ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Scienze e Tecnologie Informatiche

Worklight: potenzialità e integrazioni

Relazione finale in:
Mobile Web Design

Relatore:
Prof. Mirko Ravaioli

Presentata da:
Lorenzo Rava

Sessione III

Anno Accademico 2011/2012

Sommario

La presente tesi, realizzata all'interno di Tecla.it, azienda impegnata in progetti integrati per lo sviluppo del business digitale attraverso eCommerce, portali, marketing e mobile, si propone proprio in quest'ultimo ambito, in forte sviluppo nel recente periodo, di sviscerare a fondo le possibili tecniche realizzative per la creazione di applicazioni mobile. In particolare la realizzazione con il metodo ibrido, così che l'applicazione possa essere il più possibile cross – platform ma comunque performante. Questo perché all'interno di una azienda è difficile pensare che per realizzare una applicazione per un cliente, si debba in realtà realizzarne una per ogni piattaforma diversa, in quanto ognuna di queste prevede tecniche e linguaggi di programmazione completamente diversi.

Inoltre la tesi si pone come obiettivo, la realizzazione di un'applicazione che possa essere utilizzata in azienda come PoC (Proof of Concept) da mostrare a possibili clienti e come punto di partenza per la creazione di nuove applicazioni per i clienti stessi.

Si è scelto di integrarla con un portale di eCommerce, in particolare quello specifico di IBM, Websphere Commerce, attraverso il metodo dei servizi REST per evidenziare la facilità nel reperire informazioni dalla commerce stessa con semplici richieste al server con i metodi HTTP, in particolare GET, POST, PUT e DELETE.

Inoltre si vuole approfondire e realizzare il servizio, oggi in forte crescita, dell'invio delle notifiche push, nell'ottica futura di consentire sempre in collaborazione con la commerce, la notifica agli utenti per esempio di modifiche ad oggetti a loro interessati o l'inizio di periodi di sconti.

INDICE

1 Introduzione

1.1 Smartphone

La fine dello scorso millennio è stata caratterizzata dall'avvento di Internet, una rivoluzione tecnologica e culturale per il fatto di permettere la condivisione di informazioni in qualunque parte del mondo.

Oggi però stiamo già assistendo ad una nuova rivoluzione, quella dei dispositivi mobile. Quelle che prima erano informazioni ed applicazioni raggiungibili ed eseguibili esclusivamente da PC, ora sono accessibili da dispositivi sempre più potenti, con la caratteristica di essere mobili e di dimensioni notevolmente ridotte.

Una volta si usava il termine "*cellulari*", ma ora sono diventati veri e propri PC portatili, dove la funzione di telefono è solo una delle molteplici disponibili.

A questi dispositivi, ora definiti "*smartphone*", vanno aggiunti i "*tablet*", i quali non è corretto definire solo "smartphone più grandi", in quanto hanno caratteristiche hardware ben diverse che lo rendono idoneo a determinate funzioni che su smartphone non sarebbero possibili o che sarebbero decisamente meno comode, quali la lettura di documenti o la visualizzazione di filmati.

Quello degli smartphone è un mercato in grande crescita negli ultimi anni: secondo Strategy Analytics[1] nel primo trimestre del 2011 erano stimati 708 milioni di questi dispositivi in tutto il mondo.

[1]Strategy Analytics Inc. agenzia per l'analisi e le previsioni di mercato (http://www.strategyanalytics.com/)

Alla fine del 2012 il numero di smartphone attivi aveva superato il miliardo e, secondo le previsioni, si stima che questo numero raddoppierà entro il 2015, visto l'andamento del mercato negli ultimi tre anni[2].

Un'altra stima importante è quella che prevede che nel giro di pochi anni, il numero di smartphone supererà quello dei PC.

Sebbene un PC abbia caratteristiche software e hardware superiori a quelle degli smartphone, quest'ultimi stanno facendo grossi passi avanti a livello tecnologico. Infatti sono comparsi i primi dispositivi con microprocessori quad-core, cosa impensabile pochi anni fa[3].

1.1.1 Sistemi Operativi Mobile

Il principale aspetto che differisce fra di loro i vari smartphone è il *sistema operativo.*

Attualmente quello maggiormente diffuso è Android, che in Italia nel 2012 ha avuto il 56% delle vendite di smartphone. Il secondo sistema operativo più diffuso è invece iOS della Apple con una quota molto inferiore rispetto ad Android, circa il 19%. Windows Phone è invece il terzo più diffuso, in forte crescita, soprattutto sul mercato dei tablet.

Uno dei sistemi operativi storicamente più diffusi, Symbian, è sceso al quarto posto ed è passato dal 25% al 7% della quota mercato[4].

[2] http://punto-informatico.it/3628460/PI/News/smartphone-un-record-nove-zeri.aspx
[3] http://techcrunch.com/2012/11/06/gartner-1-2-billion-smartphones-tablets-to-be-bought-worldwide-in-2013-821-million-this-year-70-of-total-device-sales/
[4] http://www.tomshw.it/cont/news/iphone-5-traina-ios-in-italia-si-vendono-tanti-nokia-lumia/41394/1.html

Italy	100.0%	100.0%	0.0
iOS	18.0	19.0	1.0
Android	44.8	56.7	11.9
RIM	5.2	3.7	-1.5
Symbian	25.7	7.3	-18.4
Windows	3.9	11.7	7.8
Bada	1.5	1.5	0.0
Other	0.9	0.1	-0.8

Figura 1: sviluppo del mercato delle app dal 2010 al 2012

I produttori di questi sistemi operativi mettono a disposizione degli sviluppatori il necessario per interfacciarsi al sistema, come un ambiente di sviluppo, i propri tool e i linguaggi di programmazione per consentire lo sviluppo su quella specifica piattaforma.

Dettagli dei principali sistemi operativi:

- **Android**
  - *Produttore:* sviluppato da Android Inc. successivamente acquistato da Google.
  - *Versione Attuale:* 4.2 Jelly Bean, sebbene siano pochi gli smartphone o tablet aggiornati a questa versione[5].
  - *Hardware supportato:* è il sistema operativo della maggior parte degli smartphone in commercio, di numerose case produttrici. La struttura open source basata su kernel Linux permette una vastissima distribuzione.
  - *Linguaggio di programmazione:* Android fornisce agli sviluppatori un SDK[6] con svariate componenti tra le quali il simulatore. Oltre a questo viene fornito un plugin per Eclipse, ADT (Android Development Kit), con le classi e librerie necessarie per lo sviluppo. Il linguaggio utilizzato è Java.

[5] http://www.android-htc.it/20568/news/statistiche-android-jelly-bean-ancora-al-2-7/
[6] SDK (Software Development Kit)

- **iOS**
    - *Produttore:* sviluppato dalla Apple
    - *Versione Attuale:* 6.1; oltre il 60% dei dispositivi con iOS sono aggiornati almeno alla versione 6.0 o successive[7].
    - *Hardware supportato:* questo sistema operativo è disponibile esclusivamente per i prodotti della Apple, in particolare iPhone, iPodTouch, iPad e AppleTV.
    - *Linguaggio di programmazione:* per programmare applicazioni in iOS è necessario utilizzare il software apposito, XCode, disponibile esclusivamente su computer Mac OS X. Il linguaggio di programmazione è Objective-C.
- **Windows Phone**
    - *Produttore:* sviluppato dalla Microsoft.
    - *Versione Attuale:* 8.0.
    - *Hardware supportato:* è disponibile su dispositivi di diverse case produttrici.
    - *Linguaggio di programmazione:* la piattaforma di sviluppo è Visual Studio con la relativa SDK per il mobile e i linguaggi supportati sono C++ e C#.

[7] http://www.ispazio.net/391823/ios-6-sono-piu-di-300-milioni-i-dispositivi-gia-aggiornati

1.1.2 Applicazioni e Application Store

Ogni sistema operativo solitamente mette a disposizione un proprio "Application Store", cioè una piattaforma digitale che ha il compito di fornire le applicazioni agli smartphone o tablet.

Possono essere disponibili anche altri Store, forniti per esempio da compagnie private (esempio Amazon[8]), operatori telefonici o direttamente dalla casa produttrice del dispositivo.

I principali Store per numero di applicazioni contenute, come è possibile vedere dal grafico che segue, sono:

- Google Play[9]: sviluppato per Android. Possiede circa 700.000 applicazioni. Appena uno sviluppatore firma la propria applicazione, può pubblicarla su questo Store, senza un lungo processo di approvazione.
- AppStore[10]: sviluppato dalla Apple per sistemi iOS. Possiede anche questo circa 700.000 applicazioni. Per prevenire abusi, il processo di approvazione previsto per questo Store è molto rigido. Un'applicazione può essere rifiutata per numerosi motivi, sia tecnici, come malfunzionamenti, ma anche commerciali. Su questo Store il tasso di rifiuto delle applicazioni è del 20%.

[8] http://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011
[9] Fonte Wikipedia: http://it.wikipedia.org/wiki/Google_Play
[10] Fonte Wikipedia: http://it.wikipedia.org/wiki/App_Store

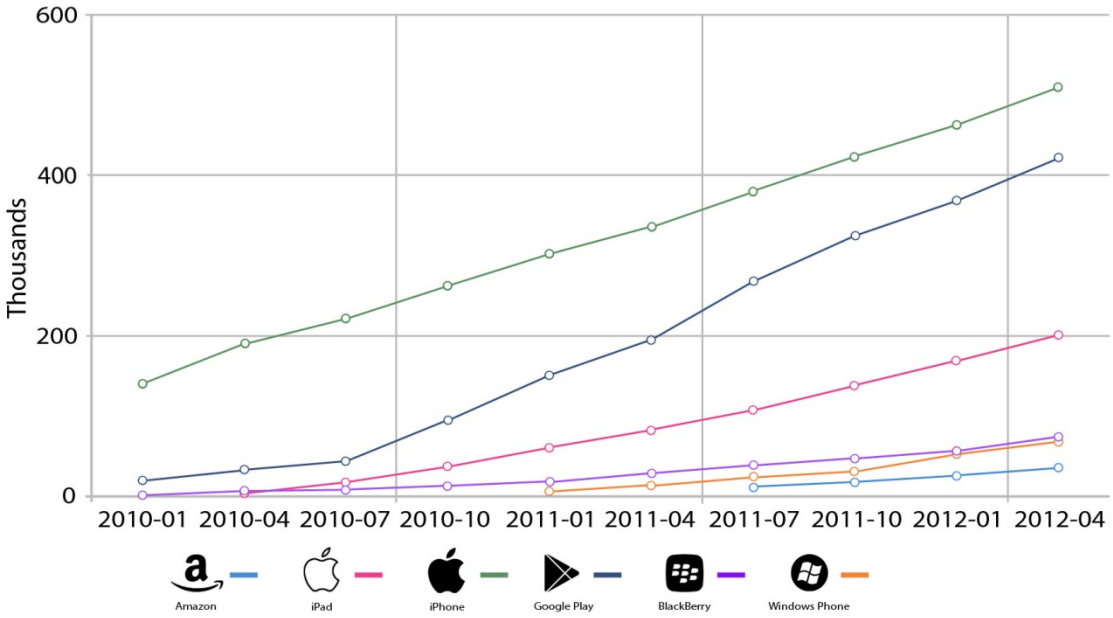

Figura 2: sviluppo del mercato delle app dal 2010 al 2012

Normalmente se si vuole realizzare un'applicazione che sia disponibile su tutti i sistemi operativi è necessario realizzarla singolarmente con le specifiche di ogni sistema operativo. In seguito si vedrà come è possibile implementarla una volta sola, in modo che sia disponibile per numerosi sistemi operativi.

