ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea in Ingegneria Informatica

SVILUPPO CROSS-PLATFORM DI APPLICAZIONI MOBILE: ANALISI DELLO STATO DELL'ARTE E SPERIMENTAZIONI IN TITANIUM

Elaborata nel corso di: Sistemi Operativi LA

Tesi di Laurea di:
FRANCESCA CIMATTI

Relatore:
Prof. ALESSANDRO RICCI

Co-relatore:
Prof. ANDREA SANTI

ANNO ACCADEMICO 2013–2014
SESSIONE I

PAROLE CHIAVE

Sviluppo di mobile app

Multi-platform

HTML5

JavaScript

Titanium

Alla mia famiglia che mi ha permesso di arrivare fino a qui.

Indice

Introduzione

L'evoluzione vertiginosa delle tecnologie che c'è stata negli ultimi anni, ed in particolare la progressiva riduzione delle dimensioni dei componenti hardware insieme all'aumentare delle capacità di calcolo e delle possibilità di interconnessione, ha determinato prima la diffusione dei pc portatili, e successivamente quella di smartphone e tablet. Contestualmente quindi è sorta l'era del *mobile computing*, ed è iniziata l'evoluzione degli approcci alla programmazione delle applicazioni mobile. Il mobile computing[1] consiste nell'interazione uomo-computer in mobilità, e comprende la comunicazione mobile, l'hardware mobile e il software mobile.
Oggi i dispositivi mobile sono diventati pervasivi, infatti vi è quasi un dispositivo per ogni persona, e il desiderio sempre più crescente da parte di utenti e aziende di sfruttarne al meglio le potenzialità, ha conferito notevole importanza al mobile computing.

Oggi è possibile classificare gli *approcci alla programmazione delle applicazioni mobile* in tre categorie principali:

- nativi;
- web-based;
- ibridi.

Gli approcci *nativi* sono quelli classici, sorti contestualmente allo sviluppo delle piattaforme mobile, che quindi sfruttano gli strumenti e i linguaggi specificamente messi a disposizione dagli sviluppatori della particolare piattaforma. Questi approcci permettono di sviluppare app perfettamente integrate con la piattaforma sottostante, con la quale interagiscono direttamente attraverso chiamate di sistema, sfruttando al meglio tutte le funzionalità hardware e software del device, e risultando così molto performanti.

Il principale svantaggio di questo approccio è la mancanza di portabilità delle app risultanti. Infatti questa caratteristica vincola gli sviluppatori a realizzare un'app per ogni piattaforma mobile per raggiungere gli utenti di ognuna di esse, con la difficoltà di dover conoscere tutto l'insieme vasto ed eterogeneo di strumenti e linguaggi di sviluppo supportati.
Gli approcci *web-based* sono sorti dall'evoluzione delle tecnologie web, in particolare HTML5, CSS3 e JavaScript sulle quali si basano per astrarre dalle piattaforme mobile sottostanti. Questi approcci consentono di sviluppare app molto portabili, eseguite in parte all'interno del browser, e in parte su server remoti, alleggerendo il carico computazionale del dispositivo che deve solo mostrare il risultato. Proprio per la loro architettura le loro performance sono vincolate da quelle del motore di rendering del browser e le funzionalità del device che si possono sfruttare sono strettamente dipendenti dal grado di evoluzione delle tecnologie web.
Gli approcci *ibridi* sono nati dalla combinazione dei due approcci precedenti, per sfruttare al meglio le potenzialità di entrambi, e si concretizzano attraverso lo sviluppo dell'essenza dell'app tramite le tecnologie web, e di un involucro in linguaggio nativo per integrarla alla piattaforma. Anche questi approcci permettono di realizzare app molto portabili, che risultano flessibili e in grado di sfruttare molte funzionalità del device, oltre ad azzerare i tempi di latenza dovuti al caricamento delle pagine, dato che l'app è installata sul device. Vantaggi e svantaggi di questi approcci dipendono molto dall'architettura e dalla bontà dello strumento utilizzato per lo sviluppo.

Vediamo ora obbiettivo e organizzazione della tesi. L'obbiettivo di questa tesi è fare una panoramica sullo stato dell'arte degli approcci di sviluppo di mobile app alternativi agli approcci nativi, e poi approfondire ulteriormente l'analisi, ed in particolare il framework Titanium, valutato come il più interessante, mediante lo sviluppo di un'app concreta e facendo quindi il confronto con lo sviluppo usando approcci nativi.
La tesi quindi è strutturata come segue:

- il capitolo 1 offre uno scenario delle applicazioni mobile attuali, individua i problemi principali dello sviluppo mobile nativo, chiarisce il significato di sviluppo mobile multi-platform, e passa in rassegna i frameworks basati sulle tecnologie web, differenziandoli per tipologie e caratteristiche;

- il capitolo 2 approfondisce i frameworks individuati nell'analisi precedente, che sono risultati più interessanti in considerazione anche dei risultati ottenuti dallo sviluppo di una semplice applicazione di test, e ne individua pregi e difetti;

- il capitolo 3 mostra il processo di sviluppo di un'app tramite Titanium, il framework di riferimento scelto in questo lavoro di tesi, analizzandone le criticità incontrate, dopo aver descritto nel dettaglio la piattaforma Titanium, considerandone le potenzialità e i principi di funzionamento;

- il capitolo 4 pone l'app sviluppata a confronto con le corrispondenti versioni native, considerando difficoltà tecniche e risultati legati alla User Experience, e trae le conseguenti conclusioni in merito all'effettiva utilità di Titanium.

Capitolo 1

Stato dell'arte dello sviluppo mobile

Il rapido e continuo progresso dell'ingegneria elettronica dell'ultimo decennio ha reso possibile la miniaturizzazione dei ciruiti integrati, dando il via alla produzione di dispositivi sempre più portatili, perché sottili e leggeri. Nonostante le dimensioni veramente ridotte di questi dispositivi, essi dispongono di display ad alta risoluzione, e garantiscono prestazioni molto elevate, grazie a processori in continua evoluzione e memorie sempre più capienti.

Figura 1.1: Evoluzione dei telefoni cellulari

Questo nuovo scenario ha contribuito all'avvento dei dispositivi mobile come smartphone e tablet rendendo il confine fra PC e questi dispositivi sempre più sfumato.

Nel 2011 le vendite mondiali di smartphone e di tablet hanno superato per la prima volta quelle dei PC. Il mercato degli smartphone però è letteralmente esploso già dal 2007, anno in cui Apple presentò il primo iPhone, reinventando il telefono cellulare. Il cellulare divenne così uno strumento ricco di funzionalità, utile non più solo per comunicare, ma per navigare in rete, ascoltare musica, giocare, scattare foto etc.
Contemporaneamente ad Apple entrò nel mercato anche Google, tramite lo sviluppo del sistema operativo Android. Successivamente si sono introdotte anche tante altre aziende proponendo i loro sistemi operativi mobile. La nascita della concorrenza ha dato impulso alla produzione di smartphone più economici, che in questo modo sono diventati fruibili dalla maggior parte della popolazione. L'estrema versatilità insieme ad interfacce utente molto user-friendly ha reso possibile l'utilizzo dei dispositivi mobile anche da parte di privati, e non più solo da parte di aziende e professionisti. Il mercato di questi dispositivi è in continua espansione, ed essi assumono un ruolo sempre più importante sia nelle aziende che nella vita privata, infatti li portiamo costantemente con noi e siamo sempre connessi.

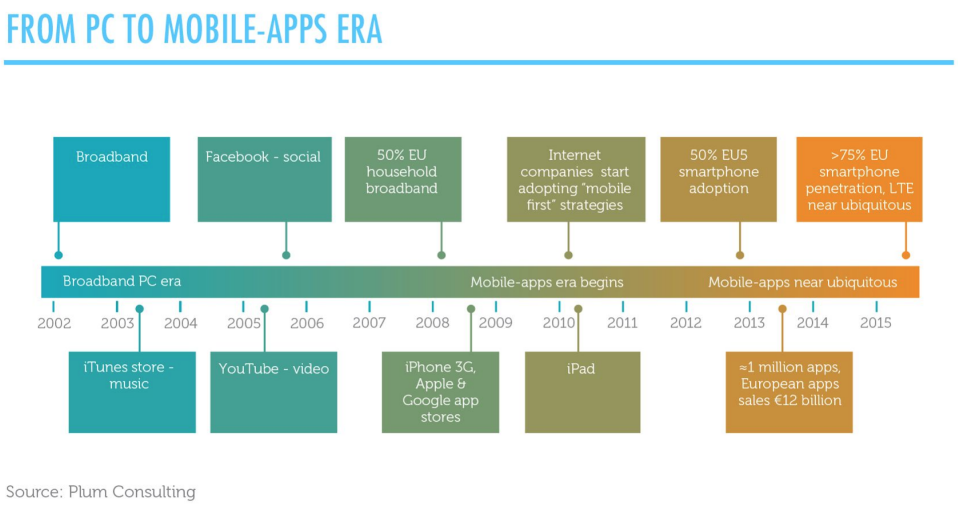

Figura 1.2: Dai PC all'era delle mobile app[2]

Attualmente la caratteristica principale degli smartphone è tuttavia la possibilità di installare app, anche di terze parti, per aumentare ulteriormente

le funzionalità del dispositivo mobile, e personalizzarlo a seconda dei propri gusti ed esigenze. Il termine app, deriva dall'abbreviazione di application, e identifica tutte quelle applicazioni software che vengono progettate per funzionare sui dispositivi mobile, gratuite o a pagamento, scaricabili dagli store online o già incluse nel device. Inizialmente lo sviluppo di mobile app era destinato solo alla produttività individuale e aziendale (CRM, project management, e-commerce, e-mail, calendario etc.), poi la popolarità delle app è cresciuta in maniera massiccia, così come il loro numero, finchè sono diventate strumenti indispensabili e irrinunciabili, abbracciando qualsiasi contesto. Oggi esistono tantissime app, e da uno studio ComScore[3] di maggio 2012 è emerso che l'utilizzo di app mobile ha addirittura superato quello della navigazione web, rispettivamente 51,1% vs 49,8%.

1.1 Tipologie di applicazioni mobile

A seconda dell'obbiettivo da raggiungere tramite lo sviluppo di un'app, potrebbe essere necessario tenere presente aspetti legati alle tipologie di utenti, a dove si trovano, o a cosa stanno svolgendo e quando, o considerare anche le condizioni più "avverse". Inoltre spesso è richiesto dal committente lo sviluppo in breve tempo e a costi sempre più bassi e competitivi, e generalmente le aspettative degli utenti rispetto alle app in stato di sviluppo aumentano se ci sono già app esistenti che soddisfano la stessa necessità dell'utente. Le app quindi possono semplificare la vita delle persone, renderla più divertente, comoda o pratica. Ogni giorno sono disponibili nuove app, e ormai ce n'è una per ogni necessità. Prima di decidere di investire nello sviluppo di un'app è importante capire quali sono le esigenze di comunicazione, quali sono i servizi che si vogliono offrire in mobilità e quali sono le esigenze dell'utente. Quindi, in base alla loro funzione, possiamo classificarle nelle seguenti categorie:

- utilità (mappe e navigazione, medicina e fitness, etc.);
- business e produttività;
- comunicazione e social networking;
- educazione;
- enterprise;

- intrattenimento (media, giochi, musica e video).

Per poter funzionare le app si appoggiano al sistema operativo, piattaforma che controlla il dispositivo mobile, perciò generalmente vengono sviluppate ad hoc considerando la sua architettura e le sue strategie di gestione delle risorse, e da questo sono vincolate. Quindi attualmente c'è una grande frammentazione del mercato mobile stesso, che comporta una divisione sostanziale del bacino di utenza in base alla piattaforma. L'eterogeneità dei sistemi operativi mobile (iOS, Android, Windows Phone 8, BlackBerry OS, Tizen, etc.) nonostante alcune caratteristiche in comune, comporta un notevole investimento, in termini di risorse umane e capitali, da parte delle aziende interessate allo sviluppo di app, per apprendere le competenze necessarie a rendere disponibile l'app su più dispositivi mobile. Questo è vero anche per le aziende che, come scelta strategica, decidono di considerare solo i sistemi operativi mobile dei dispositivi più venduti, e cioè iOS e Android. Per gestire al meglio questo problema è sempre più frequente la necessità di trovare un'alternativa che consenta alle aziende ed agli sviluppatori indipendenti di realizzare app attraverso un'unica soluzione indipendente dalla piattaforma che le ospiterà.
Le possibilità di diffusione di un'app sono tanto maggiori quanto più questa è trasversale e versatile, quindi riuscendo a funzionare in modo indifferenziato sulle diverse piattaforme mobile, sfruttandone al tempo stesso le caratteristiche. Proprio perché al momento non esiste una soluzione multi-platform che soddisfi pienamente tutte le possibili casistiche di sviluppo, in questa tesi si analizzerà lo stato dell'arte evidenziando punti di forza e limiti delle alternative attuali.
Risulta quindi molto importante suddividere le app in due grandi categorie dal punto di vista della portabilità:

- app native;
- app multi-platform.

1.2 Sviluppo nativo

Lo sviluppo nativo consiste nella programmazione di applicazioni sfruttando l'approccio platform-specific, cioè progettando l'applicazione appositamen-

te per ogni singola piattaforma. Questo è l'approccio classico, attualmente ancora molto utilizzato dagli sviluppatori, ed è quello più supportato a livello di tools di sviluppo, community, forum etc.
Le app native sono scritte e compilate per una specifica piattaforma utilizzando il linguaggio di programmazione legato al particolare sistema operativo e le librerie proprietarie.

Mobile OS	Skill Set richiesti
iOS	Objective-C (eventuali porzioni di codice in C)
android	Java (eventuali porzioni di codice in C o C++)
Windows Phone	C#, C++, XAML, Visual Basic.Net
BlackBerry OS	Java, C++
TIZEN	C++, C
ubuntu touch	QML, C, C++

Figura 1.3: Skill Set delle piattaforme attuali

In genere una determinata app nativa si presenta diversamente, a livello di UI (User Interface), a seconda della piattaforma per la quale è stata progettata e sulla quale quindi viene eseguita, nonostante offra in tutte le sue diverse versioni platform-specific le stesse funzioni. Questo accade perché un'app sviluppata con approccio nativo offre una UX (User Experience) intrinsecamente legata alle caratteristiche della piattaforma di destinazione, mantenendo lo stesso look&feel.
Occorre comunque considerare che i sistemi operativi mobile vengono periodicamente aggiornati, ma non tutti i device ricevono l'aggiornamento, specialmente se cominciano a essere obsoleti. Quindi possono crearsi problemi di incompatibilità tra un'app e alcune versioni particolarmente datate della piattaforma.
Per realizzare un'app nativa occorre installare nel proprio computer l'IDE (Integrated Development Environment) e l'SDK (Software Development

Kit) propri della piattaforma per la quale si renderà disponibile l'app. Questi strumenti vengono continuamente aggiornati inserendo nuove funzionalità e nuove API di sistema, permettendo allo sviluppatore di essere sempre al passo con le funzionalità presenti nei device più recenti. Un problema presente per quasi tutte le piattaforme principali (eccetto Tizen e Blackberry) è originato dall'imposizione di un particolare sistema operativo su cui installare gli strumenti nativi. Quindi ad esempio per sviluppare per iOS[5] tramite l'IDE nativo Xcode, è necessario un computer Mac basato su Intel e Mac OS X Leopard o una sua versione più recente, perché altri sistemi operativi e vecchie versioni di Mac OS X, non sono supportati. L'ambiente di sviluppo nativo per Windows Phone[6] richiede una delle versioni più recenti del sistema operativo Windows, e quello per Ubuntu Touch[4] richiede il sistema operativo Ubuntu. Pur considerando l'installazione su macchine virtuali o l'utilizzo in cloud degli strumenti nativi, non esistono ad oggi metodi non limitanti per sviluppare nativamente per le piattaforme sopra citate su computer senza i requisiti richiesti.
Durante lo sviluppo è possibile eseguire il debug dell'app e analizzarla all'interno dell'ambiente di sviluppo mettendola in esecuzione su appositi emulatori, e successivamente sottoporla a test sul device fisico. Gli emulatori sono programmi che emulano nel computer il look&feel di specifici device, permettendo di testare in modo economico le app anche sui dispositivi che non sono fisicamente a disposizione degli sviluppatori. In particolare per gli emulatori Android[7], vista la varietà di dispositivi che lo utilizzano, è possibile configurare anche caratteristiche specifiche come quantità di RAM, di memoria interna e di SD Card, tipo di CPU, etc. Comunque non tutti gli emulatori sono gratuiti e non tutte le piattaforme li mettono a disposizione, specie se sono recenti, come nel caso di Ubuntu Touch.
Il set di strumenti nativi permette anche di creare il package dell'app, così da poterla pubblicare direttamente nello store, senza l'utilizzo di software aggiuntivi. Per la maggior parte delle piattaforme mobile, per poter pubblicare un'app nel relativo store è necessario pagare una tariffa annuale variabile a seconda della piattaforma, inoltre le app devono essere preventivamente approvate, infatti possono essere rifiutate per motivazioni di carattere tecnico e/o commerciale.

Vediamo ora quali sono i principali vantaggi e svantaggi della metodologia di sviluppo appena illustrata.

1.2.1 Vantaggi e svantaggi

I vantaggi apportati sono:

- alte performance ed efficienza, le app hanno alte prestazioni e GUI (Grafical User Interface) molto reattiva, e sono ottimizzate in termini di spazio e risorse, perché possono effettuare direttamente chiamate di sistema senza componenti intermedi e vengono ottimizzate attraverso la compilazione in codice macchina svolta dal compilatore nativo;

- versatilità, pieno accesso dell'app a tutte le funzionalità hardware e software del dispositivo (fotocamera, GPS, accelerometro, messaggi, calendario, contatti, file system, etc.), che possono essere sfruttate appieno data l'ottimizzazione per la specifica piattaforma e l'utilizzo diretto tramite il linguaggio nativo, inoltre è possibile l'invio di notifiche push (avvisi inviati dall'app alla schermata iniziale del dispositivo mobile quando l'utente non la sta utilizzando);

- multitasking (esecuzione di più processi contemporaneamente), funzionamento dell'app anche in background in base all'algoritmo di scheduling usato dal particolare sistema operativo mobile;

- look&feel coerente a quello della specifica piattaforma e ricca UX, dato che le app sono ben integrate nel sistema operativo del dispositivo ospitante;

- funzionamento anche offline e persistenza dei dati, le app permettono l'accesso offline a contenuti e funzioni anche senza connessione ad Internet (a meno che una specifica funzione non lo renda strettamente necessario);

- facile visibilità e reperibilità dell'app nello store, "deposito" virtuale predisposto da ogni piattaforma, tramite il quale le app possono essere scaricate e installate.

Gli svantaggi sono:

- nessuna portabilità, le app non possono essere eseguite su una piattaforma diversa da quella per la quale sono state sviluppate, per renderle

disponibili al maggior numero di utenti, e quindi per più piattaforme, occorre svilupparne una versione specifica per ognuna;

- difficoltà nell'aggiornamento dell'app, che può richiedere lo sviluppo, il test e la distribuzione quando sorge un problema di compatibilità tra questa e una nuova versione della piattaforma o del dispositivo;
- insieme di competenze e tecnologie richieste vasto ed eterogeneo, per garantire il supporto della propria app sulle principali piattaforme è necessario possedere tutte le skills presenti in figura 1.3, e ciò è molto impegnativo e difficilmente raggiungibile;
- alti costi di sviluppo, in termini di costi veri e propri e in termini di tempi di sviluppo, a causa della notevole quantità ed eterogenità del know-how, e dei costi imposti dalle varie piattaforme per depositare l'app negli store e, in alcuni casi, anche per testarle nei device fisici o negli emulatori;
- la pubblicazione nello store necessita dell'approvazione del gestore che può richiedere giorni o anche settimane, ogni app deve infatti sottostare alle regole imposte dal gestore, che possono anche cambiare comportando un adeguamento della stessa, inoltre ogni aggiornamento implica per l'utente un nuovo download dell'app dallo store.

1.3 Sviluppo multi-platform

Lo sviluppo multi-platform comprende tutti quegli approcci e metodologie di sviluppo che cercano di raggiungere la filosofia di sviluppo *"Write once, run everywhere"*, ovvero *"Scrivi una volta, esegui ovunque"*, o più realisticamente *"Scrivi una volta e fai eseguire al meglio possibile su più piattaforme"*. Questo mito ha caratterizzato da sempre il mondo informatico, attento alla portabilità e ai costi di manutenzione del software, ad esempio la piattaforma Java ne è un'implementazione concreta, essendo un ambiente comune su cui realizzare software eseguibile su tutti i sistemi operativi desktop.
L'obbiettivo di questa tipologia di sviluppo è produrre software che possa funzionare in modo identico su diverse piattaforme, così da poter rendere disponibile un unico prodotto alla maggior parte degli utenti indipendentemente dalla piattaforma utilizzata dal loro dispositivo. Questo approccio

è nato dalla necessità di realizzare app disponibili alla maggioranza degli utenti, nonostante l'eterogeneità delle piattaforme, dei relativi ambienti e linguaggi di sviluppo e dei dispositivi, senza necessità di disparate conoscenze nè stanziamenti cospicui di risorse.
Esistono diverse soluzioni per superare questo problema, tutte però offrono, a loro modo, la possibilità di sviluppare app portabili su più piattaforme diverse.
Le app realizzate con questo approccio possono essere suddivise in due categorie a seconda della tecnologia e dell'architettura adottate alla base dello sviluppo cross-platform:

- web app;
- app ibride.

Comunque nella scelta della tipologia di app da realizzare, gli aspetti positivi e negativi dovrebbero essere considerati in relazione al compito che l'app dovrà assolvere. Questa semplificazione ci aiuterà a capire quali sono le due strade fondamentali seguite dalla maggior parte dei frameworks che andremo ad analizzare nel prossimo paragrafo, per affrontare al meglio il problema.

1.3.1 Web app

Queste app nascono dall'idea di utilizzare le tecnologie web attualmente disponibili per ottenere portabilità tra i sistemi operativi mobile sfruttando un mobile web browser installato sul particolare sistema operativo mobile (Chrome, Firefox for mobile, Safari, Internet Explorer Mobile, etc.). Generalmente HTML5 e CSS3 sono usate per realizzare il rendering dell'app, e JavaScript per la logica. Queste app sono web-based, quindi fruibili attraverso Internet, e in grado di funzionare su tutti i dispositivi connessi alla rete nonostante i diversi sistemi operativi, perciò non è necessario realizzarne una per ogni dispositivo. Inoltre non risiedono sul dispositivo mobile, ma vengono eseguite su un server remoto che svolge la maggior parte del lavoro computazionale, lasciando al dispositivo il compito di mostrare il risultato, perciò non hanno bisogno di essere scaricate e installate sul dispositivo. Le web app sono accessibili solo tramite un mobile browser, il quale assume il ruolo di client dell'app, consentendo l'accesso unicamente attraverso un

indirizzo URL, ed è l'ambiente di runtime delle web app. Una caratteristica importante è la possibilità di utilizzarle anche offline, sfruttando l'Application Cache e il Web Storage tramite l'HTML5. Inoltre possono essere scritte in vari linguaggi, purché il prodotto finale eseguito dal browser del dispositivo sia in HTML5, CSS3 e JavaScript, infatti è principalmente l'evoluzione di queste tecnologie che ha reso possibile lo sviluppo di web app. In sintesi una web app si può considerare come un sito web dinamico, responsive, e soprattutto capace di offrire funzionalità complesse, attraverso linguaggi di scripting lato client e lato server, mobile frameworks che implementano il RWD (Responsive Web Design), e tecnologie per riprodurre in rete comportamenti tipici delle interfacce software, che permettono di creare app simili alle app native anche a livello di UI.

Vantaggi

- elevata portabilità, essendo basate sugli standard del web le app funzionano su qualsiasi dispositivo dotato di un browser, raggiungendo la maggior parte degli utenti attraverso un'unica app fruibile per tutte le piattaforme, e quindi mantenendo un unico codice;

- funzionamento anche offline, le web app quando sono online possono scaricare risorse sfruttando l'Application Cache e il Web Storage per mostrarle in modalità offline, mantenendone così automaticamente copie locali che vengono aggiornate quando l'app è di nuovo online;

- costi e tempi di sviluppo e manutenzione contenuti, perché per gran parte dello sviluppo vengono riutilizzati competenze e strumenti già noti agli sviluppatori web (questo però dipende anche dal grado di complessità dell'app), dando la possibilità di sviluppare app anche senza conoscere linguaggi nativi;

- alleggerimento dal carico computazionale per il device, dato che la maggior parte dell'app è situata sul Web Server, quindi non incide sulla capacità di memoria del dispositivo nè sulle sue capacità di calcolo, ma addirittura potrebbe effettuare anche computazioni superiori alle potenzialità del device, essendo queste eseguite dal server;

- distribuzione e aggiornamenti rapidi e non vincolati dai tempi e dalle regole degli store, per la natura intrinseca delle web app infatti l'utente online accede direttamente alla versione più aggiornata;
- facile condivisione, le web app non richiedono l'installazione sul dispositivo, e sono condivisibili tramite il semplice link dell'URL, ad esempio all'interno di una mail, in un sms, in un post di un social network, in codici QR, etc.

