ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria Informatica

PROGETTAZIONE E SVILUPPO DI APPLICAZIONI IBRIDE IN AMBITO MOBILE

Elaborata nel corso di: Sistemi Operativi

Tesi di Laurea di:
ALESSIA PAPINI

Relatore:
Prof. ALESSANDRO RICCI

Co-relatori:
Ing. ANDREA SANTI

ANNO ACCADEMICO 2012–2013
SESSIONE II

PAROLE CHIAVE

Mobile Application

Hybrid

HTML5

Javascript

PhoneGap

Alla mia famiglia...

A mio nonno Claudio...

Indice

1 Introduzione **1**
1.1 Smartphone . . . 3
1.2 Sistemi operativi *mobile* . . . 4
1.3 Nuove tipologie di applicazioni . . . 6
1.3.1 Native Application . . . 7
1.3.2 Mobile Web Application . . . 8
1.3.3 Hybrid Application . . . 10
1.4 Caratteristiche delle Hybrid Application . . . 11
1.5 Obiettivi e organizzazione della tesi . . . 13

2 Architettura **15**
2.1 N-Tier . . . 17
2.2 Architettura sul dispositivo . . . 18
2.2.1 Applicazione . . . 19
2.2.2 API per l'interfacciamento e piattaforma *mobile* . . . 22
2.2.3 Interazione con la parte backend . . . 25
2.3 Confronto tra diverse architetture . . . 30
2.3.1 Confronto con applicazioni native . . . 30
2.3.2 Confronto con applicazioni web . . . 31

3 Approcci per lo sviluppo **33**
3.1 HTML5 . . . 33
3.1.1 Canvas . . . 34
3.1.2 Offline e Storage . . . 35
3.1.3 Geolocation . . . 37
3.1.4 Web Workers . . . 38
3.1.5 Server-Sent Events . . . 40

3.2 CSS3 40
3.2.1 Transition 41
3.2.2 Animation 42
3.2.3 Media Queries 43
3.3 Javascript 44
3.3.1 AJAX 45
3.3.2 DOM e gestione eventi 46
3.4 Model-View-Controller 49
3.4.1 Single-page application 50

4 Framework **53**
4.1 Framework per la User Interface 53
4.1.1 JQuery Mobile 54
4.1.2 SenchaTouch 55
4.2 Framework per l'accesso alle API del dispositivo: PhoneGap 55
4.2.1 Architettura 56
4.2.2 PhoneGap API 58

5 Conclusioni **67**

Capitolo 1

Introduzione

Negli ultimi anni i dispositivi *mobile* hanno conquistato un posto di primo piano nella vita di tutti i giorni, offrendo agli utenti una *user-experience* che si adatti alle richieste ed esigenze di ogni individuo. L'inizio di questa ascesa risale a 40 anni fa, quando il 3 aprile 1973 Martin Cooper, ingegnere della Motorola, effettuò la prima telefonata "*mobile*"al suo diretto concorrente, il direttore di ricerca dei Bell Laboratories (AT&T), utilizzando il prototipo DynaTAC 8000x. I dispositivi che seguirono furono di dimensioni notevoli, tanto da guadagnarsi il soprannome "*the brick*", il mattone, abbastanza pesanti, con poca autonomia e ad appannaggio solo di pochi privilegiati, rappresentando addirittura uno "*status-symbol*"della società. Nonostante questo, furono un pionieristico e coraggioso primo passo in avanti verso una rivoluzione delle telecomunicazioni, gli albori di una nuova e frenetica era della telefonia in cui il telefono fisso lasciava abitazioni e uffici per essere utilizzabile ovunque.