1.2 Web 2.0

Con il termine "*Web 2.0*[11]" s'intende uno stato dell'evoluzione del web, che permette una spiccata iterazione tra sito web e utente. Questo paradigma consente agli utenti di prendere informazioni da diversi siti simultaneamente e di distribuirle sui propri siti per nuovi scopi.

La base del paradigma Web 2.0 è Ajax[12], una tecnica di interazione più evoluta rispetto al classico paradigma client – server che consente un approccio di sviluppo web basato su JavaScript per la creazione di soluzioni web interattive. Consente alle pagine web di funzionare quasi come applicazioni desktop, superando la staticità delle pagine, caratteristica principale del web precedente al paradigma 2.0.

L'intento di tale tecnica è quello di ottenere pagine web che rispondono in maniera più rapida, grazie allo scambio in *background* di piccoli pacchetti di dati con il server, così che l'intera pagina web non debba essere ricaricata ogni volta che l'utente effettua una modifica. Questa tecnica riesce, quindi, a migliorare l'interattività, la velocità e l'usabilità di una pagina web.

1.2.1 Ajax

Come detto precedentemente, la tecnologia Ajax si basa su uno scambio di dati in background fra browser e server in modo asincrono, che consente l'aggiornamento dinamico della pagina senza un esplicito refresh dell'utente. Ajax è definito asincrono perché i dati extra richiesti al server e caricati in background non interferiscono con la pagina esistente.

[11] "Web 2.0" termine coniato da Tim O'Reilly nel 2004

[12] AJAX (Asynchronous JavaScript and XML) enunciato per la prima volta da Jesse Garrett nel 2005 sul suo blog

Nei due grafici che seguono si può notare la differenza tra la comunicazione classica tra client e server, e quella attraverso AJAX:

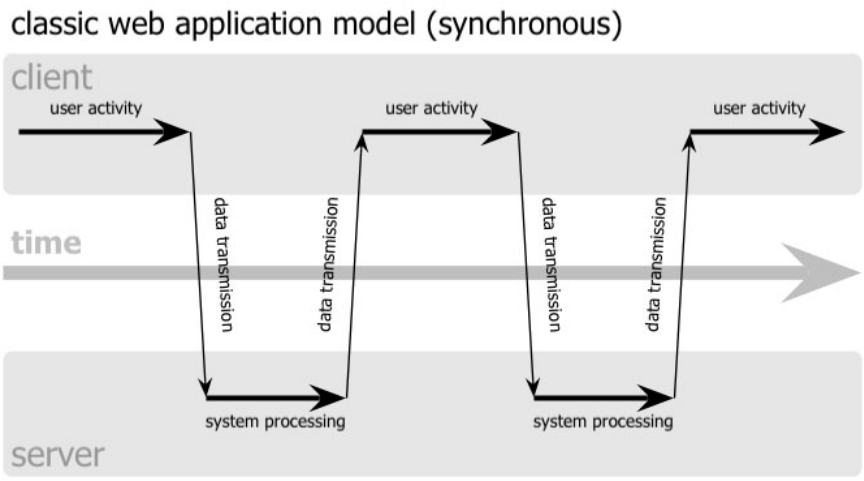

Figura 3: comunicazione classica client – server

In quella classica tra la richiesta dell'utente e la risposta del server intercorre un tempo di attesa che blocca l'utente.

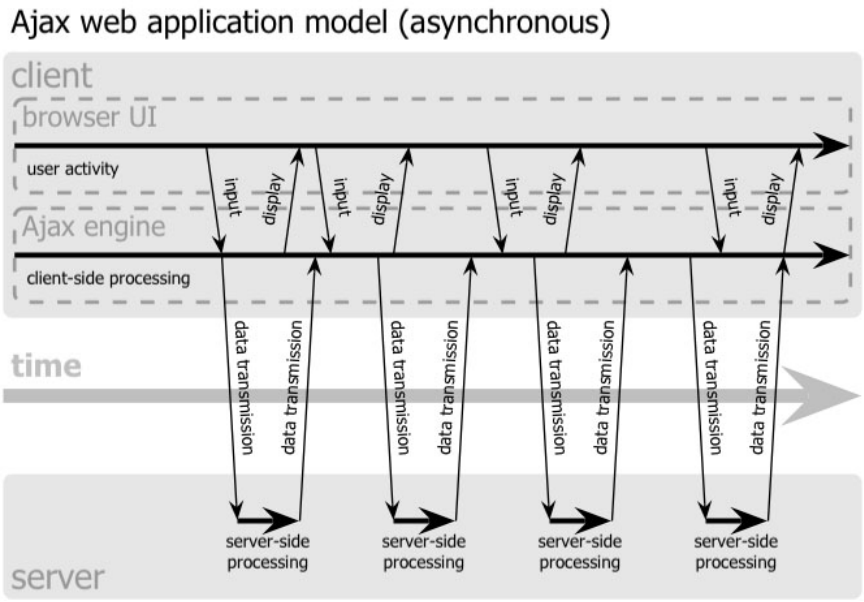

Figura 4: comunicazione client – server attraverso Ajax

Mentre nella comunicazione con AJAX, possono esserci più richieste al server completamente indipendenti, gestite dal motore AJAX.

Il linguaggio standard per effettuare richieste al server è JavaScript. In genere viene usato XML come formato di scambio dei dati, anche se di fatto è possibile utilizzarne altri, come semplice testo, HTML preformattato oppure oggetti JSON.

Attraverso la combinazione di Ajax e la manipolazione del DOM[13] è possibile aggiornare anche solo una sezione di una pagina web a seguito dell'effettuazione di un comando dall'utente, così da non dover ricaricare interamente la pagina in seguito ad ogni interazione con la pagina stessa, portando grandi vantaggi in termini di efficienza e velocità.

1.2.2 HTML5 e CSS3 per sviluppo mobile

Per creare una pagina web è necessario utilizzare lato client HTML per la struttura della pagina, CSS per lo stile e un linguaggio di scripting come JavaScript eseguibile dal browser. Mentre server side sono disponibili numerosi linguaggi, tra cui i più noti PHP, ASP o Java, che vengono elaborati dal server e poi resi disponibili all'utente.

Recentemente sono stati resi disponibili HTML5 e CSS3, ultime versioni distribuite dal W3C[14]. Queste nuove versioni, soprattutto HTML5, hanno aiutato moltissimo lo sviluppo delle applicazioni in ambito mobile[15].

Le novità principali di questa versione sono:

- *Multimedia:* Supporto audio e video nativo nel browser. Introdotti elementi specifici per il controllo dei contenuti multimediali attraverso i tag *<audio>* e *<video>*.

[13] DOM (Document Object Model) è standard ufficiale del W3C per la rappresentazione di documenti strutturati
[14] World Wide Web Consorium: organizzazione non governativa con lo scopo di sviluppare il World Wide Web
[15] http://www.solotablet.it/app/articoli/html5-si-standard-multi-piattaforma

- *Local Storage:* possibilità di salvare dati direttamente lato client, così che questi siano disponibili anche *offline,* se l'utente non è connesso.
- *Semantic:* regole per la strutturazione del testo in paragrafi, sezioni e capitoli rese più stringenti.
- *Geolocalizzation:* Introduzione della geolocalizzazione. Il browser può esporre le coordinate geografiche dell'utente fornitegli dal sistema operativo.

Grazie all'adozione dello standard HTML5 e quindi di CSS3 da parte dei produttori di piattaforme mobile è possibile oggi la costruzione di applicazioni web con look and feel complessi e simili alle interfacce native.

Grazie alle nuove API messe a disposizione da HTML5 gli sviluppatori possono contare sulla possibilità di costruire interfacce più complesse senza la necessità di dover usare plugin o di dover, per poter implementare la propria applicazione, passare alla costruzione di un'applicazione nativa.

2 Concetti Base

2.1 Come realizzare Mobile Application

Ci sono varie modalità per la realizzazione di applicazioni per dispositivi mobile, quali Smartphone o Tablet. Quale modalità scegliere dipende molto dal tipo di applicazioni che si vuole creare, tenendo anche in considerazione una previsione di costi e tempi per la realizzazione.

I metodi per creare applicazioni mobile si possono racchiudere in 3 aree:

- Web Based
- Native
- Hybrid

2.1.1 Web Based

Creare un'applicazione di tipo Web Based è il metodo più semplice e meno costoso. Di fatto sono normali pagine web "ri-disegnate" per dispositivi mobile in modo da simulare l'aspetto delle interfacce delle app native. I linguaggi utilizzati per questo tipo di applicazione sono lato client HTML5[16], CSS3 e JavaScript, mentre lato server un qualsiasi linguaggio server side (es. java, php, etc…).

Figura 5: Loghi di HTML5, CSS3 e JavaScript

[16] http://www.w3.org/TR/2011/WD-html5-20110525/

Quali sono i vantaggi di questa realizzazione:

- **I costi:** rispetto allo sviluppo di applicazioni native su più piattaforme, lo sviluppo di un'applicazione web mobile ha costi inferiori per via del linguaggio comune con il quale viene scritta. Ciò richiede allo sviluppatore competenze tecniche comuni a tutte le piattaforme quali: HTML5 e JS, e un linguaggio server side, che è indipendente dalla piattaforma.
- **Portabilità:** non è assicurato che una web application sia più veloce da realizzare rispetto ad un'applicazione nativa, tuttavia costruire applicazioni native ci limita notevolmente nella portabilità su altre piattaforme, in quanto ognuna di queste è caratterizzata da un proprio framework di sviluppo e linguaggio di programmazione differente, e tutto questo porta a costi di portabilità molto alti.
- **Distribuzione dell'applicazione:** a differenza delle applicazioni native, dove è ormai affermata la distribuzione tramite mobile Application Store, l'applicazione web viene distribuita su server remoti al device, fornendo flessibilità massima riguardo la possibilità di eseguire aggiornamenti all'applicazione, i quali sono subito disponibili agli utenti.

  Accessibilità ovunque: realizzata una volta, questa è disponibile su tutti i dispositivi, mobili e tablet.

Questo metodo di realizzazione ha però anche vari aspetti negativi:

- **Accesso alle funzionalità native del device:** non è ancora possibile raggiungere tutte le funzionalità native del device, quali la possibilità di invio di notifiche push, o l'utilizzo di dispositivi interni, come l'accelerometro o la bussola (vedi HTML5 Device API[17]).
- **Modalità offline:** se il dispositivo non è connesso ad internet, gli utenti non potranno accedervi. Per aggirare questo problema bisogna realizzare metodi di caching sul dispositivo, in modo che la connessione sia necessaria solo la prima volta e quando non vi è connessione, si visualizza la versione in cache.