Svantaggi

- HTML5, CSS3, JavaScript ed eventuali frameworks mobile utilizzati per le UI, non sono supportati allo stesso modo da tutti i browser mobile, perciò durante lo sviluppo occorre fare molta attenzione alle differenze di comportamento e interpretazione da parte dei browser, altrimenti alcune caratteristiche e funzioni della web app potrebbero non funzionare come previsto;
- performance limitate dalla presenza del browser come ambiente di esecuzione, infatti le prestazioni online dipendono in modo sensibile dalla velocità della connessione ad Internet e dalle capacità del server remoto nell'offrire l'elaborazione richiesta dall'utente, e mediamente sono meno performanti e reattive rispetto a quelle delle app native, a causa dalla latenza del trasferimento dati e del rendering delegato al browser, in particolar modo se le app contengono grafica complessa;
- accesso limitato alle funzionalità hardware e software del device e incapacità di inviare notifiche push, inoltre l'accesso alle risorse del device non è mai diretto ma sempre interfacciato da API JavaScript gestite attraverso il browser;
- nessun funzionamento in background (multitasking), perché l'app viene eseguita all'interno del browser e non è direttamente installata sulla piattaforma;
- UX universale e nessuna UI nativa, infatti si possono utilizzare solo UI che emulano il look&feel di quelle native, non quelle vere e proprie, di conseguenza la UX risulta un po' incoerente con la piattaforma,

la UI non evolve automaticamente con gli aggiornamenti del sistema operativo mobile, e questo potrebbe non essere molto apprezzato dagli utenti che sono molto legati ad aspetti di UI e UX della propria piattaforma;

- acquisizione di uno spazio web, che può essere a costi più o meno ridotti a seconda della complessità che dovrà avere la web app.

Cloud app In tempi recenti è nato un ulteriore approccio per lo sviluppo di mobile app che sfrutta il cloud computing[1] nelle sue declinazioni, cioè SaaS (Software-as-a-Service), PaaS (Platform-as-a-Service) o IaaS (Infrastructure-as-a-Service), e rappresenta una grande opportunità per le aziende, soprattutto per il diffondersi del fenomeno BYOD (Bring your own device), politiche aziendali che permettono di usare i propri dispositivi personali sul lavoro per avere accessi privilegiati alle informazioni aziendali e alle loro applicazioni.
In un'architettura cloud i dati e la maggior parte del codice di elaborazione risiedono in un data center, da qualche parte nella "nuvola", e il funzionamento è garantito da servizi di back-end che gestiscono la multi-tenancy (partizionamento virtuale dei dati e configurazione in modo che ogni client lavori con un'istanza virtuale personalizzata), l'elevata scalabilità, la sicurezza, l'affidabilità, l'integrazione, la continuità, e il supporto ai vari metodi di accesso.
In questo approccio quindi l'app non viene intesa come un prodotto, ma piuttosto come un servizio. Il web browser è solo una delle possibili modalità di accesso a questo sistema, ma non l'unica visto che è possibile accedervi anche da applicazioni installate su desktop e dispositivi mobile connessi a Internet.
Ad esempio possono essere servizi cloud i cicli di calcolo on-demand, le piattaforme di storage, etc., e sono cloud app Evernote, Dropbox, ShareFile, Gmail.

1.3.2 App ibride

L'idea alla base dell'approccio ibrido è quella di combinare tecnologie di sviluppo native con soluzioni web-based, così da racchiudere in sè alcune caratteristiche delle app native ed altre delle web app, cercando di sfruttare

al meglio le loro potenzialità. L'architettura ibrida prevede un "core" contenente l'essenza dell'app sviluppato utilizzando tecnologie web, all'interno di un "wrapper", un involucro sviluppato in linguaggio nativo.
Quindi le app ibride sono sviluppate prevalentemente in HTML5, CSS3 e JavaScript, perciò si aggiornano tramite il web, ma si interfacciano tramite un wrapper che ne facilita l'integrazione con le risorse e con le caratteristiche della piattaforma sulla quale vengono installate. Il wrapper deve essere riscritto per ogni sistema operativo per il quale si vuole distribuire l'app, e può essere realizzato direttamente dagli sviluppatori o tramite una delle tante soluzioni già disponibili all'uso, gratuite o a pagamento, cioè frameworks multi-platform come ad esempio PhoneGap. In base alle scelte dello sviluppatore ed all'eventuale framework utilizzato, il wrapper può sfruttare una certa gamma di funzionalità hardware e software del dispositivo come fotocamera, GPS, accelerometro, contatti, etc., utilizzare il motore di rendering dei linguaggi del web fornito dal framework o quello tipicamente integrato nel browser della piattaforma, e fornire o meno componenti della UI nativi. In genere lo sviluppatore può sfruttare le funzionalità del device utilizzando API JavaScript, che vengono mappate in codice nativo dal wrapper. Nel caso in cui il framework scelto non permetta di utilizzare UI native il rendering della parte grafica è affidato ad una WebView, un componente incorporato nel browser inserito nella UI dell'app.
Questo approccio offre la possibilità di creare soluzioni davvero uniche e flessibili come ad esempio realizzare UI affiancando componenti nativi a WebView.
Le app ibride vengono eseguite direttamente sulla piattaforma mobile del dispositivo, perciò sono reperibili allo stesso modo delle app native, ma il contenuto web può essere ospitato su un server e scaricato nell'app ogni volta che è attiva la connessione ad Internet ed ha subito un aggiornamento. Tutti i dati necessari per il funzionamento dell'app possono essere memorizzati sul dispositivo mobile, infatti le tecnologie di storage fornite da HTML5 permettono di realizzare il caching per rendere possibile l'utilizzo della WebView, anche nel caso in cui l'utente non sia connesso ad Internet. La persistenza dei dati può essere realizzata anche tramite l'accesso diretto a database presenti sul device.

Vantaggi

- elevata portabilità e riutilizzo su più piattaforme di tutto il codice web, o della maggior parte, occorre riscrivere per ogni piattaforma solo il wrapper, ed inoltre questo lavoro può essere facilmente delegato ad un framework, dando la possibilità di sviluppare app anche senza conoscere linguaggi nativi;

- flessibilità derivata dalla possibilità di combinare funzionalità native con tecniche di sviluppo web;

- accesso elevato alle funzionalità hardware e software del device, con la possibilità di inviare notifiche push, il tutto offerto da un framework attraverso API JavaScript;

- costi e tempi di sviluppo abbastanza contenuti, perché le app ibride sono prevalentemente sviluppate in HTML5, CSS3 e JavaScript, riutilizzando competenze e strumenti già noti agli sviluppatori web (questo però dipende anche dal grado di complessità dell'app) e la maggior parte del codice;

- UX e look&feel molto simili a quelle di un'app nativa, grazie all'accesso alle funzionalità del dispositivo e alla possibilità, offerta da alcuni frameworks, di utilizzare anche UI native;

- aggiornamento della WebView agevole e dinamico tramite l'accesso alla rete;

- funzionamento anche offline, infatti l'app è installata nel dispositivo e l'eventuale WebView utilizzata è sempre disponibile grazie alle tecnologie di storage fornite da HTML5, inoltre è possibile garantire la persistenza dei dati sfruttando l'accesso diretto a database presenti sul device;

- efficienza e performance generalmente abbastanza buone, le app ibride azzerano i tempi di latenza dovuti al caricamento delle pagine web presenti invece nelle web app, offrendo così prestazioni migliori;

- facile visibilità e reperibilità dell'app nello store.

Svantaggi

- inevitabile compromesso dovuto alla portabilità su più piattaforme, cioè dipendenza dal framework scelto per lo sviluppo, o conoscenza dei linguaggi nativi, ad esempio per poter sfruttare una nuova funzionalità di una piattaforma è necessario attendere l'aggiornamento del framework, oppure occore implementare l'interfacciamento in codice nativo;

- performance peggiori rispetto allo sviluppo della stessa app con l'approccio nativo, infatti l'accesso alle risorse del device comporta un overhead computazionale perché non è diretto ma sempre interfacciato da API JavaScript gestite da un motore di rendering dei linguaggi del web, anche effettuando la compilazione JIT (just-in-time), inoltre la UI è meno reattiva dato che la velocità di rendering della WebView non è ancora paragonabile alla velocità di rendering delle UI delle applicazioni native;

- interfacciamento con le risorse dei device dipendente dal framework utilizzato, spesso occorre usare un sottoinsieme delle risorse disponibili comune a tutte le piattaforme per averne il supporto su tutte, perché alcune funzionalità potrebbero non essere supportate per alcune piattaforme o esserlo solo parzialmente, e questo può ostacolare le prestazioni dell'app o vietare agli sviluppatori di utilizzare le funzioni avanzate, inoltre l'eventuale implementazione di funzionalità extra richiede lo sviluppo di plugin per il framework in linguaggio nativo;

- Time To Market rallentato dai tempi di approvazione dell'app per accedere agli store, inoltre l'aggiornamento dell'app richiede la ricompilazione e la ripubblicazione nello store;

- debug e test dell'app complicato e dipendente dagli strumenti messi a disposizione dal framework scelto, visto che l'app sulle diverse piattaforme può presentare comportamenti leggermente diversi o piccoli bug, ecco perché alcuni sviluppatori definiscono lo sviluppo cross-platform *"scrivi una volta, esegui il debug ovunque"*;

- se il framework non permette di utilizzare UI native, le UI che emulano il look&feel di quelle native potrebbero non essere molto apprezzate

dagli utenti che sono molto legati ad aspetti e convenzioni di UI e UX della propria piattaforma.

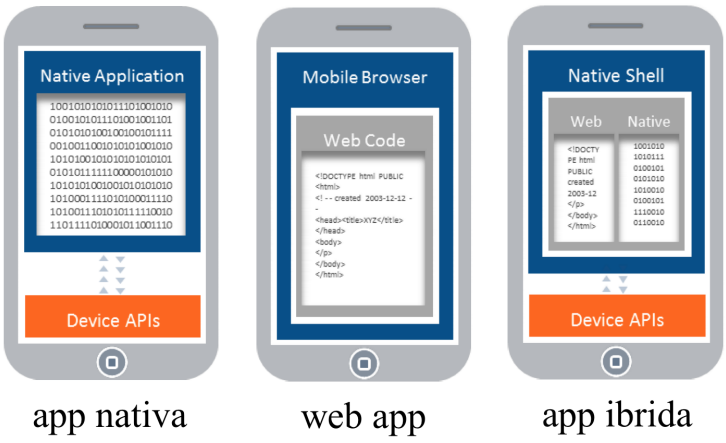

Figura 1.4: Architettura delle app

1.4 Approcci per lo sviluppo multi-platform

Nell'ottica di analizzare gli strumenti multi-platform attualmente disponibili, è importante capire lo stato dell'arte delle tecnologie del web sopra citate e sfruttate dalla maggior parte dei frameworks che andremo a considerare. Gli strumenti individuati nell'indagine, verranno suddivisi in categorie e valutati secondo criteri ben definiti allo scopo di stabilire quali di essi meritino un approfondimento ulteriore nei capitoli successivi.

1.4.1 HTML5, CSS3, JavaScript

HTML5[1][8] è un linguaggio di markup in continua evoluzione per la strutturazione dei contenuti del web, e si basa sul DOM (Document Object Model), ovvero sulla rappresentazione ad albero usata dai browser per riprodurre la pagina web. È la quinta revisione dello standard HTML, creato nel 1990 grazie a Tim Berners-Lee, e standardizzato come HTML4 nel

1997. HTML5 nasce per risolvere problemi d'inadeguatezza delle sue versioni precedenti, sorti inizialmente con la comparsa della multimedialità nello scenario di Internet, dove le pagine web non erano più formate esclusivamente da testo e immagini statiche, e per offrire agli sviluppatori web un linguaggio pronto ad essere modellato secondo le più recenti necessità, dal punto di vista della strutturazione del contenuto e da quello dello sviluppo di vere e proprie applicazioni web. Lo sviluppo di HTML5 venne avviato dal gruppo di lavoro WHATWG (Web Hypertext Application Technology Working Group) fondato nel 2004 da sviluppatori appartenenti ad Apple, Mozilla Foundation e Opera Software, secondo le seguenti linee guida[9]:

- *don't break the web*, nessuna nuova regola deve interferire con il corretto funzionamento di una pagina sviluppata in precedenza;
- *pave the cowpaths*, introdurre sperimentazioni già adottate con successo dalla maggioranza degli utenti, piuttosto che proibire o inventare nuove funzionalità;
- *be pratical*, le modifiche devono avere un riscontro pratico, nascendo da un'esigenza di utenti e sviluppatori, e devono produrre un miglioramento significativo.

Quindi l'obiettivo del gruppo è elaborare specifiche per lo sviluppo di un web più orientato alle applicazioni che ai documenti, per questo HTML5 è anche un potenziale candidato per le applicazioni mobile cross-platform. Oggi i documenti HTML5 sono in grado di incorporare molte tecnologie, come CSS, JavaScript, XML, etc., per aggiungere al documento ipertestuale controlli più sofisticati sulla resa grafica, interazioni dinamiche con l'utente, animazioni interattive e capacità multimediali senza la necessità di plugin lato client. HTML5 raccoglie l'eredità semantica dell'XHTML2, e aggiunge valore semantico a struttura e contenuto della pagina, tramite nuovi tag, come <header>, <nav>, <footer>, <aside>, <section>, <article>, per identificare elementi comuni e ricorrenti, altri per per ottenere pagine web semanticamente eccellenti per microformati e data entry. È previsto il supporto per la memorizzazione locale di grosse quantità di dati scaricati dal browser, permettendo l'utilizzo di applicazioni web offline, attraverso Application Cache, per comunicare al browser quali file salvare in locale e Web Storage, un'evoluzione dei cookies, molto più facile da gestire e soprattutto con più spazio a disposizione per salvare dati direttamente dal browser,

senza passare da un database centralizzato raggiungibile solo tramite la connessione. Queste specifiche abilitano la creazione di pagine web con grafica, disegno ed effetti 3D, tramite tag come <canvas>, e l'inserimento facile e veloce di file audio e video, come parte del DOM, tramite <audio>, <video>, inoltre definiscono nel dettaglio l'elaborazione richiesta per i documenti non validi, introducono nuove potenti funzionalità, ed eliminano i tag obsoleti. Comunque attualmente HTML5 non è supportato allo stesso modo da tutti i browser[10], e il W3C ha annunciato che le specifiche di HTML5 saranno finali e consolidate a fine 2014, mentre quelle HTML5.1 saranno finalizzate per il 2016.

CSS3[1][11] è la versione attualmente più evoluta di CSS (Cascading Style Sheet), un linguaggio che consente di definire la formattazione, lo stile e l'aspetto dei contenuti presenti in un documento scritto in un linguaggio di markup, mantenendo separato il contenuto dalla presentazione attraverso fogli di stile. A differenza delle versioni precedenti, CSS3 è diviso in moduli, cioè parti più piccole, per rendere più agile il linguaggio e consentire a W3C di sviluppare le varie parti con tempistiche differenti, quindi l'evoluzione dei vari moduli è diversa, e così è anche il supporto offerto dai browser. Tra i principali moduli ci sono: Selectors, Box Model, Backgrounds and Borders, Text Effects, 2D-3D Transformations, Animations, Multiple Column Layout e User Interface.

JavaScript[1][12] è un linguaggio di scripting orientato agli oggetti basato su prototipi, per la programmazione web, sviluppato nel 1995 da Brendan Eich come modulo del browser Netscape Navigator. Originariamente chiamato Mocha, poi LiveScript, e in seguito rinominato JavaScript, permette l'interazione con oggetti istanziati in una pagina web, è stato formalizzato con una sintassi più vicina a quella del linguaggio Java, e si occupa della logica applicativa della pagina web, permettendo l'inserimento di contenuto eseguibile che consente di compiere azioni e di interagire con l'utente. L'interpretazione effettuata da un motore lato-client, incluso nel browser, rende JavaScript automaticamente portabile su tutti i sistemi operativi che possiedono un browser compatibile. La popolarità di JavaScript è aumentata radicalmente con l'introduzione dei motori con compilatori JIT (Just In Time), e l'aumento della potenza a disposizione dei processori.
Attualmente per supportare le tantissime funzionalità di cui una applica-

zione web moderna potrebbe aver bisogno, HTML5 prevede l'utilizzo di API JavaScript per accedere a caratteristiche e dati specifici dei terminali, ad esempio per gestire la geolocalizzazione, per utilizzare la fotocamera, etc. Sono sviluppate come parte dell'iniziativa HTML5 anche le API Web Workers e Web Sockets. Web Workers permette di svincolare un'attività intensiva dal resto della UI, per consentire l'esecuzione di codice JavaScript in background in modo asincrono, così da non intaccare le performance della pagina web che si sta visualizzando che quindi continua a rimane reattiva. Web Sockets invece, permette di utilizzare il protocollo Web Sockets per stabilire un canale di comunicazione full-duplex (bidirezionale) tra browser e server, per rendere possibile la creazione di applicazioni web che richiedono uno scambio costante di informazioni fra client e server.

1.4.2 Frameworks

In relazione all'architettura delle app multi-platform, descritta nel paragrafo precedente, ora effettueremo un'indagine sugli strumenti che sono a disposizione degli sviluppatori per la loro realizzazione. Analizzandoli, e suddividendoli in categorie, cercheremo di individuare i più interessanti considerando soprattutto i benefici della potenziale strategia di sviluppo seguita ed offerta dal particolare framework.

Data la necessità sempre più crescente di raggiungere gli utenti di piattaforme diverse a minori costi di sviluppo, oggi esistono un'infinità di frameworks che permettono di realizzare, in modalità più o meno simili, lo sviluppo multi-platform tramite la stesura di un unico codice. Al fine di individuare i frameworks più interessanti da approfondire successivamente con un'applicazione di test, ne sono stati individuati diciassette tra tutti quelli attualmente disponibili e in continua evoluzione.
La prima selezione è stata effettuata prediligendo quelli completamente o parzialmente gratuiti, in modo da permettere al team di Mint di presentare offerte più competitive senza dover includere la quota parte del costo del framework, che nell'eventualità di un numero limitato di clienti per anno e un costo particolarmente elevato, diventerebbe rilevante. Anche gli altri criteri adottati per la valutazione sono stati stabiliti insieme al team dell'azienda Mint, considerando le loro esigenze nello sviluppo di app, come le necessità sempre presenti di gestire alcuni formati di dati e reperirli tramite

determinati protocolli, la loro volontà di offrire ai clienti una UX particolarmente interattiva e ricca di animazioni, e di fornire agli utenti app fruibili anche offline. Inoltre è stata data importanza ai frameworks già popolari tra gli sviluppatori e particolarmente supportati, nell'ottica di inglobare anche le valutazioni della maggior parte di loro, e le eventuali problematiche già da loro riscontrate.
Le caratteristiche ritenute più rilevanti nell'indagine sono quindi le seguenti:

- la popolarità e il potenziale supporto, proprietario e non, offerto in forum e community;
- la chiarezza e la quantità della documentazione che mettono a disposizione degli sviluppatori, ritenendo più interessanti quelli con la documentazione più completa e più comprensibile;
- la possibilità di fornire animazioni e interattività;
- la possibilità gestire particolari formati di dati (XML, SQLite, JSON, etc.) e di reperirli attraverso particolari protocolli (SOAP, etc.);
- la possibilità di gestire dati offline;
- le piattaforme supportate, giudicando più importanti quelli che permettono la creazione di app supportate da iOS, Android e Windows Phone, perché sono le piattaforme mobile più diffuse al momento.

Nel caso dei frameworks che permettono di produrre solo web app, nella determinazione delle piattaforme supportate è stata considerata la compatibilità con il browser proprietario della particolare piattaforma mobile, anche se tra i browser supportati da questi strumenti ci sono anche quelli installabili dall'utente come Firefox for mobile, Opera Mobile, etc., e i principali browser desktop.
La figura 1.5 mette in evidenza i linguaggi di sviluppo supportati dai frameworks, e il target, ovvero l'obbiettivo che è possibile raggiungere sfruttando il particolare framework, che in questo caso corrisponde alla tipologia di app che permette di sviluppare.
Proseguendo nell'indagine, gli strumenti individuati sono stati differenziati in base al tipo di funzionalità e risorse che offrono, allo scopo di comprenderne meglio le peculiarità e le differenze.

Frameworks	Linguaggi di sviluppo	Target		Piattaforme supportate		
		web app	app ibride	iOS	Android	Windows Phone
PhoneGap	HTML5, JavaScript, CSS3	no	si	3.1+	si	7 e 8
Titanium	JavaScript, XML, HTML5, CSS3	si	si	si	si	no
Adobe air	Flash, Flex, ActionScript3, HTML5, JavaScript, CSS3	si	si	5.1+	2.3	no
Motorola RhoMobile Suite	Ruby, JavaScript, HTML5, CSS3	no	si	6+	2.3+	8 solo sui dispositivi più recenti
jQuery Mobile	HTML5, JavaScript, CSS3	si	no	3.2-6.1	2.1-4.2	7.5-8
Sencha Touch	HTML5, JavaScript, CSS3	si	si	4+	2.3+	8 solo su Nokia Lumia 900 e 920
Wink	HTML5, JavaScript, CSS3	si	no	x	si	7
LG Enyo	HTML5, JavaScript, CSS3	si	no	4+	2.3+	8, 7.5 parzialmente
Qooxdoo Mobile	JavaScript, HTML5, CSS3	si	no	si	1.6+	8 in fase di sviluppo
Jo	HTML5, JavaScript, CSS3	si	no	3+	2.1+	no
PhoneJS	HTML5, JavaScript, CSS3	si	no	5+	2.3+	8
Dojo Mobile	HTML5, JavaScript, CSS3	si	no	5+	2.2+	no
DHTMLX touch	HTML5, JavaScript, CSS3	si	no	si	si	no
Chocolate chip UI	HTML5, JavaScript, CSS3	si	no	si	si	si
IBM Worklight	HTML5, JavaScript, CSS3, Java	si	si	si	2.2+	7.5 e 8
Vaadin	Java, HTML5, JavaScript, CSS3	si	no	5+	2.3+	8 prossimamente
Monocross	C#, HTML5, JavaScript, CSS3	si	si	si	si	7

Figura 1.5: Scenario dei principali frameworks multi-platform individuati nella fase introduttiva

Quindi dopo aver raccolto informazioni sulla loro architettura e sul loro funzionamento, è stato possibile identificare tre categorie in cui suddividere i frameworks:

- librerie;
- "wrappers";
- web application frameworks.

L'ordine nel quale sono state elencate le categorie non è casuale, perché segue il livello di complessità e astrazione dei frameworks, partendo da quella contenente gli strumenti meno complessi.

Tra le *librerie* rientrano tutti quei frameworks che offrono, tramite i linguaggi del web, componenti per riprodurre e simulare i comportamenti tipici delle interfacce native, in termini di grafica e interattività delle UI, e funzioni per semplificare lo sviluppo di web app, combinando versatilità ed estensibilità, agevolando manipolazione del DOM, gestione degli eventi, animazione e utilizzo dell'approccio AJAX, con API di facile utilizzo che funzionano attraverso una moltitudine di browser.
Questi frameworks permettono di realizzare web app con grafica Touch-Optimized per i dispositivi mobile, identificando il nome e la versione del browser e della piattaforma mobile del device per dedurre se JavaScript e alcune funzionalità HTML5 e CSS3 sono supportate, e adattare automaticamente il layout dell'app al display del device, in accordo con le tecniche di RWD. In questo modo la web app può funzionare in modo ottimale in funzione dell'ambiente nel quale viene eseguita (pc, tablet, smartphone, con una vasta gamma di risoluzioni di dispositivi, densità dello schermo e modalità di interazione) con la stessa base di codice sottostante.
Alcune di queste librerie permettono di sfruttare il pattern architetturale MVC (Model-View-Controller), tutte sono più o meno compatibili con uno o più wrapper, e gratuite.
In particolare Sencha Touch[14] include tra gli strumenti offerti anche un tool per produrre anche app ibride, ma possiamo considerarla comunque una libreria visto che per fare questo sfrutta il wrapper PhoneGap.

