La grande diffusione nell'ultimo quinquennio di smartphone (e più recentemente di tablet) dotati di moderni sistemi operativi *general-purpose* ha portato allo sviluppo di applicazioni prima impensabili e al coinvolgimento di persone precedentemente poco avvezze alla tecnologia.

In questo contesto, durante il corso di "Programmazione di Sistemi Embedded" è stato proposto lo sviluppo di un progetto commissionato dal gruppo di ricerca dell'Università di Padova di P.I.P.P.I. (Programma di Intervento Per la Prevenzione dell'Istituzionalizzazione), consistente nel creare un'applicazione per tablet che, facilitando il lavoro di raccolta dati dell'operatore sociale in visita alle famiglie, permettesse anche modi nuovi di relazionarsi con il bambino.

Successivamente, poiché i dati inseriti nell'applicazione rimanevano in locale, per far sì che esprimesse le sue piene potenzialità e fosse veramente utilizzabile dagli operatori, come supporto ai preesistenti strumenti Web, è stato da me sviluppato un sistema di sincronizzazione dei dati tra l'app Android e l'applicazione web in uso, così da poter trasferire le modifiche effettuate dall'app al server centrale e viceversa.

Nel primo capitolo viene presentata una breve introduzione a P.I.P.P.I. e all'app sviluppata durante il corso. E' oggetto del terzo capitolo la descrizione dei concetti chiave e degli strumenti utilizzati è oggetto del secondo capitolo, necessaria per la piena comprensione delle tecniche e delle soluzioni implementate per la realizzazione dell'obiettivo, trattati nel terzo ed ultimo capitolo.

Sommario

Capitolo 1: Cosa è stato fatto

1.1 Cos'è P.I.P.P.I.

Nell'ultimo quarto di secolo l'attenzione ai temi della protezione e tutela dell'infanzia e dell'adolescenza è notevolmente cresciuta, rinforzando la sensibilizzazione dell'opinione pubblica, volta a far emergere le situazioni di maltrattamento e trascuratezza dei bambini e dei ragazzi[1].

P.I.P.P.I., acronimo di Programma di Intervento Per la Prevenzione dell'Istituzionalizzazione, si inserisce in questo contesto proponendosi di aiutare le famiglie definite "*negligenti*", sostenendole intensivamente: con il supporto offerto dagli operatori, i genitori imparano modi nuovi e funzionali di relazionarsi con i propri figli, passando più tempo insieme e gestendo il quotidiano.

Nello specifico, P.I.P.P.I. è un programma di intervento finanziato dal Ministero del Lavoro e delle Politiche Sociali, rivolto a 10 nuclei familiari per ogni città aderente, con figli da 0 a 11 anni, finalizzato a ridurre il rischio di allontanamento dei figli, causato dalla trascuratezza dei genitori e dalle carenze educative, o a limitarlo nel tempo, nel caso in cui l'allontanamento fosse inevitabile, facilitando così la riunificazione familiare.
Il programma si inserisce nella cerchia dei progetti di *Preservation Families* in un momento storico decisamente poco favorevole, in cui le risorse del *welfare* diminuiscono bruscamente.
Appare quindi evidente l'urgenza di implementare programmi di intervento *evidence based* in modo che sia gli operatori che le famiglie possano documentare con precisione gli esiti dell'intervento in atto e correlare tali esiti (outcome) alle risorse investite (input) e al processo generale dell'intervento (output), usando i finanziamenti in modo da rispondere in maniera pertinente ai bisogni delle famiglie.

[1] Il presente paragrafo si compone di una sintesi della presentazione del programma P.I.P.P.I. pubblicata in [1].

Il programma si basa essenzialmente sui seguenti 6 punti.

1. Realizzazione di una equipe multidisciplinare di professionisti coinvolta attivamente nelle scelte relative al programma.
2. Pieno coinvolgimento delle famiglie e dei bambini, veri protagonisti di questo programma attuato in un'ottica bio-ecologica dello sviluppo umano (sul modello di quella proposta da Bronfenbrenner).
3. Coinvolgimento di poche famiglie per volta, in modo da poterle seguire in maniera approfondita e stabile.
4. Capacità di coniugare la pratica dell'intervento e la pratica della valutazione in modo che gli operatori possano valutare i progressi del loro intervento.
5. Utilizzo di strumenti che permettano di confrontare la valutazione iniziale del bambino con le valutazione in itinere nei diversi momenti di intervento: ad esempio T0 sarà la fase iniziale, T1 la fase intermedia e T2 la fase conclusiva.
6. Sperimentazione di metodi di interazione tra il mondo scolastico del bambino e il mondo sociale.

L'intervento viene monitorato grazie alla somministrazione di questionari atti a verificare la soddisfazione genitoriale, i bisogni dei bambini (o degli adolescenti), il sostegno sociale eventualmente ricevuto e il funzionamento del nucleo familiare.
P.I.P.P.I. si propone come una *ricerca-formazione-intervento* partecipata, finalizzata ad assicurare un rigoroso sviluppo della ricerca senza perdere di vista gli scopi dell'intervento. I ricercatori e i professionisti possono, dunque, contribuire all'integrazione del programma nel quadro standard delle prassi dei servizi di tutela dei minori, e far sì che gli strumenti utilizzati nella sperimentazione entrino a far parte del *modus operandi* ordinario dei servizi, garantendone così la piena replicabilità.

In definitiva, il presente progetto ha come *focus* principale la "protezione della relazione genitore-figlio", piuttosto che la sola protezione del bambino, e l'individuazione di possibili strade che permettano di evitare l'allontanamento, aiutando i genitori a prendersi adeguatamente cura dei propri figli e a rispondere ai loro bisogni di sviluppo.
Le famiglie negligenti presentano spesso problematiche e bisogni diversi: molte di loro esibiscono serie difficoltà di carattere sociale e relazionale (quali povertà,

esclusione dal mondo del lavoro, basso livello di istruzione, violenza coniugale, frequenti traslochi che portano all'isolamento). Malgrado il numero preoccupante di queste famiglie che arriva ai servizi, le metodologie di intervento non hanno registrato un chiaro e consapevole cambiamento.
Sono cambiate e migliorate le modalità di accoglienza dei bambini fuori dalla famiglia, ma il problema sembra lontano dall'essere risolto.

In questo contesto si inserisce RPM, la "*scheda di Rilevazione, Progettazione e Monitoraggio del benessere del bambino/a e della sua famiglia*". L'idea nasce nel 2010 come percorso di ricerca-intervento partecipato fra il Dipartimento di Scienze dell'Educazione dell'Università di Padova e altre realtà del territorio. Il progetto di ricerca, denominato *"A cosa serve allontanare i minori dalle famiglie di origine?*", è finalizzato a supportare e accompagnare il lavoro degli operatori che si occupano di protezione e cura dei bambini, sia nella fase di analisi della situazione che in quella più prettamente progettuale (definizione degli obiettivi, delle azioni di intervento e degli indicatori di risultato).
A partire dal 2011, la scheda RPM è stata adottata anche da P.I.P.P.I., portando alla realizzazione dell'omonima applicazione web RPMonline, a cura del Centro Servizi Informatici d'Ateneo, che viene tuttora utilizzata dagli operatori come piattaforma di reportistica.

Il progetto è stato finanziato dal Ministero per ulteriori due anni (2013-2014) durante i quali si è provveduto ad un radicale rifacimento e semplificazione di RPMonline (d'ora in poi spesso chiamato semplicemente RPM), sul quale si è basato il lavoro che descriveremo in questa sede.

1.2 Cosa abbiamo fatto

Durante il corso di “Programmazione di Sistemi Embedded” tenuto dal Professor Carlo Fantozzi è stata data la possibilità agli studenti di poter sviluppare dei progetti speciali proposti da committenti reali.
Uno di questi era P.I.P.P.I., scelto dal gruppo composto da me e due colleghi.
Il committente richiedeva la trasposizione come app per dispositivi mobili delle funzioni utilizzate dagli operatori in RPM. E’ stato quindi scelto di sviluppare un’applicazione per tablet Android per due ragioni: la densità di informazioni da visualizzare in alcune schermate e il costo accessibile dei tablet Android.
Successivamente l’applicazione ha subito alcune aggiunte come parte del lavoro di tesi di uno dei miei colleghi, quindi qui illustrerò il prodotto finale.

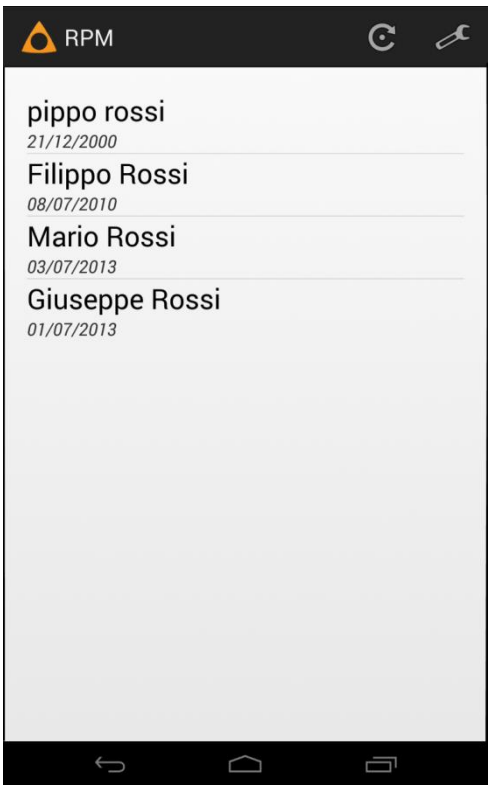


Figura 1: Lista dei bambini

Figura 2: Lista dei tempi

In Figura 1 è possibile vedere la schermata iniziale dell’applicazione: presenta la lista dei bambini, con relativa dati di nascita, seguiti dall’operatore; selezionando un elemento si apre la lista dei tempi (Figura 2).