2.1.2 Native Application

Un'applicazione nativa è realizzata nel linguaggio specifico per ogni sistema operativo mobile (esempio Objective-C per Apple iOS, Java per Android, C# o C++ per Windows Phone, etc…). Questo metodo è completamente diverso dal precedente e richiede requisiti tecnici di diverso tipo.

I vantaggi:

- **Accesso alle funzionalità native del device:** si ha possibilità di accedere a tutti i sensori hardware del dispositivo, quali accelerometro, GPS, fotocamera e altri, che possono essere gestiti direttamente e compiere esempio azioni diverse in caso di movimento del dispositivo. Inoltre è possibile l'utilizzo delle **notifiche push** e l'accesso alle funzioni interne quali i contatti o le immagini salvate nel dispositivo.

[17] http://mobilehtml5.org/

- **Distribuzione dell'applicazione:** avere la possibilità di distribuire la propria applicazione attraverso i mobile Application Store, consente di entrare in un mercato in grande espansione come numero di potenziali utenti.
- **Modalità offline:** attraverso gli Application Store, l'applicazione viene installata sul device e non risiede su server remoti. Di conseguenza non necessita di ulteriori download per le esecuzioni successive (se non per eventuali aggiornamenti).
- **User Experience:** è possibile realizzare interfacce grafiche più reattive e fluide, in quanto si effettuano chiamate dirette al sistema e il codice non viene più interpretato come nelle web browser, ma è già in linguaggio macchina.

Sebbene i vantaggi siano molteplici, ci sono anche alcuni svantaggi:

- **Portabilità:** le app native funzionano solo per specifici dispositivi per cui sono state scritte e compilate. Se la portabilità dell'applicazione su altre piattaforme è un requisito richiesto, realizzarla in modalità nativa comporta per lo sviluppatore la necessità di conoscere differenti linguaggi di programmazione e questo comporta tempi di sviluppo maggiori e problemi di manutenibilità del software. Nel caso la portabilità non fosse un requisito, con l'applicazione nativa si possono raggiungere ottimi risultati di efficienza e di sfruttamento dell'hardware sottostante.
- **Poca flessibilità:** se è necessario un aggiornamento della GUI o variazioni nell'usabilità bisogna sempre passare attraverso l'approvazione all'interno del relativo Application Store. Questo può essere un problema per esempio per lo Store della Apple che ha un tasso del 20% di rifiuto oltre a tempi di validazione

decisamente lunghi, anche un paio di settimane. Apple mette a disposizione due documenti, "*App Store Review Guidelines*" e "*Apple iOS Human Interface Guidelines*"[18], che evidenziano tutte le linee guida da rispettare per evitare un rifiuto.

Questo tipo di applicazione è una ottima scelta nel caso si voglia realizzare una applicazione complessa, che sfrutti tutte le funzionalità hardware e software del dispositivo, tenendo però in considerazione i costi, i tempi di realizzazione e la complessità nel realizzare e pubblicare eventuali aggiornamenti.

2.1.3 Hybrid Application

La realizzazione di una applicazione attraverso l'approccio ibrido cerca di prendere il meglio dei due metodi visti precedentemente. Si vedrà nel prossimo capitolo che con questo metodo si riducono i costi di portabilità e al contempo si riesce ad avere accesso a tutte le features dei dispositivi, facendo uso di un linguaggio comune a molti quale HTML5, CSS e JavaScript.

Per fare tutto questo si sfruttano anche vari framework, opensource e privati, per la realizzazione di una User Interface il più possibile rientrante nelle Human Interface Guideline di ciascuna piattaforma e per l'accesso e l'uso dei componenti interni dei dispositivi, quali fotocamera, GPS o altri, che non sono supportati da HTML5.

Di seguito una immagine che confronta graficamente i tre metodi visti, in funzione dell'efficienza e della portabilità:

[18] https://developer.apple.com/appstore/guidelines.html

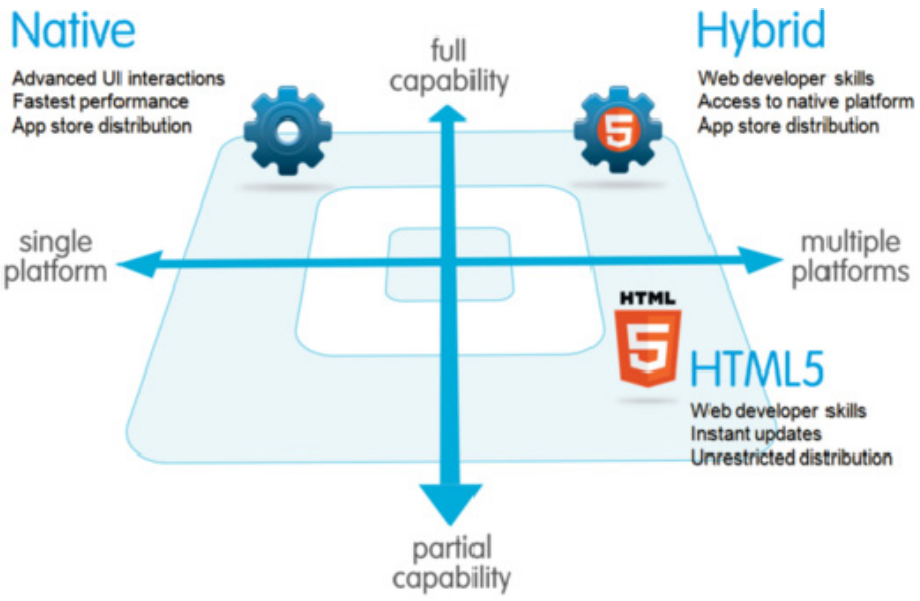

Figura 6: Confronto tra Applicazioni Native, Ibride e Web

2.1.4 Meccanismo utilizzato dalle Hybrid Application

Per realizzare applicazioni ibride, il codice viene scritto con un linguaggio comune per tutti i dispositivi (appunto HTML5, CSS e JavaScript) e successivamente impacchettato all'interno del wrapper nativo in funzione del sistema operativo scelto con il quale mettere in commercio l'applicazione.

Purtroppo per ora i browser non hanno completato l'implementazione di tutte le "*Device API*" facenti parte della specifica HTML5 pertanto non è ancora possibile accedere a tutte le features native dei device[19]. Per esempio Chrome e Safari per mobile non supportano l'uso della fotocamera, o l'utilizzo delle notifiche web (il W3C sta lavorando in questo senso per cercare di supportare il maggior numero di funzionalità dei dispositivi[20]).

[19] http://www.hostingtalk.it/it/c_000000IT/l-implementazione-di-html5-e-css3-tra-teoria-e-pratica
[20] http://www.w3.org/2009/dap/?ocid=aff-n-we-loc--DEV40909&WT.mc_id=aff-n-we-loc--DEV40909

Questo limite si è cercato di superarlo come detto in precedenza attraverso alcuni framework (esempio PhoneGap, vedi capitolo 2.2.2) che si pongono come ponte tra le API del dispositivo e il codice HTML - JavaScript.

Altri framework invece (esempio Sencha Touch, JQuery Mobile, etc…, vedi capitolo 2.2.1) vengono spesso utilizzati per facilitare la realizzazione dell'interfaccia grafica, in modo da avere, al momento dell'impacchettamento all'interno del wrapper nativo, una user interface rientrante nelle *Human Interface Guidelines* della piattaforma di riferimento, in modo da rendere quasi indistinguibile la differenza tra app nativa e app ibrida.

Vantaggi e Svantaggi

Il grande vantaggio delle hybrid application è che alla fine si produce una applicazione nativa, distribuibile tramite mobile Application Store, ma con il cuore sempre scritto attraverso HTML, CSS e JS. Avendo una porzione di codice in comune a tutte le piattaforme, si riesce a limitare i costi di portabilità.

Un altro vantaggio di queste applicazioni rispetto a quelle web, sta nella possibilità di poterle sfruttare in modalità offline. Essendo il codice dell'applicazione all'interno del device alla pari delle applicazioni native, possono essere eseguite in locale senza necessità di una connessione Internet attiva (sempre se non richiesta dall'applicazione stessa).

In confronto alle web app, quelle ibride azzerano i tempi di latenza dovuti al caricamento delle pagine Web, ma il rendering grafico è in ogni caso affidato al browser e non alle API grafiche fornite dal sistema operativo,

come nel caso delle applicazioni native. Questo significa che se l'applicazione che si vuole realizzare necessità di una notevole potenza grafica, come può essere un gioco, solo con le applicazioni native si riesce ad avere performance accettabili.

Un eventuale punto a sfavore per questo metodo di sviluppo, può essere la dipendenza dai framework, come scrive Benjamin Sandofsky, Teach Lead presso Twitter in un suo articolo:

> "*Shell apps require a bridge for their pages to interact with native chrome and call native APIs. Where does it come from?*
>
> *You could use an open source shell-app framework like PhoneGap, but that leaves you at the mercy of their schedule. If the native platform introduces a new API, or you run into an edge case that requires extending the shell framework, it could be months before you can implement your own app's functionality. One solution is to start from an open source framework and maintain a fork when the need arises. (…)"* [21]

In pratica vuole dire che se la piattaforma nativa introduce un aggiornamento con delle nuove funzionalità che si vorrebbero subito sfruttare, bisogna invece aspettare anche l'aggiornamento del framework. Se ci si viene a trovare in un caso in cui il framework non risulta adeguato per le necessità della propria applicazione bisogna provvedere ad estendere il framework stesso e questo significa un ritardo molto significativo nello sviluppo dell'app.

[21] Benjamin Sandofsky, "Shell Apps and Silver Bullets" (http://sandofsky.com/blog/shell-apps.html)

In definitiva si può dire che per applicazioni che non hanno bisogno di elaborazioni grafiche estremamente complesse, la soluzione ibrida è un ottimo compromesso. Fornisce una soluzione conveniente per la realizzazione di una vasta gamma di applicazioni, permettendo di avere tempi e costi accettabili, ma soprattutto consentendo la pubblicazione su molteplici piattaforme.

2.2 Framework

Per semplificare e rendere possibile la creazione di queste hybrid application, è possibile sfruttare dei framework, ovvero delle librerie JavaScript, che possono essere open source o proprietarie.

2.2.1 Framework per la User Interface

Quelli che vediamo ora, sono specifici nella realizzazione dell'interfaccia grafica, permettendo per esempio allo sviluppatore di non doversi creare tutti gli oggetti grafici tramite HTML5 e CSS, ma effettuando dei semplici Drag&Drop dei template presenti.

Questi framework infatti, possiedono delle collezioni di elementi già realizzati, i quali possono essere "trascinati" nella nostra applicazione ed è poi il framework stesso che si preoccupa della traduzione dell'elemento in codice.

Prendiamo in analisi due tra i più utilizzati, JQuery Mobile e SenchaTouch[22].

[22] http://www.codefessions.com/2012/05/which-mobile-javascript-framework-is.html

JQuery Mobile

JQueryMobile[23] è il più conosciuto e utilizzato tra i framework di questo genere. Il suo vantaggio principale sugli altri è che essendo “figlio” di JQuery, ha una curva di apprendimento minima per chi ha familiarità con questa nota tecnologia ed inoltre, essendo opensource, ha una grande comunità di developers.