La categoria dei *"wrappers"* raggruppa tutti i frameworks che consentono principalmente lo sviluppo di app ibride, offrendo agli sviluppatori una certa gamma di funzionalità hardware e software dei device supportati, come fotocamera, GPS, accelerometro, contatti, etc., ed eventualmente anche componenti di UI nativi, come nel caso di Titanium[24]. Questi strumenti automatizzano la creazione di un involucro nativo, un package nel formato della particolare piattaforma, che racchiude l'app creata sfruttando le tecnologie del web, dandole la capacità di comunicare con il device sottostante sul quale verrà installata ed eseguita.
In genere lo sviluppatore può sfruttare le funzionalità e/o componenti di UI nativi dei device utilizzando API appositamente messe a disposizione e automaticamente mappate in codice nativo dal wrapper, senza doversi preoccupare di conoscere il linguaggio nativo. Quindi il funzionamento dei wrappers si basa sul pattern Adapter, che fornisce una soluzione astratta al

problema dell'interoperabilità tra interfacce differenti, in questo caso quelle dei linguaggi del web e quelle della specifica piattaforma.

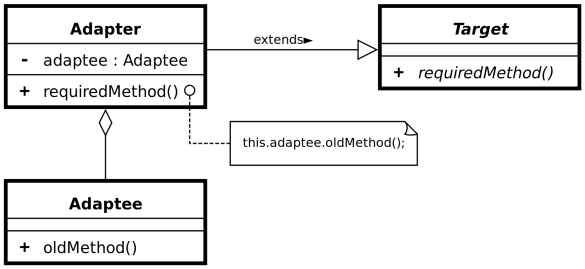

Figura 1.6: Diagramma UML del pattern Adapter

Alcuni wrapper per l'esecuzione del codice web offrono un proprio motore di rendering dei linguaggi del web, altri permettono di sfruttare quello tipicamente integrato nel browser della piattaforma.

Wrappers	How to	Accesso a componenti di UI nativi
PhoneGap	A build time incorpora codice HTML5, CSS3 e JavaScript dentro una WebView nativa, in grado di comunicare con il codice JavaScript in esecuzione al suo interno, che attraverso API JavaScript controlla a runtime le funzionalità del device chiamando funzioni native. Quindi il rendering avviene all'interno di una istanza del browser, che interpreta l'HTML5, CSS3 e JavaScript.	no
Titanium	Titanium implementa classi proprietarie per offrire funzionalità del device e componenti di UI nativi tramite API JavaScript. Tutto il codice sorgente dell'app, scritto in JavaScript, viene inglobato in un file nativo (Java o Objective-C) per essere distribuito alla piattaforma mobile. L'app viene eseguita dentro un interprete JavaScript incorporato nel codice sorgente durante il build process, che valuta il codice a runtime, facendolo interagire con l'ambiente nativo attraverso oggetti proxy, che esistono in entrambi gli ambienti.	si
Adobe air	Sistema di runtime multi-piattaforma per la creazione di RIA (Rich Internet Application) per desktop, dispositivi mobile e TV, sviluppate con Adobe Flash, Adobe Flex, ActionScript3, HTML5, CSS3 e JavaScript. Adobe AIR supporta applicazioni Flash eseguendole all'interno di un'istanza di Flash Player, e applicazioni web eseguendole all'interno di un web browser, e permette di realizzare app ibride, fornendo l'accesso alle funzionalità del device, tramite Adobe Flash, Adobe Flex e ActionScript3, e applicazioni web tramite HTML5, CSS3 e JavaScript. Inoltre consente l'utilizzo di file locali, di database SQLite locale, di web services, ... e di eseguire contenuto JavaScript con alcune limitazioni di sicurezza.	no
Motorola RhoMobile Suite	L'app viene eseguita dentro una WebView nativa. Il codice sorgente può essere scritto in Ruby o JavaScript, e le funzionalità del device sono messe a disposizione dai componenti Rhodes e RhoElements tramite API Ruby e API JavaScript. La struttura dell'app di default include jQuery Mobile e i CSS associati per gestire la UI, che perciò è priva di un supporto grafico avanzato. La sua architettura poggia su una virtual machine molto performante per interpretare Ruby, linguaggio compatto e poco oneroso dal punto di vista computazionale, e fare il rendering HTML5 a runtime. Offre funzionalità per la gestione avanzata dei dati che lo rendono indicato per lo sviluppo di applicazioni B2B.	si

Figura 1.7: "Wrappers"

Quelli più complessi come Titanium e RhoMobile[20] danno anche la possibilità di sfruttare il pattern MVC ed offrono strumenti per il debug e la simulazione dell'app mentre quelli più semplici come PhoneGap[19] richiedono in input una web app precedentemente realizzata e non forniscono strumenti per il debug dell'app impacchettata. Tutti sono compatibili con le librerie della categoria precedente e sono parzialmente gratuiti. La realizzazione del livello di astrazione tra il livello nativo e quello web può essere effettuato anche in cloud, come nel caso dei servizi PhoneGap Build e RhoHub, dando la possibilità agli sviluppatori che non utilizzano un Mac di sviluppare app per iOS aggirando facilmente i requisiti Apple, e permettendo a tutti di utilizzare sempre la versione più aggiornata degli SDK nativi.

Infine per *"web application frameworks"* s'intende fare riferimento a tutti quei frameworks che racchiudono una certa complessità architetturale, e sono stati progettati per fornire un ambiente completo per lo sviluppo di applicazioni avanzate multipiattaforma. Questi frameworks danno allo sviluppatore la possibilità di creare applicazioni web desktop e mobile, supportandone in maniera approfondita l'intero ciclo di vita, quindi lo sviluppo, l'integrazione back-end, l'autenticazione, offrendo servizi in cloud, ponendo molta attenzione agli aspetti di sicurezza e gestione dell'app. La maggior parte di essi fornisce strumenti per integrare le app con le risorse dell'impresa, rispondendo all'esigenza comune a molte aziende di portare le funzionalità di applicazioni già esistenti sul mobile.
Nel caso di Monocross[23] è possibile creare anche app ibride attraverso l'uso del tool Xamarin. Inoltre sono tutti parzialmente gratuiti e con componenti estensibili e personalizzabili. Infatti IBM Worklight[21] è compatibile con alcune delle librerie sopra citate, Vaadin[22] può essere esteso con widget GWT (Google Web Toolkit) e i temi possono essere personalizzati con CSS, e Monocross è anch'esso compatibile con librerie della stessa tipologia di quelle citate, e in più offre la possibilità di utilizzare componenti di UI nativi mettendoli a disposizione tramite API C#.

Web application frameworks	Cosa permette di realizzare
IBM Worklight	Framework che include un ambiente completo per sviluppare web app, app ibride, applicazioni web desktop e per altri sistemi embedded con Java ME. È progettata per supportare l'intero ciclo di vita dell'applicazione, l'integrazione back-end, meccanismi di autenticazione, la gestione post-distribuzione, la comunicazione crittografata server-client, la crittografia on-device e l'autenticazione offline, offre una console amministrativa web-based per la gestione unificata del portfolio di app e collegare e sincronizzare i dati enterprise, e per l'analisi, funzionalità avanzate di sicurezza, la possibilità di realizzare enterprise application store, potenzia la sicurezza e la governance delle applicazioni, … Inoltre permette di sfruttare le funzionalità del device tramite Apache Cordova, di utilizzare UI native, o affiancare WebView a UI native, e di integrare nell'ambiente anche app native già esistenti. Fornisce Worklight Server, un middleware mobile-optimized che opera come un gateway tra applicazioni, sistemi back-end e servizi cloud-based. È uno dei finalisti del Mobile Solution IT del 2013 nell'ambito Application Development & Platforms.
Vaadin	Framework per lo sviluppo di RIA (Rich Internet Application) con architettura server-side (la maggior parte della logica dell'applicazione è eseguita sul server) basato su Java, comprendente anche strumenti di sviluppo client-side basati su GWT (Google Web Toolkit) e HTML5. Il motore client-side AJAX-based di Vaadin comprende un compilatore, che permette la compilazione di codice Java in codice JavaScript, e gestisce il rendering delle UI nel web browser. Per personalizzare l'aspetto dell'applicazione offre un compilatore SASS che semplifica la gestione di CSS. Vaadin aggiunge la validazione dei dati lato server per tutte le azioni. Il lato client si può estendere utilizzando HTML5, JavaScript e GWT per creare nuovi componenti dell'interfaccia utente o applicazioni offline.
Monocross	Framework basato su C# e .NET che utilizza Xamarin per lo sviluppo di app ibride, e ASP.NET per lo sviluppo di web app, che condividono modello di dati (Model) e logica di funzionamento (Controller). La View viene implementata per ogni specifica piattaforma per offrire le UI native del device attraverso Xamarin.iOS, compilando AOT (Ahead-of-Time) per produrre ARM binario, e Xamarin.Android, compilando JIT (Just In Time) direttamente sul dispositivo Android. Windows Phone utilizzando C# .NET come ambiente nativo non richiede accorgimenti per la distribuzione sul device. Xamarin.Mobile offre accesso completo alle funzionalità del device. Alcuni componenti cross-platform che forniscono servizi comuni sono: SQLite-NET, web services, IO, multithreading, application logging, data serialization, shared business logic, data access, shared data access.

Figura 1.8: "web application frameworks"

1.4.3 Considerazioni sui frameworks

Delle dieci *librerie* individuate nello scenario di partenza, si è deciso di approfondire solo le sei presenti in figura 1.10 con il voto finale ponderato maggiore, calcolato dando un peso alle caratteristiche come mostrato in figura 1.9. Quindi per ogni libreria è stato dato un punteggio, da uno a dieci, ad ognuna delle caratteristiche presenti in figura 1.9.
In particolare per valutare "popolarità e supporto" sono stati esaminati i forum e le community ufficiali, dando punteggi più alti alle librerie con quelli più attivi e con più utenti, e considerando la maggiore notorietà sul motore di ricerca Google e sul sito StackOverflow. Le altre caratteristiche sono state stimate consultando i siti e i blog ufficiali di ogni libreria. Il punteggio per "quantità dei componenti offerti" e "possibilità di personalizzazione dei componenti" è maggiore per le librerie che offrono molte tipologie di componenti, e che possono essere facilmente estensibili e personalizzabili. Per

giudicare la "fluidità degli esempi ufficiali" e il "supporto iOS, Android e Windows Phone" sono stati testati demo e kitchensink ufficiali su alcuni smartphone e tablet, dando punteggio maggiore a quelli con componenti più fluidi e maggiore compatibilità con le piattaforme.
Il numero sei è stato ritenuto il giusto compromesso tra le risorse per l'approfondimento e una buona varietà di librerie.

Caratteristiche	**Pesi**
popolarità e supporto	30,5%
quantità dei componenti offerti	7,5%
possibilità di personalizzazione dei componenti	7,5%
fluidità degli esempi ufficiali (su desktop e su dispositivi mobile)	26,0%
supporto iOS	9,5%
supporto Android	9,5%
supporto Windows Phone	9,5%
Totale	**100,0%**

Figura 1.9: Caratteristiche delle librerie e relativi pesi

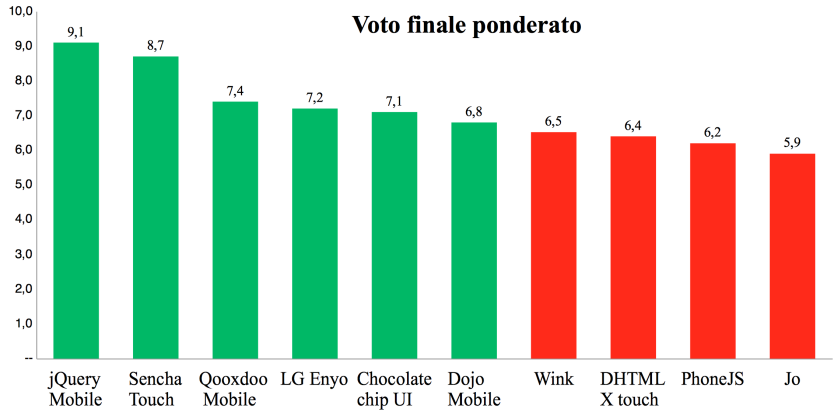

Figura 1.10: Voto finale ponderato delle librerie

Nel capitolo successivo sarà esposto l'approfondimento delle tre librerie, che tramite lo svolgimento di un'applicazione di test sono risultate particolarmente utili, e che ne ha rilevato pregi e difetti.

Inoltre è stato ritenuto di particolare interesse approfondire anche il *"wrapper"* Titanium, in considerazione delle potenziali performance e UX che può offrire, dato l'accesso alla maggior parte delle funzionalità hardware e software dei device e ai componenti di UI nativi tramite il solo linguaggio JavaScript, con la possibilità di sfruttare il pattern MVC attraverso il componente Alloy e XML. Infatti la sua architettura permette di evitare l'utilizzo di WebView, eseguendo l'app dentro un interprete JavaScript, incorporato nel codice sorgente, che valuta il codice a runtime, così da evitare che la qualità della UI vari in base alla qualità della WebView e del motore di rendering della piattaforma.
Quindi nel terzo capitolo verranno messe in luce le sue caratteristiche tramite lo sviluppo di un prototipo di app ibrida, che successivamente verrà confrontata con la corrispondente versione nativa, al fine di individuare l'effettivo valore aggiunto offerto da questo wrapper.

Nonostante sarebbe stato molto interessante, i *"web application frameworks"* non sono stati ulteriormente approfonditi, perché avrebbe richiesto troppo tempo e soprattutto esulerebbe dalla semplice valutazione di approcci multiplatform alternativi alla programmazione nativa, data la loro complessità, e la quantità e varietà degli aspetti della programmazione che abbracciano.

Capitolo 2

Librerie di particolare interesse

Dallo sviluppo di una semplice applicazione di test sfruttando le sei librerie precedentemente individuate, ne sono state individuate tre risultate particolarmente vantaggiose. Questo capitolo chiarirà come effettivamente possono essere sfruttate nello sviluppo di web app, e perché si distinguono dalle altre, mettendone in luce punti di forza ma anche di debolezza, ed eventuali limitazioni tramite un semplice esercizio.

2.1 Descrizione applicazione di test

L'*applicazione di test* è la stessa per tutte le librerie e consiste nella creazione di una lista, partendo da dati disponibili in formato JSON (JavaScript Object Notation), con la quale l'utente può interagire toccandone un elemento per visualizzarne il dettaglio.
Insieme al team di Mint è stato deciso di far risiedere le risorse (dati e immagini) in locale, per evitare problemi di same-domain-policy, una restrizione presente nei browser che impedisce l'accesso alla maggior parte dei metodi e delle proprietà tra pagine provenienti da siti diversi tramite qualsiasi tipo di richiesta HTTP. Nonostante ciò, la lista verrà sempre implementata in modo dinamico perché solitamente il contenuto è dinamico, quindi il numero di elementi in lista, i relativi dettagli, le immagini, ed eventualmente l'intestazione della pagina principale devono poter cambiare in base alle risorse che solitamente risiedono sul server, vengono aggiornate, e richieste al server tramite chiamata AJAX (Asynchronous JavaScript and XML).

2.2 Caratteristiche comuni

Tutte e tre le librerie sono open source, e offrono un sistema di UI progettato per rendere responsive le web app, quindi in grado di adattare il proprio layout in modo automatico ai dispositivi con i quali vengono visualizzati, e per avvicinarle maggiormente a look&feel e convenzioni d'interazione delle app native, predisponendo UI mobile touch-optimized. Inoltre correlano le UI gestures alle corrispondenti logiche di business, permettendo la gestione di eventi touch e multitouch, che normalmente vengono originati dall'interazione dell'utente con l'app nativa attraverso un display touchscreen, come il tocco, il doppio tocco, il pinch, la rotazione, scroll, etc. Queste librerie semplificano la manipolazione del DOM (Document Object Model), la gestione di particolari formati di dati (XML, SQLite, JSON, etc.) e il loro reperimento attraverso particolari protocolli (SOAP, etc.), e permettono di governare tramite AJAX la comunicazione con i server e la navigazione delle pagine, che può avvenire anche con effetti grafici[1].
Tutte e tre le librerie sono compatibili con vari strumenti di test (AutoMate, JsUnit, etc.).

2.3 jQuery Mobile

jQuery Mobile[13], progetto della jQuery Foundation, è una delle librerie più conosciute ed utilizzate, ed offre un buon numero di UI basato su HTML5. È ben documentata e flessibile dato che il suo sviluppo segue un approccio progressive enhancement (potenziamento progressivo), infatti prevede un core essenziale e largamente compatibile, che viene arricchito progressivamente con ulteriori componenti, fruibili dai device a seconda del loro livello di supporto. Quindi i device vengono raggruppati in gruppi a seconda di tre diversi livelli di supporto, A (completo), B (completo escluso AJAX), C (HTML essenziale), così che i device più avanzati possano disporre di un'esperienza ed un'interfaccia più ricca di quelli più arretrati, ma entrambi possano comunque beneficiare dei contenuti e delle funzionalità offerte dall'app.

Peculiarità:

- il core leggero e la configurazione HTML5-driven per il layout delle pagine con minimal scripting ne ottimizzano la velocità, in modo da non incidere in maniera pesante sulle performance dell'app;
- compatibilità con la maggior parte dei dispositivi e dei browser desktop, parzialmente anche con quelli più obsoleti;
- curva di apprendimento quasi piatta, in particolar modo per chi ha già dimestichezza con la libreria jQuery;
- facile estensibilità tramite plugin, temi, strumenti, e framework proprietari come ThemeRoller, o di terze parti come PhoneGap.

2.3.1 Applicazione di test

L'implementazione dell'applicazione di test è stata suddivisa in due file, jqmEs.html e jqmEs.js.
Nel file **jqmEs.html** ne ho implementato in modo statico la struttura principale. Questo file è un semplice documento HTML che mi ha permesso attraverso il browser di visualizzare le modifiche dell'applicazione in tempo reale durante tutto lo sviluppo. In particolare la seguente linea di codice da me inserita nell'header impedisce alle piattaforme mobile di applicare i loro algoritmi di zoom alla pagina, in modo che l'applicazione sia concepita in modo responsive (basato su dimensioni fluide che si adattano al display del device) sfruttando la libreria jQuery Mobile.

```
<meta name="viewport"
      content="width=device-width, initial-scale=1">
```

Il documento include nell'header le librerie jQuery, jQuery Mobile ed un foglio di stile CSS predefinito, attraverso le tre seguenti linee di codice, in versione CDN (Content Delivery Networks) minificata, così da ridurne la dimensione del codice sorgente rimuovendo da esso elementi non necessari come tabulazioni per indentare il codice, commenti, etc., e rendere il caricamento della libreria più veloce.

```
<script src="http://code.jquery.com/jquery-1.8.3.min.js">
</script>
<script src="http://code.jquery.com/mobile/1.2.1/
```

```
            jquery.mobile-1.2.1.min.js">
</script>
<link rel="stylesheet"
      href="http://code.jquery.com/mobile/1.2.1/
            jquery.mobile-1.2.1.min.css"/>
```

Ho scelto di utilizzare le versioni ospitate sui server di jQuery perché solitamente questi repository sono geolocalizzati, quindi le risorse vengono scaricate dal server più vicino all'utente, ed inoltre è probabile che siano già state utilizzate da altri siti precedentemente visitati dall'utente, e quindi siano già presenti nella cache del mobile browser, evitando così inutili richieste HTTP.
La pagina principale che ospiterà la lista è stata creata staticamente all'interno del body, in cui viene implementato un template single page.
jQuery Mobile sfrutta la struttura semantica delle pagine HTML5 e gli attributi *data-* per definire le diverse parti dell'interfaccia, e agevola la creazione di UI mettendo a disposizione dei *widgets*, componenti predefiniti che trasformano un elemento HTML in un componente interattivo, e hanno metodi ed eventi ad essi correlati. I widgets vengono definiti dichiarando l'attributo *data-role="nomeWidget"* sul tag HTML che li dovrà contenere. Quindi i seguenti tag * *, in base all'attributo *data-role* rappresentano la pagina nel suo complesso, e porzioni diverse della pagina.

```
<div data-role="page">
  <div data-role="header">
    <h1></h1>
  </div>
  <div data-role="content">
  </div>
  <div data-role="footer">
    <h4></h4>
  </div>
</div>
```

Una volta caricata la pagina, jQueryMobile utilizzerà questa struttura per arricchirla con altri tag, creati dinamicamente nel file JavaScript, e agganciarvi eventi e interazioni.
La seguente linea di codice include nel documento HTML il file **jqmEs.js** che andremo ad analizzare.

```
<script type="text/javascript" src="jqm.js"/></script>
```

Tutto ciò che è implementato nel file JavaScript viene eseguito quando il DOM (Document Object Model) è stato completamente caricato, perciò fa parte di una funzione definita all'interno del seguente metodo di jQuery.

```
$(document).ready(function(){})
```

Nello specifico *$()* è una funzione che trasforma ciò che è tra le parentesi in un oggetto jQuery, cioè qualcosa con la quale jQuery può lavorare.
Quindi gli step effettuati in questo file sono:

- creazione lista;
- creazione delle pagine di dettaglio per ogni elemento della lista;
- impostazione dell'intestazione a piè della pagina principale.

Ho utilizzato direttamente le informazioni in formato JSON assegnandole alla variabile *myObject*.
Ho creato la lista sfruttando il widget denominato *Thumbnails listview*, una semplice lista non ordinata, in cui ogni elemento della lista deve contenere un link. jQuery Mobile applicherà tutti gli stili necessari per mostrarla in modo mobile-friendly.
Quando l'utente toccherà un elemento della lista, la libreria innescherà un click sul link (valore univoco dell'attributo id della rispettiva pagina di dettaglio) relativo alla voce di elenco, emetterà una richiesta AJAX per l'URL nel link, aggiunge la nuova pagina nel DOM, e avvierà la transizione dalla pagina principale a quella di dettaglio. Inoltre aggiunge automaticamente alle pagine di dettaglio una freccia per permettere all'utente di tornare alla pagina principale, ma per far sì che funzioni correttamente è necessario assegnare un identificatore univoco alla pagina principale tramite l'attributo *id*.
Il codice seguente mostra l'implementazione della lista e parte del ciclo per creare i suoi elementi.

```
var $lista = $("<ul></ul>");
$lista.attr("data-role","listview");
var $divContent = $("div div:nth-child(2)");
$divContent.append($lista);
var $events = $(myObject.eventi);
for (i=0; i < $events.length; i++){
    var $li = $("<li></li>");
    $lista.append($li);
```

```
    var $a = $("<a></a>");
    $a.attr("href","#dettaglio"+i);
    $a.attr("data-transition","slidefade");
    $li.append($a);
    var $newPage = $("<div></div>");
    $newPage.attr("data-role","page");
    $newPage.attr("id","dettaglio"+i);
    $("body").append($newPage);
```

2.4 Qooxdoo Mobile

Qooxdoo Mobile[15] è una libreria JavaScript per realizzare mobile app senza scrivere codice HTML, ed è parte del progetto Qooxdoo, che è supportato da 1&1, una delle principali web host del mondo.
Qooxdoo ha architettura modulare adattabile al tipo di applicazione che si vuole realizzare, e comprende Core, Website, Desktop (per sviluppare applicazioni web con interfaccia desktop-like), Mobile, Server (per sviluppare applicazioni lato server, o Web Workers per l'esecuzione in background di uno o più threads JavaScript). In particolare il Core, modello di OOP alla base di tutti i componenti, arricchisce JavaScript con classi, interfacce e mixins, facilitando il riutilizzo del codice e garantendo coerenza tra le tipologie di progetti.
Il set di strumenti include un parser JavaScript, che è parte integrante del processo di generazione automatica e viene utilizzato per ottimizzare, comprimere, collegare, distribuire nel web le app e generarne la documentazione.

Peculiarità:

- fornisce le funzionalità chiave della OOP, come la definizione di classi e di interfacce, e il supporto all'ereditarietà, anche multipla dal momento che una classe può implementare/includere uno o più interfacce/mixins, che a loro volta possono estenderne altre/i;

- comprende mixins[1], classi che contengono una combinazione (non ottenuta tramite ereditarietà) di metodi di altre classi;

- fornisce un potente e semplice sistema per personalizzare layout e tematizzazione tramite SCSS, la nuova sintassi del linguaggio di scrip-

ting Sass (Syntactically Awesome Stylesheets) che estende CSS rendendo i fogli di stile più modulari e gestibili, le cui parti principali sono mantenute dal framework e che dev'essere compilato in CSS;

- supporta internazionalizzazione e localizzazione, ovvero l'adattamento dell'app senza particolari modifiche per diverse lingue e requisiti tecnici di un particolare target di mercato;
- ricca documentazione con numerosi esempi dimostrativi e la possibilità di eseguire e modificare il codice in un Playground online.

2.4.1 Applicazione di test

Dopo aver installato la libreria sul mio pc, ho richiesto la generazione automatica dello scheletro dell'applicazione di test QApp eseguendo il seguente comando da command line:

```
./tool/bin/create-application.py --type=mobile --name=QApp
--out=..
```

Dopo essermi posizionata all'interno della directory "QApp" ho richiesto la generazione dei sorgenti eseguendo il seguente comando nella command line:

```
./generate.py source
```

Il risultato dei due comandi precedenti è quello mostrato in figura 2.1.