Il costante progresso tecnologico nel campo dell'elettronica ha consentito la miniaturizzazione dei ciruiti integrati permettendo alle aziende del settore di produrre modelli di cellulare sempre più sottili, tecnologicamente avanzati, dotati di molteplici funzioni, maneggevoli, con display grandi ad alta risoluzione e più economici, fruibili quindi dalla maggior parte della popolazione. Il telefono cellulare, infatti, è una delle tecnologie che ha ottenuto il successo più rapido della storia rappresentando il futuro della comunicazione *mobile* e diffondendosi velocemente in tutto il mondo, nel 2007 il 50% della popolazione mondiale possedeva un cellulare, all'inizio del 2009 la percentuale è salita fino al 61% [13]. Un anno decisivo è stato il

2011, in cui è avvenuto lo storico sorpasso dei dispositivi mobili sui pc, questo non rappresentò solamente una semplice inversione di tendenza a livello di mercato, ma una vera e propria svolta tecnologica e sociale. Cambiarono le modalità di fruizione e i contenuti, cambiò il modo di pensare degli utenti, di interagire tra loro, di effettuare tutte quelle operazioni quotidiane che prima richiedevano, se non la presenza fisica in un determinato luogo, almeno l'utilizzo di un dispositivo fisso o relativamente ingombrante come un computer o un notebook.

La volontà di rendere ogni individuo in grado di poter comunicare in mobilità e con un modo semplice e intuitivo con altre persone ha portato, oltre all'incessante sviluppo della tecnologia *mobile*, all'esigenza di sviluppare un supporto che permettesse lo scambio e la fruizione di informazioni più articolate ed elaborate. L'idea era di realizzare una rete informatica che permettesse agli utenti di differenti computer di comunicare tra loro. Essa si sviluppò in diverse tappe che portarono alla nascita di una rete mondiale di reti di computer ad accesso pubblico, definita come "rete delle reti", ovvero Internet, che attualmente rappresenta il principale mezzo di comunicazione di massa, in grado di offrire all'utente una vasta serie di contenuti informativi e servizi. Uno dei servizi principali di Internet è il Web, uno spazio elettronico e digitale che permette di navigare ed usufruire di un insieme vastissimo di contenuti collegati tra loro tramite i cosiddetti *link* o collegamenti. I contenuti del Web sono costantemente *on-line* e sono organizzati in *siti web* a loro volta strutturati in *pagine web*, le quali si presentano come composizioni di testo e/o grafica visualizzate tramite un browser web. La peculiarità dei contenuti del Web è quella di non essere memorizzati su un unico computer ma di essere distribuiti su più computer, caratteristica da cui discende un'alta efficienza, in quanto non vincolati ad una particolare localizzazione fisica. Tale peculiarità è realizzata dal protocollo di rete HTTP il quale permette di considerare i contenuti del Web come un unico insieme anche se fisicamente essi risiedono su una moltitudine di computer in Internet.

L'opportunità di sfruttare un tale servizio ovunque, disponendo di un collegamento Internet, è concretizzata dalla convergenza e integrazione di diverse tecnologie di comunicazione all'interno di un unico dispositivo *mobile*, lo *smartphone*, diffuso oggi presso più della metà della popolazione mondiale. Oggi però il cellulare non è più solo uno strumento di comunicazione, ma un vero e proprio "*Social Medium*", "l'estensione digitale"del

consumatore che lo accompagna lungo tutta la giornata e gli permette di svolgere molteplici attività, non solo legate al fabbisogno di comunicazione e informazione.

1.1 Smartphone

Dai primi telefoni radiomobili agli attuali *smartphone* l'evoluzione è stata tanto veloce quanto pervasiva nella nostra vita. L'uso del termine cellulare (o telefonino) viene oggi utilizzato per indicare dispositivi telefonici mobili semplici, con poche funzioni e in media senza connessione ad Internet mentre il termine *smartphone* indica un telefono dotato di un sistema operativo più evoluto e aperto all'installazione di applicazioni addizionali da parte dell'utente, capace di integrare le funzioni del cellulare con molte altre, in particolare quelle connesse a Internet, agli ambienti di lavoro e con finalità sociali, multimediali e ludiche. A rendere gli smartphone così performanti e funzionali rispetto a telefoni cellulari di precedente generazione sono l'aumento delle prestazioni grazie a processori *multi-core* e memorie RAM con capacità superiori ad 1 GB, unite a sistemi operativi sviluppati *ad hoc* e ad interfacce utente sempre più *user-friendly* che consentono molto spesso la realizzazione di display di dimensioni maggiori a parità di spazio disponibile eliminando in molti casi la necessità di tastierine fisiche [9] [16].