La possibilità principale di questo framework è quella di poter creare più pagine in un unico file. Questo è possibile grazie all'uso del tag * * dove data-role="page" sta appunto a dichiarare al browser che ciò che è racchiuso in questo tag è da considerarsi una pagina indipendente dal resto del contenuto del file. Qualora volessimo quindi creare più pagine basterà ripetere questo tag più volte con id diversi.

Le pagine vengono caricate tramite AJAX, avendo JQuery un motore di navigazione AJAX che si attiva automaticamente. Nel DOM quindi entrano solo gli elementi all’interno del tag *div data-role="page".* Solo la pagina iniziale carica il tag <head>.

Attraverso la sua semplicità e il suo meccanismo di switch tra le pagine, si può dire che JQuery Mobile è particolarmente adatto ad applicazioni con contenuti statici.

[23] http://jquerymobile.com/

Sencha Touch

Sencha Touch[24] è il primo che ha ricostruito con precisione l'intera interfaccia grafica di iOS. Utilizza un paradigma ad oggetti per la realizzazione dell'interfaccia grafica, dove le varie classi rappresentano gli elementi dell'UI.

Sencha Touch deriva da una versione Desktop del framework, che si chiama Ext JS e da questa eredita la struttura delle classi.

Questo framework adotta il paradigma MVC come fondamenta sul quale costruire le applicazioni. Questo modello si basa su tre concetti di base:

- *Model*: rappresenta le entità che entreranno in gioco nell'applicazione. Definisce i dati esponendo alla *view* e al *controller* le funzionalità per l'accesso ad essi e il loro aggiornamento.
- *View*: la logica di presentazione dei dati viene gestita dalla *view*. Questo modulo gestisce la costruzione dell'interfaccia grafica. La *view* interpreta il *model* e delega al *controller* l'esecuzione dei processi richiesti dall'utente dopo averne catturato gli input e la scelta delle eventuali schermate da presentare.
- *Controller*: Implementa la logica di controllo dell'applicazione. Coordina *model* e *view* per ottenere il risultato voluto. È il responsabile del caricamento e salvataggio dei dati e ha la responsabilità di trasformare le interazioni dell'utente nella *view*.

Nella creazione di una applicazione con SenchaTouch vi sono sempre tre cartelle 'js/models', 'js/views', 'js/controllers', le quali ognuna contiene i JavaScript responsabili delle funzioni sopra descritte.

[24] http://www.sencha.com/products/touch

Un altro concetto fondamentale in Sencha sono gli *Store,* cioè la fonte principale per recuperare dati. Possono essere collegati direttamente ai componenti dell'interfaccia grafica (ad esempio una lista) oppure utilizzati direttamente dai *controller* per il caricamento di dati da un server.

Come primo impatto per gli sviluppatori, Sencha risulta più complesso ma, come si enuncia da IBM stessa in un suo articolo, di grande potenzialità:

> "*The maturity and functionality of this framework in such a new and evolving space—the mobile web—is noteworthy. In many ways, Sencha Touch has quickly become the gold standard that other frameworks are measured against."* [25]

Dal punto di vista pratico l'unico linguaggio su cui si lavora è JavaScript poiché il codice HTML viene generato automaticamente e sono già presenti template in CSS3 che riprendono le interfacce di Android, iOS, etc.

Sencha inoltre mette a disposizione anche un rad WYSIWYG[26] "Sencha Architect[27]" per la realizzazione vera e propria dell'interfaccia grafica, in modo molto simile all'Interface Builder di XCode per iOS, fornendoti l'immagine del dispositivo dentro la quale poter trascinare gli elementi.

[25] IBM developerWorks, http://www.ibm.com/developerworks/opensource/library/wa-senchawebdev/
[26] Rapid Application Development del tipo WYSIWYG ("quello che vedi è quello che è")
[27] http://www.sencha.com/products/architect

2.2.2 Framework per l'accesso alle API del dispositivo

Uno dei framework maggiormente adottati nella costruzione di applicazioni ibride è PhoneGap[28]. Questo framework specifico per il mobile è stato realizzato da Nitobi Software e successivamente acquistato da Adobe Systems nel 2011. Il software alla base di PhoneGap è Apache Cordova ed è totalmente opensource.

Con questo framework è possibile utilizzare la maggior parte dei dispositivi interni ai device quali GPS, accelerometro o fotocamera, ma anche di interfacciarsi con i servizi interni, quali contatti, calendario, impostazioni, etc.

PhoneGap supporta praticamente tutti i sistemi operativi mobile, iOS, Android, Windows Phone, BlackBerry, Bada, Symbian, etc.

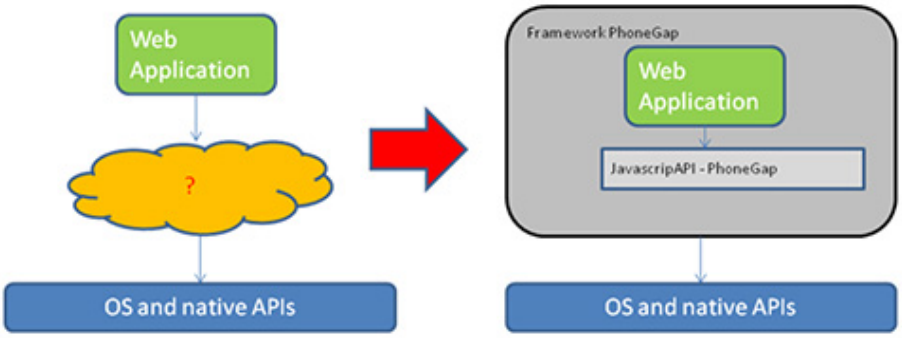

Figura 7: Funzionamento del framework PhoneGap

Come illustra la figura sopra, Phonegap, come altri framework per app ibride, esegue l'applicazione scritta in HTML e JS (web app), all'interno di un contenitore (una sorta di wrapper). Le chiamate tra web e nativo, a differenza di un'applicazione web eseguita in un browser, possono contare su un modulo che fa da ponte tra questi due.

[28] http://phonegap.com/

Per rendere il tutto indipendente dalle modalità realizzative del ponte e dalle differenze della piattaforma sottostante, Phonegap ha realizzato un livello di astrazione in javascript che permette allo sviluppatore di scrivere codice che funzioni su tutte le piattaforme.

PhoneGap utilizza il linguaggio nativo della piattaforma per accedere alle risorse hardware e software in modo da aggiungere le funzionalità di base al motore JavaScript e renderle così facilmente utilizzabili dall'applicazione come fossero tradizionali metodi di libreria[29].

Nell'immagine che segue è possibile vedere tutte le features che PhoneGap supporta dei vari sistemi operativi:

	iOS iPhone / iPhone 3G	iOS iPhone 3GS and newer	Android	OS 4.6-4.7	OS 5.x	OS 6.0+	WebOS	WP7	Symbian	Bada
ACCELEROMETER	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
CAMERA	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
COMPASS	✗	✓	✓	✗	✗	✗	✗	✓	✗	✓
CONTACTS	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
FILE	✓	✓	✓	✗	✓	✓	✗	✓	✗	✗
GEOLOCATION	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MEDIA	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗
NETWORK	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NOTIFICATION (ALERT)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NOTIFICATION (SOUND)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NOTIFICATION (VIBRATION)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
STORAGE	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗

Figura 8: Features supportate da PhoneGap

[29] http://www2.mokabyte.it/cms/article.run?articleId=O5R-R6L-HN8-8R8_7f000001_18359738_2ff5bd55

3 Descrizione del progetto

3.1 Descrizione generale

A conclusione dell'analisi fatta si è deciso di creare un progetto pilota che potesse in azienda essere usato sia come PoC (Proof of Concept) sia come base per lo sviluppo di applicazioni cross - platform in ambito Commerce IBM. A tale scopo sono stati analizzati:

- IBM Worklight, prodotto proprietario di IBM basato su Phonegap per la costruzione di applicazioni ibride, con lo scopo di testare la portabilità, l'accesso alle API native del device (come le push notification) e la possibilità di distribuzione dell'applicazione sugli Store.
- i servizi REST messi a disposizione dalla Commerce IBM (a partire dalla Feature Pack 4) per il login, la visualizzazione dei prodotti di un categoria, il checkout, etc. necessari per la realizzazione dello Store mobile.

Si è scelto di implementare un'applicazione di tipo hybrid, per poter offrire al cliente una applicazione facilmente distribuibile su numerose piattaforme, ma allo stesso tempo abbastanza performante per essere paragonata ad una di tipo nativo.

3.1.1 IBM Worklight

IBM Worklight[30] è una piattaforma di sviluppo per applicazioni mobile e tablet cross platform.

[30] http://www-01.ibm.com/software/mobile-solutions/worklight/

Si divide in 4 componenti principali:

- **IBM Worklight Studio** è un IDE basato su Eclipse[31] che consente agli sviluppatori di gestire tutte le attività di programmazione.
- **IBM Worklight Server** è un server basato su Java che opera come un gateway tra applicazioni e sistemi di back-end.
- **IBM Worklight Device Runtime Component** fornisce codice runtime che integra le funzionalità del server. Queste API facilitano l'accesso alle funzionalità native del device. Tra queste è utilizzato PhoneGap come bridge tra le tecnologie Web e le funzioni native.
- **IBM Worklight Console** è un'interfaccia utente amministrativa, web based, utilizzata per server, adapter, applicazioni e servizi push.

Le principali funzionalità dell'ambiente **IBM Worklight Studio** sono:

- Sviluppo di applicazioni web, native e hybrid per tutti i dispositivi che si basano su codice nativo e tecnologie web standard.
- API di accesso al dispositivo tramite codice nativo o linguaggi web standard attraverso il framework PhoneGap.
- Utilizzo di entrambe i linguaggi, nativo e web, nella stessa app.
- le API client side sono indipendenti dal tipo di framework usato e questo consente l'uso di librerie e framework di terze parti, quali Dojo Mobile, jQuery Mobile e SenchaTouch.

Le principali funzionalità di **IBM Worklight Server** invece sono:

- una gestione trasparente della scalabilità facendo cache delle richieste e limitando il carico verso i servizi per il quale è stato creato un adapter.

[31] http://www.eclipse.org/

- *Direct Update:* Aggiornamento diretto di applicazioni ibride e web senza passare dai relativi Store.
- Semplifica la gestione delle notifiche push, infatti permette di non occuparsi delle diversità presenti su ogni piattaforma, in quanto si occupa della gestione a bassa livello del *subscription* effettuato dagli utenti per ogni applicazione deploiata sul server.

IBM Worklight Device Runtime invece ha come funzionalità principali:

- Possibilità di autenticazione offline.
- Database locale per l'archiviazione di dati in modalità offline.
- Aggiornamento diretto delle risorse delle web app.

IBM Worklight Console svolge il ruolo di analizzare le statistiche dell'utente e di controllo del meccanismo di invio delle notifiche push.

