikipedia.org/wiki/Android_Runtime

Elenco delle figure