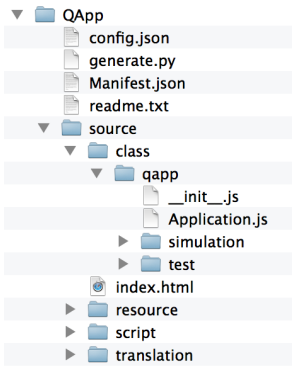

Figura 2.1: Scheletro dell'app QApp

Per sviluppare l'applicazione di test ho sfruttato i file index.html e Application.js.
Il file predefinito **index.html** è un semplice documento HTML con body vuoto, che non è stato necessario modificare in quanto presentava già i metadati adeguati al caso, e che quindi ho utilizzato solo per visualizzare attraverso il browser le modifiche dell'applicazione in tempo reale durante tutto lo sviluppo.
L'applicazione di test è stata implementata nel main del file **Application.js**. Il codice seguente di questo file è stato generato automaticamente alla creazione del progetto, e definisce la classe principale dell'applicazione che implementa la funzione *main()*, punto di inizio per l'esecuzione dell'applicazione:

```
qx.Class.define("QApp.Application",{
  extend : qx.application.Mobile,
  members :{
    main : function(){
      //Call super class
      this.base(arguments);
      //Enable logging in debug variant
      if (qx.core.Environment.get("qx.debug")){
        //support native logging capabilities
        qx.log.appender.Native;
        //support additional cross-browser console
        qx.log.appender.Console;
      }
```

Per prima cosa ho creato la pagina principale per contenere la lista attraverso un oggetto di classe *NavigationPage*, e un manager delle pagine attraverso un oggetto di classe *Manager*, al quale ho aggiunto la pagina per poterla gestire, come mostrato nel codice seguente:

```
var page = new qx.ui.mobile.page.NavigationPage();
var manager = new qx.ui.mobile.page.Manager(false);
```

Il parametro false passato al costruttore del manager innesca il gestore di layout per dispositivi mobile ad esclusione dei tablet.
Ho utilizzato direttamente le informazioni in formato JSON assegnandole alla variabile *myObject* e utilizzando l'oggetto *eventi* tramite il seguente codice.

```
var events = myObject.eventi;
```

Successivamente ho implementato lista e pagine per mostrare il dettaglio di ogni suo elemento all'interno dell'ascoltatore (funzione callback) dell'evento *initialize* (inizializzazione del ciclo di vita). Ho realizzato la lista creando un oggetto di classe *Lista* e passando come parametro al costruttore, il modello dei dati che devono essere mostrati nella lista. L'infrastruttura virtuale alla base della classe *Lista* ha notevoli vantaggi, soprattutto quando c'è una grande quantità di elementi, perché crea solo i widgets (componenti predefiniti che rendono interattivo un elemento HTML) per gli elementi visibili nella lista e li riutilizza, risparmiando tempo di creazione e memoria.
Per rendere visibile la pagina di dettaglio corrispondente all'elemento selezionato dall'utente ho aggiunto alla lista l'ascoltatore dell'evento *changeSelection*.
Quanto appena detto è esposto nel codice seguente.

```
var list = new qx.ui.mobile.list.List({
    configureItem : function(item,data,row){
        item.setImage(data.immagine);
        item.setTitle(data.titolo);
        item.setSubtitle(data.sottotitolo);
        item.setSelectable(true);
        item.setShowArrow(true);
     }
});
list.addListener("changeSelection", function(evt){
    pageD[evt.getData()].show();
}, this);
```

Successivamente all'interno di un ciclo ho creato un array di pagine di dettaglio, oggetti per i quali la loro classe *NavigationPage* prevede l'aggiunta automatica di un back button, e ho memorizzato in un ulteriore array le informazioni di dettaglio per ogni elemento.
Inoltre ho impostato il contenuto delle pagine di dettaglio e gestito la pressione del pulsante back.

```
for(var i=0; i<events.length; i++){
    arrayData.push({
        titolo:events[i].titolo,
        sottotitolo:dataE,
        immagine:events[i].url_img,
        abstract:events[i].abstract,
        contenuto:events[i].contenuto
    });
```

```
    pageD[i] = new qx.ui.mobile.page.NavigationPage();
    pageD[i].setUserData("abstract",events[i].abstract);
    pageD[i].addListener("initialize", function(){
        var label1 = new qx.ui.mobile.basic.Label(
            this.getUserData("abstract")
        );
        this.getContent().add(label1);
    },pageD[i]);
    pageD[i].addListener("back", function(){
        page.show({reverse:true});
    }, this);
    manager.addDetail(pageD[i]);
}
```

L'ultima riga di codice del ciclo aggiunge le pagine di dettaglio al manager delle pagine.
Inoltre ho settato la proprietà model della lista con l'array di informazioni di dettaglio e aggiunto la lista al container della pagina, come di seguito.

```
list.setModel(new qx.data.Array(arrayData));
page.getContent().add(list);
```

Infine ho aggiunto la pagina principale al detailContainer del manager delle pagine, e reso visibile la pagina principale tramite le seguenti due righe di codice.

```
manager.addDetail(page);
page.show({reverse:true});
```

2.5 Sencha Touch

Sencha Touch[14] è una libreria specializzata in mobile app che cerca di offrire l'esperienza di un linguaggio OOP, e fornisce astrazioni di alto livello. Inoltre supporta internamente tutto il ciclo di vita dell'app, fornendo anche la possibilità di racchiudere la web app in un pacchetto nativo tramite il Sencha Cmd.

Peculiarità:

- offre un sistema di classi proprietario che emula i modelli OOP, utilizzando oggetti JavaScript in stile JSON, con un sistema di ereditarietà e modelli di oggetti, simili al concetto di classi;

- adotta il pattern architetturale MVC, attraverso le classi estendibili Application e Controller per coordinare l'interazione tra componenti di UI e i Model e gli Store;
- vasto numero di componenti built-in tematizzabili, compreso grafici animati e interattivi;
- consente di sfruttare l'accelerazione hardware, quando messa a disposizione dal dispositivo, per aumentare la velocità dell'app all'avvio e durante l'interazione con l'utente;
- fornisce temi predefiniti che ricalcano quelli delle app native, per adattare il tema dei componenti di UI all'OS mobile, utilizzando Sass (estensione di CSS3 che permette di aggiungere regole nidificate, variabili o mixins);
- estendibile con strumenti proprietari come Sencha Architect (un visual HTML5 app builder), Sencha Space (per gestire app, utenti e dati), etc., e librerie di terze parti;
- ricca documentazione con esempi, forum proprietario e corsi di formazione.

2.5.1 Applicazione di test

Dopo aver installato la libreria sul mio pc ho richiesto la generazione automatica dello scheletro dell'applicazione di test STApp eseguendo il seguente comando nel Sencha Cmd:

```
sencha generate app STApp ../STApp
```

Per sviluppare l'applicazione ho sfruttato il file predefinito **index.html** solamente per visualizzare attraverso il browser le modifiche dell'applicazione in tempo reale durante tutto lo sviluppo. L'implementazione dell'applicazione di test è stata suddivisa in quattro file.
La funzione implementata nel file **app.js** carica l'applicazione, identificata dalla classe *Ext.app.Application*. L'applicazione consiste in una pagina singola con la configurazione passata come parametro d'ingresso alla funzione appena citata, che definisce il set di Models, Stores, Controllers, e Views dei quali consiste l'app, ognuno nella rispettiva directory all'interno della

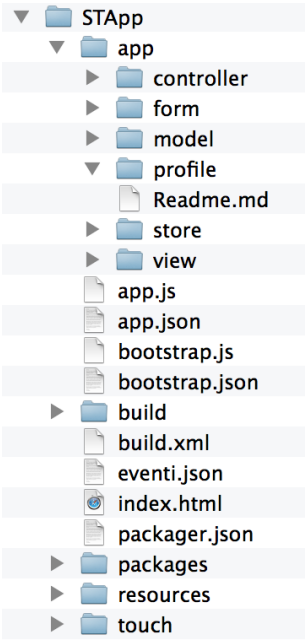

Figura 2.2: Scheletro dell'app STApp

directory "app". In particolare i Models rappresentano gli oggetti dati da gestire, gli Stores sono responsabili del caricamento dei dati nell'app, i Controllers agiscono di conseguenza alle interazioni dell'utente, e i Views sono responsabili della visualizzazione dei dati.
La funzione *launch()* avvia l'applicazione creando un Viewport, che rappresenta la finestra del browser, al quale ho aggiunto un'istanza di classe *Main*, che rappresenta la schermata principale, per renderla visibile.
Quanto appena descritto è esposto nel codice seguente.

```
Ext.application({
    name: 'STApp',
    requires: [
        'Ext.dataview.NestedList',
        'Ext.data.TreeStore',
        'Ext.TitleBar',
        'Ext.data.Model',
        'Ext.XTemplate',
        'Ext.data.StoreManager',
        'Ext.data.NodeStore',
```

```
        'Ext.Container',
        'Ext.Panel',
        'STApp.model.ListItem'
    ],
    models: ['ListItem'],
    stores: ['TreeStore'],
    controllers: [],
    views: ['Main'],
    launch: function() {
        Ext.Viewport.add(Ext.create('STApp.view.Main'));
    },
```

Nel file **Main.js** ho definito la rispettiva classe estendendo *Ext.navigation.View*, classe che rappresenta un container che rende visibile solo una schermata alla volta e gestisce automaticamente la transizione da quella principale a quella di dettaglio e viceversa, aggiungendo anche un back button alle schermate di dettaglio.
In particolare tramite il codice seguente ho creato e configurato una lista nidificata di classe *Ext.dataview.NestedList* che caricherà i dati da un TreeStore, da me definito nel rispettivo file. Ho configurato la schermata di dettaglio detailCard come un pannello scorrevole, e ho implementato l'ascoltatore di leafitemtap in modo che imposti il contenuto della schermata di dettaglio con i campi del modello di dati relativi all'elemento in lista che è stato selezionato.

```
items: [{
   xtype: 'nestedlist',
   title: 'EVENTI',
   scrollable: {
      direction: 'vertical',
      directionLock: true
   },
   displayField: 'titolo',
   store: 'TreeStore',
   detailCard: {
      xtype: 'panel',
      scrollable: true,
      styleHtmlContent: true
   },
   listeners: {
      leafitemtap: function(
         nestedList, list, index, element, record
      ){
```

```
            this.getDetailCard().setHtml(
                record.get('abstract')+record.get('contenuto')
            );
        }
    }
}]
```

Nel file **TreeStore.js** ho definito la classe *TreeStore* estendendo l'omonima predefinita che funge da archivio per i dati nidificati, e configurando lo store con il Model di classe *ListItem* associato ad esso, e con le informazioni in formato JSON come root (nodo principale), come nel seguente codice.

```
Ext.define('STApp.store.TreeStore', {
    extend: 'Ext.data.TreeStore',
    requires: ['STApp.model.ListItem'],
    config: {
        model: 'STApp.model.ListItem',
        defaultRootProperty: 'eventi',
```

I Models vengono automaticamente registrati nel model manager per essere utilizzati dagli Stores. Nel file **ListItem.js** ListItem ho definito la classe omonima estendendo la classe predefinita *Ext.data.Model* e definendo un array di campi che identifica le informazioni presenti per ogni oggetto JSON nello Store, come esposto nel codice seguente.

```
Ext.define('STApp.model.ListItem', {
    extend: 'Ext.data.Model',
    config: {
        fields: [
            'titolo','abstract','contenuto',
            'data','url_img_thumb','img_url'
        ]
    }
});
```

2.6 Considerazioni finali

Dallo sviluppo della stessa applicazione di test sfruttando le sei librerie individuate dopo la fase di analisi descritta nel primo capitolo, sono affiorati concretamente i loro pregi e difetti, consentendomi di identificarne tre di maggior interesse.

In particolare ho riscontrato l'effettiva semplicità di utilizzo di *jQueryMobile*, che risulta quindi essere quella con curva di apprendimento minore. Credo però che questa libreria possa risultare veramente utile solo in progetti molto semplici, come nel caso in esame, perché offre la manipolazione del DOM attraverso un'astrazione che definirei di basso livello, permettendo di manipolarlo attraverso widget molto semplici, che rendono interattivi alcuni elementi HTML, costringendo lo sviluppatore a continuare a progettare l'app in termini di singoli tag HTML necessari. Infatti utilizzando jQuery Mobile ogni oggetto corrisponde praticamente ad un tag. Quindi appena la complessità della web app aumenta, l'approccio offerto da questa libreria tende a diminuire rendendo il codice eccessivamente prolisso, e poco chiaro.

Qooxdoo Mobile invece offre un astrazione di livello maggiore rispetto a jQueryMobile, generando automaticamente la struttura dell'app, e permettendo di organizzare il codice attraverso il concetto di classi, che ne favoriscono leggibilità, modularità e riuso. Nello sviluppo tramite Qooxdoo Mobile ho potuto concepire gli oggetti in maniera più astratta e lontana dal semplice tag. Inoltre questa libreria è risultata essere molto flessibile, perché permette di manipolare il DOM con una buona astrazione, senza rimuovere la possibilità di manipolazione diretta.

Infine *Secha Touch* è quella a mio parere più completa in termini di offerta, perché oltre a offrire un sistema di classi proprietario che emula i modelli di OOP, adotta il pattern architetturale MVC che consente di migliorare la gestione e la manutenzione di progetti di grandi dimensioni, e per i quali quindi risulta quindi la più adatta. La curva di apprendimento per imparare a sfruttare Sencha Touch è risultata essere effettivamente piuttosto ripida, perché la libreria si basa su molte API proprietarie, e la documentazione seppur ricca è a mio parere poco chiara. Anche tramite questa libreria è possibile manipolare il DOM ad alto livello, astraendo notevolmente dal concetto di tag.

Penso quindi che per sviluppare una web app sia conveniente scegliere una delle tre sopra citate, in considerazione della complessità del progetto da realizzare.

Capitolo 3

Sviluppo caso di studio applicativo su Titanium

In questo capitolo verranno testate le peculiarità del framework Titanium mostrando lo sviluppo di un prototipo di app ibrida, al fine di ottenere un riscontro pratico delle sue caratteristiche, ponendola anche a confronto con la rispettive versioni native.
Inizialmente però verranno analizzate le specifiche di Titanium, individuando le strategie che adotta per affrontare le problematiche dello sviluppo mobile multi-platform, e rendere la progettazione e l'implementazione delle app più rapide, agili ed economiche.

3.1 Caratteristiche generali

Titanium[1][24] è un ambiente di sviluppo estensibile ed open source, ideato per permettere la creazione di mobile app multi-platform, principalmente ibride (JavaScript ed eventualmente XML) ma anche web (HTML5, CSS3, JavaScript), è attualmente alla versione 3.2 ed è un sistema in continua evoluzione. Le piattaforme attualmente supportate sono iOS, Android, BlackBerry OS, Tizen, i mobile stock browser e probabilmente a fine 2014 anche Windows Phone, anche se il supporto non è esattamente dello stesso livello per tutte le piattaforme, infatti iOS e Android godono di un maggior supporto rispetto agli altri OS.
Questo framework è parte del progetto più evoluto Appcelerator Platform,

una piattaforma open source completa che va incontro alle esigenze aziendali integrando in un unico strumento funzioni di sviluppo, implementazione e analisi, e che appartiene ad Appcelerator, società di mobile technology con sede a Mountain View, California.
L'obiettivo di Titanium è quello di aiutare gli sviluppatori a sfruttare le loro abilità JavaScript per costruire applicazioni mobile che girano su più piattaforme, offrendo un ambiente di runtime e delle API JavaScript per costruire app con look&feel nativo e performance il più possibile vicine a quelle native, che si adattano bene all'interno dell'ecosistema di ogni piattaforma.
Titanium infatti è stato costruito secondo le due asserzioni seguenti[25]:

- c'è un nucleo di API per lo sviluppo mobile che può essere normalizzato tra le piattaforme, e questo dovrebbe essere oggetto di riutilizzo del codice;
- ci sono API platform-specific, convenzioni per le UI, e caratteristiche che gli sviluppatori dovrebbero includere nello sviluppo per le specifiche piattaforme, quindi il codice platform-specific dovrebbe esistere per questi casi d'uso per fornire la migliore esperienza possibile.

Secondo Appcelerator le app dovrebbero, quando opportuno, trarre vantaggio dalla familiarità e dalle alte prestazioni dei componenti di UI nativi, senza però avere la necessità di imparare le API platform-specific per funzionalità generiche come ad esempio disegnare un rettangolo o fare una richiesta HTTP.
Titanium quindi è un tentativo di raggiungere il riutilizzo del codice tramite API JavaScript, mantenendo le caratteristiche specifiche di ogni piattaforma e le prestazioni native per soddisfare le aspettative degli utenti, perciò il motto di Appcelerator è *"write once, adapt everywhere"*. Inoltre mette a disposizione API cross-platform per l'accesso ai componenti nativi di UI come ad esempio navigation bars e menu, e alle funzionalità dei device inclusi file system, geolocalizzazione, accelerometro, etc., e consente anche l'accesso trasparente alle funzionalità native non ancora coperte dalle API. Quindi durante l'esecuzione dell'app non vi è nessuna emulazione visiva in corso (attraverso l'uso di CSS, o il rendering di widget UI utilizzando OpenGL o Flash), animazioni e comportamenti corrispondono a ciò che l'utente si aspetta, perché si sta utilizzando lo stesso controllo di UI che viene usato nelle app native.

3.1.1 Architettura della piattaforma

Titanium permette di sviluppare web app tramite HTML5, CSS3 e JavaScript, oppure app ibride con o senza l'utilizzo di WebView tramite principalmente JavaScript. Generalmente però viene utilizzato per sviluppare app ibride permettendo di evitare le limitazioni imposte dall'utilizzo di WebView, sfruttando così le classi proprietarie che implementa per consentire agli sviluppatori di eseguire il rendering di elementi di UI nativi, oltre ad offrire il controllo delle funzionalità dei device, esposte attraverso API JavaScript.
Tutto il codice sorgente dell'app viene distribuito al dispositivo mobile, in cui viene interpretato a runtime utilizzando un motore JavaScript, in particolare V8 JavaScript engine di Google è utilizzato su Android e BlackBerry OS, e JavaScriptCore di Apple su iOS (altrimenti nel caso si sfrutti una WebView viene utilizzato il built-in JavaScript engine). Titanium[25] funge da "ponte" tra la piattaforma mobile e il codice sorgente dell'app, offrendo un livello di collegamento che associa le funzioni JavaScript alle API native, quindi la chiamata delle funzioni a *Titanium.some_function* invoca il codice nativo sottostante.

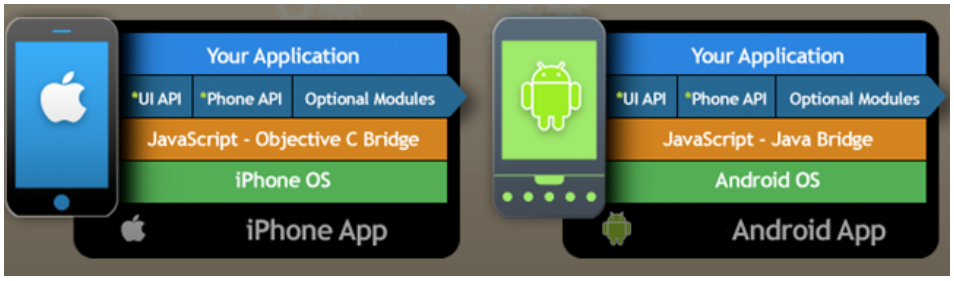

Figura 3.1: Architettura delle app sviluppate con Titanium

Esaminando l'architettura più in profondità, a runtime viene creato un ambiente di esecuzione JavaScript in codice nativo, e il codice JavaScript viene iniettato come un oggetto allineato e abbinato agli oggetti nativi all'interno dell'ambiente di runtime. L'abbinamento è one-to-one allo stesso modo in cui viene eseguito codice JavaScript all'interno di codice HTML.
In sostanza, nell'ambiente di runtime JavaScript dell'app vengono iniettati oggetti "proxy", cioè oggetti JavaScript ognuno dei quali ha un oggetto asso-

ciato in codice nativo. Ogni oggetto proxy esiste sia in ambiente JavaScript che in ambiente nativo, e funge da "ponte" tra i due. Nel codice JavaScript, quando viene chiamata una funzione sull'oggetto globale *Titanium* o *TI*, come *var b = Ti.UI.createButton();*, questo invocherà un metodo nativo che creerà un oggetto di UI nativo, e un oggetto proxy (*b*) che espone in JavaScript le proprietà e i metodi dell'oggetto nativo di UI sottostante. Kroll, parte dell'SDK, è il particolare componente che traduce le chiamate alle funzioni nei loro equivalenti nativi. Quindi ad esempio quando si modifica un Titanium Button, Kroll applica le modifiche corrispondenti all'equivalente nativo, e quando si verifica un evento in ambiente nativo, Kroll lo cattura, lo gestisce e poi lo propaga all'ambiente JavaScript. Componenti di UI (view proxy) possono essere organizzati gerarchicamente per creare UI complesse. Oggetti proxy che rappresentano una interfaccia ad API non visive, come filesystem I/O o database access, vengono eseguiti in codice nativo, e in modo sincrono o asincrono, per API come network access, restituiscono un risultato in JavaScript.
A runtime l'app è formata da tre componenti principali:

- il codice sorgente JavaScript inglobato in un file nativo e compilato come una stringa codificata;
- l'implementazione delle API di Titanium platform-specific in linguaggio nativo;
- l'interprete JavaScript, incorporato nel codice sorgente durante il build process, che viene utilizzato per valutare il codice a runtime.

Titanium perciò non richiede l'uso di una WebView (è possibile creare una WebView come widget nativo di UI, ma questa non viene utilizzata per valutare il codice sorgente scritto con Titanium), e non c'è cross-compilazione in codice nativo (compilazione di un codice sorgente ottenendo un file binario eseguibile su di un elaboratore con architettura diversa da quella della macchina su cui si è lanciato il cross-compilatore stesso).

Ad alto livello Titanium è composto dalla combinazione dei seguenti componenti:

- Titanium JavaScript API (più di 5000), che consentono l'accesso a centinaia di UI native, e alle funzionalità dei device;

- Titanium SDK, componente fondamentale che comprende un set di utilities basate su Node.js (framework event-driven), e strumenti che lavorano con gli SDK nativi combinando il sorgente JavaScript, l'interprete JavaScript e le risorse statiche in un file binario pronto per l'installazione;

- Titanium Studio, IDE Eclipse-based derivato da Aptana Studio che Appcelerator ha acquisito nel gennaio 2011, per scrivere, testare, fare il debug (sul device e sull'emulatore), distribuire l'app, e gestire gli aggiornamenti di Titanium SDK e l'utilizzo di moduli, inoltre comprende template e semplici app di esempio;

- moduli, che estendono il nucleo di Titanium e le funzionalità chiave delle API, e possono essere liberamente realizzati dagli sviluppatori per aggiungere funzionalità del device e componenti di UI esposti in JavaScript, implementando un Proxy e/o un'interfaccia View Proxy in codice nativo, che possono essere distribuiti attraverso il Marketplace di Titanium;

- Alloy, un framework MVC che consente di sviluppare app in modo strutturato, utilizzando anche XML, per facilitare la creazione di componenti modulari, e migliorare significativamente leggibilità, manutenibilità e riusabilità del codice, diminuendo tempi e costi di sviluppo;

- Cloud Services, un MBaaS (Mobile Backend as a Service), cioè servizi di back-end e analytics che forniscono statistiche di utilizzo come quanto spesso viene utilizzata l'app e su quali piattaforme, realizzati con approccio modulare e riutilizzabile in modo che un singolo backend cloud-based sia in grado di supportare più app, inoltre permettono di registrare analisi personalizzate in modo da poter monitorare l'utilizzo di caratteristiche specifiche, come i click sui pulsanti, l'accesso ai dati, o qualsiasi altro tipo di interazione con l'utente.

Nel giugno 2013 Jeff Haynie, CEO di Appcelerator, ha annunciato che la società ha avviato lo sviluppo di Ti.Next[25], un progetto per riscrivere Titanium SDK in Javascript per migliorarne le performance e portare gli utenti finali di Titanium, che scrivono in JavaScript, più vicini al codice interno. Per questo progetto è stato ideato un piccolo microkernel, chiamato

TiRuntime, con bootstrap in linguaggio nativo minimale, che comunica con un set comune di compilatori, tools e un'unica JavaScript VM. Il team di Titanium ha trovato un modo per far funzionare WebKit KJS VM su più piattaforme mobile evitando così l'utilizzo di una VM diversa per ogni piattaforma. In questo modo è stato ottimizzato il microkernel, riducendo le linee di codice per piattaforma all'incirca da centomila a cinquemila, così da semplificare notevolmente ottimizzazioni, mantenimento e profiling. Inoltre una parte molto importante di Ti.Next è Hyperloop, l'infrastruttura del compilatore di nuova generazione che migliorerà le prestazioni, l'uniformità tra le piattaforme, la scalabilità e la manutenibilità di Titanium SDK, senza cambiare le API.
L'uscita ufficiale di Ti.Next è prevista nel quarto quadrimestre del 2014, e includerà il supporto per Windows Phone.