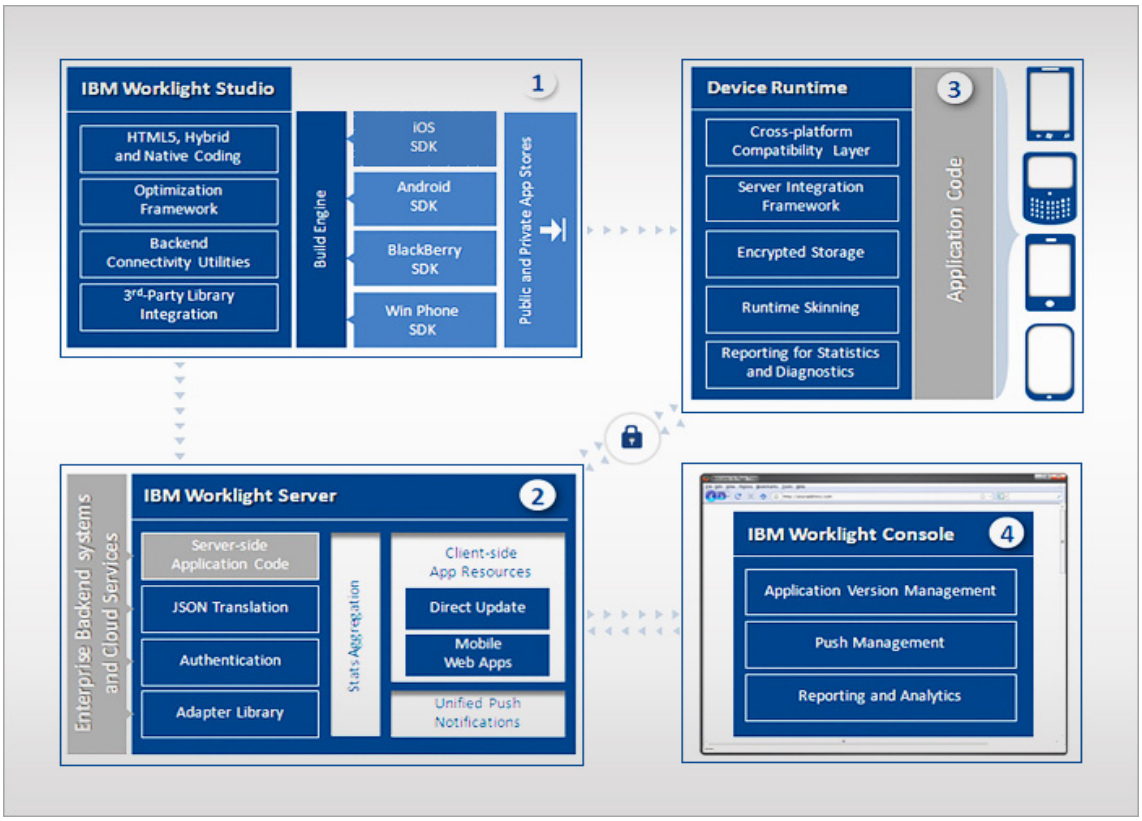

Figura 9: Componenti di Worklight

Push Notification

Le notifiche push sono la possibilità di un dispositivo di ricevere comunicazioni inviate da un server. Questi messaggi possono essere ricevuti indipendentemente dal fatto che l'applicazione sia in esecuzione oppure no.

Si è scelto di sfruttare questa tecnologia, in quanto risulta molto efficace a livelli di marketing per mantenere aggiornati gli utenti sia per quanto riguarda un prodotto in offerta sia per notificare un cambiamento della disponibilità di un prodotto.

3.1.2 Servizi REST

REST[32] è uno stile architetturale per la comunicazione client – server, alternativo rispetto a quello dei web service SOAP – based, basato esclusivamente su richieste HTTP, utilizzando i metodi standard GET, POST, PUT e DELETE.

È un alternativa leggera ai Web Service[33] e proprio per questo si rende adatto ad applicazioni di tipo mobile. Inoltre un servizio REST ha i vantaggi di essere:

- indipendente dalla piattaforma
- Indipendente dal linguaggio di programmazione
- Basato su un protocollo di comunicazione standard (HTTP)
- Utilizzando la porta 80 non necessita di configurazioni particolari.

[32] Representational State Transfer
[33] http://it.wikipedia.org/wiki/Web_service

All' interno di questo progetto, vengono chiamati i servizi REST implementati su IBM Websphere Commerce, soluzione leader per i progetti di commercio elettronico utilizzata dalle più importanti aziende italiane e internazionali[34].

Attraverso questi servizi è possibile effettuare tutte le operazioni principali per l'acquisto di un prodotto da uno Store.

Dall'infocenter[35] appunto di Websphere Commerce si trova la documentazione che illustra i dettagli per effettuare la richiesta ad un servizio:

- Protocollo: HTTP o HTTPS
- Autenticazione: richiesta oppure no
- HTTP Method: GET, POST, PUT o DELETE
- URL: a cui effettuare la richiesta
- Parametri: se richiesti
- Esempio
- Formato di scambio dati: JSON.

Nella realizzazione, per semplificare i test e non imbatterci nella SOP (Same Origin Policy[36]), sono stati implementati tutti i servizi utilizzando il protocollo HTTPS.

[34] http://ecommerce.tecla.it/ibm-websphere-commerce/
[35] http://pic.dhe.ibm.com/infocenter/wchelp/v7r0m0/index.jsp?topic=%2Fcom.ibm.commerce.starterstores.doc%2Fconcepts%2Fcwvrestapi.htm
[36] http://en.wikipedia.org/wiki/Same_origin_policy

4 Implementazione del progetto

4.1 Scelte implementative

Al momento di iniziare la realizzazione dell'applicazione è stato necessario fare delle scelte progettuali:

- *Sencha Touch* per la realizzazione dell'UI. Dopo l'analisi effettuata nel capitolo 2.2.1 si è scelto appunto questo framework per la sua maturità raggiunta e per la vasta comunità che può essere di supporto.
- *IBM Worklight* per la possibilità di rendere l'applicazione distribuibile su svariati Application Store, per la possibilità di accedere alle API native e per il supporto nell'interazione con la commerce lato server.
- *Android*, come ambiente di test. Grazie alla validità della soluzione ibrida scegliere Android o iOS o qualsiasi altra piattaforma è ininfluente, ma avendo l'azienda messo a disposizione un dispositivo Android, si è scelto questa come piattaforma di test.

4.2 Realizzazione

La realizzazione del progetto è stata suddivisa in due fasi. La prima riguardante lo sviluppo dell'applicazione web attraverso l'uso di Sencha Architect sulla base dei servizi REST della Commerce.

La seconda relativa all'incapsulamento dell'applicazione Sencha in una applicazione Worklight, così da poter realizzare l'ultima parte del progetto, cioè l'integrazione con le push notification.

Per effettuare le chiamate ai servizi REST sono state usate due metodologie:

- Chiamata AJAX dal controller: quando è necessario ottenere esclusivamente una conferma se la richiesta è andata a buon fine o no.
- Utilizzo degli Store di Sencha: quando la richiesta prevede di reperire una lista di dati (esempio i prodotti nel carrello o le categorie di prodotti). Uno Store per effettuare una chiamata ad un servizio REST utilizza un "*AJAX proxy*", nel quale è possibile settare tutte le informazioni quali URL, parametri o header. Come detto precedentemente viene utilizzato esclusivamente il formato JSON nei servizi REST di Websphere Commerce, quindi lo Store, che è indipendente dal formati di interscambio JSON o XML, è da configurare con un oggetto, "*JSON Reader*", che si occupa della parserizzazione della risposta sulla base del *model* impostato allo Store.

4.2.1 Prima fase: Sencha

Descrizione del flusso implementato

- Login
- Carrello e CheckOut
- Visualizzazione delle categorie e dei prodotti associati ad esse
- Inserimento di un nuovo prodotto nel carrello
- Logout

Il login è necessario in quanto tutti i successivi servizi hanno necessità di essere eseguiti da un utente registrato.

I servizi REST sono stateless e possono essere richiamati senza un concetto di sessione. Per questo motivo e per garantire che le chiamate vengano effettuate da un utente autenticato, viene utilizzato un sistema a due token, che vengono forniti dal server in risposta alla richiesta di login. Questi token devono poi essere passati in tutte le successive chiamate nell'intestazione della richiesta, per far si che lato server si possa riconoscere il client che sta effettuando la richiesta.

Sulla IBM Websphere Commerce sono previsti due tipi diversi di autenticazione:

- Autenticazione standard in seguito ad una registrazione, con inserimento di username e password.
- Autenticazione di tipo di *guest,* quindi per gli utenti ospiti che non intendono registrarsi ma possono comunque effettuare ordini.

Nell'applicazione si è scelto di implementare il primo metodo.

Effettuando il login è poi possibile richiamare le API REST per recuperare i dati relativi ai prodotti presenti nel carrello personale oppure richiamare il

servizio REST responsabile della conferma del carrello effettuando l'operazione di checkout, dopo la quale l'ordine viene chiuso per poi passare al pagamento (funzionalità non implementata nell'applicazione).

In seguito all'autenticazione è inoltre possibile richiamare il servizio responsabile dell'ottenimento delle categorie dei prodotti nello Store di riferimento e gli item che ognuna comprende.

Per finire è possibile aggiungere un nuovo prodotto nel carrello, inserendo l'id del prodotto e la quantità voluta.

Nell'immagine che segue è possibile vedere l'organizzazione del progetto. Tutti gli elementi nei paragrafi successivi fanno riferimento alle voci presenti nell'immagine.

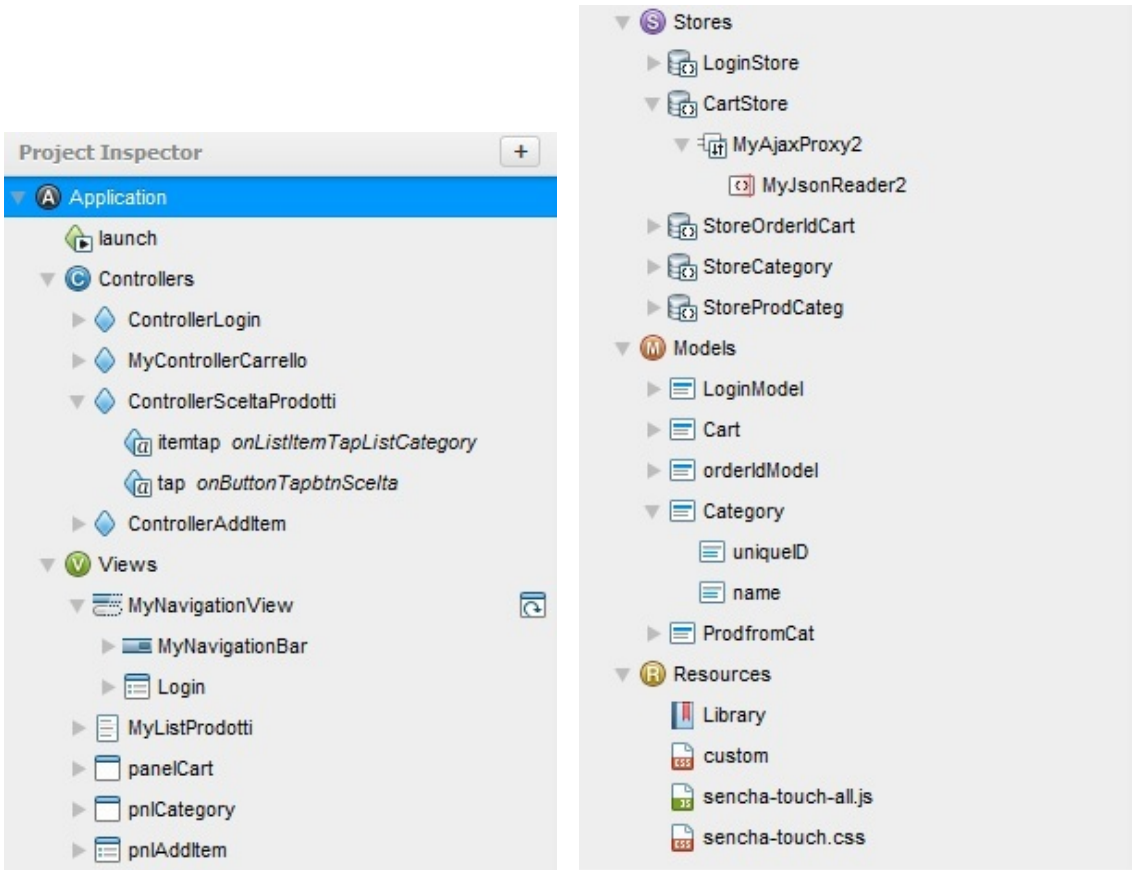

Figura 10: Project Inspector dell'applicazione

Le View e l'interfaccia grafica

La realizzazione dell'interfaccia grafica è avvenuta parallelamente alla realizzazione dei servizi REST.