Ogni tempo rappresenta un periodo di lavoro dell’équipe multidisciplinare con la famiglia (bambini e genitori) . Per ogni tempo si descrivono la situazione e le problematiche presenti nonché eventuali punti di forza, per fissare quindi degli obiettivi che dovrebbero essere raggiunti entro il tempo stabilito. Solitamente ad ogni bambino si assegnano tre tempi.

Selezionando un tempo si accede al *Mondo del Bambino* (Figura 3)

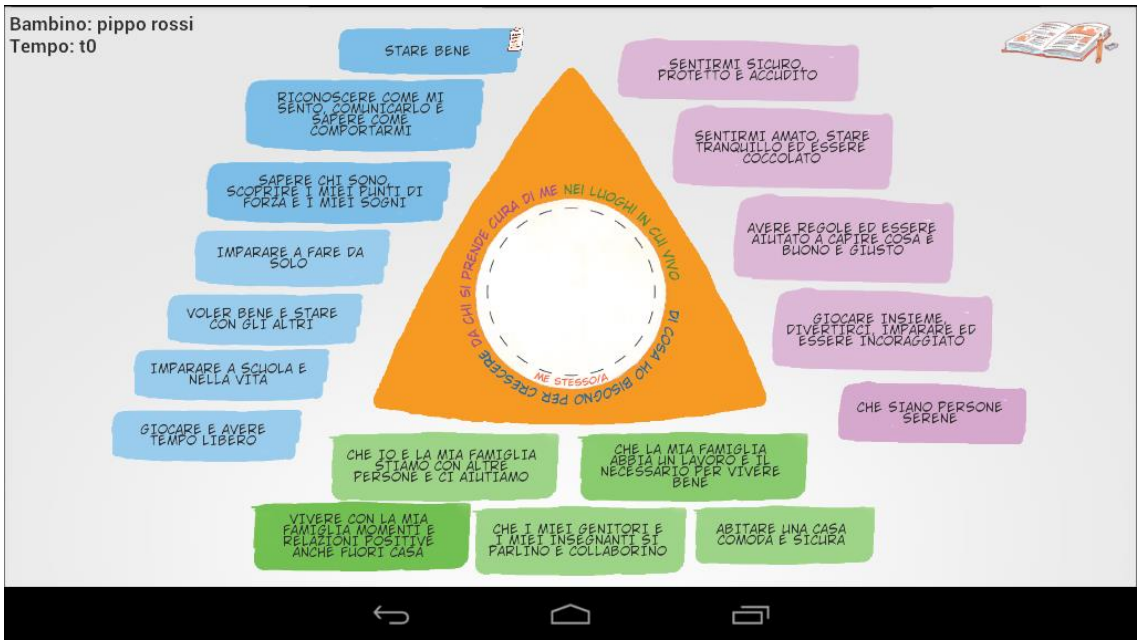


Figura 3: Il Mondo del Bambino

"*Il Mondo del Bambino*", nucleo del progetto P.I.P.P.I. nonché dell'app stessa, descrive di cosa il bambino ha bisogno per crescere e chi si prende cura di lui nei luoghi in cui vive, delineando una serie di "*sottodimensioni*" disposte sui tre lati secondo le seguenti dimensioni: bambino (a sinistra), ambiente (in basso), famiglia (a destra).

Ogni sottodimensione (nel resto del progetto definita ***nuvola***) aprirà una schermata come in Figura 4.

Lo stato di completezza della compilazione, come la presenza dell'*assessment* o del livello attuale, influenzerà l'aspetto della sottodimensione, che diventerà di un colore più acceso man mano che viene compilata e comincerà a fluttuare qualora la compilazione fosse completa di tutti i campi.

Ad ogni sottodimensione è possibile associare una progettazione tramite l'apposito pulsante in figura. La progettazione consiste (vedi Figura 5)

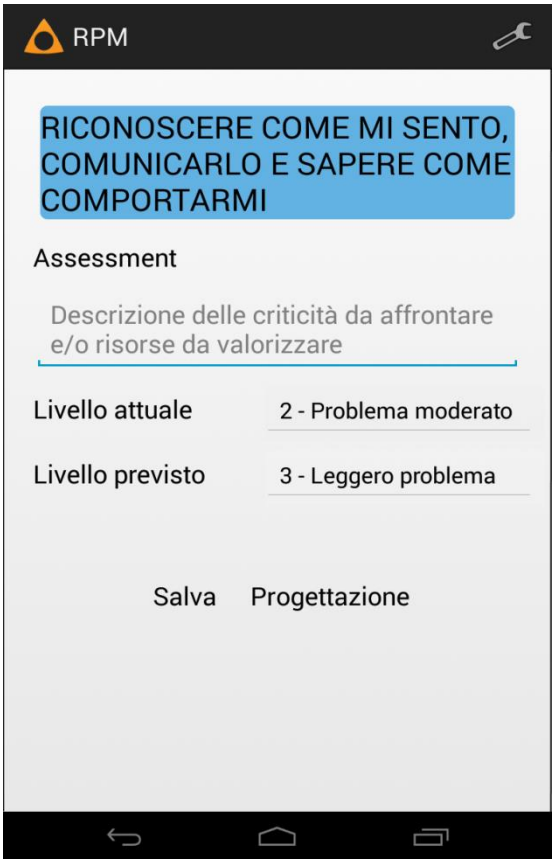


Figura 4: Modifica di una sottodimensione

nella definizione di metodologie e attori per raggiungere un determinato obiettivo. Se viene compilata la progettazione comparirà un foglietto appeso all'angolo in alto a destra della relativa sottodimensione (vedasi "Stare bene" in Figura 3).

Figura 5: Compilazione di una Progettazione

Selezionando il libro dal *Mondo del Bambino* (Figura 3 in alto a destra) è possibile passare alla lista degli incontri tra operatori e bambino e/o membri della famiglia. Diversamente alle parti dell'applicazione viste finora, dove il controllo dell'utente si limita alla modifica di sezioni statiche preesistenti, dalla lista degli incontri (qui non mostrata) è possibile creare o cancellare un determinato incontro; vediamo un esempio di compilazione in Figura 6.

Figura 6: Compilazione di un Incontro

1.3 Cosa vogliamo fare

L'app durante il corso è stata sviluppata per lavorare completamente in locale. Questo vuol dire che l'operatore, dopo essere stato dalla famiglia e aver memorizzato dei dati nell'applicazione, arrivato a casa doveva recuperarli e trascriverli in RPM.

Obiettivo fondamentale era quindi integrare l'app con il *back-end*[2] di RPM, in modo che i dati inseriti dall'app fossero visibili anche in RPM, e viceversa, senza dover trascriverli manualmente.

Inizialmente si era optato per la soluzione più immediata, ovvero un reperimento *real-time* dei dati: cliccando sul nome di un bambino l'app effettua la richiesta al server, i dati ricevuti vengono processati e infine la lista dei tempi viene visualizzata dall'utente. Nel dispositivo non viene utilizzato alcun database né vengono memorizzati dati in modo permanente: questo semplifica di molto l'implementazione ma obbliga l'operatore ad avere una connessione attiva ad ogni operazione dell'app.
I tablet a basso costo sono venduti senza supporto per *SIM*[3] e allo stesso tempo non è neanche sempre possibile contare sulla presenza di una connessione Wi-Fi disponibile (o del *tethering*[4]). Si è deciso quindi, quasi a lavoro completato, di riutilizzare i metodi di comunicazione sviluppati per RPM per popolare un database permanente nel dispositivo in modo che l'app avesse sempre disponibile una copia dei dati, aggiungendo infine appositi metodi di sincronizzazione per salvare nel server le modifiche effettuate nell'app e viceversa.

[2] In informatica, il front-end è responsabile di collezionare i dati in input dall'utente e processarli secondo una specifica che il back-end possa usare. Il front-end è l'interfaccia tra l'utente e il back-end. In questo caso il front-end è il sito web in cui gli operatori inseriscono i dati, il back-end è il database sottostante che contiene i dati strutturati in tabelle.
[3] Subscriber Identity Module, smart card che memorizza l'identificativo unico dell'abbonato e permette all'operatore telefonico di associare il dispositivo mobile in cui è inserita la SIM al profilo di un determinato cliente.
[4] Utilizzo dello smartphone e della sua connessione dati come punto d'accesso (solitamente wireless) per offrire accesso alla rete ai dispositivi che ne sono provvisti.

Capitolo 2: Strumenti

2.1 Concetti chiave

HTTP

L'HyperText Transfer Protocol (protocollo di trasferimento di un ipertesto) è usato come principale sistema per la trasmissione d'informazioni sul web. HTTP funziona come protocollo di *richiesta-risposta* nel modello *client-server*. Un web browser, per esempio, potrebbe essere il *client* e un'applicazione in esecuzione sul computer ospitante un sito web potrebbe essere il *server*. Il *client* invia un messaggio di richiesta HTTP al *server*: quest'ultimo, che fornisce una serie di risorse come file HTML e altri contenuti, o esegue altre operazioni per conto del client, ritorna un messaggio di risposta al *client*.

La specifica *HTTP/1.0* (1996) [3] fornisce i seguenti metodi di richiesta:

- **GET**: richiede una rappresentazione della risorsa specificata nell'URI[5]; le richieste GET dovrebbero solo restituire dei dati e non avere altri effetti.
- **HEAD**: richiede la stessa risposta che sarebbe restituita con una GET ma viene restituito solamente l'*header* senza il *body*[6].
- **POST**: richiede che il server accetti l'entità contenuta nella richiesta come nuovo subordinato della risorsa specificata nell'URI, ovvero comporta una modifica dello stato della risorsa. Viene utilizzato ad esempio per inviare delle credenziali d'accesso o un messaggio in un forum.