3.2 Un caso applicativo: MetropolisTitan

Questo paragrafo fornisce un esempio pratico di come un prototipo di app ibrida possa essere implementata concretamente sfruttando Titanium.
Il processo di sviluppo per la realizzazione del prototipo di app è stato scomposto nelle attività fondamentali del ciclo di vita del software, e cioè le fasi preliminari di analisi e progettazione, la fase implementativa in cui si scelgono gli strumenti ottimali e si realizzano i componenti, e la fase di testing e debugging.
Lo sviluppo del prototipo di app è avvenuto all'interno dell'azienda Mint nel corso di un tirocinio, allo scopo di confrontarlo con la rispettiva versione nativa sviluppata precedentemente dal team di Mint, così da verificare concretamente i vantaggi dello sviluppo multi-platform attraverso l'utilizzo del framework Titanium.

3.2.1 Analisi dei requisiti

Questa fase ha lo scopo di definire, il più precisamente possibile, il problema da risolvere, così da determinare cosa farà il sistema, e generalmente è costituita anche dalla raccolta dei dati tramite colloqui tra cliente/committente e relativi sviluppatori. In questo caso l'obiettivo è quello di creare un prototipo di app ibrida che permetta di accedere alle informazioni su Metropolis - Paints for Lifestyle, la divisione decorativi di IVAS S.p.A.

Tenendo presente che il framework Titanium permette lo sviluppo multi-platform mettendo a disposizione non solo le funzionalità del dispositivo, ma anche i componenti di UI nativi, e che lo scopo dello sviluppo di questa app è il confronto con la rispettiva versione nativa, come stabilito con il team di Mint, è importante che la UI dell'app sia, il più possibile, uguale a quella di Metropolis, la rispettiva app nativa precedentemente realizzata sia per iOS che per Android dagli sviluppatori di Mint. La parte dell'aspetto grafico ritenuta importante da *replicare* è soprattutto *il layout*, quindi tutti i dettagli secondari come le descrizioni dei prodotti, le scritte nelle immagini nella schermata principale, etc. non sono richiesti. Quindi nonostante Titanium supporti anche lo sviluppo di app ibride per Tizen e BlackBerry OS, e considerato l'obbiettivo di questo progetto, l'app è stata sviluppata solamente per le piattaforme iOS e Android.
Vediamo ora le caratteristiche principali e le funzionalità che deve offrire, nelle seguenti specifiche funzionali in ordine di priorità:

- appena messa in esecuzione l'app deve mostrare per qualche secondo l'immagine contenente il simbolo della divisione Metropolis detta *"splash screen"*, e poi mostrare all'utente una *schermata principale* che gli permetta di accedere a quattro ulteriori viste, ognuna a schermo intero, tramite la selezione di uno dei quattro elementi (Tabs di una TabBar), l'interattività delle tre immagini della schermata principale non è richiesta;

- l'app deve implementare la funzionalità *"Scelta Guidata"* dell'app Metropolis, che permette all'utente di scegliere il prodotto adatto alle proprie esigenze, offrendo quindi la ricerca guidata dei prodotti attraverso la scelta del tipo di superficie, della tonalità di colore e del tipo di ambiente, mostrando i nomi dei prodotti risultanti dalla selezione dell'utente con la relativa immagine, suddivisi per linea di prodotto (Colors, Urban Effects, Resins), sfruttando un file.plist;

- l'app deve implementare la funzionalità *"Rivenditori"* dell'app Metropolis, mostrando la lista di tutti i rivenditori ordinata secondo la loro ubicazione geografica partendo dal più vicino all'utente, con alla fine la lista dei manager di area, mostrando per ogni rivenditore nome, numero di telefono, e-mail, tipologia, chilometri di distanza dall'utente, e per ogni manager nome, cognome, numero di telefono,

e-mail, e qualifica, sfruttando un file.csv e la geolocalizzazione; inoltre deve offrire all'utente la possibilità di visualizzare una mappa navigabile che indichi la posizione attuale dell'utente e quella di tutti i rivenditori, sfruttando la geolocalizzazione.

A seguire alcuni screenshots dell'app originale Metropolis, rispettivamente per Android e per iPhone.

Figura 3.2: Schermata principale della versione per Android

Figura 3.3: Mappa con i rivenditori della versione per Android

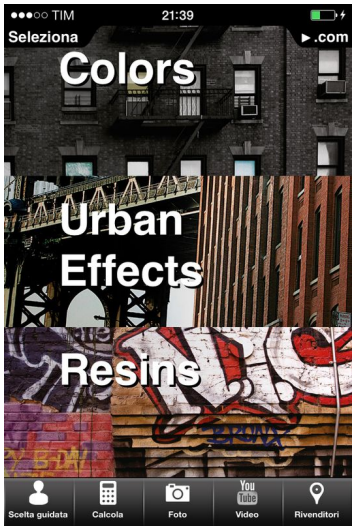

Figura 3.4: Schermata principale della versione per iPhone

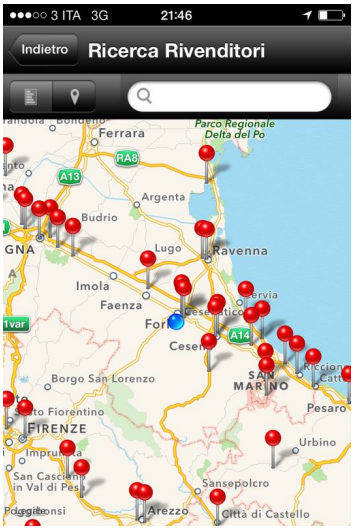

Figura 3.5: Mappa con i rivenditori della versione per iPhone

3.2.2 Progettazione

Nell'attività di progettazione si entra nel merito della struttura che dovrà avere il sistema software da realizzare, definendone le linee guida in funzione dei requisiti evidenziati dall'analisi, per stabilire in che modo saranno soddisfatti dal sistema.
Ora andremo a definire le linee guida da seguire durante l'implementazione vera e propria, progettando l'architettura dell'app, e definendo le scelte progettuali adottate per implementare le funzionalità definite in analisi.
L'idea alla base di tutta la progettazione è quella di evitare strategie specifiche per ognuna delle due piattaforme, preferendo un unico approccio progettuale così da ridurre i tempi di sviluppo.
Per quanto riguarda l'intera architettura dell'app, ho ritenuto opportuno adottare il pattern architetturale MVC, che consiste nel dividere l'app in tre parti interconnesse tra loro (Model, View, Controller). Questo pattern consente di realizzare un'architettura che separa le rappresentazioni interne delle informazioni, dalla logica di presentazione dei dati all'utente, e dalla logica di controllo, permettendo di realizzare un'implementazione più chiara, aumentandone la flessibilità e il riutilizzo.
Quindi l'app è stata strutturata principalmente secondo la seguente architettura:

- Model, consiste nel modello dei dati (qui risiedono i file in cui vengono conservate le informazioni che utilizza l'app), quindi riceve le richieste del Controller al quale passa i dati;

- View, implementa la logica di presentazione, perciò questo componente crea l'interfaccia utente, anche sfruttando i dati che gli passa il Controller;

- Controller, realizza la business logic, quindi richiede i dati al Model, li elabora e passa i risultati al View.

Dopo aver velocemente appurato che l'utilizzo di Titanium tramite Alloy, framework MVC di Titanium che sfrutta anche XML, avrebbe reso ancor più ripida la curva di apprendimento dell'intero framework, si è deciso insieme al team di Mint di utilizzare solamente Titanium SDK, scrivendo perciò l'intero codice sorgente in JavaScript. Ciò nonostante, ho deciso di suddividere il sistema in più componenti raggruppandoli in directory diverse a

seconda del ruolo che svolgono nel pattern MVC.
Inoltre ho deciso di organizzare il progetto in moduli, per separare concettualmente le diverse risorse utilizzate dall'app rendendo più facile la fase d'implementazione, e scomporre la soluzione da implementare in sottoattività meno complesse.
Quindi in seguito a quanto appena detto, la struttura dell'app è risultata essere quella in figura 3.6.

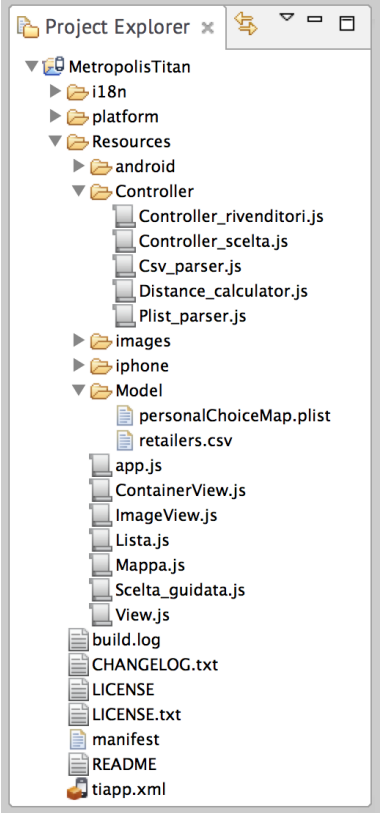

Figura 3.6: Struttura del progetto MetropolisTitan

In particolare all'esterno della directory "Resources" sono presenti solo i file riguardanti la configurazione del progetto, i componenti che svolgono il ruolo di View sono i file all'interno della directory "Resources" ma al di fuori di tutte le sue subdirectory, e le risorse dell'app sono state racchiuse nella directory "images", mentre quelle specifiche per Android e per iPhone

si trovano nelle rispettive directory.
La suddivisione in più moduli, separando le varie parti dell'app e racchiudendole in diversi file, è stata concepita anche al fine di realizzare un sistema composto, nel quale sia più semplice e veloce effettuare modifiche, o correzioni di errori, risultando più circoscritti. Questo approccio rende più comprensibile il funzionamento dell'app, e ne facilita l'estensione e il miglioramento. Inoltre comporta anche vantaggi per eventuali progetti futuri, infatti nel caso sia necessario sviluppare altre app che presentano alcune funzionalità uguali a queste, è possibile risparmiare tempo riutilizzando le parti già sviluppate in precedenza.

Allo scopo di realizzare un'app che fosse il più possibile fluida e reattiva agli input dell'utente, ho deciso di progettarla in modo che crei tutte le UI necessarie per il suo funzionamento non appena viene avviata dall'utente, aggiungendo eventualmente dinamicamente i dati completamente dipendenti dalle scelte dell'utente.
Ora vediamo più nel dettaglio quali strategie sono state adottate per affrontare alcune problematiche sorte per la realizzazione delle funzionalità inidividuate in analisi.
Insieme al team di Mint si è deciso di permettere all'app l'accesso diretto alle risorse da utilizzare (immagini, file.csv e file.plist), facendole risiedere al suo interno anzichè su un eventuale server, puntando ad una più facile e veloce implementazione della business logic. Sempre insieme e per la stessa motivazione si è scelto di sfruttare moduli precedentemente sviluppati da terzi per la realizzazione delle sottoattività più comuni come parsing di file.plist e file.csv, e il calcolo della distanza tra due luoghi, ovviamente scegliendoli dopo averne verificato l'efficacia.
Inoltre, per dare priorità all'implementazione delle funzionalità richieste nei requisiti, abbiamo deciso di adottare una semplificazione generale che consiste nella realizzazione della TabBar in modo persistente, quindi presente anche nelle schermate delle funzionalità, anzichè essere nascosta come nell'app originale, e conseguente assenza della NavigationBar in iOS e della gestione del BackButton in Android.

Il comportamento del sistema in merito alla funzionalità *"Scelta guidata"* è stato gestito in base al semplice modello di ASF (automa a stati finiti), nello specifico quello di una macchina di Mealy, automa che genera un'uscita a

partire dagli stati d'ingresso e dallo stato corrente. Quindi ho sfruttato il file.plist che mi è stato dato dal team di Mint, contenente le informazioni ramificate in base ai rapporti di dipendenza dei prodotti dalla tipologia del prodotto, dal colore, e dalla tipologia di ambiente, seguendo l'albero in esso contenuto attraverso le tre scelte sequenziali dell'utente, memorizzate in tre variabili. Le tre variabili che mantengono l'input dell'utente vengono sovrascritte ogni volta che l'utente effettua una nuova scelta, e i prodotti risultanti dalla selezione vengono aggiunti dinamicamente ad una lista vuota, creata staticamente all'avvio dell'app.

Per realizzare la funzionalità *"Rivenditori"* ho sfruttato il file.csv che mi è stato dato dal team di Mint e la geolocalizzazione, in modo che l'app reperisca le informazioni sui rivenditori e sull'ubicazione dell'utente al suo avvio, creando un'unica view in cui aggiungere e rimuove la lista e la mappa in base all'input dell'utente.

Per comprendere meglio l'architettura basata sul pattern MVC, e l'interazione tra le sue parti, di seguito ne vedremo rispettivamente una rappresentazione UML tramite object diagram e sequence diagram.

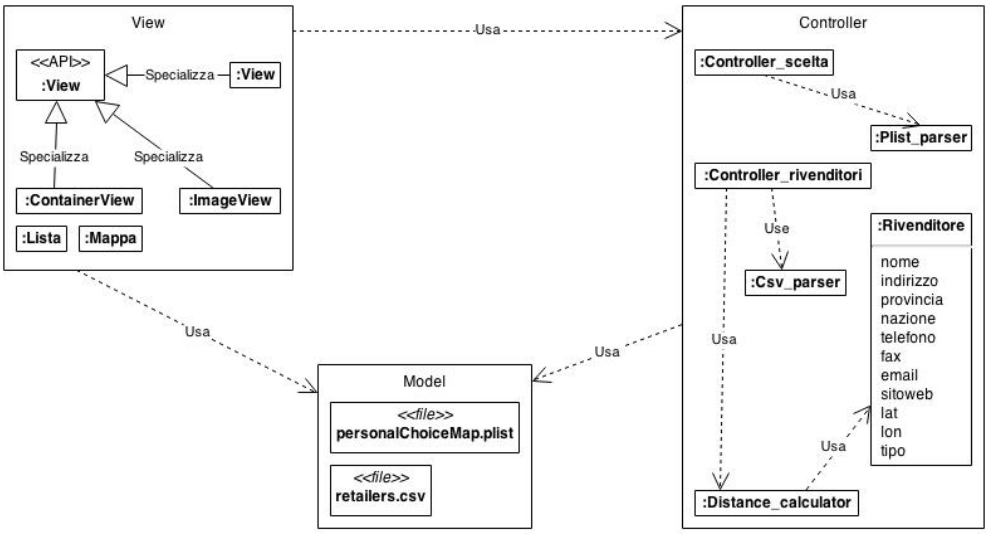

Figura 3.7: Architettura del progetto MetropolisTitan adottando il pattern MVC

L'object diagram di figura 3.7 mostra le tipologie di oggetti che ho deciso di progettare personalmente allo scopo, utilizzando le funzioni per simulare in qualche modo le classi. Tutti gli oggetti di UI utilizzati direttamente tramite API di Titanium, come Button, Label, etc., non sono stati inseriti nel diagramma ritenendo poco importante il legame tra essi che è solo legato a questioni di layout. L'unico oggetto appartenente alle API di Titanium che ho ritenuto rilevante mostrare in figura è View, perché è stato specializzato.

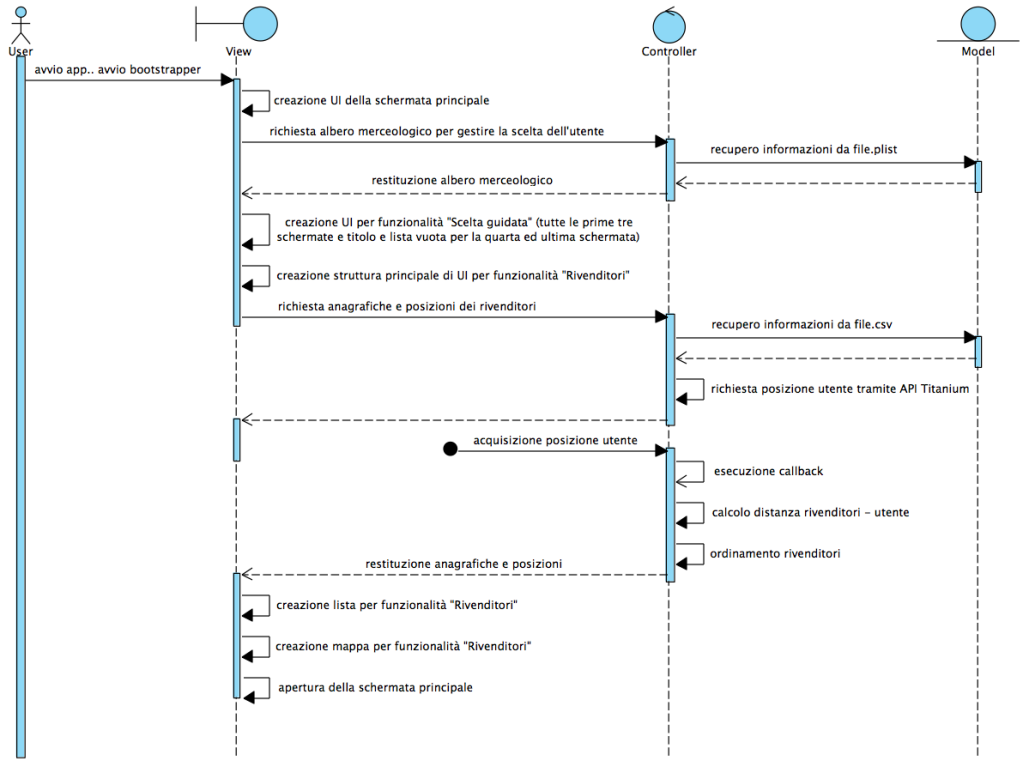

Figura 3.8: Interazione tra Model, View e Control del progetto MetropolisTitan

Le dinamiche delle interazioni nel tempo tra le varie tipologie di oggetti in merito alla fase di bootstrap, ovvero a ciò che dovrebbe accadere all'avvio dell'app da parte dell'utente, sono mostrate in figura 3.8, e sono state pen-

sate dopo aver dato un breve sguardo alla API di Titanium da sfruttare per il recupero della posizione dell'utente.

3.2.3 Implementazione

In questa fase viene messa in luce la concreta realizzazione della soluzione, traducendo in un particolare linguaggio di programmazione le decisioni prese in sede di progettazione, spesso coinvolgendo anche tecnologie diverse (database, etc.). Tipicamente questa attività viene effettuata sfruttando un determinato ambiente di sviluppo IDE.
Quindi scelto Titanium SDK come strumento di sviluppo, l'implementazione dell'app/prototipo è stata realizzata tramite i tools che offre, raccolti nell'IDE ufficiale Titanium Studio scaricabile gratuitamente dal sito ufficiale. Inoltre, data l'architettura di Titanium, è stato necessario da subito installare l'Android SDK e l'iOS SDK.
Questa fase è stata affiancata e supportata dai frequenti risultati della fase di test, descritta nel paragrafo successivo.
Il prodotto finale dell'implementazione sarà un prototipo di app denominato MetropolisTitan.

Creazione del progetto

Per creare un progetto mobile in Titanium Studio occorre inserire e scegliere alcune meta-informazioni:

- Deployment Targets, ovvero le piattaforme di distribuzione;
- Project name, cioè il nome con il quale l'app verrà visualizzata dagli utenti;
- App Id, che rappresenta l'Application Package Name per Android (l'identificatore univoco utilizzato da Android per gestire le app installate) e il CF Bundle Identifier (la stringa di identificazione che specifica il tipo di pacchetto) per iPhone;
- Project Template, cioè la tipologia di template (facoltativo).

Come tipologia di template e quindi struttura fondamentale di partenza, ho deciso di adottare il template predefinito *Single Window Application*, perché consiste in una singola finestra principale vuota con il ruolo di top-level container alla quale possono essere aggiunti componenti View (schermate).
Avrei potuto sfruttare il template predefinito Tabbed Application, punto di

partenza per applicazioni basate sui Tabs, ma esso utilizza come componenti di UI Tabs e TabGroup, che non avrebbero permesso all'app di soddisfare appieno i requisiti. Infatti per la creazione e la gestione di Tabs, Titanium mette a disposizione un'API unica valida sia per iOS che per Android, come per la maggior parte dei componenti che fornisce. Questa API in particolare però viene interpretata dalle piattaforme secondo i rispettivi principi di UX e look&feel, quindi in iOS i Tabs vengono posti nella parte bassa del display, mentre in Android nella parte alta del display. Considerando che secondo i requisiti il prototipo deve avere la stessa UI dell'app da replicare, e che questa mostra i Tabs nella parte bassa dello schermo in entrambe le piattaforme, ho scelto di implementarli in modo personalizzato, come vedremo in seguito.
Quindi alla creazione del progetto Titanium Studio ha generato la directory "root" del progetto contenente un file di configurazione, localization files, e una directory per contenere immagini, assets, e sorgenti JavaScript. I file della struttura predefinita del progetto che hanno maggiore importanza sono la directory "Resources", in cui mantenere sorgenti e grafica del progetto, il bootstrap file *"app.js"*, file di avvio caricato per primo quando l'app viene lanciata, e *"tiapp.xml"*, contenente informazioni di configurazione.
Nonostante Titanium metta a disposizione alcune API specifiche per ogni piattaforma da esso supportata, la maggior parte di esse è comune a tutte e due le piattaforme (Android, iPhone) offrendo quindi una stessa interfaccia consistente per entrambe, perciò ho deciso di sfruttare quest'ultime il più possibile, per massimizzare la riusabilità del codice e ridurre al minimo l'implementazione specifica per ogni piattaforma. Ora vedremo l'implementazione dando particolare attenzione a come sono state gestite alcune problematiche dovute alle differenze di funzionamento, e di visualizzazione tra Android e iOS.

Gestione risoluzione degli schermi

Per gestire al meglio le differenti risoluzioni degli schermi, Titanium ha introdotto un nuovo sistema di layout e un concetto chiamato *Universal Unit Support*, che permettono allo sviluppatore di specificare le unità per valori di dimensione e posizione. Quindi affinchè il layout di ogni componente di UI utilizzato, ed unico per entrambe le piattaforme, supporti tutte le risoluzioni dei device Android e iOS, ho modificato l'unità di default del

sistema[24], che altrimenti sarebbe il pixel in Android, e il *DIP* (density-independent pixel) in iOS, che è equivalente agli "Apple points". Infatti l'unità di default del sistema in Titanium è l'unità di misura sullo schermo del dispositivo, o la misura relativa alle informazioni dal parent View (componente contenitore) per alcune tipologie di componenti di UI. Il modo più semplice per raggiungere questo obiettivo è stato quello di utilizzare il DIP per entrambe le piattaforme, impostandolo tramite la modifica della proprietà della seguente linea di codice, presente nel file *"tiapp.xml"*:

```
<property name="ti.ui.defaultunit" type="string">
    dip
</property>
```

Nonostante questo però è stato necessario tenere presente che l'interpretazione del DIP varia a seconda della piattaforma[24], come è possibile vedere in figura 3.9

Interpretazione di 1 DIP (density-independent pixel) nelle piattaforme	
Android	1 pixel su un display 160DPI px=dip*(screen density)/160
iOS su iPhone senza display Retina	1 pixel su un display 163DPI px=dip*(screen density)/163
iOS su iPhone con display Retina	2 pixel di larghezza o altezza

DPI = dots per inch

Figura 3.9: Interpretazioni del DIP

Per questo motivo ho cercato di inserire soprattutto impostazioni relative, configurando proprietà come *width, left, right, height, top, bottom*, di alcuni componenti di UI come percentuali delle dimensioni dal componente contenitore.

Gestione "splash screen"

Quando viene mostrato lo "splash screen" (immagine che deve comparire per qualche secondo all'avvio dell'app) in Android occorre nascondere la

TitleBar, mentre per iOS essa non compare per default. Android consente di impostare l'aspetto dell'app utilizzando dei temi, per specificare colori predefiniti, font e immagini, per un'attività Android o un'intera app, perciò è stato necessario creare le directory "platform/android/res/values" ed inserirvi all'interno il file *"theme.xml"* con il seguente contenuto:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<style name   ="Theme.Titanium"
       parent ="android:Theme.NoTitleBar">
    <item name="android:windowBackground">
        @drawable/background
    </item>
</style>
</resources>
```

In questo modo l'app utilizza il tema predefinito offerto da Titanium, che definisce lo "splash screen" all'avvio dell'app e per il resto rimanda al tema predefinito del dispositivo in cui viene messa in esecuzione, nascondendo però la TitleBar durante la visualizzazione dell'immagine.

Gestione schermata principale

L'implementazione della maggior parte delle UI e l'attività d'integrazione dei vari moduli di UI con ruolo di View nel pattern MVC è stata realizzata all'interno del file di bootstrap *"app.js"*. In questo file vengono definite le variabili globali, viene creata la finestra principale dell'app, componente top-level dell'app di tipo Window, alla quale vengono aggiunti i componenti di UI della schermata principale, e tutti i componenti di tipo View che conterranno le schermate delle funzionalità da implementare. Ho realizzato la UI della schermata principale tramite l'utilizzo di due container, uno per racchiudere le tre immagini, e uno contenente i "Tabs" che servono per navigare l'app scegliendone le funzionalità, in questo modo è stato più semplice configurare e gestirne il layout. In particolare nelle prime due righe del codice seguente vengono richiamati i moduli che implementano i costruttori di due oggetti creati subito dopo, di tipo *ImageView* e *ContainerView*, che estendono l'oggetto di UI di tipo View predefinito di Titanium, specializzandolo in due configurazioni diverse e ricorrenti, per ridurre le linee di codice riutilizzandolo maggiormente.