Nella prima view vi è la form destinata ai dati per il login e tutti i *button* responsabili dell'attivazione dei principali servizi visti precedentemente.

Figura 11: View di partenza dell'applicazione

In questa view, che è quella iniziale, viene anche importato il *navigation view*, meccanismo che consente la navigazione attraverso le viste già percorse. All'interno del *navigation view* è stato inserito il bottone per il *logout,* in modo che sia disponibile in tutte le viste successive.

In questa vista sono stati introdotti i seguenti componenti:

- Un form per inserire username e password

- Button per effettuare il login
- Button per aprire il carrello
- Button per visualizzare le categorie
- Button per l'inserimento di un nuovo prodotto nel carrello
- Button per i settaggi delle Push Notification (argomento analizzato nel paragrafo 4.2.2)

Nella logica MVC del framework è stato necessario associare ad ogni button della view un *controller action*, nel quale è stata poi scritta la procedura per la gestione del **tap**.

Alla pressione del button "**effettua login**" viene lanciato un alert che comunica la riuscita o meno dell'autenticazione.

Effettuando il tap sul bottone "**Carrello**" si visualizza questa view, cioè una lista con un elenco di prodotti recuperati dal rispettivo Store:

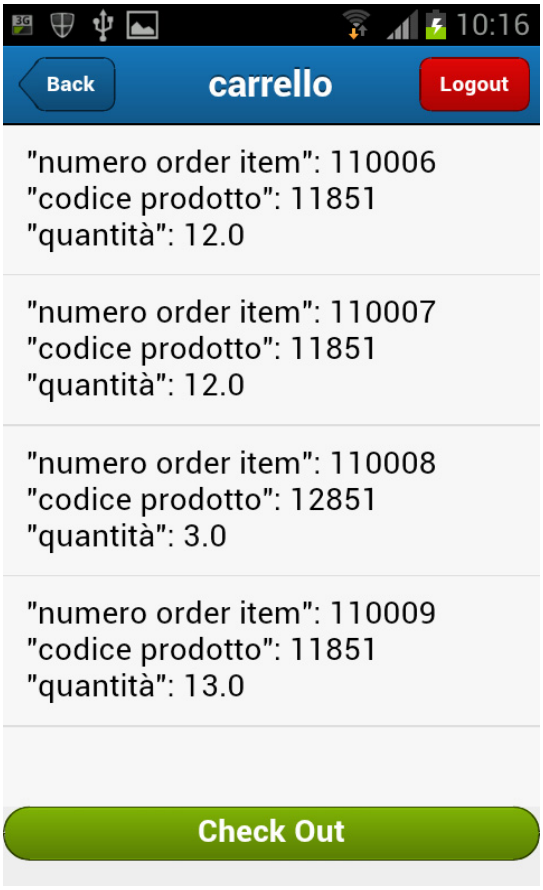

Figura 12: View per la visualizzazione del carrello

Premendo poi sul bottone “**check out**” si visualizza un alert come per il login, che riferisce se l’operazione è andata a buon fine. Effettuato il checkOut, se l’utente cercasse ti tornare in questa view, la lista verrebbe visualizzata senza elementi, in quanto l’ordine non è più nel carrello, ma pronto per la spedizione.

Premendo invece il button “**visualizza categorie**” dalla prima view, si passa ad una nuova vista con una lista simile a quella del checkout. Ottenuto l’elenco delle categorie, è poi possibile effettuare il **tap** su una di queste. In questo modo si ottiene un ulteriore lista con i dettagli dei prodotti presenti in quella specifica categoria.

In questo caso nella barra di navigazione viene caricato come titolo il nome della categoria su cui si è fatto il tap:

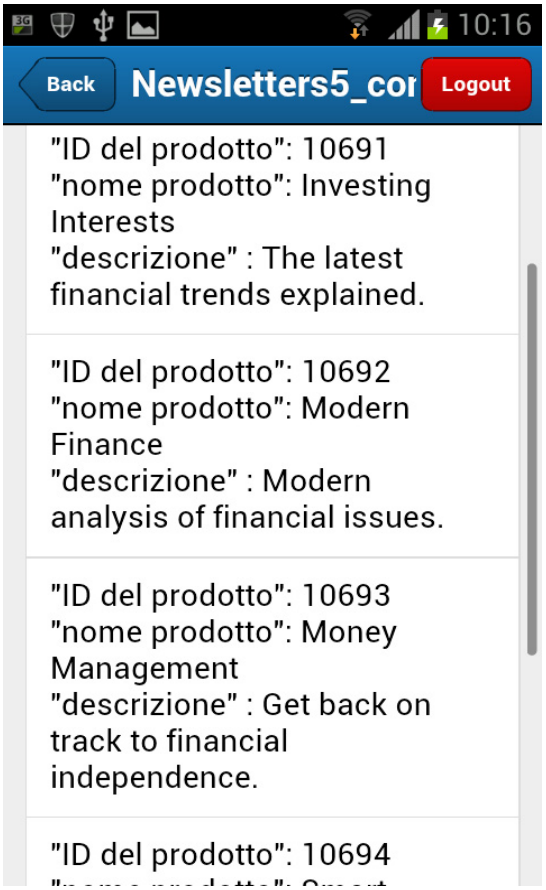

Figura 13: view dei prodotti per categoria

L'ultima azione effettuabile nell'applicazione è l'inserimento di un nuovo prodotto nel carrello, effettuando il tap sul button "**aggiungi prodotto**" nella prima view.

In questo modo si apre una nuova vista con una form in cui poter inserire il codice del prodotto e la quantità.

Il rispettivo controller action recupera questi dati dalla form e li invia come parametri nella richiesta REST apposita, che, se va a buon fine, rende visibile un *alert* di conferma:

Figura 14: alert di conferma inserimento prodotto

Gli elementi del flusso e i rispettivi Controller

Login

Il primo servizio utilizzato è il *login.* Questo servizio richiede di effettuare una richiesta di tipo POST con username e password dell'utente.

La risposta del server è un JSON con 4 valori:

```
{
   "WCToken" : "xxxxxxxxxxxxxxxxxxxxxx",
   "WCTrustedToken" : "xxxxxxxxxxxxxxxxxxxxxxxx"
   "personalizationID" : "1320436510481-2",
   "userId" : "3"
}
```

I primi due valori, *WCToken* e *WCTrustedToken*, sono quelli necessari da inserire nell'header di tutte le richieste degli altri servizi, in modo che vengano considerati autenticati.

Di seguito il codice del *controller* relativo al login, `onButtonTapbtnLoggo: function(button, e, options)` con il settaggio dell'header della richiesta:

```
//richieta REST per il login
var store = Ext.getStore("LoginStore");
//proxy per poter inserire i dettagli necessari per la chiamata
var proxy = store.getProxy();
//setWriter serve affinchè i dati in setExtraParam vengano
considerati dei JSON
proxy.setWriter({
    type : 'json',
    encodeRequest: true
});
proxy.setHeaders("content-type","application/json");
proxy.setExtraParam("logonId", form.getValues().Username);
proxy.setExtraParam("logonPassword", form.getValues().Password);
```

Come è possibile vedere dal codice, il servizio è chiamato attraverso lo Store *LoginStore*, che appunto recupera i token attraverso il model *LoginModel.*

Il WCToken e il WCTrustedToken vengono salvati in variabili globali dell'applicazione (`MyApp.WCToken` e `MyApp.WCTrustedToken`), in modo da poter essere disponibili successivamente in ambiti diversi dal controller del login.

Carrello

Uno dei servizi richiamabili solo in seguito al login è la visualizzazione del carrello. Il servizio si chiama *getCart.*

La richiesta viene effettuata con metodo GET e necessita l'inserimento nell'header dei *token* del login, necessari perché vengano visualizzati i dati relativi al carrello dello specifico utente loggato.

La chiamata viene effettuata dallo Store *CartStore* che è associato al model *Cart* per recuperare da ogni elemento del carrello le informazioni necessarie, quali codice del prodotto e quantità.

Il controller relativo a questo servizio è molto semplice, consiste esclusivamente nel settaggio dei parametri dell'header e nella chiamata al load dello Store.

Checkout

L'operazione di conferma dell'ordine nel carrello consiste in due servizi separati, che poi vengono chiamati uno di seguito all'altro. Il *preCheckout* e il *checkOut.* Il primo blocca l'ordine, mentre il secondo cambia proprio lo stato confermandolo definitivamente. Questi due servizi devono necessariamente essere richiamati in questo ordine.

Entrambi i metodi necessitano chiaramente dell'autenticazione, quindi impostano nell'header i token come visto precedentemente.

Il *preCheckout* effettua il servizio con metodo PUT mentre il *checkOut* con metodo POST.

Entrambi devono spedire come oggetto della richiesta un JSON con l'*orderId* dell'ordine nel carrello. Questo viene recuperato attraverso lo Store *StoreOrderIdCart,* che attraverso il model ad esso associato, *orderIdModel,* effettua una chiamata uguale a *getCart*, ma recupera esclusivamente l'*orderId* e non tutti i dettagli del carrello.

Di seguito un frammento del controller `onButtonTapChechOut: function(button, e, options)` che evidenzia l'AJAX request per il *preCheckout*:

```
Ext.Ajax.request({
    url : ' https://.../10151/cart/@self/precheckout',
    jsonData: {'orderId' : MyApp.idOrdine},
    method: 'PUT',
    headers: {
        'content-type':"application/json",
        'WCToken':MyApp.WCToken,
        'WCTrustedToken':MyApp.WCTrustedToken
    },
    success :function(response, options) {
      console.log("precheckout andato a buon fine");
    },
    failure : function(response, options) {
        console.log("precheckout NON effettuato");
    },
    async:false
});
```

In questa funzione si può vedere l'ultimo parametro "`async:false`"; è stato inserito perché essendo le due richieste di preCheckout e checkOut in cascata, e sapendo che AJAX è asincrono, la seconda delle due veniva richiamata prima del termine della precedente.