La specifica *HTTP/1.1* (1997) aggiunge 5 nuovi metodi (*OPTIONS, PUT, DELETE, TRACE* e *CONNECT*) la cui descrizione verrà tralasciata non facendo parte della nostra sfera d'interesse.

Si noti che, anche se le POST dovrebbero essere usate solo per richieste che comportano una modifica dello stato del server, oggigiorno sono spesso usate anche come sostituto delle GET per due motivi:

[5] Uniform Resource Identifier è una stringa che identifica univocamente una risorsa generica. L'URL è un URI che identifica risorse web.

[6] Un pacchetto HTTP è composto da 3 parti: riga di richiesta/stato (richiesta/risposta), header (informazioni aggiuntive) e body (corpo del messaggio).

- Gli URL hanno una lunghezza massima, quindi il numero massimo di elementi esprimibili nella query inserita nell'URL è limitato. Mentre *Apache HTTP Server* può gestire URL lunghi fino a 4000 caratteri, *Microsoft Internet Explorer* supporta al massimo 2048 caratteri.
- Anche se si sta usando una connessione crittografata, rendendo quindi i pacchetti protetti durante il viaggio tra *client* e *server*, solitamente entrambi memorizzano nei log gli indirizzi in chiaro, rendendo così visibili a malitenzionati che avessero accesso al sistema eventuali dati sensibili contenuti nell'URL.

HTML

L'HyperText Markup Language è il linguaggio di *markup*[7] utilizzato nella formattazione di documenti ipertestuali sotto forma di pagine web.

Il contenuto delle pagine web consiste in un documento HTML contenente una struttura ad albero annidato di elementi definiti da marcatori detti *tag*.

Un elemento HTML (spesso chiamato anche tag) contiene un tag di apertura, uno o più attributi di tale elemento con i rispettivi valori (che possono definirne colore, stile, posizione, etc), il contenuto informativo (che può essere una stringa di testo o una struttura di altri elementi) e infine un tag di chiusura.

Un documento comincia con la definizione del tipo di documento, la quale segnala al browser l'URL delle specifiche HTML utilizzate per il documento, che permettono l'identificazione delle regole di interpretazione per visualizzare correttamente la pagina.

Segue poi la struttura degli elementi che compongono la pagina, compresa tra i tag <*html*> e <*/html*>, suddivisa in due sezioni:

- *Header*, o intestazione, contiene alcune informazioni di controllo solitamente non visualizzate;
- *Body*, o corpo, contiene la parte informativa vera e propria che sarà visualizzata dal browser.

[7] Un linguaggio di markup consiste in un insieme di regole che descrivono i meccanismi di rappresentazione di un testo utilizzabile su più supporti. Tali regole sono esprimibili appunto tramite l'uso di marcatori (o espressioni codificate).

APEX

Oracle Application Express (precedentemente Oracle HTML DB) è un ambiente di sviluppo di applicazioni web basate su database [2].

APEX può essere utilizzato per sviluppare applicazioni per desktop e dispositivi mobili velocemente utilizzando soltanto un web browser, senza conoscenze approfondite di programmazione.

RPM è stato sviluppato utilizzando APEX.

Web service

Secondo la definizione data dal World Wide Web Consortium (W3C), un web service è "*un sistema software progettato per supportare l'interoperabilità tra diversi elaboratori su di una medesima rete ovvero in un contesto distribuito*"[6].

Tale risultato viene raggiunto associando all'applicazione un'interfaccia software che espone all'esterno il servizio associato, permettendo ad altri sistemi di interagire con l'applicazione stessa inviando determinati messaggi di richiesta formattati secondo lo standard XML attraverso protocolli web (quasi sempre HTTP).

Il server tipicamente risponde restituendo un documento HTML, XML o JSON[8] ma anche un qualsiasi altro tipo di risorsa.

Questa caratteristica permette a diversi sistemi operativi e diversi linguaggi di programmazione di essere interoperabili tra di loro grazie all'uso di questi standard; semplifica inoltre la manutenzione di software comunicanti tra loro, gli sviluppatori infatti devono solamente preoccuparsi di mantenere invariate le interfacce esterne.

Fino a qualche anno fa l'approccio principale per un web service era quello basato sul protocollo *SOAP*, che espone un insieme di metodi richiamabili da remoto da parte del client, utilizzando richieste in formato XML strutturate secondo ben determinate specifiche del protocollo.

Oggigiorno invece si tende a preferire *REST*, un'architettura che si limita a definire una serie di componenti e principi chiave, senza specificare i dettagli dell'implementazione [16]. Al contrario di SOAP, dove ogni operazione è portata a termine utilizzando un determinato metodo, in REST ogni singola risorsa è identificata dal proprio URI e le operazioni su di essa si compiono utilizzando alcune delle canoniche richieste HTTP (GET, POST, PUT, DELETE). Il risultato delle singole operazioni dipende però dall'implementazione scelta dallo

[8] Formato adatto per l'interscambio di dati. Si è imposto negli ultimi anni come alternativa a XML grazie alla sua semplicità d'uso in applicazioni JavaScript/AJAX.

sviluppatore, essendo REST solamente uno stile di architettura e non un protocollo.

App

Per app si intende un'applicazione sviluppata per dispositivi mobili moderni, come smartphone e tablet. In particolare, un'app per Android è composta da una serie di *activity*, che potremmo definire come la singola schermata che svolge una particolare funzione all'interno dell'applicazione. In Android esistono poi altri componenti, come i *service*, che sono eseguiti in background e svolgono operazioni che si protraggono per diverso tempo (come download, riproduzione di musica, etc.), nonché altri componenti di gestione degli eventi di sistema e di scambio di dati tra applicazioni diverse.

2.2 Strumenti e librerie utilizzate

Android SDK

Contiene le librerie di Android, codice sorgente, esempi e documentazione.

Eclipse

Eclipse è un IDE (Integrated Development Environment) usato principalmente nello sviluppo e testing di applicazioni Java.

ADT

Android Development Tools è un plugin per Eclipse che permette di integrare una serie di tools dedicati allo sviluppo di applicazioni Android, ad esempio: wizard guidato per la creazione di nuovi progetti Android; editor grafico XML per progettare interfacce utente; analisi statica del codice; un pannello di debug utilizzabile sia con dispositivi fisici che con dispositivi emulati; distribuzione e firma dei pacchetti *.apk* dell'applicazione.

jsoup

jsoup è una libreria Java che permette di estrarre e manipolare dati da codice HTML prelevato da un URL o una stringa, utilizzando un attraversamento DOM[9] (gerarchia ad albero) o dei selettori CSS[10] (in grado di trovare uno o più elementi per tipo, attributo, etc.) [8].

[9] Document Object Model: modello a oggetti del documento, è una rappresentazione ad albero dei documenti strutturati, ottenuto per esempio dai browser dall'interpretazione di un documento HTML.
[10] Il CSS è un linguaggio utilizzato per definire la formattazione di documenti HTML, XHML e XML.

HttpFox

HttpFox è un plugin per Mozilla Firefox che monitora e analizza tutto il traffico HTTP in entrata ed uscita tra il browser e Internet, visualizzando in maniera chiara e strutturata l'intero contenuto dei pacchetti scambiati, compresi i cookie[11] utilizzati [13].

HttpClient

Parte del progetto Apache HttpComponents [9], una serie di componenti Java di basso livello focalizzati su HTTP e protocolli associati, HttpClient è un agente *HTTP/1.1* finemente personalizzabile che fornisce strumenti di autenticazione e gestione di connessioni HTTP, capace di gestire tutti i metodi di richiesta visti precedentemente; attualmente (Ottobre 2013) si trova alla versione 4.3.

La libreria fa parte del framework di Android fin dagli albori di questo: sembrerebbe[10] che Google abbia preso agli inizi del 2009 uno *snapshot*[12] casuale dell'allora ramo in sviluppo della versione 4.0, poi rilasciata stabilmente nell'Agosto dello stesso anno, ma da allora non è più stata aggiornata; tuttavia le funzionalità necessarie in questa sede si sono rilevate stabili quindi non è stato necessario integrare esternamente una versione aggiornata della libreria.

Per questo motivo Google stessa consiglia di utilizzare la loro libreria *HttpURLConnection*, più aggiornata, ma che offre minori possibilità di personalizzazione e ho riscontrato grossi problemi con il reindirizzamento automatico utilizzato da APEX: da qui la scelta di utilizzare la soluzione di Apache.

Gson

Gson è una libreria Java sviluppata da Google che permette di convertire oggetti Java nella loro rappresentazione JSON [15]. Viceversa, da una stringa JSON è possibile ottenere l'oggetto rappresentato.

Al contrario di soluzioni native come la serializzazione Java, molto lenta e che produce dati di grandi dimensioni anche per oggetti con poche variabili native, Gson è capace di produrre, tranne rare eccezioni, in modo efficiente una stringa sintetica rappresentante perfettamente lo stato di un oggetto.

[11] Stringhe di testo nei browser utilizzate dai siti web per memorizzare informazioni.
[12] Codice sorgente della libreria in un dato momento temporale.

Capitolo 3: Implementazione progetto

In questo capitolo descriveremo come sono stati utilizzati gli strumenti descritti precedentemente per implementare la sincronizzazione tra app e server.

Oggigiorno, quando il fornitore di un servizio vuole permettere ad applicazioni esterne collegate in rete di scambiarsi dati e poter effettuare richieste *custom* viene implementato un web service, potendo così anche sfruttare una serie di librerie che permettono la gestione ad alto livello delle più comuni implementazioni di web service.

APEX permette di abilitare nativamente un web service ma, per la difficoltà del Centro Servizi Informatici di Ateneo di potersi dedicare allo sviluppo in tempi brevi, come sarebbe stato necessario, si è dovuto optare per l'unica alternativa possibile, ovvero la comunicazione diretta a basso livello tra client e server tramite richieste HTTP.