```
var ContainerView = require('ContainerView');
```

```
var ImageView = require('ImageView');

var im = new ContainerView();
im.height = Ti.UI.FILL;
im.bottom = '10%';

var colors = new ImageView();
colors.backgroundImage =
    'images/line_colors_portrait_iphone.jpg';

var urbanEff = new ImageView();
urbanEff.backgroundImage =
    'images/line_urbaneffects_portrait_iphone.jpg';

var resins = new ImageView();
resins.backgroundImage =
    'images/line_resins_portrait_iphone.jpg';

resins.height = '33.34%';

im.add(colors);
im.add(urbanEff);
im.add(resins);
```

Come accennato in precedenza, avendo scelto di sfruttare il template Single Window Application anzichè quello Tabbed Application, per rispettare i requisiti dell'analisi, ho implementato i "Tabs" in modo personalizzato, sfruttando quattro Buttons e una *ContainerView* per contenerli. In questo modo ho utilizzato componenti di Titanium che sono supportati allo stesso modo da entrambe le piattaforme, senza ricorrere a implementazioni platform-specific, nè alle API specifiche di Titanium, Tab e TabGroup (contenitore dei Tabs).
Quindi ho impostato la *ContainerView* in modo che mostri i Buttons nella parte bassa del display, e ho gestito l'evento *click* sui Buttons in modo che essi funzionino come se fossero dei "Tabs", ognuno rendendo visibile solo la View o il primo elemento dell'array di View ad esso correlato, tramite il metodo *setVisible()*. Credo che l'utilizzo di questo metodo permetta di ottimizzare le performance con le quali l'app offre il passaggio da una funzionalità all'altra durante la navigazione dell'utente, riducendo il carico di lavoro del processore, dato che le View appartengono già alla Window quando l'utente preme uno dei Buttons.

Quanto appena descritto è stato implementato nel codice seguente.

```
var tabC = new ContainerView();
tabC.layout = 'horizontal';
tabC.height = '10%';
tabC.bottom = 0;

var b = [];
for (var i=0; i<4; i++){
    b[i] = Titanium.UI.createButton({
        left: 0,
        layout: 'vertical',
        height: Ti.UI.FILL,
        width: '25%',
        backgroundColor: 'black',
        font: {fontSize:10,fontFamily:'Helvetica'},
        textAlign: 'center',
        color: 'white',
    });
}

b[0].title = 'Scelta guidata';
b[1].title = 'Calcola';
b[2].title = 'Video';
b[3].title = 'Rivenditori';

for (var i=0; i<4; i++){
    tabC.add(b[i]);
}

b[0].addEventListener('click', function(e){
    svT0[0].setVisible(true);
    svT0[1].setVisible(false);
    svT0[2].setVisible(false);
    svT0[3].setVisible(false);
    svT1.setVisible(false);
    svT2.setVisible(false);
    svT3.setVisible(false);
    im.setVisible(false);
});

b[1].addEventListener('click', function(e){
    svT0[0].setVisible(false);
    svT0[1].setVisible(false);
    svT0[2].setVisible(false);
```

```
    svT0[3].setVisible(false);
    svT1.setVisible(true);
    svT2.setVisible(false);
    svT3.setVisible(false);
    im.setVisible(false);
});

b[2].addEventListener('click', function(e){
    svT0[0].setVisible(false);
    svT0[1].setVisible(false);
    svT0[2].setVisible(false);
    svT0[3].setVisible(false);
    svT1.setVisible(false);
    svT2.setVisible(true);
    svT3.setVisible(false);
    im.setVisible(false);
});

b[3].addEventListener('click', function(e){
    svT3.setVisible(true);
    svT0[0].setVisible(false);
    svT0[1].setVisible(false);
    svT0[2].setVisible(false);
    svT0[3].setVisible(false);
    svT1.setVisible(false);
    svT2.setVisible(false);
    im.setVisible(false);
});
```

Codice platform-specific

L'unica porzione di codice platform-specific necessaria per l'intero prototipo riguarda la configurazione e l'apertura della Window di cui abbiamo già accennato, e le configurazioni di posizione dei componenti direttamente aggiunti ad essa. Infatti è stato necessario considerare che iOS e Android presentano una particolare differenza nella gestione della Window tramite Titanium.
iOS considera la *StatusBar*, ovvero la barra che mostra l'indicatore di carica della batteria, l'orario, etc., inglobata nella Window, mentre Android dedica alla Window lo spazio del display escluso quello occupato dalla StatusBar. Quindi per mantenere la StatusBar sempre visibile e fare in modo che i com-

ponenti direttamente aggiunti alla Window occupino lo spazio disponibile tra la StatusBar e il contenitore di "Tabs", ho settato la proprietà *top* per ogni elemento aggiunto direttamente alla Window che deve essere mostrato subito sotto la StatusBar, cioè del componente che contiene le immagini della schermata principale, e di tutte le View utilizzate per implementare le funzionalità. Il valore di questa proprietà specifica dove il componente aggiunto alla Window deve essere posizionato rispetto ad essa, in particolare la posizione superiore del componente aggiunto. Nello specifico 20dp corrispondono a 20 Apple points, l'altezza esatta della StatusBar in iOS.
Altre caratteristiche presenti in entrambe le versioni native dell'app Metropolis, e quindi da replicare secondo i requisiti, sono la StatusBar di colore bianco e l'effetto dissolvenza quando appare la schermata principale e scompare lo splash screen. Queste configurazioni fanno parte delle impostazioni di default per l'esecuzione dell'app in Android, mentre per soddisfare questi requisiti in iOS ho dovuto impostare due particolari specifiche. Tutti i problemi platform-specific illustrati in precedenza sono stati gestiti, tramite JavaScript anzichè codice nativo, sfruttando il *feature detection* tramite l'API di Titanium nel seguente modo.

```
var osname = Ti.Platform.osname;
if(osname === 'iphone'){
   im.top = '20dp';
   for (var i=0; i<4; i++){
     svT0[i].top = '20dp';
   }
   svT1.top = '20dp';
   svT2.top = '20dp';
   svT3.top = '20dp';
   win.setStatusBarStyle(
     Titanium.UI.iPhone.StatusBar.LIGHT_CONTENT
   );
   win.modal = true;
   win.open({
     modalTransitionStyle:
     Titanium.UI.iPhone.MODAL_TRANSITION_STYLE_CROSS_DISSOLVE

   });
}else if (osname === 'android'){
   im.top = 0;
   for (var i=0; i<4; i++){
     svT0[i].top = 0;
   }
```

```
    svT1.top = 0;
    svT2.top = 0;
    svT3.top = 0;
    win.open();
}
```

Vediamo ora l'implementazione dei singoli moduli che costituiscono il sistema, seguendo l'organizzazione e la suddivisione dei compiti delineati nella fase di analisi.

Funzionalità "Scelta guidata"

Nel file *"app.js"* viene affidata la creazione della UI della funzionalità "Scelta guidata" all'oggetto di tipo *Scelta_guidata*, come è mostrato nelle linee di codice seguenti.

```
var svT0 = [];
var Scelta_guidata = require('Scelta_guidata');
var svT0 = new Scelta_guidata();
```

L'implementazione di questa funzionalità è stata suddivisa tra i due moduli seguenti.
Nel file *"Controller_scelta.js"* viene implementato il recupero dei dati dal file *"personalChoiceMap.plist"*, sfruttando un oggetto di tipo *Plist_ parser* implementato in un modulo precedentemente sviluppato da terzi, ed importato nel progetto, che restituisce l'albero merceologico sotto forma di oggetto navigabile in formato JSON.

```
function Controller_scelta(rivenditori, propName) {
    var Plist_parser = require('Controller/Plist_parser');
    var PlParser = new Plist_parser();
    var fObj = Ti.Filesystem.getFile(
        Ti.Filesystem.resourcesDirectory,
        'Model/personalChoiceMap.plist'
    );
    var xmlStr = fObj.read().text;
    var myObj = PlParser.parse(xmlStr);
    return myObj;
}
module.exports = Controller_scelta;
```

Il file *"Scelta_guidata.js"* implementa il costruttore della rispettiva tipologia di oggetto, quindi per prima cosa vengono richieste al controller specifico

della funzionalità le informazioni che serviranno per gestire i risultati da esporre all'utente in base alle sue scelte, ovvero l'albero merceologico, come è mostrato nel seguente stralcio di codice.

```
var Controller_scelta = require(
    'Controller/Controller_scelta'
);
var myObj = new Controller_scelta();
```

Inoltre sempre in questo file vengono creati staticamente Buttons, Labels e una TableView, cioè una lista vuota. L'evento *click* su uno dei Buttons viene gestito dal rispettivo listener che mantiene la scelta del particolare Button in una delle tre variabili denominate *scelta1, scelta2, scelta3*. In particolare *scelta1* contiene il tipo di prodotto scelto, *scelta2* contiene il tipo di colore scelto e *scelta3* contiene la tipologia di ambiente scelto, e vengono sovrascritte ognivolta che l'utente effettua nuove scelte.
Sempre in questo file vengono implementate le funzioni *crea_elementi_lista()* e *aggiorna_lista()*, invocate dai listeners dei Buttons che ricevono l'ultima scelta dell'utente, per gestire la lista risultante dalle sue tre scelte.
La funzione *aggiorna_lista()* aggiorna la lista dinamicamente, semplicemente settando sezioni e righe risultanti dall'invocazione di *crea_elementi_lista()*, come si può vedere nel seguente stralcio di codice.

```
tableView.setData(crea_elementi_lista());
```

La funzione *crea_elementi_lista()* crea dinamicamente le sezioni e le righe da mostrare nella lista, navigando l'oggetto JSON in base alle tre scelte effettuate dall'utente, e gestendo le informazioni sugli elementi risultanti tramite il ciclo implementato nel seguente stralcio di codice.

```
for (
  var i=0;
  i < Object.keys(myObj[scelta1][scelta2][scelta3]).length;
  i++
){
    str[i] = myObj[scelta1][scelta2][scelta3][i];
    if (str[i].charAt(0) == 'c'){
        nC++;
        if (nC == 1){
            row = Ti.UI.createTableViewRow({
                className: 'sez', //migliora le performance
                backgroundColor: '#333333',
                color: 'white',
```

```
                title: 'Colors',
                height: 25,
                width: Titanium.UI.FILL,
            });
            tableData.push(row);
        }
    }else if (str[i].charAt(0) == 'u') {
        nU++;
        if (nU == 1){
            row = Ti.UI.createTableViewRow({
                className: 'sez', //migliora le performance
                backgroundColor: '#333333',
                color: 'white',
                title: 'Urban Effects',
                height: 25,
                width: Titanium.UI.FILL
            });
            tableData.push(row);
        }
    }else {
        nR++;
        if (nR == 1){
            row = Ti.UI.createTableViewRow({
                className: 'sez', //migliora le performance
                backgroundColor: '#333333',
                color: 'white',
                title: 'Resins',
                height: 25,
                width: Titanium.UI.FILL
            });
            tableData.push(row);
        }
    }
    elem[i] = str[i].substr(2, (str[i].length)-2);

    row = Ti.UI.createTableViewRow({
        className:'scG', //migliora le performance
        backgroundColor: '#4B4B4B',
        rowIndex:i,
        height: 100,
        layout: 'horizontal'
    });

    image = Ti.UI.createImageView({
        image: 'images/'+elem[i]+'.jpg',
```

```
        left: 0,
        top: 0,
        width: 137,
        height: Titanium.UI.FILL
    });
    row.add(image);

    l = Ti.UI.createLabel({
        backgroundColor: '#4B4B4B',
        color:'white',
        text: elem[i],
        left: 5,
        top: 0,
        width: Titanium.UI.SIZE,
        height: Titanium.UI.SIZE
    });
    row.add(l);
    tableData.push(row);
}
```

L'array *tableData*, utilizzato per contenere le righe e le sezioni risultanti dalla scelta dell'utente, viene svuotato e riempito ogni volta che l'utente ha effettuato tre nuove scelte.

Funzionalità "Rivenditori"

Per implementare la UI della funzionalità "Rivenditori" ho deciso di creare nel file *"app.js"* un'unica View principale contenente un'ulteriore View, che funge da "TopBar" e a sua volta contiene una SearchBar e due Buttons. La scelta è stata condizionata dalla necessità di facilitare la realizzazione delle due viste, che in parte mostrano gli stessi dati sotto due forme diverse (lista e mappa), e che condividono la stessa "TopBar", permettendomi in questo modo di riutilizzare il codice.
I due Buttons implementano i componenti che l'utente deve utilizzare per poter vedere la lista o la mappa cercando di farli apparire e funzionare il più possibile come quelli dell'app originale, per la quale è stato utilizzato il componente SegmentedControl in iOS e una versione personalizzata dei componenti RadioGroup e RadioButton in Android. Purtroppo Titanium non offriva nessun componente completo e già adatto allo scopo che fosse unico e consistente per entrambe le piattaforme, ma solo il componente ButtonBar supportato da iOS. Quindi ho sfruttato due Buttons per comporre

lo stesso componente attraverso un'unica implementazione per entrambe le pittaforme, anzichè diversificata. I listeners dei Button gestiscono l'aggiunta alla View principale, o la rimozione da essa, delle informazioni sui rivenditori sottoforma di lista o di mappa. Alla fine del codice seguente dopo aver dichiarato quattro variabili globali, viene richiesto il modulo nonché controller di questa funzionalità e creato il rispettivo oggetto, il cui costruttore è implementato nel file *"Controller_rivenditori.js"*.

```
var svT3 = new View();
svT3.backgroundColor = 'black';

var topBar = Ti.UI.createView({
    height: Ti.UI.SIZE,
    width: Ti.UI.SIZE,
    top: 0,
    backgroundColor: 'black',
    layout: 'horizontal',
    horizontalWrap: false,
    visible: true
});

var bBar = [];
for (var i=0; i<2; i++){
    bBar[i] = Titanium.UI.createButton({
        height: 36,
        width: 36,
        borderRadius: 5,
        borderColor: 'darkgray',
        backgroundColor: '#333333'
    });
}

bBar[0].setImage('images/icon_list_iphone@2x.png');
bBar[0].left = 6;
bBar[1].left = 0;
bBar[1].setImage('images/icon_mark_iphone@2x.png');

bBar[0].addEventListener('click', function(e){
    if(tableVRadded == false){
        svT3.add(tableViewR);
        svT3.remove(mapView);
        tableVRadded = true;
        mapVadded = false;
    }
```

```
});

bBar[1].addEventListener('click', function(e){
    if(mapVadded == false){
        svT3.add(mapView);
        svT3.remove(tableViewR);
        tableVRadded = false;
        mapVadded = true;
    }
});

var search = Titanium.UI.createSearchBar({
    barColor: '#000',
    showCancel: false,
    height: '40',
    width: Ti.UI.FILL,
    left: 2
});

search.addEventListener('return', function(e){
    search.blur();
});

topBar.add(bBar[0]);
topBar.add(bBar[1]);
topBar.add(search);
svT3.add(topBar);

var tableViewR;
var mapView;
var tableVRadded;
var mapVadded;
var Controller_rivenditori = require(
    'Controller/Controller_rivenditori'
);
var controller_ri = new Controller_rivenditori();
```

L'implementazione prosegue nei tre moduli che andremo a descrivere, percorrendo le fasi seguenti:

- recupero delle informazioni sui rivenditori;
- recupero della posizione corrente dell'utente;

- calcolo della distanza in km tra la posizione corrente dell'utente e le posizioni dei rivenditori;
- ordinamento dei rivenditori a partire dal più vicino all'utente;
- creazione della lista;
- creazione della mappa.

Il controller della funzionalità per prima cosa recupera i dati dal file *"retailers.csv"*, sfruttando l'oggetto *Csv_parser* implementato in un modulo precedentemente sviluppato da terzi, ed importato nel progetto, che restituisce un array di array. Successivamente viene creato un array di oggetti di tipo *Rivenditore*, aggiungendo loro dinamicamente le proprietà *nome, indirizzo, provincia, nazione, telefono, fax, email, sitoweb, latitudine, longitudine, tipo*, come esposto nello stralcio di codice che segue.

```
//definisco la "classe" Rivenditore
function Rivenditore(){};
for (var j=0; j<resultCsv[0].length; j++){
    propName[j] = resultCsv[0][j].toLowerCase().replace(
        ' ',''
    );
}
for (var i=1; i<resultCsv.length-1; i++){
    //istanzio un nuovo oggetto
    rivenditori[i-1] = new Rivenditore();
    for (var j=0; j<propName.length; j++){
        rivenditori[i-1][propName[j]] = resultCsv[i][j];
    }
}
```

Titanium mette a disposizione due modalità per ottenere informazioni sulla posizione dell'utente, entrambe valide sia per Android che per iOS:

- fare una richiesta una tantum con il metodo *getCurrentPosition()*;
- registrarsi per ricevere aggiornamenti sulla posizione tramite un listener dell'evento *location*, attivo finchè non viene rimosso.

Sapendo che l'utilizzo di servizi di localizzazione può avere un impatto significativo sulla durata della batteria del dispositivo, e che il consumo di

energia è fortemente influenzato dalla precisione e dalla frequenza degli aggiornamenti della posizione richiesti dall'app, è importante usarli nel modo più efficiente possibile. Per questo motivo e considerando lo scopo per il quale l'app necessita di conoscere la posizione dell'utente, ho ritenuto non essenziale il monitoraggio continuo, ed ho scelto di utilizzare *getCurrentPosition()*, metodo che recupera l'ultima posizione conosciuta dal dispositivo e la passa come primo parametro alla funzione di callback. Questo metodo viene gestito in modo leggermente diverso dalle piattaforme sottostanti, infatti in Android viene utilizzata una posizione memorizzata nella cache del device, senza bisogno di attivare il WiFi, mentre in iOS il WiFi può essere utilizzato se la posizione è considerata troppo "vecchia"[24].

```
Ti.Geolocation.getCurrentPosition(function callback(e){
    if (!e.success || e.error){
        var currentLocErr='error:'+JSON.stringify(e.error);
        Ti.API.info("Code translation:"+translateErrorCode(
            e.code
        ));
        alert('error'+JSON.stringify(e.error));
        return;
    }
    lat = e.coords.latitude;
    lon = e.coords.longitude;
    usa_coordinate(lat,lon);
});
```

Come è possibile vedere dallo stralcio di codice precedente, non appena viene recuperata la posizione dell'utente viene eseguita la callback che la assegna alle variabili *lat* e *lon*, e invoca la funzione *usa_coordinate()*, implementata sempre in questo file. All'interno di questa funzione viene affidato il calcolo della distanza tra l'utente e i rivenditori all'oggetto *Distance_calculator* implementato nel rispettivo modulo esterno, precedentemente realizzato da terzi e inglobato nel progetto, e successivamente i rivenditori vengono ordinati a partire dal più vicino all'utente.

```
function usa_coordinate(lat, lon){
   for (var i=0; i<rivenditori.length; i++){
      rivenditori[i].km = Distance_calculator.distance(
         rivenditori[i][propName[8]],
         rivenditori[i][propName[9]],
         lat,
         lon,
```

```
            'K'
        );
    }
    rivenditori.sort(function(a,b){
        return a.km - b.km;
    });
    var Lista = require('Lista');
    Lista.crea_lista(rivenditori, propName);
    var Mappa = require('Mappa');
    Mappa.crea_mappa(lat, lon, rivenditori, propName);
}
```

Infine viene affidata la creazione e l'aggiunta alla View di lista e mappa dei rivenditori a due oggetti, uno di tipo *Lista* e l'altro di tipo *Mappa*, i cui costruttori sono implementati nei rispettivi moduli. In particolare la mappa vera e propria, identificata dalla variabile *mapView*, è stata integrata nel progetto sfruttando il modulo built-in gratuito di Titanium denominato *ti.map*. Questo modulo aggiuntivo, che permette di usare le Google Maps v2 nei device Android e l'iOS Map Kit framework nei device iPhone, non è supportato su emulatore Android. Per abilitarne l'utilizzo è stato necessario ottenere la Google Maps API key per il progetto, inserendo nella Google Console il certificato digitale SHA-1 automaticamente generato da Titanium alla creazione del progetto, e aggiungere le seguenti righe di codice nel file *"tiapp.xml"*.

```
<module platform="android">ti.map</module>
<module platform="iphone">ti.map</module>
```

```
<uses-permission
  android:name="android.permission.INTERNET"/>
<uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission
  android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission
  android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission
  android:name="com.google.android.providers.gsf.permission.
                READ_GSERVICES"/>
<uses-feature
  android:glEsVersion="0x00020000"
  android:required="true"/>
<uses-permission
```