Se il cellulare nasce come appendice del telefono fisso e quindi come semplice mezzo di comunicazione vocale interpersonale (cui al massimo si aggiungeva la novità degli sms), oggi questa funzione è solo una delle tante e, pur rimanendo centrale, oramai è su altre che si concentra l'interesse di tutti, utenti e costruttori. Funzioni che possono essere considerate accessorie sono ormai una presenza fondamentale in uno smartphone, come semplici giochi, applicazioni che permettono l'integrazione tra telefonia cellulare e Internet e che ci consentono di essere sempre connessi ai *social network*, di effettuare videochiamate, di utilizzare una fotocamera integrata, sistemi di localizzazione GPS e sfruttare il collegamento ad Internet. Alcune di queste funzionalità erano presenti già in modelli di precedente generazione mentre altre sono rese possibili dall'integrazione nel dispositivo *mobile* di sensori quali accelerometri, sensore di prossimità, giroscopi, magnetometri ed altri ancora come sensori di luce, di temperatura o di pressione. Un'ulteriore importante tecnologia sviluppata è il *touchscreen*, che permette di

interagire direttamente con l'interfaccia grafica attraverso il tocco delle dita o altri oggetti sullo schermo; inoltre esistono particolari schermi tattili che permettono il *multi-touch*, ovvero la capacità riconoscere la presenza in contemporanea di più dita sullo schermo, utilizzando tecnologie resistive o capacitive.

Oggi esistono smartphone con connessione GSM/GPRS/EDGE/UMTS/HSDPA/HSUPA/LTE, che utilizzano le tecnologie *Bluetooth* e *Wi-Fi* per le comunicazioni con altri dispositivi. Alcuni smartphone, inoltre, offrono anche possibilità di *tethering* in *Wi-Fi* (modem Internet) verso dispositivi quali altri smartphone o cellulari, laptop o PC fissi, mentre la totalità di essi, così come la maggior parte dei cellulari permettono il tethering cablato ovvero attraverso cavo USB.

Lo smartphone entra prepotentemente nella vita degli individui e la forza dei nuovi dispositivi *mobile* sta proprio, nella capacità di essere a portata di mano in qualsiasi momento e di dare all'utente la possibilità di gestire operazioni e consultare informazioni che fino a pochi anni fa implicavano una modalità di fruizione totalmente diversa.

L'ingresso e la diffusione degli smartphone sul mercato, ha determinato numerosi cambiamenti nella società: dal modo di fare le foto, al processo selettivo nello shopping, passando per l'istruzione, senza contare le applicazioni che permettono all'utente di fare qualsiasi cosa.

L'Italia è al primo posto in Europa per il numero di smartphone posseduti dalla popolazione. Secondo il decimo rapporto Rapporto Censis/Ucsi 2012 [12] l'utenza degli smartphone è cresciuta del 10% e il 54,8% dei giovani italiani ne possiede uno. In particolare, tra il 2009 e il 2012 è avvenuto un vero e proprio boom degli smartphone che sono passati dal 15% al 27,7% della popolazione.

La caratteristica principale degli smartphone moderni è rappresentata, tuttavia, dalla possibilità di installare applicazioni di terze parti (software, giochi, temi) per aumentare le funzionalità del dispositivo *mobile*.

1.2 Sistemi operativi *mobile*

Un'aspetto fondamentale che caratterizza gli smartphone è la presenza di un sistema operativo, che permette di interagire ad alto livello con le molteplici funzionalità di questi dispositivi. Inizialmente si trattava di sistemi mini-

mali che dovevano soddisfare semplici funzioni del telefono come chiamate e messaggi, senza permettere alcuna variazione dei contenuti. L'aumento progressivo delle funzioni e l'integrazione di nuove caratteristiche hardware, ha costretto ad un evoluzione i sistemi operativi, che hanno dovuto cambiare radicalmente la loro semplice struttura a favore di una più complessa che potesse dare all'utente la possibilità di interagire con il proprio dispositivo in modo intuitivo e garantendo, inoltre, un maggior livello di personalizzazione.