Si è quindi scelto di usare HttpFox per ispezionare i pacchetti scambiati tra browser e server e poterne effettuare così un *reverse-engineering* per replicarne la struttura nell'app.

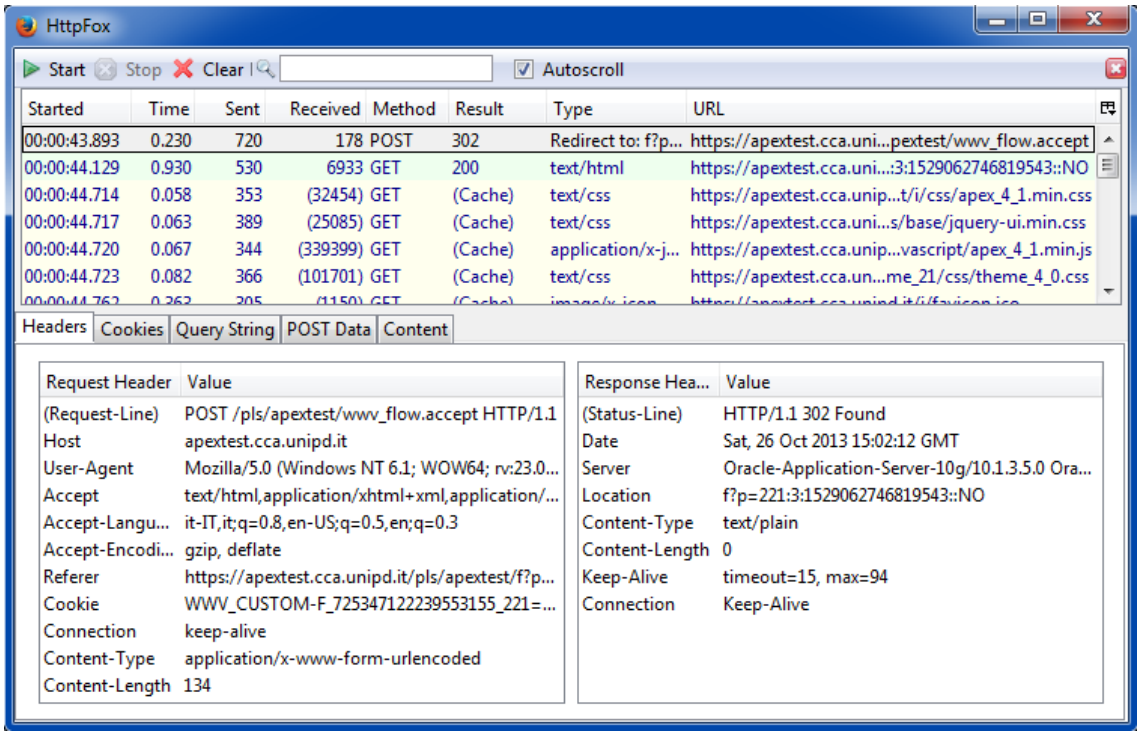
3.1 Utilizzo di HttpFox

HttpFox è un tool tanto semplice quanto efficace.
Interponendosi a livello di estensione del browser, il traffico protetto *SSL/TLS*[13] viene naturalmente visualizzato in chiaro, a differenza di applicativi come *WireShark* che ponendosi a livello di applicazione di sistema devono essere configurati opportunamente (e i cui scopi vanno ben oltre le necessità qui richieste).

La schermata del plugin (Figura 7a) si presenta con due pulsanti per iniziare e fermare la cattura del traffico e una lista delle richieste inviate con relativa risposta (quest'ultima non presente nel caso di risorse recuperate dalla cache del browser).

[13] Transport Layer Security (TLS) e il suo predecessore SSL (Secure Sockets Layer) sono protocolli crittografici sviluppati per fornire sicurezza di comunicazione tramite Internet. Il funzionamento può essere riassunto in 3 fasi: negoziazione tra le parti dell'algoritmo da utilizzare; scambio delle chiavi e autenticazione; cifratura simmetrica e autenticazione dei messaggi.

Per ogni elemento della lista vengono visualizzati: *header* della richiesta, *header* della risposta, cookie inviati e ricevuti, eventuali parametri aggiuntivi della richiesta (contenuti nell'URL), contenuto della richiesta se è una POST, contenuto del *body* della risposta.



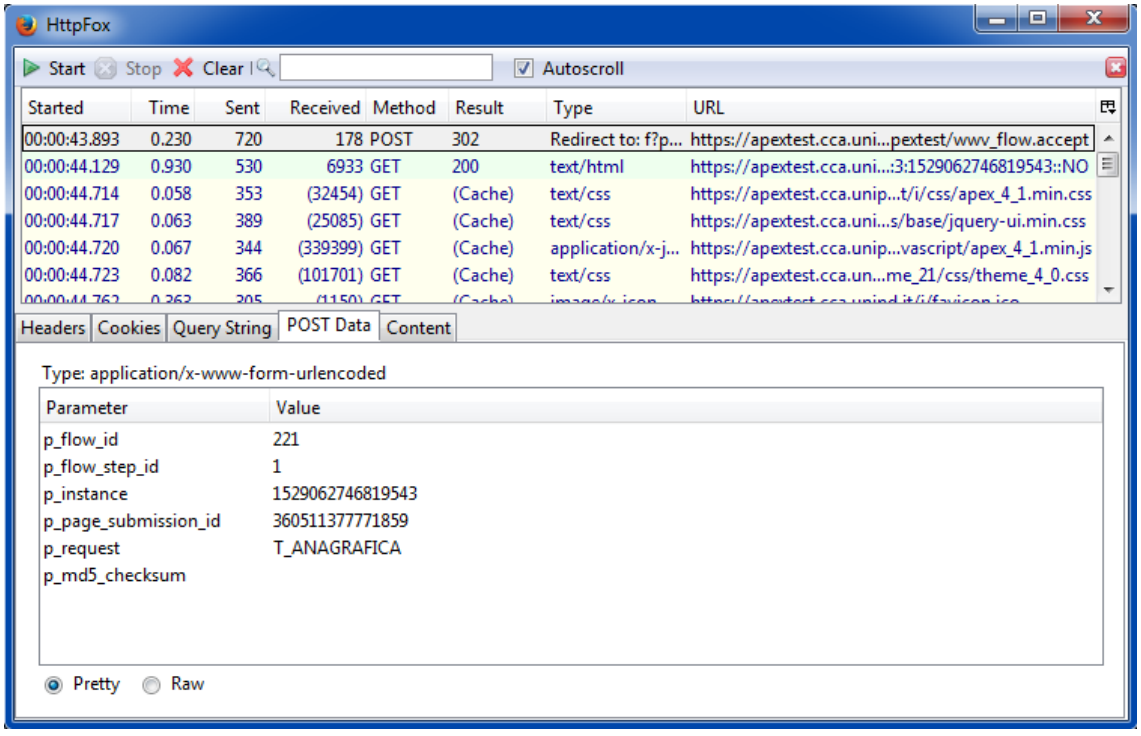


Figura 7a: Schermata di HttpFox; Figura 7b: Contenuto di una POST

Nel caso di una GET il body della risposta conterrà il codice HTML (se abbiamo richiesto una pagina web) o un altro tipo di dati, nel caso di una POST la risposta potrebbe contenere una risorsa come una GET o essere vuota.
In Figura 7b è possibile esaminare un esempio di richiesta POST.

3.2 Utilizzo di jsoup

jsoup permette di effettuare facilmente il *parsing*[14] di una pagina HTML fornita come stringa o direttamente come URL.

Per effettuare il parsing a partire da una stringa Java, ad esempio, si utilizzerà il seguente metodo statico:

```
String html = "<html><head><title>First parse</title></head>"
  + "<body><p>Parsed HTML into a doc.</p></body></html>";
Document doc = Jsoup.parse(html);
```

Se invece vogliamo ottenere il documento HTML direttamente dall'URL:

```
Document doc = Jsoup.connect("http://example.com/").get();
```

L'oggetto *Document* contiene tutti gli elementi della pagina, disposti in un albero mantenendone la struttura gerarchica originaria.

L'oggetto *Element* è il fulcro centrale della libreria: esso infatti rappresenta un elemento HTML con relativa sottostruttura (vedasi quanto detto a pagina 10); a partire da esso è possibile estrarre dati ed attraversare il grafo dei nodi.

Document è una semplice sottoclasse di Element con l'aggiunta di alcuni metodi specifici per la radice di una pagina HTML, tra cui un metodo per ottenerne il titolo e due metodi per ottenere l'Element di *header* o *body* della pagina.

Ricordando che un Document altro non è che l'Element radice di tutto l'albero, vediamo ora che metodi sono applicabili su un Element per attraversare la struttura.

- ***getElementById*(String id)**: individua l'elemento con l'id indicato. L'id è un semplice attributo che deve però assumere per ogni elemento un valore univoco in tutto il documento HTML.

[14] Identificazione di uno schema che permette di riconoscere ed estrarre determinati dati.

- ***getElementsByTag*(String tag)**: restituisce gli elementi di un certo tipo, ovvero quelli con il tag di apertura coincidente a quello da noi indicato (vedasi pagina 10).
- ***getElementsByClass*(String className)**: restituisce gli elementi con l'attributo *class* che assume il valore specificato.
- ***getElementsByAttribute*(String key)**: restituisce gli elementi che contengono un attributo così denominato. Esistono inoltre metodi associati che permettono di trovare elementi con attributi che iniziano, terminano o contengono la stringa desiderata; similmente esistono metodi che permettono di individuare elementi con attributi che assumono un valore desiderato (o soddisfano una certa espressione).

Seguono i metodi per attraversare la struttura.