Poiché è necessario che il preCheckout sia terminato prima che parta il checkOut, questa clausola impone al primo di non eseguirsi in modo

asincrono. Cioè passare il controllo alle operazioni successive, solo quando terminato.

Categorie

La visualizzazione delle categorie di prodotti avviene attraverso il servizio *findTopCategories.* Come per la visualizzazione del carrello, è un servizio che si richiama attraverso una richiesta di tipo GET e dovendo restituire un elenco di categorie, non è stato richiamato attraverso una semplice richiesta AJAX, ma facendo uso di uno Store specifico per la richiesta.

Lo Store di riferimento è *StoreCategory* e ha il compito di recuperare l'elenco delle categorie.

Esattamente come per il servizio *getCart,* il controller associato è molto semplice. Oltre al load dello Store, vi è il settaggio dell'header dei token. Mentre l'url viene inserito direttamente nei settaggi dell'AJAX Proxy dello Store.

Prodotti delle categorie

Il servizio che restituisce la lista dei prodotti partendo dall'identificativo di una categoria si chiama *findProductsByCategory* e prevede una chiamata in GET. L'id della categoria non viene passato come parametro nel campo dati della richiesta, ma viene inserito al termine dell'URL, come in questo esempio:

*" . . . /wcs/resources/store/10151/productview/byCategory/**10002** "*

Questo servizio si differenzia dagli altri, in quanto l'URL non è possibile inserirlo nell'AJAX Proxy direttamente, ma deve essere inserito dal

controller action in modo dinamico, poiché si differenzia in funzione della categoria che si seleziona tra quelle ritornate dal servizio precedente.

Questo è possibile farlo in quanto il *controller action* associato all'*itemtap* su un elemento di una lista, mette a disposizione un parametro, `record`, che memorizza i dati dell'elemento su cui è stato effettuato il tap:

```
onListItemTapListCategory: function(dataview, index, target, record, e, options) {. . .}
```

Attraverso le proprietà del parametro `record`, riesco a recuperare l'`uniqueID` della category da inserire in coda all'url per il recupero dei prodotti:

```
var idCategory = record.get('uniqueID');

proxy.setUrl(". . ./store/10151/productview/byCategory/"+ idCategory);
```

Lo Store utilizzato è *StoreProdCateg,* associato al model *ProdfromCat.*

Aggiunta prodotto al carrello

Il servizio da richiamare per aggiungere un nuovo prodotto al carrello è *addOrderItem.*

Si effettua con autenticazione e con metodo POST, passando come parametri il *productId* e la *quantity* del prodotto che si vuole aggiungere.

Non avendo la necessità di collegare direttamente i dati letti ad un componente della *view* si è deciso di usare una chiamata AJAX direttamente dal controller, visualizzando tramite *alert* il successo o il fallimento dell'aggiunta.

Questo è il codice del controller, `onButtonTapbtnConfirmAdd function (button, e, options)`:

```
var form = Ext.getCmp('pnlAddItem');
Ext.Ajax.request({
    url : "https://localhost:443/wcs/resources/store/10151/cart",
    jsonData:{  'orderItem': [{'productId':
form.getValues().productId,'quantity': form.getValues().quantity}]},
    method: 'POST',
    headers: {
        //'content-type':"application/json",
        'WCToken':MyApp.WCToken,
        'WCTrustedToken':MyApp.WCTrustedToken
    },
    success :function(response, options) {

        Ext.Msg.alert("Prodotto aggiunto: ", response.responseText);
        console.log("successo, prodotto aggiunto : " +
response.responseText);
    },
    failure : function(response, options) {
        Ext.Msg.alert("Prodotto NON aggiunto: ",
response.responseText);
        console.log("fallimento, prodotto non aggiunto : " +
response.responseText);
    }
});
```

Logout

Il servizio che effettua il *logout* dell'utente invalida i token ricevuti precedentemente con il servizio di login, quindi determina la fine della sessione autenticata. Richiamato questo servizio, non è poi più possibile utilizzare i token ricevuti nella fase di login precedente per nessun servizio.

La chiamata avviene attraverso una DELETE, ma sempre da autenticati, quindi con il passaggio dei token nell'header. Anche in questo caso, i dati letti non vanno associati a nessuna view, quindi il servizio è stato richiamato con una richiesta AJAX nel controller action `onButtonTapbtnLogout function(button, e, options).`

4.2.2 Seconda fase: Worklight

Importazione in ambiente Worklight

Realizzata la parte su Sencha si è passati alla seconda fase; la sperimentazione delle push notification sfruttando l'ambiente di IBM Worklight.

Per importare l'applicazione creata su Sencha Architect in Worklight è necessario copiare i file creati da Sencha al momento della pubblicazione, all'interno del progetto di Worklight. Un primo adattamento, poiché l'applicazione non veniva più eseguita su un application server, è stato quello di impostare tutti gli URL da relativi ad assoluti.

È stato necessario inoltre effettuare una seconda modifica riguardante l'inizializzazione di Worklight. Al posto di richiamare l'*initialize* di Worklight al dom ready, si è deciso di richiamarlo al ready dell'applicazione sencha, in modo da avere le inizializzazioni di Worklight e Sencha in un javascript unico, *app.js*:

```
launch: function() {
     Ext.create('MyApp.view.MyNavigationView', {fullscreen: true});
     try{
            //Inizializzazione di Worklight
            WL.Client.init(wlInitOptions);
     }
     catch(e){
            WL.Toast.show("errore nell'init di worklight" + e);
     }
```

Effettuati questi accorgimenti è stato possibile eseguire l'applicazione così come era stata creata su Sencha Architect, direttamente su dispositivo e verificare anche in questo ambito il corretto funzionamento dei servizi.

A questo punto è stato possibile iniziare la realizzazione delle push notification.

Realizzazione Push Notification

Per l'utilizzo delle push notification è necessario agire su 3 livelli differenti:

- Client side nell'applicazione
- Server side con la costruzione di un adapter in Worklight
- A livello di piattaforma, con la configurazione di un Push Service Mediator.

A *livello applicativo* per aggiungere tutta la logica necessaria per la gestione delle push notification: la verifica dell'abilitazione della ricezione delle notifiche push, il subscribe e il codice relativo da richiamare all'atto della ricezione di una notifica.

A *livello server side* con la realizzazione di un Adapter. Un *Adapter* è un livello di trasporto utilizzato da Worklight per connettere il client ai vari servizi di back-end. Gli adapter sono usati per recuperare informazioni ma anche per effettuare azioni non implementate nel client. Per le notifiche push si utilizza un HTTP Adapter. Nell'immagine che segue è possibile vedere l'iterazione dell'adapter tra client e back-end:

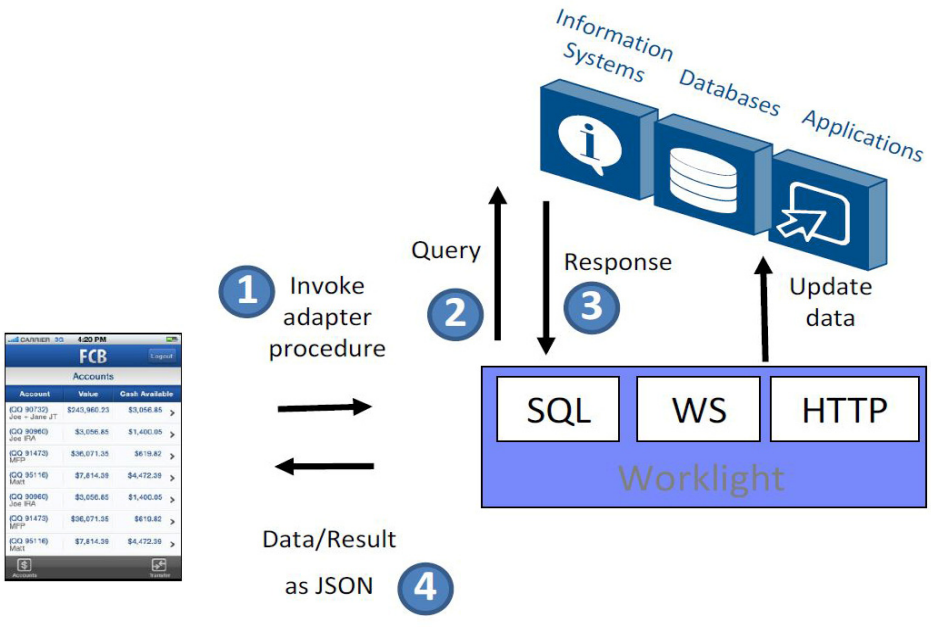

Figura 15: funzionamento degli adapter

A *livello di piattaforma* la configurazione di Google Cloud Messaging[37] come Push Service Mediators.

Ogni dispositivo si deve registrare a questo servizio, il quale poi si occupa di effettuare il push della notifica (ricevuta dall'adapter) sui rispettivi dispositivi.

Per iniziare a ricevere notifiche un'applicazione deve per prima cosa effettuare il *subscribe* ad un *event source* apposito. Un event source è un canale a cui una applicazione si può registrare e viene dichiarato nell'adapter.

Effettuando il *subscribe*, Worklight si occupa automaticamente di ottenere il token del device dal push service e di sottoscrivere il dispositivo all'event source.

Flusso necessario per l'invio e la ricezione di una notifica push:

- una notifica viene recuperata dall'event source dell'adapter.
- l'adapter processa la notifica e la invia a GCM
- GCM invia la notifica ai device
- I device visualizzano la notifica.

Nella configurazione dell'adapter la prima cosa da effettuare è la creazione dell'event source. Questo viene fatto nel file dell'adapter *testingAdapter-impl.js:*

```
WL.Server.createEventSource({
      name: 'PushEventSource',
      onDeviceSubscribe: 'deviceSubscribeFunc',
      onDeviceUnsubscribe: 'deviceUnsubscribeFunc'
});
```

[37] http://developer.android.com/google/gcm/index.html

Sempre in questo file vi è l'implementazione della procedura responsabile dell'invio della notifica:

```
function submitNotification(userId, notificationText){
	//si ottiene le informazioni su tutte le sottoscrizioni
dell'utente. Le può avere per numerosi device.
	var userSubscription =
	WL.Server.getUserNotificationSubscription(
	'testingAdapter.PushEventSource', userId);

	if (userSubscription==null){
		return { result: "No subscription for user " + userId };
	}

	WL.Server.notifyAllDevices(userSubscription, {
		badge: 1,
		sound: "sound.mp3",
		activateButtonLabel: "ClickMe",
		alert: notificationText,
		payload: {
			foo: 'bar'
		}
	});
}
```

Client side, invece, i settaggi da fare riguardano la registrazione all'*event source* e il *subscribe* da parte dell'utente.

Nel javascript principale, *app.js*, la prima cosa fatta è stata la registrazione all'*event source:*

```
function wlCommonInit(){
	WL.Client.Push.onReadyToSubscribe = function(){
		WL.Toast.show("onReadyToSubscribe");
		WL.Client.Push.registerEventSourceCallback(
				"myPush",
				"testingAdapter",
				"PushEventSource",
				pushNotificationReceived);
	};
};
```

Questa funzione `wlCommonInit()` viene chiamata non appena è stato inizializzato Worklight.