```
  android:name="com.mintlab.metropolis.titan.permission.
                MAPS_RECEIVE"/>
<permission
  android:name="com.mintlab.metropolis.titan.permission.
                MAPS_RECEIVE"
  android:protectionLevel="signature"/>
<application>
  <meta-data
   android:name="com.google.android.maps.v2.API_KEY"
   android:value="AIzaSyAnAPCh4mQt7WaTymnqbXEbK03VMkt2ApI"/>
</application>
```

Il secondo stralcio di codice appena mostrato riguarda solo la piattaforma Android, e permette alle API di scaricare i dati dal server di Google Map, di salvarli in cache, di utilizzare il GPS per la localizzazione del dispositivo, di utilizzare il Wi-Fi o la connessione mobile per la posizione del dispositivo, di accedere ai servizi web-based di Google e per specificare OpenGL ES 2.0 come requisito.

3.2.4 Testing e debugging

L'attività di test e debug per questo progetto è stata eseguita parallelamente all'implementazione, ed è stata mirata alla verifica del raggiungimento totale o parziale dei requisiti, della correttezza dell'implementazione dei singoli moduli e del funzionamento complessivo del sistema. In particolare questa fase ha consentito di produrre feedback per migliorare l'attività d'implementazione con *semplici test manuali tramite emulatore, simulatore e device.* Nello specifico sia il debugging che il testing del prototipo sono stati effettuati su:

- device Nexus S con Android 4.4.2 (KitKat), RAM 343 MB(128MB GPU, 384MB OS), e schermo 4" hdpi;
- device Galaxy Nexus, con Android 4.3 (Jelly Bean), RAM 1GB, schermo 4.65" xhdpi;
- device Moto X con Android 4.4.2 (KitKat), RAM 2GB, schermo 4.7" xhdpi;

- desktop tramite gli ambienti di test di Android e iOS, messi a disposizione dal framework Titanium sfruttando gli SDK nativi.

È importante ricordare che emulatore/simulatore non sono comunque una rappresentazione perfetta del dispositivo mobile vero e proprio, infatti l'emulatore Android fornisce un ambiente hardware virtuale che mette in esecuzione il sistema operativo Android, i componenti della piattaforma e l'app, mentre per il simulatore iOS il codice dell'app viene cross-compilato e trasformato in un eseguibile OS X, che quindi lo mette in esecuzione simulando l'ambiente interno di un iPhone. Questi strumenti mi hanno permesso di emulare/simulare l'esecuzione del prototipo su differenti device Android e iPhone che non erano fisicamente a disposizione mia o del team di Mint. È stato possibile configurare simulatori ed emulatori scegliendo la versione dell'OS, la tipologia di device e la tipologia di schermo, e per gli emulatori Android anche il tipo di CPU, le dimensioni di RAM, di memoria interna e di VM Heap. Pertanto ho emulato device diversi, in modo da testare il prototipo con la maggior parte delle configurazioni possibili.

A seguire vedremo alcuni screenshots del prototipo MetropolisTitan in esecuzione sul *simulatore di iPhone* Retina 3.5" con iOS 7.1 (non è stato possibile testare il prototipo su iPhone non Retina non avendo a disposizione nè il device nè il rispettivo simulatore).

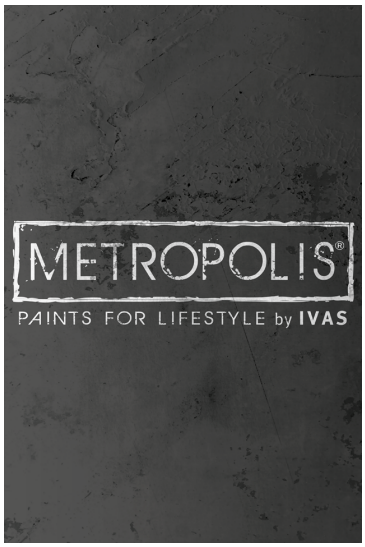

Figura 3.10: Splash screen

Figura 3.11: Schermata principale

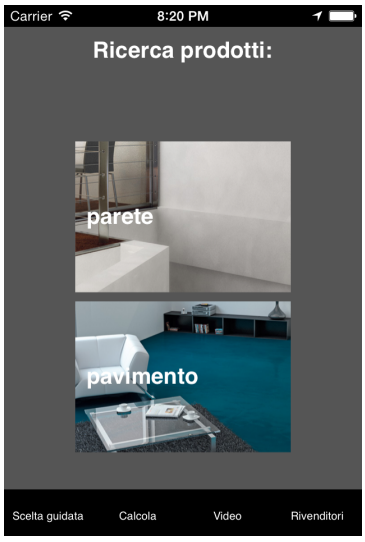

Figura 3.12: Funzionalità "Scelta guidata"

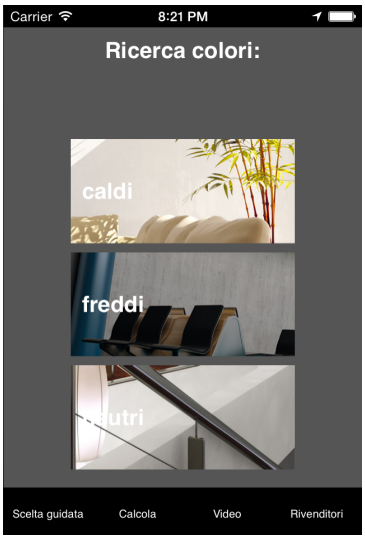

Figura 3.13: Funzionalità "Scelta guidata"

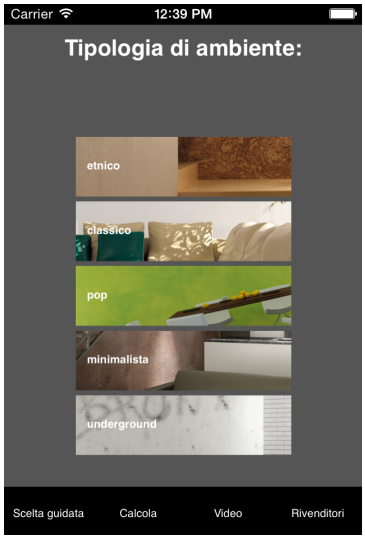

Figura 3.14: Funzionalità "Scelta guidata"

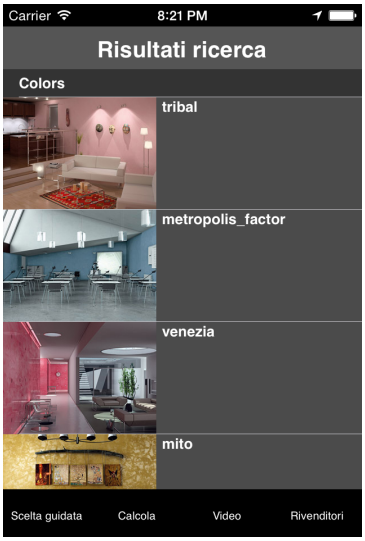

Figura 3.15: Funzionalità "Scelta guidata"

Figura 3.16: Funzionalità "Rivenditori"

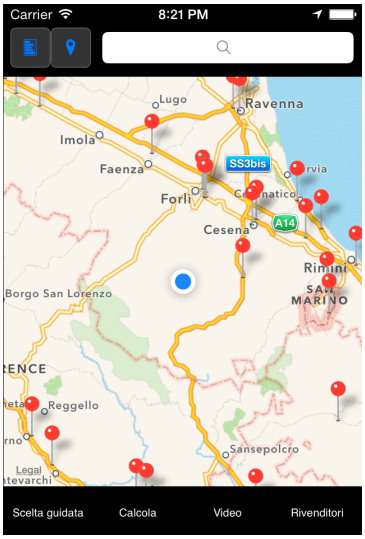

Figura 3.17: Funzionalità "Rivenditori"

A seguire vederemo gli screenshots riguardanti l'esecuzione del prototipo MetropolisTitan sull'*emulatore di Nexus One*, che presenta le seguenti caratteristiche:
Android 2.3.3, RAM 512MB, VM Heap 32MB, schermo 3.7" normal e densità hdpi (WVGA854 skin).
Ho deciso di emulare principalmente questo device in consideraizone delle statistiche dettagliate offerte dalla Google Play Developer Console[7], che in base al numero dei dispositivi attivi nell'ecosistema Android e Google Play, mostrano le percentuali di dispositivi che eseguono una determinata versione di Android, e le percentuali di quelli con una particolare configurazione dello schermo, definita dalla combinazione di dimensioni e densità. Quindi ho scelto di emulare un device che avesse una delle versioni Android maggiormente distribuite (Jelly Bean, Gingerbread, KitKat), escludendo quelle presenti nei device fisici a mia disposizione, e uno degli schermi maggiormente diffusi (normal hdpi o xhdpi). Non è stato possibile testare sull'emulatore Android la funzionalità *"Rivenditori"* perché esso non supporta la geolocalizzazione e il modulo di Titanium utilizzato per mostrare la mappa.

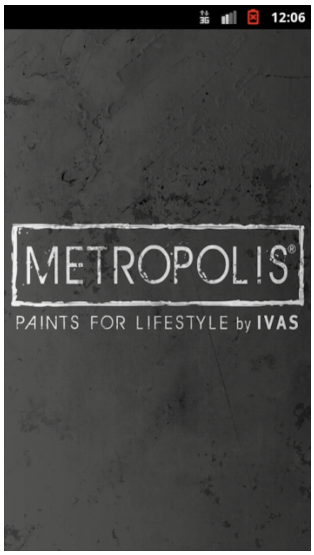

Figura 3.18: Splash screen

Figura 3.19: Schermata principale

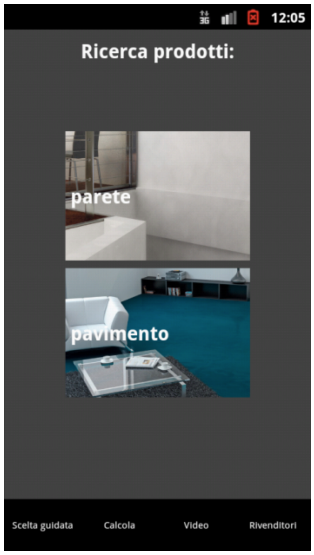

Figura 3.20: Funzionalità "Scelta guidata"

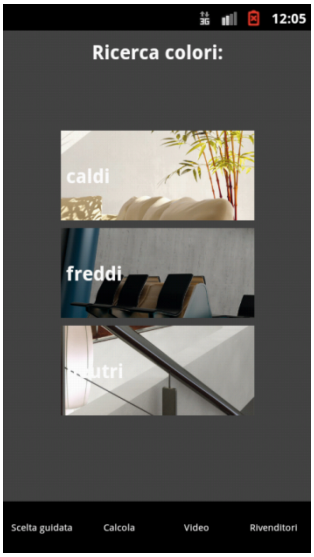

Figura 3.21: Funzionalità "Scelta guidata"

Figura 3.22: Funzionalità "Scelta guidata"

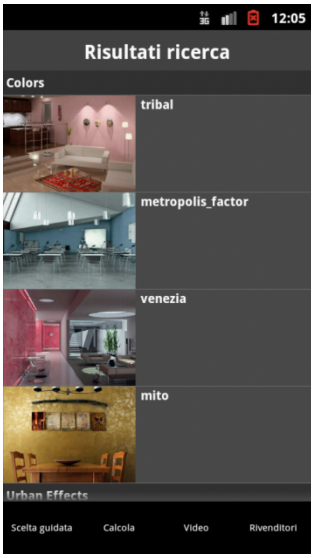

Figura 3.23: Funzionalità "Scelta guidata"

Capitolo 4

Confronto con le versioni native e considerazioni finali su Titanium

Questo capitolo analizza e valuta il prototipo MetropolisTitan sviluppato nel capitolo precedente, individuando i risultati raggiunti attraverso il confronto con le rispettive versioni native, allo scopo di evincere se l'utilizzo del framework Titanium nello sviluppo multi-platform sia stato effettivamente vantaggioso, in che modo ed eventualmente sotto quali aspetti.

4.1 Valutazione del prototipo

Al fine di valutare al meglio il prototipo MetropolisTitan, risultante dal processo di sviluppo descritto nel capitolo precedente, verrà messo a confronto con l'app Metropolis, precedentemente sviluppata dal team di Mint nelle due versioni native, una per iOS e una per Android.
Il confronto è stato effettuato sulla base dei device a disposizione con le seguenti modalità:

- valutazione delle *performance*, in termini di velocità e fluidità, confrontando Metropolis per Android con il prototipo dopo averle installate sugli stessi device fisici a mia disposizione;

- valutazione di *layout e look&feel* confrontando Metropolis nelle versioni native, con il prototipo rispettivamente installato su device Android e in esecuzione sul simulatore di iPhone.

Ora valuteremo l'esecuzione del prototipo multi-platform sulle due piattaforme di destinazione.

4.1.1 Prototipo sul simulatore iPhone

Nonostante la versione per iOS di MetropolisTitan sia stata eseguita sul simulatore, l'app risulta essere molto fluida e veloce, e senza nessun tipo di ritardo.
Inoltre come è possibile vedere dagli screenshots seguenti, layout è abbastanza conforme a quello originale, e il look&feel è veramente molto simile a quello percepito utilizzando la versione nativa, proprio perché i componenti di UI utilizzati attraverso Titanium sono effettivamente quelli offerti dall'SDK nativo di iOS.

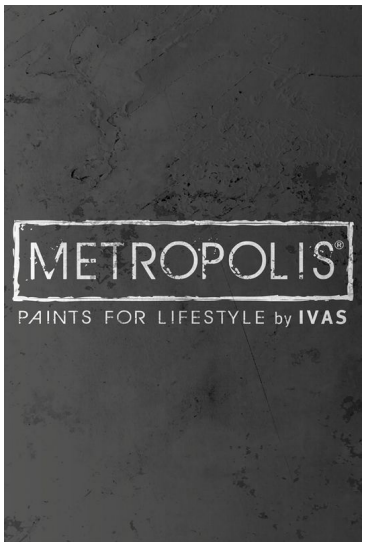

Figura 4.1: Splash screen dell'app originale su iPhone

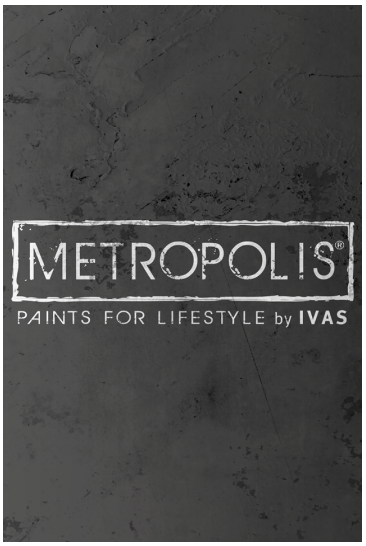

Figura 4.2: Splash screen del prototipo sul simulatore

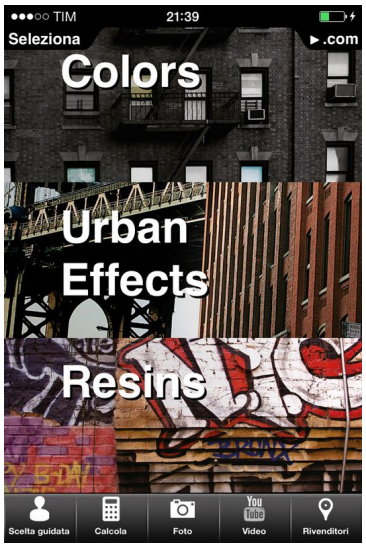

Figura 4.3: Schermata principale dell'app originale su iPhone

Figura 4.4: Schermata principale del prototipo sul simulatore

Figura 4.5: Funzionalità "Scelta guidata" dell'app originale su iPhone

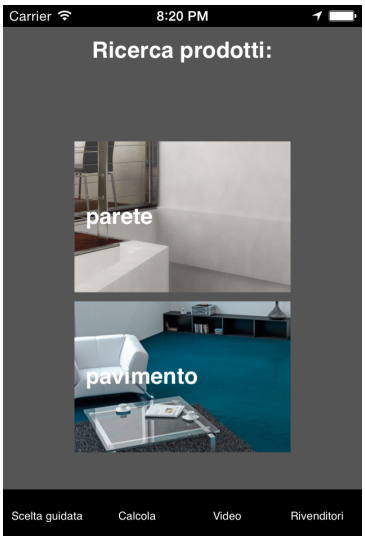

Figura 4.6: Funzionalità "Scelta guidata" del prototipo sul simulatore

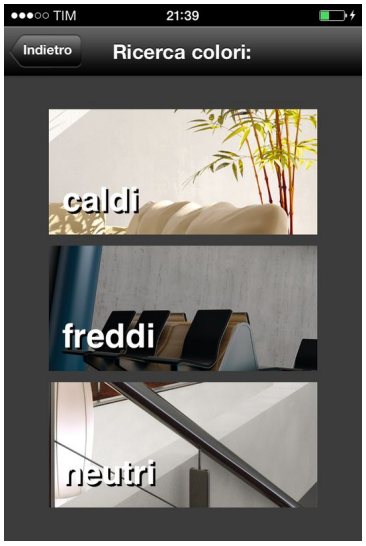

Figura 4.7: Funzionalità "Scelta guidata" dell'app originale su iPhone

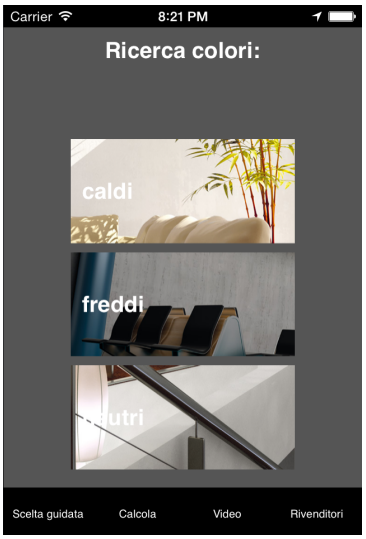

Figura 4.8: Funzionalità "Scelta guidata" del prototipo sul simulatore

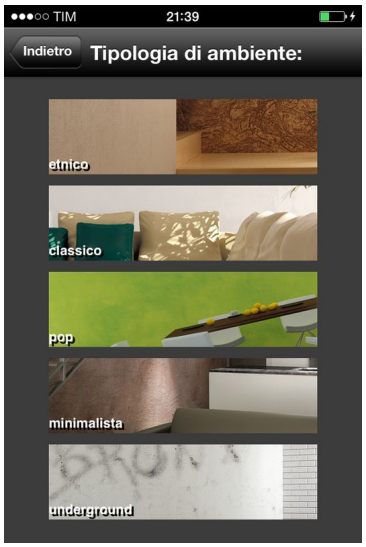

Figura 4.9: Funzionalità "Scelta guidata" dell'app originale su iPhone

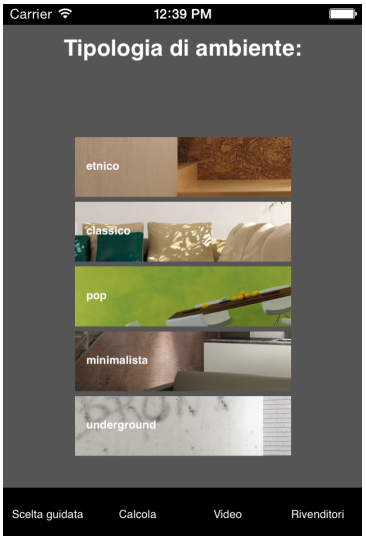

Figura 4.10: Funzionalità "Scelta guidata" del prototipo sul simulatore

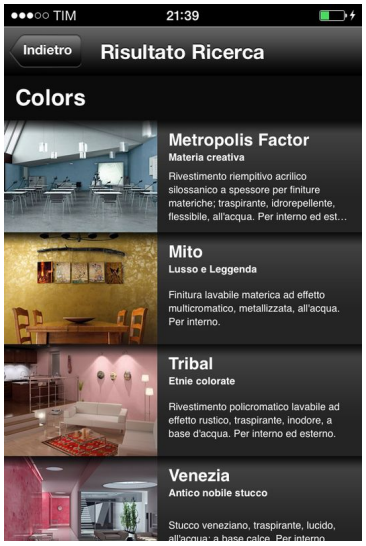

Figura 4.11: Funzionalità "Scelta guidata" dell'app originale su iPhone

Figura 4.12: Funzionalità "Scelta guidata" del prototipo sul simulatore

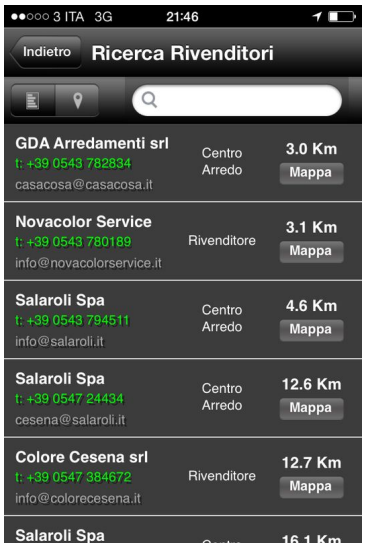

Figura 4.13: Funzionalità "Rivenditori" dell'app originale su iPhone

Figura 4.14: Funzionalità "Rivenditori" del prototipo sul simulatore

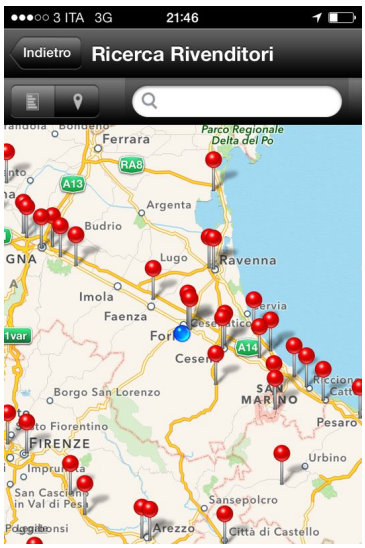

Figura 4.15: Funzionalità "Rivenditori" dell'app originale su iPhone

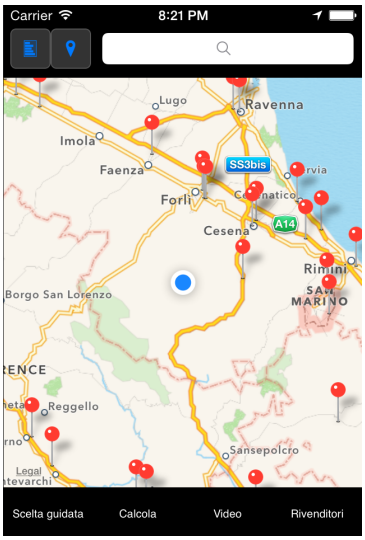

Figura 4.16: Funzionalità "Rivenditori" del prototipo sul simulatore

4.1.2 Prototipo sui device Android

Allo scopo di rendere più dettagliato il confronto tra le caratteristiche del prototipo su device Android e quelle dell'app originale nella versione per Android, ho realizzato una tabella comparativa. Il confronto è stato effettuato eseguendo le due app su Galaxy Nexus in quanto è un dispositivo con caratteristiche hardware che rientrano abbastanza nella media, e con versione Android nè troppo recente nè obsoleta.
Quindi per ogni componente di UI, utilizzato nel prototipo, ne ho descritto la funzione e valutato performance (in termini di tempistiche e fluidità), layout e look&feel, tenendo in considerazione i requisiti realmente implementati. In particolare sono stati evidenziati con caratteri rossi le tempistiche in secondi nel caso in cui la reattività del componente non è istantanea al tocco dell'utente, e per sottolineare la durata della schermata di caricamento iniziale (splash screen). A seguire sarà mostrata la tabella suddivisa in tre sezioni, la prima confronta splash screen e schermata iniziale delle due app, la seconda esamina la funzionalità "Scelta guidata" e la terza analizza la funzionalità "Rivenditori".
Successivamente saranno messi a confronto gli screenshots dell'app originale e quelli del prototipo, in esecuzione su device Galaxy Nexus.

oggetto della valutazione	componente di Titanium utilizzato	Prototipo su Galaxy Nexus			App originale su Galaxy Nexus		
		funzione	performance	layout e look&feel	funzione	performance	layout e look&feel
Splash screen	nessuno	invariata rispetto all'originale	immagine di splash screen per ≈3s	immagine di splash screen visibile per tutto il caricamento e leggermente ingrandita rispetto all'originale (StatusBar e NavigationBar sempre visibili)	schermata di caricamento	schermo nero per ≈1s e immagine di splash screen per ≈3s	prima schermo nero, poi immagine di splash screen (StatusBar e NavigationBar sempre visibili)
Top-level container	Window	contiene gli elementi della schermata principale e le View delle funzionalità	invariate rispetto all'originale (dissolvenza leggermente più veloce)	invariati rispetto all'originale	contiene gli elementi della schermata principale	compare senza ritardi e in modo fluido con effetto dissolvenza	mantiene visibile StatusBar e NavigationBar
Tab bar	View	invariata rispetto all'originale	n.a	sempre visibile e leggermente più alta rispetto all'originale	contiene quattro Buttons	n.a	visibile solo nella schermata principale
Buttons x4	Buttons	invariata rispetto all'originale	invariate rispetto all'originale	nome funzionalità senza icona	permette di accedere alle funzionalità dell'app	reagisce istantaneamente al tocco	nome funzionalità e icona
Contenitore di immagini	View	contiene tre View	n.a	non è più visibile dopo aver scelto una delle funzionalità	-	n.a	-
Immagine x3	View	contiene un'immagine	n.a	ogni immagine occupa lo stesso spazio nel display	contiene un'immagine interattiva	reagisce istantaneamente al tocco	ogni immagine ha un nome e occupa lo stesso spazio nel display
Back button	n.a	toccato una volta mostra lo splash screen, due volte permette di uscire dall'app	invariate rispetto all'originale	invariati rispetto all'originale	toccato mostra la schermata precedente, toccato nella schermata principale permette di uscire dall'app	reagisce istantaneamente al tocco	default di sistema

- = nessuno/a, l'oggetto della valutazione non è presente nell'app originale
n.a = non applicabile perchè il componente non è interattivo o l'interattività non era richiesta nei requisiti

Figura 4.17: Comparazione di splash screen e schermata iniziale

oggetto della valutazione	componente di Titanium utilizzato	Prototipo su dispositivi Android			App originale su dispositivi Android		
		funzione	performance	layout e look&feel	funzione	performance	layout e look&feel
Schermata x4	View	invariata rispetto all'originale	la prima compare un po' più velocemente dell'originale ma senza effetti, le altre compaiono come le originali	come l'originale ma mantiene visibile anche la Tab bar	le prime tre contengono una Label e dei Button, l'ultima contiene la Label e una lista	compare senza ritardi e in modo fluido (la prima compare con effetto pop)	sfondo grigio, mantiene visibile StatusBar e NavigationBar, mostra una Label al centro del lato superiore e dei Button centrali
Titolo x4	Label	invariata rispetto all'originale	n.a	come l'originale ma testo leggermente più piccolo	contiene il titolo della schermata	n.a	testo bianco su sfondo grigio
Immagine da scegliere (x2, x3, x5)	Button	invariata rispetto all'originale	invariate rispetto all'originale	immagine più piccola rispetto all'originale, con testo allineato al centro del suo lato sinistro	contiene un'immagine che selezionata mostra la schermata successiva	reagisce istantaneamente al tocco	immagine con testo allineato al suo angolo in basso a sinistra
Lista	TableView	invariata rispetto all'originale	invariate rispetto all'originale	sfondo più chiaro e senza ombreggiature	contiene sezioni, e righe risultanti dalle scelte dell'utente	permette di scorrere i suoi elementi molto velocemente, in modo fluido e senza ritardi	sfondo grigio scuro con ombreggiature tra una riga e l'altra
Sezione	TableViewRow	invariata rispetto all'originale	n.a	testo bianco e sezione leggermente più sottile dell'originale	contiene il nome di una sezione	n.a	sfondo grigio scuro con testo grigio chiaro
Riga	TableViewRow	contiene un'immagine affiancata da testo (solo nome del prodotto)	n.a	invariati rispetto all'originale ma con sfondo leggermente più chiaro	contiene un'immagine affiancata da testo, toccata mostra una nuova schermata	reagisce istantaneamente al tocco	sfondo grigio e altezza pari alla dimensione dell'immagine
Immagine prodotto	ImageView	invariata rispetto all'originale	n.a	invariati rispetto all'originale	contiene l'immagine del prodotto	n.a	allineata a sinistra nella riga
Nome prodotto	Label	invariata rispetto all'originale	n.a	testo un po' più piccolo dell'originale e sfondo leggermente più chiaro	contiene il nome del prodotto	n.a	testo bianco su sfondo grigio a destra dell'immagine

n.a = non applicabile perchè il componente non è interattivo o l'interattività non era richiesta nei requisiti

Figura 4.18: Comparazione della funzionalità "Scelta guidata"

oggetto della valutazione	componente di Titanium utilizzato	Prototipo su dispositivi Android			App originale su dispositivi Android		
		funzione	performance	layout e look&feel	funzione	performance	layout e look&feel