- ***siblingElements*()**: restituisce i fratelli dell'elemento.
- ***firstElementSibling*(), *lastElementSibling*()**
- ***nextElementSibling*(), *previousElementSibling*()**
- ***parent*()**: restituisce il genitore.
- ***parents*()**: restituisce il genitore e tutti gli ascendenti fino alla radice del documento compresa.
- ***children*()**: restituisce tutti i figli.
- ***child*(int index)**: restituisce il figlio con l'indice indicato.

Un metodo più potente per trovare elementi consiste nell'uso dei selettori. Un selettore consiste in una stringa con una sintassi simile a quella utilizzata da CSS[10] e jQuery[15].

Vediamo velocemente i selettori principali:

- **tagname**: trova gli elementi per tag;
- **#id**: trova gli elementi per id;
- **.class**: trova gli elementi per classe;
- **[attr], [^attr]**: elementi con nome dell'attributo rispettivamente coincidente o con tale prefisso;
- **[attr=value]**: elementi con attributo che assume il valore dato;
- **[attr^=value], [attr$=value], [attr*=value]**: elementi con valore dell'attributo che inizia per, finisce o contiene la stringa data.

[15] Libreria JavaScript molto utilizzata che presenta navigazione di documenti HTML, gestione di eventi, animazioni e altre funzioni molto usate nelle pagine web moderne.

Vediamo ora alcune combinazioni di selettori:

- **ancestor child**: trova gli elementi *child* discendenti dall'elemento *ancestor;*
- **parent > child**: elementi *child* con genitore diretto *parent;*
- **siblingA + siblingB**: trova gli elementi *siblingB* con fratello immediatamente precedente *siblingA.*

Tutti i selettori possono essere combinati tra loro, da qui la loro potenza rispetto ai singoli metodi visti precedentemente. Facciamo alcuni esempi.

"a[href].highlight" trova gli elementi di tipo a con attributo href e classe highlight, ad esempio *Cliccami*.
"td[headers=PR_PROBLEMA_RISORSE] > table.radiogroup" trova gli elementi *td* con attributo *headers* che assume il valore specificato, direttamente discendenti da elementi *table* di classe *radiogroup*.

Esistono infine alcuni pseudo selettori che permettono di trovare elementi che contengono o non contengono elementi che soddisfano un certo selettore, o che corrispondono ad una espressione regolare.

I selettori si applicano utilizzando il metodo ***select*(String cssQuery)** della classe *Element*. Il metodo, come tutti gli altri metodi visti inizialmente, restituisce la lista di elementi come un oggetto di tipo *Elements*, implementazione dell'interfaccia *List* di Java.

Per estrarre dati da un oggetto Element si utilizzano i seguenti metodi principali:

- ***text*()** : restituisce il testo puro contenuto tra tag di apertura e tag di chiusura
- ***attr*(String name)** : restituisce il valore dell'attributo *name*
- ***html*()** : restituisce il codice HTML racchiuso tra tag di apertura e chiusura
- ***outerHtml*()** : restituisce il codice HTML esterno di definizione dell'elemento

Esistono infine metodi che consentono di aggiungere, modificare ed eliminare attributi dell'elemento e metodi che permettono di modificare l'albero stesso, aggiungere o rimuovendo elementi secondo la parentela desiderata.
Questo consente, se desiderato, di usare jsoup per modificare o persino creare da zero un documento HTML in modo procedurale con una chiara gerarchia.

3.3 Utilizzo di HttpClient

Prima di poter utilizzare la libreria HttpClient è necessario inizializzare e configurare il client HTTP che sarà utilizzato per effettuare le richieste.

La connessione tra client e server APEX dev'essere protetta, quindi è necessario creare una connessione *TLS* che riconosca come valido il certificato del server. Non bisogna inoltre dimenticare di abilitare la memorizzazione automatica dei cookie scambiati, poiché sono fondamentali per mantenere i dati identificativi della sessione e permettere al server di riconoscere le nostre richieste.
Il codice dettagliato dell'implementazione può essere letto nell'appendice.

Ora che abbiamo un client in grado di comunicare correttamente con RPM possiamo vedere come effettuare una **GET**:

```
1. HttpRequestBase request = new HttpGet(url);
2. HttpResponse response = httpClient.execute(request);
3. HttpEntity entity = response.getEntity();
4. String html = EntityUtils.toString(entity);
5. String responseStatus = response.getStatusLine().toString();
```

Commentiamo il codice riga per riga.

1. Viene creata la richiesta con l'URL specificato.
2. Viene eseguita la richiesta con il nostro client e il programma rimane in attesa finchè non arriva la risposta.
3. Ottiene il messaggio (*entity*) contenuto nella risposta, se esiste. Nel nostro caso si tratterà sempre di codice HTML, ma potrebbe essere una qualunque risorsa.
4. Si estrae la rappresentazione come stringa del messaggio. Quest'ultimo viene "*consumato*", ovvero rilasciato dalla memoria e reso inutilizzabile.
5. Opzionalmente si può estrarre anche il codice di risposta per controllare che l'operazione sia andata a buon fine (nel qual caso il server risponderà 200, in caso contrario con un messaggio che può andare da 100 a 500).

Nel caso di una **POST** dobbiamo innanzitutto impostare i parametri contenuti nella richiesta; se prendiamo come esempio la POST di Figura 7b (richiesta dell'elenco dei bambini), essa si tradurrà nel seguente oggetto.

```
List<NameValuePair> params = new ArrayList<NameValuePair>();
params.add(new BasicNameValuePair("p_flow_id", "221"));
params.add(new BasicNameValuePair("p_flow_step_id", "1"));
params.add(new BasicNameValuePair("p_instance", 1529062746819543));
params.add(new BasicNameValuePair("p_page_submission_id", 360511377771));
params.add(new BasicNameValuePair("p_request", "T_ANAGRAFICA"));
```

Impostati i parametri della richiesta, possiamo ora creare ed inviare la richiesta vera e propria:

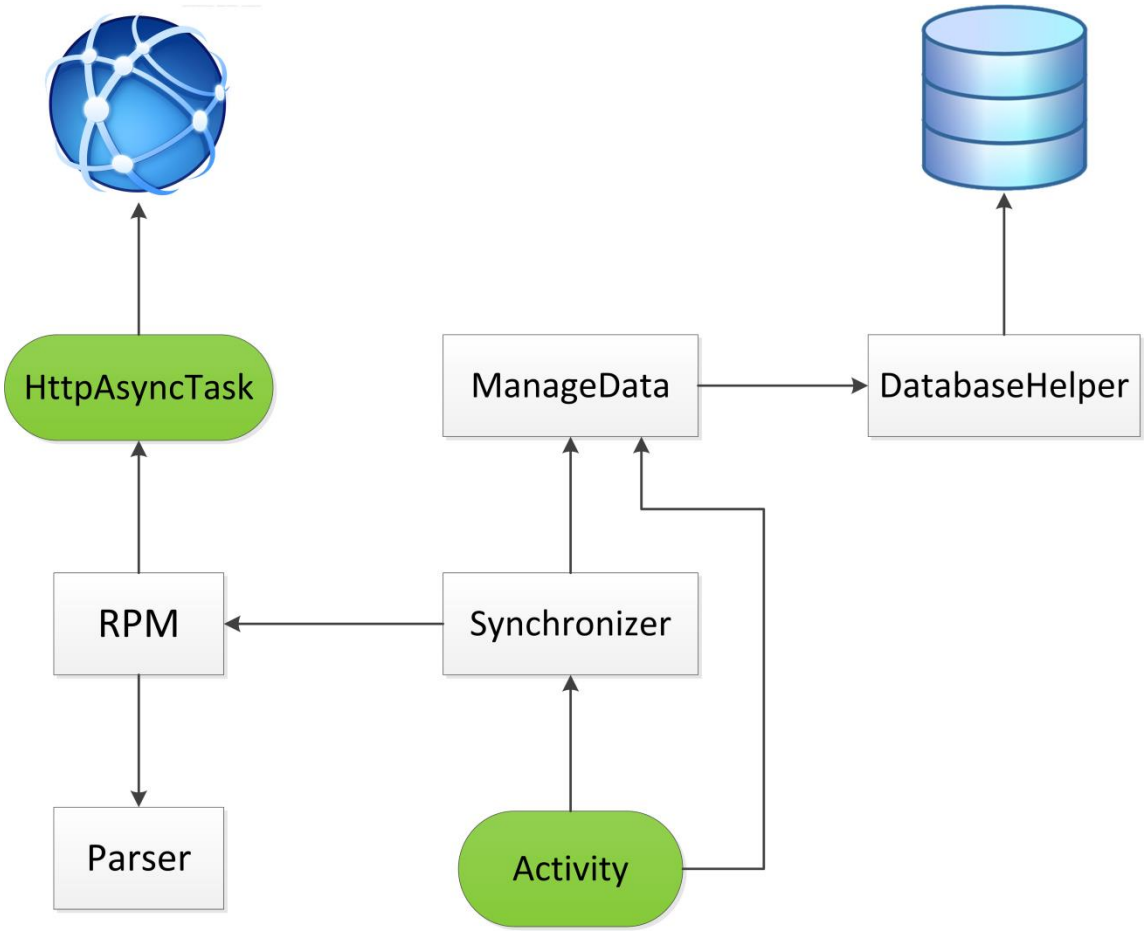
```
1. HttpRequestBase request = new HttpPost(url);
2. request.setEntity(new UrlEncodedFormEntity(params, "UTF-8"));
3. HttpResponse response = httpClient.execute(request);
```

Viene creata la richiesta indicando l'URL del web service a cui inviarla, quindi nella seconda riga inseriamo nel corpo della richiesta i parametri impostati precedentemente, codificandoli nel formato utilizzato negli URL.
Tale meccanismo, chiamato *URL encoding* (ma spesso definito *percent-encoding*), prevede la concatenazione di stringhe utilizzando l'operatore *&*, la sostituzione degli spazi con il simbolo *+* e la trasformazione di una ventina di caratteri riservati nella loro rappresentazione percentile[16].
Si prosegue quindi come nel caso della GET, potendo poi effettuare le stesse operazioni di manipolazione della risposta.