Con il metodo `WL.Client.Push.onReadyToSubscribe` si predispone al *subscribe* da parte dell'utente, registrando il nome dell'adapter utilizzato,

l'event source e un funzione, la quale viene chiamata alla ricezione della notifica, `pushNotificationReceived`:

```
function pushNotificationReceived(props, payload){
      alert("notifica :: " + JSON.stringify(props.alert));
      WL.Logger.debug("!notifica::" +JSON.stringify(props.alert));
}
```

Per consentire all'utente di decidere se effettuare il *subscribe* o nel caso di effettuare l'*unsubscribe*, è stato necessario aggiungere una view nel progetto di Sencha con quattro button e i relativi controller:

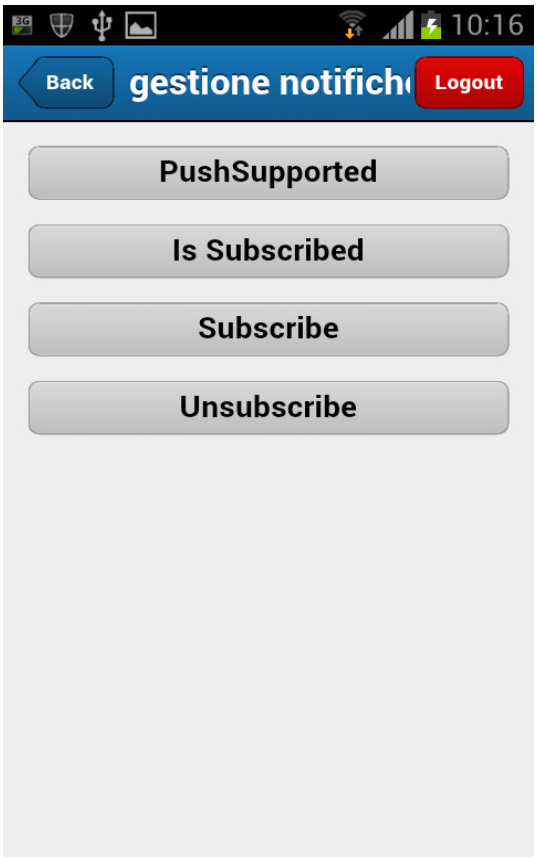

Figura 16: view per la gestione delle push notifications

Alla pressione di "PushSupported" viene chiamato il metodo `WL.Client.Push.isPushSupported()` che visualizza un alert con true se il device supporta le notifiche push.

Premendo "Is Subscribed" si esegue il metodo `WL.Client.Push.isSubscribed('myPush')` che come quello precedente ritorna *true* in un alert se il subscribed è già avvenuto. La funzione associata alla pressione di "Subscribe" è:

```
onButtonTapbtnSubscribe: function(button, e, options) {
        WL.Client.Push.subscribe("myPush", {
            onSuccess: pushSubscribe_Callback,
            onFailure: pushSubscribe_Callback
      });
    }

function pushSubscribe_Callback(response){
       Ext.Msg.alert("Subscribe", WL.Client.Push.isPushSupported());
}
```

Il metodo `WL.Client.Push.subscribe` effettua il *subscribe* mentre `WL.Client.Push.isPushSupported()` restituisce *true* se è avvenuto correttamente.

In modo analogo è la funzione per la pressione su "*Unsubscribe*".

Realizzati questi settaggi, è necessario registrarsi al Google Cloud Messaging e per fare questo bisogna avere un account Google.

Da Google APIs Console[38] è necessario abilitare il Google Cloud Messaging e memorizzarsi l'*APY key* e il *Project ID* [39].

Ottenuti questi due valori, è poi necessario, settarli nell'*application-descriptor.xml* della nostra applicazione:

```
<pushSender key='AIzaSyA5L5dTW8dsHyMxfXGkAePB-IDEcgKo6zg'
            senderId='777851127789'/>
```

[38] https://code.google.com/apis/console/

[39] Identificativo alfanumerico rilasciato dalla APIs console di Google che abilita il servizio GCM per una determinata applicazione.

Fatto questo ultimo passaggio, l'applicazione è predisposta completamente alla ricezione di notifiche push.

Per testarla non si è scelto di realizzare un ulteriore applicativo con il solo scopo di gestire la creazione e l'invio delle notifiche, in quanto non era negli obiettivi del progetto.

Perciò si è semplicemente scelto di invocare manualmente la procedura dell'adapter `submitNotification(userId, notificationText)` specificando l'userId del dispositivo in uso e il testo da inserire nella notifica, come è possibile verificare nell'immagine seguente:

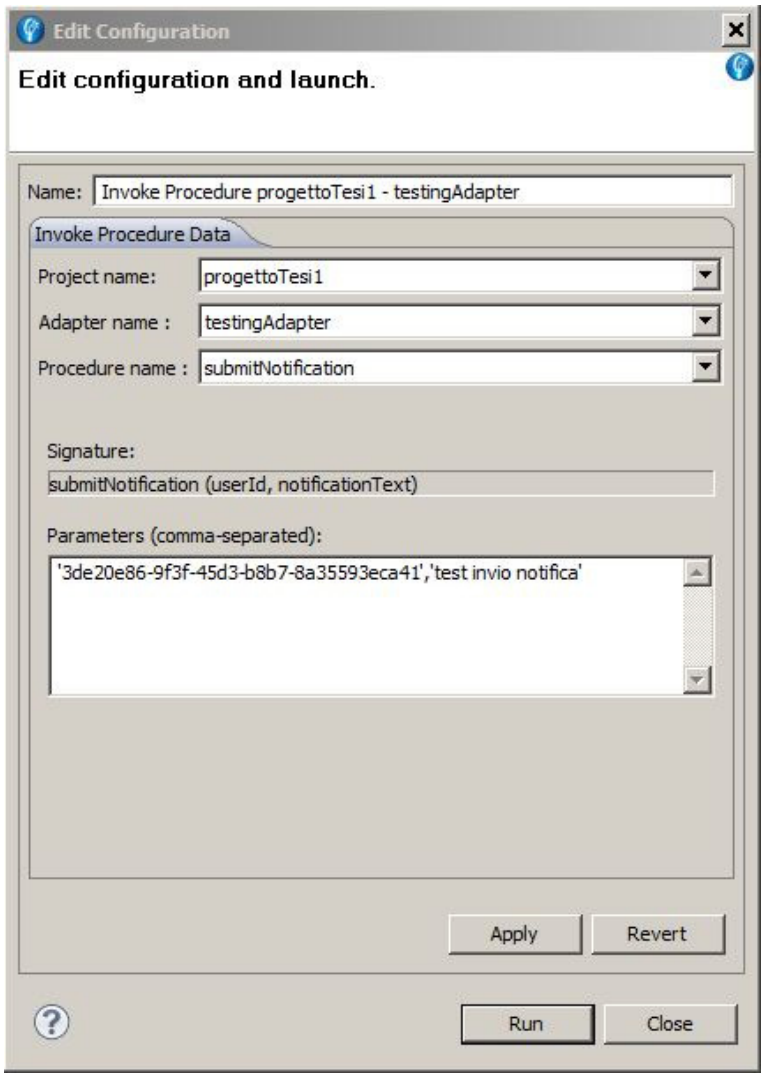

Figura 17: chiamata della procedura dell'adapter

Di seguito uno screenshot della ricezione della notifica:

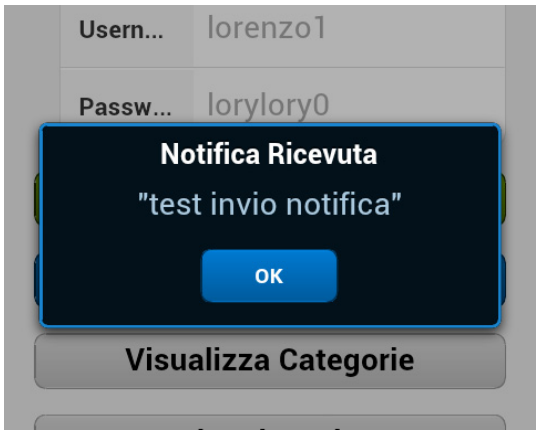

Figura 18: screenshot al momento della ricezione di una notifica

5 Conclusioni

L'obiettivo della tesi era la realizzazione di un'applicazione che facesse da punto di partenza nell'azienda per la realizzazione di future applicazioni per i clienti.

L'applicazione si può ritenere complessivamente completa e in linea con gli obiettivi che si erano dati, raggiungendo un discreto risultato, essendo riusciti non solo a risolvere il problema di portabilità su più piattaforme attraverso il supporto di Worklight, ma rendendo anche l'applicazione distribuibile come web app (escludendo in questo caso le push notifications, in quanto superflue per applicazione di tipo web).

Possibili sviluppi futuri potrebbero essere sicuramente una maggiore attenzione alla grafica. Tra le richieste dell'azienda, non vi era di prendere in considerazione particolari aspetti grafici, ma esclusivamente la parte funzionale dell'applicazione. Per questo motivo sono stati utilizzati esclusivamente i template di default del framework Sencha.

Una possibile estensione potrebbe essere la realizzazione di un servizio che si interfaccia direttamente con l'adapter, così da poter gestire le notifiche push in modo più completo, invece che richiamare manualmente la procedura.

Un'ulteriore revisione potrebbe comprendere la realizzazione anche su altre piattaforme come Windows Phone e iOS, per osservare ad esempio quali sono i metodi da loro utilizzati per la gestione appunto delle push notifications.

Questo potrebbe essere molto utile anche per poter confrontare le applicazioni create sulle diverse piattaforme e così analizzare le possibili differenze, sia grafiche che di prestazioni.

Gli sviluppi futuri in generale possono essere molteplici, soprattutto in relazione all'azienda in cui è stato realizzato questo progetto. Infatti l'azienda offre una grande varietà di servizi ai clienti e le possibili realizzazioni di questi in ambito mobile porterebbero certamente un allargamento dell'offerta da parte dell'azienda.

6 Bibliografia e Guide

1. Anthony T. Holdener III, *Ajax: The Definitive Guide*, O'Reilly, 2008
2. Prof. Mirko Ravaioli, *Dispense del corso di Mobile Web Design*, 2011/2012
3. Pellegrino Principe, *HTML5 CSS3 e javascript*, APOGEO, 2012
4. Adrian Kosmaczewski, *Sencha Touch 2 Up and Running*, O'Really, 2013
5. Massimo Carli, *Android 3 Guida per lo sviluppatore*, Apogeo, 2011
6. *Guida allo sviluppo Android*, http://android.devapp.it/
7. HTML.it, *Sencha Touch 2.0*, http://www.html.it/articoli/sencha-touch-20-1/
8. *Sencha Documentation*, http://docs.sencha.com/architect/2/#!/guide
9. *IBM Worklight Documentation*, http://www.ibm.com/developerworks/mobile/worklight/getting-started.html
10. *Documentazione ufficiale Google Cloud Messaging*, http://developer.android.com/google/gcm/index.html
11. *Documentazione ufficiale IBM Worklight,* http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp?topic=/com.ibm.iea.worklight/plugin_coverpage.html