Il sistema operativo *mobile* più diffuso è *Android* che vanta il primo posto con il 79,3% del mercato mondiale, mantenendo la sua ledership grazie all'elevato numero di device sul quale è installato e ad un grande contributo, in questi anni, determinato dall'uscita dei nuovi modelli *Samsung*, mentre al secondo posto troviamo *IOS* con il 13,2% che perde qualche punto percentuale rispetto all'anno precedente. Nonostante il suo rilascio sia avvenuto più recentemente rispetto ai già citati OS, *Windows Phone* ricopre già il terzo posto con il 3,7% destinato nei prossimi anni ad aumentare, mentre uno dei sistemi operativi storicamente più diffuso come *Symbian* subisce un crollo tale da arrivare allo 0,2% del mercato mondiale [18].

Alcuni dettagli dei principali sistemi operativi *mobile*:

- **Android** [15]
  Sviluppatore: Android Inc, successivamente acquisita da Google Inc.
  Release Corrente: 4.4 Kit Kat
  Linguaggio di programmazione: Java
  Caratteristiche: Questo OS è stato protagonista di una grande diffusione, scalando il mercato del sistemi operativi *mobile* in pochi anni. E' presente su dispositivi smartphone e tablet di diverse marche e offre una grande quantità di applicazioni sul proprio market, *Google Play*. Ha una struttura *open-source* ed è bastato sul *kernel* Linux. A disposizione degli sviluppatori è presente sia un *SDK* con alcuni strumenti fondamentali per la realizzazione di applicazioni che permettano di ampliare le funzionalità del dispositivo, sia un *ADT* (*Android Development Tools*) plugin per *Eclipse* con classi e librerie necessarie allo sviluppo.

- **IOS** [11]
  Sviluppatore: Apple Inc.
  Release Corrente: 7
  Linguaggio di programmazione: Objective-C

Caratteristiche: E' sviluppato per un determinata serie di dispositivi quali *IPhone*, *IPad*, *IPod touch* ed *Apple TV*. Apple sviluppa sia l'hardware che il software dei suoi dispositivi e questo garantisce un'integrazione massima delle migliaia di *app* fornite dallo *App store* ufficiale, le uniche a poter essere installate. Per gli sviluppatori è a disposizione un *SDK*, previa registrazione a pagamento ad uno specifico programma di *development* Apple, che permette di creare applicazioni e testarle nel simulatore supportato dal kit. L'ambiente di sviluppo per l'SDK è *XCode* presente esclusivamente su computer *Mac OS X*.

- **Windows Phone** [20]
  Sviluppatore: Microsoft Corporation
  Release Corrente: 8.0
  Linguaggio di programmazione: C++ C#
  Caratteristiche: La versione 7.0, rilasciata alla fine del 2010, rappresentò un decisivo distacco dalle precedenti versioni, non più compatibili, di *Windows Mobile* sia per quanto riguarda l'interfaccia grafica che per le funzionalità aggiuntive. Gli sviluppatori registrati a *Windows Phone* ed *Xbox Live* possono inserire e modificare le loro applicazioni per la propria piattaforma attraverso l'applicazione online *Windows Phone Dev Center*, che fornisce un pacchetto software gratuito, contenente al suo interno diversi applicativi, tra i quali, *Visual Studio 2012 Express* per *Windows Phone*, *Windows Phone Emulator*, *XNA Game Studio 4.0*, *Silverlight* per Windows Phone e *Expression Blend*. Le applicazioni dedicate a questa piattaforma possono essere scaricate dal *Windows Phone Store*.

1.3 Nuove tipologie di applicazioni

Il successo dei dispositivi mobili si diffonde di pari passo con la crescita delle cosiddette *app*, sostantivo, questo, che è ormai entrato a far parte del vocabolario comune di milioni di persone, nonostante fino a pochi anni fa non avesse alcun genere di significato rilevante. *App*, deriva dal termine *application* e tecnicamente si tratta di applicazioni di vario genere, gratuite o meno, scaricabili dagli *application store* od incluse in smartphone e tablet. Il punto di forza di questi piccoli software è sicuramente la facilità di utilizzo e la possibilità di usufruirne in qualsisi momento lo si voglia. Nel

corso dell'ultimo anno, il 37,5% di chi usa lo smartphone ha scaricato applicazioni e il 16,4% lo fa in modo frequente. Soprattutto giochi, ricercati dal 63,8% di chi scarica *app*, meteo (33,3%), mappe (32,5%), social network (27,4%), news (25,8%) e sistemi di comunicazione (messaggistica istantanea e telefonate tramite Internet): 23,2% [10].