[16] SI convertono prima come sequenza di byte nella codifica UTF-8, poi si prende la loro rappresentazione esadecimale anteponendoci il segno di percentuale.

3.4 Struttura delle classi

Di seguito è possibile esaminare una rappresentazione schematica delle classi utilizzate e delle relazioni tra loro.



Gli oggetti di sfondo bianco sono classi che implementano un *singleton*[17] *pattern* o sono a loro volta inizializzate da classi singleton e quindi ne esiste una sola istanza. Gli oggetti arrotondati di sfondo verde indicano una generica istanza di quella classe, come vedremo meglio in seguito.
Una relazione da A a B indica che A chiama un metodo di B (statico o d'istanza).

Le classi *HttpAsyncTask, RPM, ManageData, DatabaseHelper, Synchronizer e Parser* fanno parte del pacchetto ***server***, che si preoccupa di gestire i dati sia remoti che locali. E' poi presente un pacchetto ***model***, qui non illustrato, che racchiude le classi rappresentative degli oggetti utilizzati all'interno dell'applicazione (si veda il paragrafo 3.4.3 per i dettagli).

[17] Il design pattern singleton permette di avere una sola istanza di una classe esistente in ogni momento del ciclo di vita dell'applicazione, accessibile tramite un apposito metodo statico.

Si può osservare fin da subito che le uniche classi che interagiscono direttamente con l'app (ovvero invocate dalle varie *activity*) sono ManageData e Synchronizer, che si preoccupano rispettivamente di restituire i dati salvati in locale e sincronizzarli col server.

3.4.1 HttpAsyncTask

Questa classe permette di eseguire una richiesta HTTP in un *thread* asincrono rispetto a quello principale dell'app.

In Android il codice di ogni applicazione, se non specificato diversamente, è eseguito in un unico thread, quello della UI (interfaccia utente); se vengono eseguiti processi di input/output (o del codice computazionalmente molto esigente) l'applicazione smette di rispondere finchè la procedura non viene completata.
Eseguendo ogni richiesta HTTP in un thread indipendente si evitano blocchi dell'applicazione e si migliora il tempo di esecuzione complessivo delle operazioni, essendo i thread eseguiti in parallelo.

Vediamo ora com'è strutturato il costruttore della classe:

HttpAsyncTask(String url, boolean httpGet, List<NameValuePair> params, HtmlCallback callback)

Il costruttore richiede una stringa con l'URL della richiesta, un *boolean* che indica se si tratta di una GET o una POST, eventuali parametri da inviare con la richiesta (solo nel caso di una POST) ed infine una callback che sarà eseguita al completamento della richiesta restituendo il codice HTML della risposta.

La callback è un "blocco di codice" che viene passato come argomento ad una funzione e viene da questa eseguito in un determinato momento. Mentre in linguaggi come C o C++ questo risultato può essere ottenuto passando semplicemente il puntatore di una funzione, in Java la callback può essere simulata passando l'istanza di un'interfaccia, della quale la funzione ricevitrice chiamerà uno o più metodi, mentre la funzione chiamante fornisce l'implementazione concreta.
In questo esempio, la callback è così definita:

```
public interface HtmlCallback {
    public void onCompleted(String html);
}
```

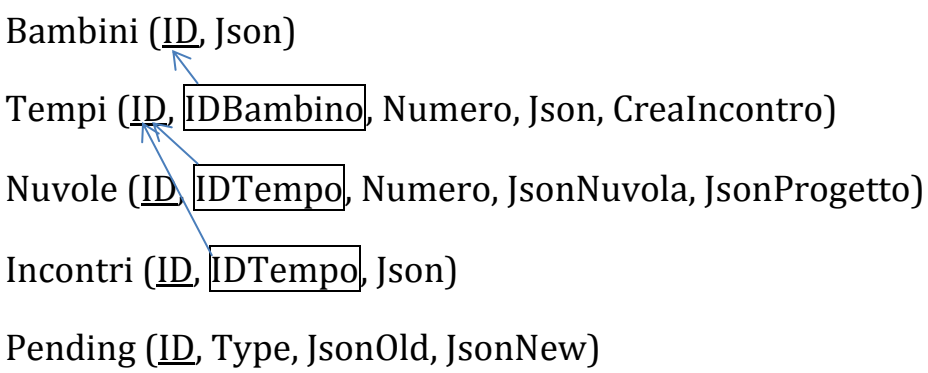
L'oggetto HttpAsyncTask potrà quindi al completamento della richiesta HTTP invocare il metodo *onCompleted* dell'istanza precedentemente passata al costruttore, permettendo così al metodo chiamante di ricevere il codice HTML della risposta.

3.4.2 DatabaseHelper

E' l'implementazione di SQLiteOpenHelper, una classe astratta di Android che si occupa di aprire un database se esiste, se necessario crearlo ed eventualmente effettuare un aggiornamento, attraverso i seguenti due metodi:

- *public void onCreate* (SQLiteDatabase db)
- *public void onUpgrade* (SQLiteDatabase db, int oldVersion, int newVersion)

Al momento *onUpgrade* non è stato implementato, non avendo finora avuto la necessità di gestire la distribuzione di versioni differenti dell'applicazione, l'implementazione del metodo *onCreate* invece si limita ad eseguire i comandi SQL di creazione delle tabelle nel database in input, secondo il seguente schema logico:



Gli ID sono chiavi autoincrementanti, l'intero 'Numero' nelle tabelle Tempi e Nuvole è utilizzato come indice di ordinamento, infine i vari Json sono stringhe di oggetti Java nella loro rappresentazione JSON, tramite la libreria Gson di Google (si veda il paragrafo 2.2).

Pending rappresenta la lista delle modifiche effettuate dall'utente che devono ancora essere inviate al server.

Ogni elemento è composto da un ID, una stringa Type che descrive la modifica effettuata (a scelta tra *NUVOLA, PROGETTO, SALVA_INCONTRO, CREA_INCONTRO, ELIMINA_INCONTRO* che rappresentano rispettivamente la modifica di una nuvola, la modifica di un progetto, la modifica di un incontro esistente, la creazione di un nuovo incontro e l'eliminazione di un incontro), le forme codificate dell'oggetto originale e dell'oggetto modificato.

3.4.3 ManageData

E' la classe di gestione del database locale, con tutti i metodi di lettura e scrittura necessari al funzionamento dell'applicazione: è infatti la classe di comunicazione tra le activity e il database.
Utilizza il design pattern *singleton*: l'istanza, inizializzata all'avvio dell'applicazione, è accessibile attraverso il metodo statico *getInstance()*.

La classe contiene i seguenti metodi di gestione diretta del database:

- *void openDB*()
- *void closeDB*()
- *void deleteDB*()

Facciamo soltanto una piccola menzione al codice per aprire il database:

```
SQLiteDatabase db = dbHelper.getWritableDatabase();
db.execSQL("PRAGMA foreign_keys = ON;");
```

La prima riga ottiene il riferimento al database e questo viene abilitato alla scrittura dal nostro oggetto DatabaseHelper, se non esiste questo è il momento in cui viene chiamato il metodo *onCreate*. Android utilizza SQLite come backend per i database e il supporto alle chiavi esterne è implementato ma disabilitato di default, quindi la seconda riga esegue ogni volta che viene aperta una connessione al database una istruzione SQL che ne abilita l'utilizzo.

Di seguito i metodi di accesso ai dati, strutturati seguendo il modello di RPM:

- *List<Bambino> getBambini*()
- *List<Tempo> getTempi*(Bambino)
- *List<Nuvola> getNuvole*(Tempo)
- *Progetto getProgetto*(Nuvola)
- *List<Incontro> getIncontri*(Tempo)

Esistono poi i seguenti metodi di modifica:

- *saveNuvola* (Nuvola vecchiaNuvola, Nuvola nuovaNuvola)
- *saveProgetto* (Progetto vecchioProgetto, Progetto nuovoProgetto)
- *saveIncontro* (Incontro vecchioIncontro, Incontro nuovoIncontro)
- *createIncontro* (Tempo tempo, Incontro nuovoIncontro)
- *deleteIncontro* (Incontro incontro)

Si può osservare che nei metodi di salvataggio vengono richieste due versioni dell'oggetto: la versione originale precedente alle modifiche e la versione modificata. Ciò è necessario per controllare, come vedremo meglio nel paragrafo successivo, che non si sovrascrivano involontariamente modifiche degli stessi dati effettuate da terzi.

Come descritto alla fine del paragrafo 1.2, la quasi totalità (ad esclusione degli incontri) delle sezioni modificabili nell'applicazione sono preesistenti. Esse vengono create al momento della sincronizzazione col server dalla classe Synchronizer, che utilizza i seguenti metodi *protected*[18] di ManageData per popolare il database:

- *long createChild* (Bambino child)
- *long createPeriod* (long childId, long periodNumber, Tempo tempo)
- *boolean setUrlIncontriTempo* (long periodId, String creaIncontro)
- *long createCloud* (long periodNumber, int cloudNumber, Nuvola cloud)
- *boolean updateProject* (long periodNumber, int cloudNumber, Progetto project)
- *long creaIncontro* (long idTempo, Incontro incontro)

Esistono infine i seguenti metodi di gestione delle modifiche pendenti:

- *boolean addPendingSave* (SaveType type, Object oldObject, Object newObject)
- *boolean updatePendingSave* (Object oldObject, Object newObject)
- *boolean removePendingSave* (Pending obj)
- *List<Pending> getPendingSaves*()
- *int getPendingSavesCount*()

[18] Metodi visibili solo dalle classi dello stesso pacchetto o dalle sottoclassi. Questo impedisce alle classi al di fuori del pacchetto *server* (che contiene le classi qui descritte) come le varie activity dell'app, di poter effettuare operazioni potenzialmente pericolose.