Schermata	View	contiene due Button, una barra di ricerca, e mostra la lista o la mappa a seconda dell'ultima scelta, e nessun avviso se non c'è connessione a Internet, accedendovi da "app recenti" quando mostra la lista mostra la tastiera	compare un po' più velocemente dell'orginale ma senza effetti	come l'originale ma mantiene visibile anche la Tab bar	contiene due Button, una barra di ricerca, e lista o mappa, inizialmente mostra sempre la lista, e una finestra pop-up se non c'è connessione a Internet	compare senza ritardi e in modo fluido con effetto pop	mantiene visibile StatusBar e NavigationBar
Top bar	View	contiene due Button e una SearchBar	n.a	barra nera sempre visibile nella schermata	-	n.a	-
Button x2	Button	invariata rispetto all'originale	quello che mostra la lista reagisce con ~5s di ritardo, quello che mostra la mappa reagisce con un ritardo <1s	più piccolo dell'originale, non cambia colore se toccato e l'icona è più piccola e non centrata	toccato mostra la lista/mappa	reagisce istantaneamente al tocco	da grigio diventa nero se toccato e possiede un'icona centrale
Barra di ricerca	SearchBar	mostra la tastiera spostando su di essa la Tab bar, ricerca n.i	reagisce istantaneamente al tocco	più sottile e senza icona	permette la ricerca in lista per città o nome del rivenditore	reagisce istantaneamente all'immissione del testo	casella di testo contenente un icona allineata al suo lato sinistro
Lista	TableView	invariata rispetto all'originale	invariate rispetto all'originale	come l'originale ma con sfondo più chiaro	contiene righe con i rivenditori ordinati dal più vicino all'utente, e in fondo i manager d'area	permette di scorrere i suoi elementi molto velocemente, in modo fluido e senza ritardi	sfondo grigio scuro separatore di riga bianco
Riga	TableViewRow	invariata rispetto all'originale	n.a	come l'originale ma il tipo è centrato, e gli elementi non sono orizzontalmente allineati	contiene informazioni su rivenditori e manager e Button "mappa"	n.a	nome, telefono, mail allineati a sinistra, tipo, km e Button "mappa" allineati a destra, tutti orizzontalmente allineati
Info rivenditori	Label	contiene info	n.a	invariati rispetto all'originale	contiene info, toccando l'indirizzo mail o il numero di telefono crea rispettivamente una mail, o imposta la chiamata	reagisce istantaneamente al tocco	nome bianco, telefono verde, mail grigia, km bianchi, tutti su sfondo grigio
Button "mappa"	Button	contiene testo	n.a	senza riquadro, testo bianco su sfondo grigio persistente	contiene testo e toccato mostra sulla mappa il rispettivo rivenditore	reagisce istantaneamente al tocco	riquadro con testo bianco, su sfondo grigio che toccato diventa nero
Mappa	Modules.Map.View	invariata rispetto all'originale	invariate rispetto all'originale	mappa (visibile anche quando l'utente non è connesso a Internet) con pin di default	mappa con i rivenditori zoomata sulla posizione dell'utente, toccando un rivenditore ne mostra nome e indirizzo in un fumetto	reagisce istantaneamente toccando un rivenditore e quando ci si sposta nella mappa o si zooma	mappa (visibile in parte anche quando l'utente non è connesso a Internet) con pin customizzati

n.i = non implementata per mancanza di tempo
- = nessuno/a, l'oggetto della valutazione non è presente nell'app originale
n.a = non applicabile perchè il componente non è interativo o l'interattività non era richiesta nei requisiti

Figura 4.19: Comparazione della funzionalità "Rivenditori"

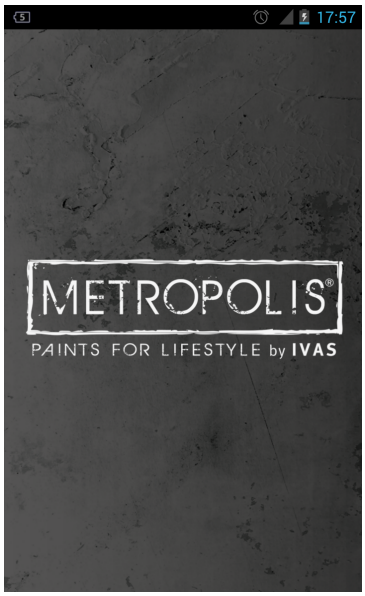

Figura 4.20: Splash screen originale

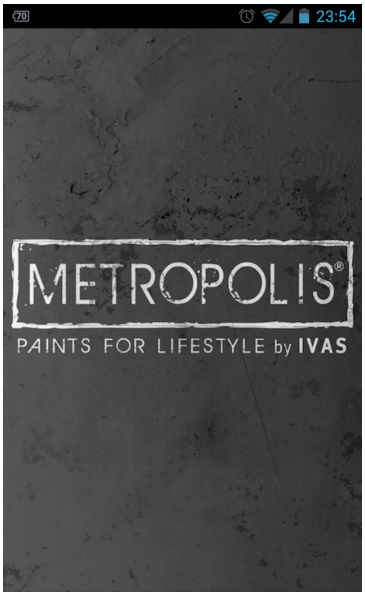

Figura 4.21: Splash screen del prototipo

Figura 4.22: Schermata principale originale

Figura 4.23: Schermata principale prototipo

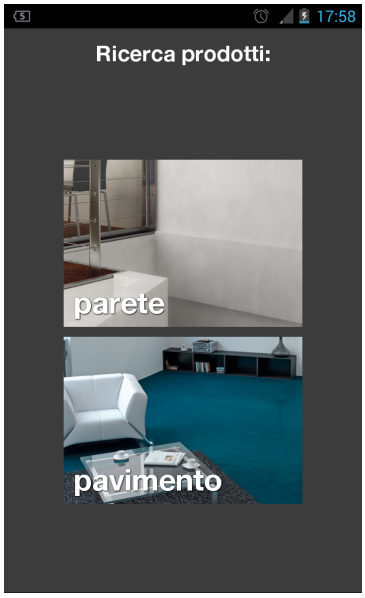

Figura 4.24: Funzionalità "Scelta guidata" originale

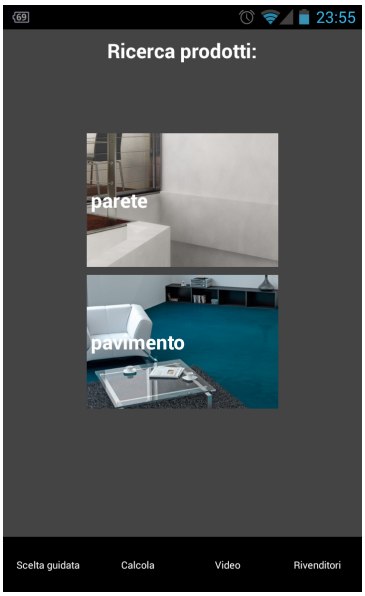

Figura 4.25: Funzionalità "Scelta guidata" prototipo

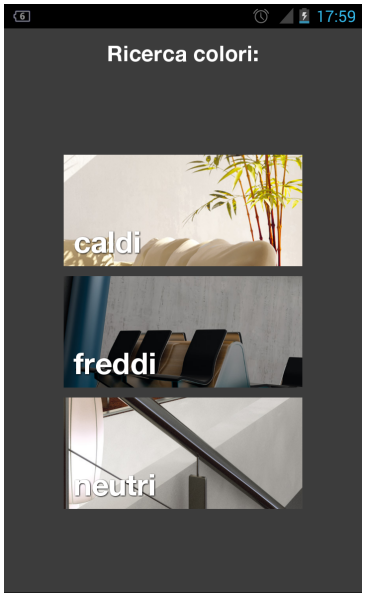

Figura 4.26: Funzionalità "Scelta guidata" originale

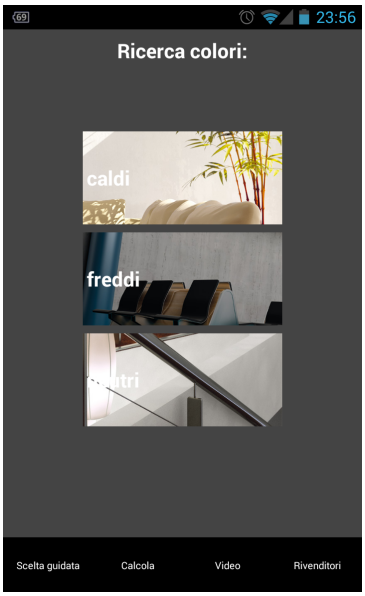

Figura 4.27: Funzionalità "Scelta guidata" prototipo

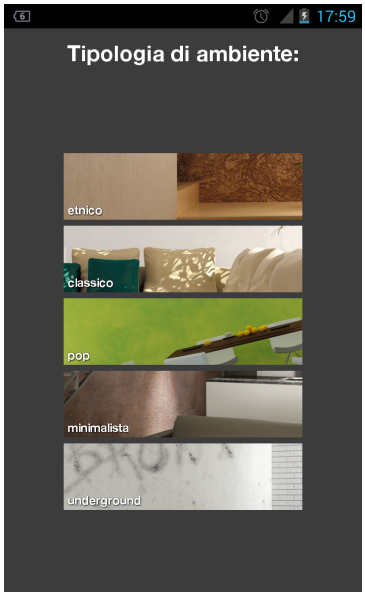

Figura 4.28: Funzionalità "Scelta guidata" originale

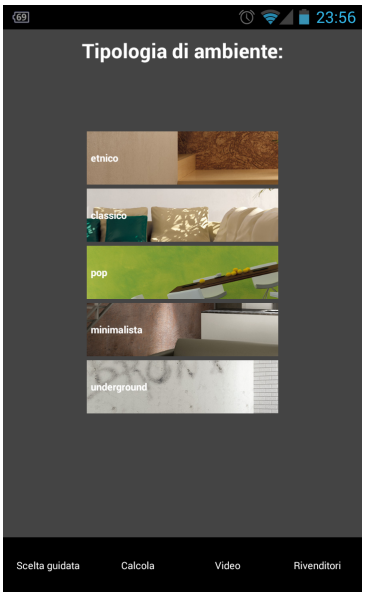

Figura 4.29: Funzionalità "Scelta guidata" prototipo

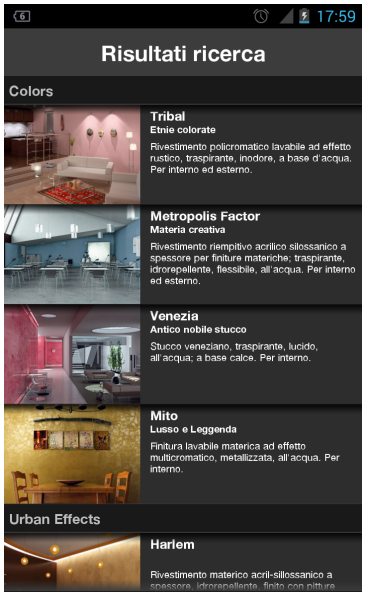

Figura 4.30: Funzionalità "Scelta guidata" originale

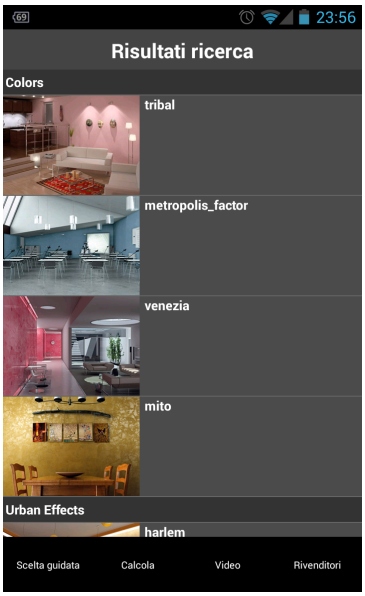

Figura 4.31: Funzionalità "Scelta guidata" prototipo

Figura 4.32: Funzionalità "Rivenditori" originale

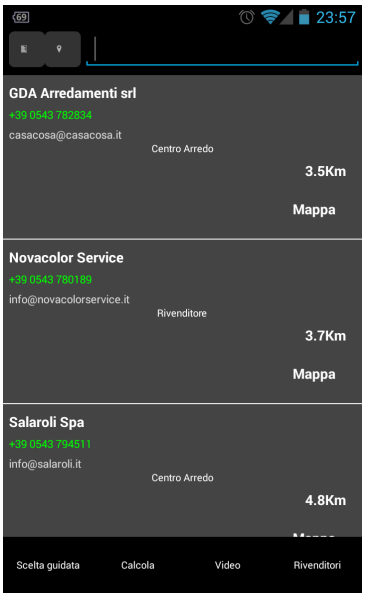

Figura 4.33: Funzionalità "Rivenditori" prototipo

Figura 4.34: Funzionalità "Rivenditori" originale

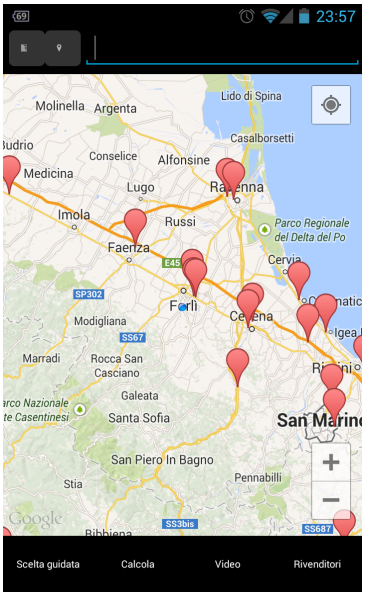

Figura 4.35: Funzionalità "Rivenditori" prototipo

4.2 Considerazioni finali

L'obbiettivo di sviluppare un prototipo di app cross-platform si può considerare raggiunto, come dimostrano gli screenshots in questo capitolo e nel paragrafo 3.2.4 che mostrano MetropolisTitan in esecuzione rispettivamente su simulatore iPhone, su emulatore Android e su device Android. Il prototipo infatti funziona bene su tutti i dispositivi Android considerati, su tutte le tipologie di emulatori Android (la funzionalità "Rivenditori" non è stata testata sugli emulatori perché non supportano la geolocalizzazione), e sul simulatore iPhone.
Nel complesso il prototipo risulta fluido e reattivo e la UX è molto simile a quella nativa. L'unico difetto realmente significativo a livello di tempistiche è il ritardo di più o meno cinque secondi con il quale il Button mostra la lista dei rivenditori, evidenziato in caratteri rossi in figura 4.19. Questo problema potrebbe essere stato causato dall'utilizzo dei metodi add() e remove() non del tutto adeguati per gestire l'aggiunta e la rimozione di lista e mappa dalla schermata, o al layout non del tutto corretto della funzionalità "Rivenditori". Perciò conoscendo un po' più approfonditamente le API di Titanium e gestendo il layout in maniera ottimale, secondo le regole della "Composite UI layout Specification" ideata appositamente per il framework, credo che questo difetto si possa correggere facilmente.
Un ulteriore accorgimento potrebbe consistere nello sfruttare Alloy, il framework MVC di supporto a Titanium, per raggiungere una organizzazione dei contenuti ottimale sfruttando XML.
Lo sviluppo del prototipo tramite Titanium è stato relativamente semplice e mi ha permesso di effettuare alcune osservazioni sul framework.
Ho apprezzato molto la notevole offerta di API cross-platform, quindi supportate sia da iOS che da Android. Proprio questo infatti mi ha permesso di implementare solo poche righe di codice diversificate a seconda della piattaforma, come è possibile vedere nel paragrafo 3.2.3.
Titanium concede la possibilità di utilizzare i componenti di UI nativi più complessi offerti dagli SDK nativi, o realizzarli in maniera customizzata sfruttando la composizione di componenti di UI nativi di base, come è stato per la Tab bar del prototipo, concedendo la stessa libertà che si ha nello sviluppo nativo. Inoltre possiede in minima parte anche API JavaScript platform-specific, perché alcuni componenti di UI nativi sono offerti solamente da uno degli SDK nativi e non da entrambi, come nel caso della

funzionalità del componente SegmentedControl in iOS, offerta da Titanium attraverso il componente TabbedBar. Al fine di garantire il corretto funzionamento dell'app, per le API che si vogliono utilizzare è necessario verificare se sono supportate da entambe le piattaforme, e se presentano eventuali lievi differenze di funzionamento e interpretazione tra esse, anche se fortunatamente solo una piccola parte delle API presenta questa particolarità. Nello sviluppo cross-platform occorre sempre tenere presente la grande varietà di display appartenenti ai device nei quali l'app verrà eseguita, quindi risulta molto importante gestire al meglio il layout, e può essere necessario diversificarlo effettuando alcuni piccoli accorgimenti specifici, che Titanium mi ha permesso di realizzare sfruttando il feature detection tramite API JavaScript.
Lo sviluppo del prototipo tramite Titanium è risultato molto interessante, anche se purtroppo non è stato possibile mettere in luce completamene tutto il potenziale che Titanium può offrire, a causa del tempo limitato a mia disposizione e della ripida curva di apprendimento per imparare a sfruttare il framework, che è JavaScript-based ma costituito per la maggior parte da API proprietarie.
Secondo me Titanium è un ottimo strumento, perché permette di mantenere praticamente gli stessi look&feel e performance delle app native, dato che consente di sfruttare la maggior parte dei componenti di UI nativi. Inizialmente sarà necessario impiegare un po' di tempo per imparare a sfruttarlo al meglio, ma poi consentirà di minimizzare i tempi e costi di sviluppo, realizzando un'unica app eseguibile in maniera efficiente sulle due piattaforme attuamente più diffuse al mondo, senza necessità di conoscerne i rispettivi linguaggi nativi. Inoltre a breve supporterà anche lo sviluppo di app per la piattaforma Window Phone 8[25].

Conclusioni e sviluppi futuri

L'*obbiettivo* di questa tesi è stato quello di fare una panoramica sullo stato dell'arte degli approcci di sviluppo di mobile app alternativi agli approcci nativi, e poi approfondire ulteriormente l'analisi, ed in particolare il framework ritenuto più interessante, mediante lo sviluppo di una app concreta, facendo quindi il confronto con lo sviluppo usando approcci nativi.
Credo che la prima parte dell'obbiettivo sia stata raggiunta con successo, avendo individuato nel primo capitolo tre principali tipologie di strumenti per la realizzazione di web app e app ibride, e attraverso l'approfondimento di due di esse nel secondo e terzo capitolo. Gli approfondimenti sono stati effettuati sulle librerie tramite lo svolgimento dell'applicazione di test esposta nel secondo capitolo, e soprattutto tramite lo sviluppo del prototipo e la sua valutazione, esposti rispettivamente nel terzo e nel quarto capitolo.

Attualmente penso che nessun framework sia in grado di soddisfare appieno e in modo ottimale tutte le esigenze che gravitano attorno allo sviluppo multi-platform, perchè quello che potrebbe essere considerato il migliore per app che devono raggiungere una certa tipologia di obbiettivi e offrire certe funzionalità, potrebbe non esserlo per altre che hanno obbiettivi e offrono funzionalità diversi. É possibile però stabilire il migliore in relazione ad una particolare categoria di obbiettivi da raggiungere e di funzionalità da offrire. Quindi ad esempio se le performance sono una caratteristica di fondamentale importanza per l'app, come nel caso di videogiochi fruibili come app su più piattaforme, o di app con particolari animazioni, lo sviluppo nativo rimane la scelta migliore.
Entrando nel merito delle librerie approfondite nel secondo capitolo, credo che le app risultanti da questo tipo di approccio siano ancora deboli, perchè non in grado di supportare completamente le funzionalità richieste ad un'app e di garantire un'esperienza utente di valore. Infatti nonostan-

te le tecnologie web sulle quali si basano abbiano definito un orientamento verso lo sviluppo di applicazioni per dispositivi mobile, le app risultanti presentano ancora alcune limitazioni. Infatti le loro performance sono limitate dalla presenza del browser come ambiente di esecuzione, che interfacciando l'accesso alle risorse del device rende la UI meno reattiva dato che la velocità di rendering delle WebView non è ancora del tutto paragonabile alla velocità di rendering delle UI native, specialmente se le app contengono grafica complessa. Inoltre l'accesso alle funzionalità hardware e software del device non è ancora completo e coerente su tutte le piattaforme, riducendone il range delle funzionalità da offrire.
Quindi penso che l'approccio attualmente più vantaggioso per affrontare lo sviluppo cross-platform sia quello offerto da Titanium, essendo un valido strumento per realizzare app cross-platform mantenendo il giusto compromesso tra portabilità e UX nativa, come esposto nel terzo capitolo.

Lanciando uno sguardo verso il *futuro*, credo che lo sviluppo multi-platform possa dirigersi verso un approccio ibrido, basato sul modello offerto da Titanium che promuove la sinergia tra UX nativa e riutilizzo del codice, che permetta di raggiungere le prestazioni native attraverso il miglioramento dell'interprete JavaScript, e di superare i limiti di JavaScript, che a mio parere non è ancora del tutto maturo per sostenere lo sviluppo di applicazioni complesse perchè è debolmente tipizzato e debolmente orientato agli oggetti. Credo infatti che nuovi linguaggi come Dart[28] o superset di Javascript come TypeScript[27] potrebbero affermarsi e diffondersi colmandone le lacune attraverso l'inclusione di classi, interfacce, tipizzazione forte, etc., raggiungendo robustezza e sicurezza.

Bibliografia

[1] Sito ufficiale inglese di Wikipedia
http://en.wikipedia.org

[2] Sito ufficiale di VisionMobile, società di ricerca
http://www.visionmobile.com

[3] ComScore Reports May 2012 U.S. Mobile Subscriber Market Share
http://www.comscore.com/ita/Insights/Press_Releases/2012/7/
comScore_Reports_May_2012_U.S._Mobile_Subscriber_Market_Share

[4] Sito ufficiale di Ubuntu
http://www.ubuntu.com/phone

[5] iOS Dev Center
https://developer.apple.com/devcenter/ios/index.action

[6] Windows Phone Dev Center
http://developer.windowsphone.com/en-us/develop

[7] Android Developers
http://developer.android.com/index.html

[8] HTML5 Specification
http://www.w3.org/TR/html5/

[9] MacDonald Matthew, O'Reilly Media (2011),
HTML5: The Missing Manual.

[10] Can I use
http://caniuse.com/

[11] Introduction to CSS3
http://www.w3.org/TR/2001/WD-css3-roadmap-20010523/

[12] Javascript APIs
http://www.w3.org/standards/techs/js#w3c_all

[13] Sito ufficiale di jQuery Mobile
http://www.jquerymobile.com

[14] Sito ufficiale di Sencha Touch
http://www.sencha.com/products/touch/

[15] Sito ufficiale di Qooxdoo
http://www.qooxdoo.org

[16] Sito ufficiale di Enyo
http://www.enyojs.com

[17] Sito ufficiale di ChocolateChip-UI
http://www.chocolatechip-ui.com

[18] Sito ufficiale di Dojo Mobile
http://www.dojotoolkit.org/features/mobile

[19] Sito ufficiale di Phonegap
http://www.phonegap.com/

[20] Sito ufficiale di Motorola RhoMobile Suite
http://www.motorolasolutions.com

[21] Sito ufficiale di IBM Worklight
http://www-03.ibm.com/software/products/it/worklight/

[22] Sito ufficiale di Vaadin
https://www.vaadin.com

[23] Sito ufficiale di Monocross
http://www.monocross.net/

[24] Sito ufficiale di Titanium
http://www.appcelerator.com/titanium/

[25] Appcelerator Blog
http://www.appcelerator.com/blog/

[26] Titanium Documentation
http://docs.appcelerator.com/titanium/latest/

[27] Sito ufficiale di TypeScript
http://www.typescriptlang.org/

[28] Sito ufficiale di Dart
https://www.dartlang.org/

Ringraziamenti

Ringrazio tutti coloro che mi sono stati vicini a loro modo, sostenendomi anche nei momenti di sconforto.
Grazie al Prof. Alessandro Ricci e al Prof. Andrea Santi che mi hanno seguito ed aiutato nella stesura di questa tesi.
Un ringraziamento particolare va ad Alex Benini, Claudio Buda e Giulio Piancastelli, che mi hanno dato la possibilità di svolgere il tirocinio presso l'azienda Mint, permettendomi così di redigere questa tesi e soprattuto di fare un'importante e indimenticabile esperienza.