In tutti i metodi di modifica visti precedentemente, dopo aver aggiornato la copia locale dei dati, viene verificato se esiste già una modifica pendente dello stesso oggetto con il metodo *updatePendingSave*, se non esiste viene creata col metodo *addPendingSave*.
Spieghiamo meglio con un esempio come funziona il confronto e il perché di questa scelta.
Eseguiamo due modifiche consecutive allo stesso oggetto, che quindi passerà attraverso tre stati: T0 prima di ogni modifica, T1 dopo la prima modifica e T2 dopo l'ultima modifica.
Alla prima modifica alla coppia *(versioneOriginale, versioneModificata)* sarà associata la coppia *(T0, T1)*; con la seconda modifica avremo *(T1, T2). S*e confrontiamo la versione originale dell'ultima modifica effettuata (T1) con la versione aggiornata delle varie modifiche pendenti già memorizzate nel database, scopriamo che esiste una corrispondenza per T1 e quindi andremo ad aggiornare la modifica pendente già presente al valore *(T0, T2).*
Questo permette, come vedremo meglio nel paragrafo 3.4.5, di tenere memorizzati gli unici dati che sono effettivamente necessari per inviare le modifiche al server: lo stato originale risalente all'ultima sincronizzazione e lo stato attuale dell'oggetto. Modifiche intermedie non ci interessano.

3.4.4 RPM

RPM è una classe *singleton* che gestisce tutte le comunicazioni col server.

Al momento dell'inizializzazione della classe viene anche inizializzato il client HTTP, così come descritto nel paragrafo 3.3.

Una rilevanza particolare assumono i seguenti due metodi.

- void **login**(String, String, Callback) : richiede in input username, password e opzionalmente una callback che sarà chiamata al completamento dell'operazione. Memorizza nella classe i dati della sessione corrente.
- String **getUpdatedUrl**(String) : restituisce l'URL aggiornato alla sessione corrente dell'URL in ingresso. Infatti, ogni URL in APEX contiene un *session id*: per permettere ai metodi di salvataggio di utilizzare gli URL di sessioni precedenti memorizzati all'interno degli oggetti è necessario aggiornare tale *id* a quello della sessione corrente.

Possiamo ora iniziare con la trattazione dell'utilizzo della classe.

Essa presenta i seguenti metodi di interrogazione:

- getChildrenList(ChildrenCallback)
- getTimesList(Bambino, TimesCallback)
- getTriangolo(Tempo, TriangoloCallback)
- getCloud(NuvolaTriangolo, CloudCallback)
- getProject(Nuvola, ProjectCallback)
- getListaIncontri(String, ListaIncontriCallback)
- getIncontro(IncontroLista, IncontroCallback)

Si noti che tutti sono dotati di una callback individuale, in modo che si abbia per ogni metodo un ben preciso tipo di dato restituito.
Il metodo estrae dall'oggetto passato in input l'URL della pagina che contiene i dati necessari ed effettua una richiesta GET invocando la classe *HttpAsyncTask*.
Al completamento della richiesta, viene quindi invocato il rispettivo metodo della classe *Parser* che effettuando il parsing del codice HTML estrarrà i dati richiesti e costruirà l'oggetto/i, restituendocelo/i.

Di seguito troviamo i metodi di salvataggio:

- saveCloud(Nuvola oldVersion, Nuvola newVersion, Callback)
- saveProject(Progetto oldVersion, Progetto newVersion, Callback)
- deleteIncontro(Incontro, Callback)
- createIncontro(Tempo, Incontro, Callback)
- saveIncontro(Incontro oldVersion, Incontro newVersion, Callback)
- modifyIncontro(Incontro oldVersion, Incontro newVersion, int requestType, Callback)

I tre metodi di modifica di un incontro (cancellazione, creazione e modifica) invocano tutti il metodo *modifyIncontro*: infatti, nelle 3 operazioni cambia soltanto un singolo valore della POST di richiesta.

Vediamo ora, con un esempio di richiesta HTTP inviata al server, che tipo di problematiche possono insorgere.

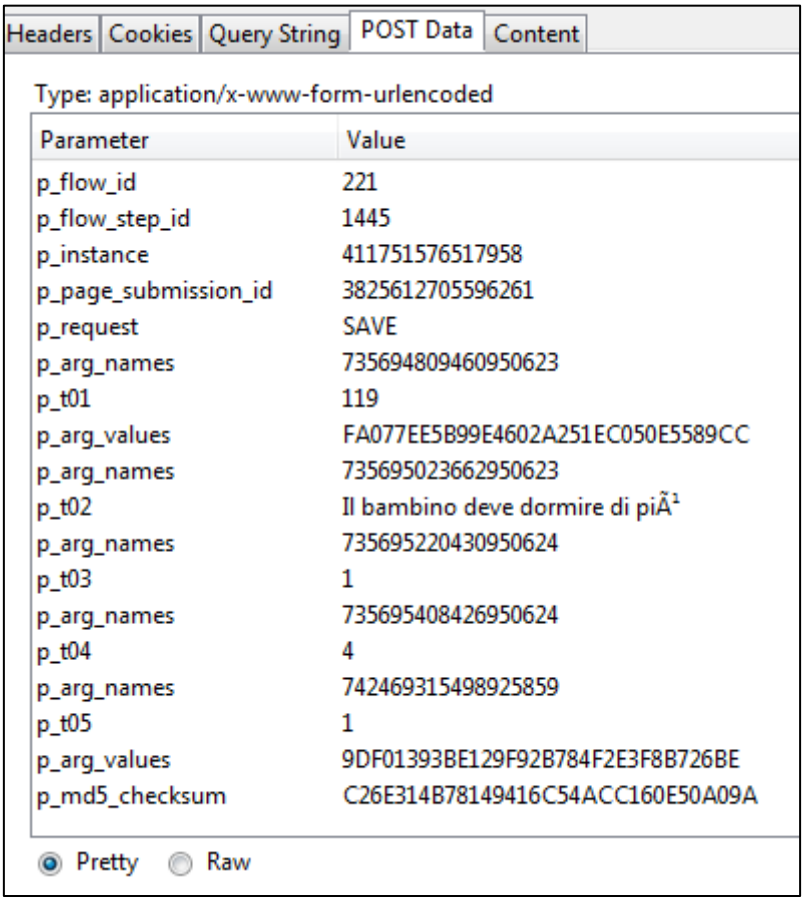


Figura 8: Esempio di POST di salvataggio di una sottodimensione

In figura è rappresentata una POST di salvataggio di una sottodimensione; da varie prove è risultato che *p_t02, p_t03* e *p_t04* corrispondono rispettivamente ad *assessment*, livello attuale e livello previsto. I vari *p_arg_names* ed alcuni altri campi assumono valori costanti (ad esempio, indicano le coordinate della sottodimensione modificata), mentre i diversi *p_arg_values* e il *p_md5_checksum* finale cambiano ad ogni richiesta.
Si è scoperto poi che tali valori sono nascosti all'interno del codice HTML della pagina: memorizzando durante il parsing quei dati all'interno dell'oggetto, al rispettivo metodo di salvataggio sarebbe bastato estrarre il set di dati dall'oggetto originale passatogli per poter ricostruire la richiesta HTTP.
Con questo metodo il primo salvataggio funzionava correttamente ma i successivi non venivano accettati dal server; si è quindi dedotto che tali valori venissero rigenerati dopo ogni salvataggio, invalidando tutti i set precedenti.

C'era anche un altro problema in sospeso, ovvero la gestione dei conflitti delle modifiche. Facciamo un esempio: l'utente A sta utilizzando l'app sul suo tablet, scarica tutti i dati dal server e inizia a lavorare sulla copia in locale, senza sincronizzare le modifiche con il server. Nel frattempo l'utente B utilizzando

RPMonline effettua una modifica allo stesso oggetto. Quando A deciderà di inviare a RPMonline le proprie modifiche, sarà salvata la sua versione dell'oggetto, sovrascrivendo tutte le modifiche apportate da B.

Come possiamo quindi gestire conflitti del genere? Se avessimo il controllo completo sui dati si potrebbe pensare di fare una copia di backup dei dati in conflitto e mantenere l'ultimo tentativo di salvataggio effettuato. Purtroppo non possiamo, ma possiamo invece controllare che lo stato dell'oggetto attualmente presente in RPMonline corrisponda allo stato di partenza della modifica che stiamo inviando e segnalare così eventuali conflitti lasciando all'utente la libertà di decidere se sovrascrivere la modifica o meno. Così facendo risolviamo anche il problema dei sopracitati dati variabili.

I metodi di salvataggio richiedono sia la versione originale (precedente ad ogni modifica) che la versione modificata dell'oggetto; il metodo si collega alla pagina web dell'oggetto, effettua il parsing e lo confronta con l'oggetto originale (tramite un semplice *override* del metodo *equals* nella classe dell'oggetto): se coincidono significa che nessuno ha effettuato salvataggi da quando noi abbiamo iniziato a modificarlo, quindi il salvataggio può procedere estraendo i dati variabili nascosti nell'HTML (i dati sono sicuramente validi dato che la pagina è stata appena richiesta) e costruendo la POST di richiesta.

3.4.5 Synchronizer

Si preoccupa di popolare il database e gestire le successive sincronizzazioni. Essa utilizza il design pattern *singleton* come le altri classi di controllo del pacchetto *server* e presenta, oltre al *getInstance*, soltanto due metodi pubblici:

- void **syncSaves** (String user, String psw, Callback callback)
- void **syncAll** (String user, String psw, Callback callback)

Il primo metodo sincronizza le modifiche in sospeso effettuate dall'utente utilizzando l'app.
Viene chiamato il metodo *getPendingSaves* della classe *ManageData* che restituisce una lista di oggetti Pending, ciascuno contenente la versione corrente dell'entità (che può essere una nuvola, una progettazione o un incontro) aggiornata all'ultima modifica e la versione originale dell'entità risalente all'ultima sincronizzazione, prima di ogni modifica (per ulteriori dettagli si vedano i paragrafi 3.4.2 e 3.4.3).

Per ogni tipo di modifica viene invocato il rispettivo metodo della classe RPM, che come descritto nel paragrafo precedente verificherà che non ci siano conflitti. Nel caso ce ne fossero, non possiamo decidere a priori se sovrascrivere o meno le modifiche effettuate da terzi, quindi l'app segnalerà con un popup il conflitto e lascerà all'utente la libertà di decidere cosa fare, abilitando l'apposita opzione "Sovrascrivi modifiche terze".
Potrebbe anche verificarsi un errore di tipo generico, probabilmente imputato ad una modifica della struttura dei dati in APEX: anche in questo caso l'errore viene segnalato e la modifica rimane in sospeso in attesa che il problema venga risolto.

Figura 9: Pannello di controllo

Il metodo *syncAll* prima chiama *syncSaves*, poi, al completamento di quest'ultima operazione, elimina l'intero database locale e riscarica tutti i dati.
E' necessario effettuare questa sincronizzazione totale ogniqualvolta viene effettuata una modifica da RPMonline, perché purtroppo non esiste un metodo di interrogazione che consenta di ottenere solo le modifiche: bisogna invece confrontare l'intero l'albero di pagine, quindi la decisione di inizializzare il database da zero si è resa la scelta più immediata.

Ricapitoliamo ora, dopo aver fatto una panoramica di tutte le classi chiave del progetto, che processi attraversano lo scaricamento dei dati dal server e l'invio di una modifica pendente. Cominciamo con quest'ultima.

Viene richiesta la lista delle modifiche pendenti alla classe *ManageData* e queste vengono esaminate una per volta. Ipotizziamo di dover salvare una *nuvola: Synchronizer* legge che è una modifica di tipo *NUVOLA* (3.4.2), quindi chiama il metodo *saveCloud* di RPM passandogli due versioni dell'oggetto Nuvola: lo stato originale antecedente le modifiche e lo stato attuale da salvare.
Il metodo saveCloud effettua una GET alla pagina della nuvola tramite la classe HttpAsyncTask, utilizzando l'URL contenuto nella versione originale dell'oggetto.
Ottenuto il codice HTML della pagina attraverso la callback, questo viene processato dalla classe *Parser* e otteniamo quindi l'oggetto Nuvola rappresentante lo stato attuale dell'entità sul server. Questo viene confrontato con la versione originale in nostro possesso: se corrispondono, possiamo procedere (si veda alla fine di 3.4.4 per i dettagli). Estratti i dati nascosti dall'HTML appena ottenuto possiamo ora costruire la POST di salvataggio; la risposta del server viene quindi processata per controllare se l'operazione è andata a buon fine e il risultato viene restituito tramite callback a Synchronizer, che potrà quindi procedere con la modifica successiva.

Vediamo quindi un riepilogo veloce del processo attraversato per scaricare i dati dal server: Synchronizer chiama il metodo *getChildrenList* di RPM per ottenere la lista dei bambini. Per ogni bambino si dovrà poi ottenere la lista dei tempi e così via, fino ad arrivare alle foglie di quest'albero di entità, rappresentate da progettazioni e incontri.
Questo risultato viene ottenuto tramite l'innesto delle callback dei vari metodi di RPM, l'una dentro l'altra, in modo che non appena termina un metodo con i dati necessari per far partire i successivi questi vengono eseguiti in cascata. Man mano che i metodi di RPM terminano, i dati vengono memorizzati nel database

locale utilizzando i metodi di ManageData (3.4.3), finché le GET in esecuzione raggiungono quota zero e il processo è quindi terminato.

Capitolo 4: Conclusioni

In questa tesi è stato visto come sia possibile implementare una funzione di dialogo con un server senza alcuna conoscenza a priori di quest'ultimo.

Tutte le applicazioni *non ufficiali* che si interfacciano con un determinato servizio sono state realizzate allo stesso modo: attenta ispezione dei pacchetti HTTP scambiati tra browser e server, identificazione dei campi aiutandosi eventualmente anche con l'analisi del codice HTML, per arrivare infine alla loro replicazione sistematica. Nel nostro caso, inoltre, si è anche dovuto introdurre un lavoro di interpretazione e parsing del codice HTML. Per effettuare correttamente l'identificazione degli elementi di nostro interesse in una pagina HTML in alcuni casi è stato necessario introdurre una stretta correlazione tra elementi differenti, portando quindi all'introduzione di errori non appena la struttura viene modificata. Le stesse problematiche riguardano le richieste HTTP, le cui strutture e identificativi dei campi devono rimanere invariati nel tempo.
Durante questo lavoro una parte considerevole di tempo è stata spesa infatti anche in interventi di manutenzione in seguito a qualche modifica nel server.
I web service moderni, come discusso nel paragrafo 2.1, scambiano messaggi in formati standard, facilitando enormemente la fase di interpretazione delle richieste e delle risposte.
Inoltre, il loro utilizzo introdurrebbe anche maggior flessibilità alle modifiche e la possibilità di eseguire richieste complesse per reperire determinati dati senza dover invece seguire la struttura delle pagine presente nell'interfaccia di *RPMonline*.

La struttura del codice è stata progettata per separare in strati differenti i vari concetti visti in questa tesi: per sostituire uno di questi (ad esempio risposte JSON anziché HTML, o modifiche dei metodi di richiesta) è sufficiente modificare solamente la classe chiave, lasciando invariato tutto il resto.
Questo permetterà in futuro di introdurre facilmente un web service, necessario per garantire all'applicazione ulteriori sviluppi e una manutenzione agevole.

Appendice: codice di inizializzazione client HTTP con connessione TLS

```
private DefaultHttpClient initializeHttpClient() {
   DefaultHttpClient httpClient = new DefaultHttpClient();
   StrictMode.ThreadPolicy policy = new StrictMode.ThreadPolicy.Builder().permitAll().build();
   StrictMode.setThreadPolicy(policy);

   // creo TrustManager per considerare tutti i certificati server validi
   X509TrustManager tm = new X509TrustManager() {
      public java.security.cert.X509Certificate[] getAcceptedIssuers() {
         return null;
      }
      public void checkClientTrusted(java.security.cert.X509Certificate[] certs, String authType) {
      }
      public void checkServerTrusted(java.security.cert.X509Certificate[] certs, String authType) {
      }
   };

   try {
      // Creo il contesto SSL utilizzando il trust manager creato
      SSLContext ctx = SSLContext.getInstance("TLS");
      ctx.init(null, new TrustManager[]{tm}, null);
      // Creo la connessione https
      SSLSocketFactory ssf = new MySSLSocketFactory(ctx);
      ssf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
      ThreadSafeClientConnManager ccm = new ThreadSafeClientConnManager(httpClient.getParams(),
          httpClient.getConnectionManager().getSchemeRegistry());
      SchemeRegistry sr = ccm.getSchemeRegistry();
      sr.register(new Scheme("https", ssf, 443));
      httpClient = new DefaultHttpClient(ccm, httpClient.getParams());
      CookieStore cookieStore = new BasicCookieStore();
      httpClient.setCookieStore(cookieStore);

      return httpClient;
   }
   catch(Exception e) {
      e.printStackTrace();
      return null;
   }
}

private class MySSLSocketFactory extends SSLSocketFactory {
   SSLContext sslContext;

   protected MySSLSocketFactory(SSLContext context) throws KeyManagementException,
          NoSuchAlgorithmException, KeyStoreException, UnrecoverableKeyException {
      super(null);
      sslContext = context;
   }

   @Override
   public Socket createSocket(Socket socket, String host, int port, boolean autoClose) throws
          IOException, UnknownHostException {
      return sslContext.getSocketFactory().createSocket(socket, host, port, autoClose);
   }

   @Override
   public Socket createSocket() throws IOException {
      return sslContext.getSocketFactory().createSocket();
   }
}
```

Bibliografia e sitografia

1. Milani P., Di Masi D., Ius M., Serbati S., Tuggia M., Zanon O. (2013**), Il Quaderno di P.I.P.P.I.. Teorie, Metodi e strumenti per l'implementazione del programma**, Sommacampagna (VR), BeccoGiallo.
2. **Oracle APEX**, http://www.oracle.com/technetwork/developer-tools/apex/overview/index.html
3. **HTTP Reference**, http://tools.ietf.org/html/rfc1945
4. **HTML**, http://www.w3.org/html/
5. **HTML Tags**, http://www.w3schools.com/tags
6. **Web Service Activity Statement**, http://www.w3.org/2002/ws/Activity
7. **Android API Reference**, http://developer.android.com/reference
8. **Jsoup**, http://www.jsoup.org
9. **Apache HttpComponents**, http://hc.apache.org/
10. **"What version of Apache HttpClient is bundled in Android 1.6?"**, http://stackoverflow.com/questions/2618573/what-version-of-apache-http-client-is-bundled-in-android-1-6
11. **TLS**, http://tools.ietf.org/html/rfc5246
12. **Uniform Resource Identifier** (URI), http://tools.ietf.org/html/rfc3986
13. **HttpFox**, http://addons.mozilla.org/it/firefox/addon/httpfox/
14. Fantozzi Carlo, **Embedded Systems Programming**, http://esp1213.wikispaces.com/
15. **Google Gson**, https://code.google.com/p/google-gson/
16. Fielding, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. Doctoral dissertation, University of California, Irvine, 2000.
17. **Varie**, http://en.wikipedia.org/