Nella progettazione di applicazioni per dispositivi mobili si possono seguire due approcci principali:

- *Native*
- *Web-based*

Uno è la realizzazione di un applicativo mirato e specifico per un tipo di dispositivo e/o sistema operativo, l'altro è rappresentato dallo sviluppo di una applicazione accessibile via web. Ognuno di questi due approcci presenta sia vantaggi che svantaggi sia per chi progetta il servizio, sia per chi ne fruisce e non c'è un approccio che a priori risulti essere migliore dell'altro. E' importante tenere conto delle caratteristiche proprie del servizio da progettare e sviluppare, per poter stabilire quale dei due approcci risulta essere più adatto, anche in considerazione dei termini e dei costi di realizzazione. Attraverso questi due approcci di sviluppo si possono realizzare diversi tipi di applicazioni che utilizzano l'uno, l'altro o entrambi e che saranno analizzate di seguito.

1.3.1 Native Application

Una *native application* è un'applicazione creata per un determinato dispositivo (smartphone, tablet, etc) e sviluppata in linguaggi di programmazione specifici per ogni sistema operativo *mobile* (Objective C per iOs, Java per Android, C# o C++ per Windows Phone, etc). Gli utenti possono scaricare queste applicazioni tramite uno store online, come ad esempio l'*Apple Store* o *Google Play Store* e installarle direttamente sul dispositivo.

Vantaggi

- **Accesso completo a tutte le funzionalità native del device**: si ha la possibilità di interagire con i sensori hardware del dispositivo come GPS, fotocamera, accelerometro. Inoltre è possibile l'utilizzo di

notifiche *push* e l'accesso a funzioni interne come la rubrica contatti o le immagini salvate nel dispositivo.

- **User-Experience**: si possono realizzare interfacce utente ottimizzate, in quanto le chiamate a sistema sono già in linguaggio macchina e il codice non deve essere interpretato da un browser web. Questo assicura alte prestazioni in termini di fluidità e reattività.
- **Distribuzione**: è possibile distribuire l'applicazione attraverso store online che, raggiungendo un gran numero di utenti, garantiscono maggior visibilità e diffusione.
- **Modalità Offline**: dopo essere stata scaricata dallo store online, l'applicazione è installata direttamente sul dispositivo e non necessità di ulteriori dowload per essere utilizzata.

Svantaggi

- **Portabilità**: le native application, essendo sviluppate in uno specifico linguaggio di programmazione, sono vincolate ad una determinata tipologia di dispositivi. Avere la possibilità di sviluppare e di utilizzare questo tipo di applicazioni su qualsiasi dispositivo implica la conoscenza, da parte dello sviluppatore, di diversi linguaggi determinando inevitabilmente maggiori costi di sviluppo e di manutenzione.
- **Flessibilità**: la necessità di correggere *bug* in modo tempestivo, sviluppare aggiornamenti o modificare semplicemente l'interfaccia grafica è vincolata all'approvazione dello store in cui l'app è pubblicata. Appare evidente come il ciclo di sviluppo, modifica e rilascio mediante uno store possa essere un problema sia per i tempi elevati di validazione che per un eventuale rifiuto per non aver rispettato tutte le *policies* di validazione.

1.3.2 Mobile Web Application

Le *mobile web application* sono applicazioni *Internet-enabled* che possono essere fruite da un tablet o uno smartphone. Sono accessibili tramite browser del dispositivo *mobile* e non hanno bisogno di essere scaricate e installate sul dispositivo. L'utilizzo della memoria *cache* del telefono permette spesso

di utilizzare alcune funzionalità in modalità offline permettendo in tal modo di far avvicinare ulteriormente le *mobile* web application alle applicazioni *mobile* native. E' importante precisare che una *mobile* web application non è un semplice sito web ottimizzato ma consiste in un sito dinamico *responsive*, in grado di offrire funzionalità complesse simili a quelle fornite dalle comuni app installate sul dispositivo. Ciò è possibile grazie ai linguaggi di scripting (sia lato client che lato server) e tecnologie come AJAX, Jquery, HTML5, CSS3, Javascript che consentono di creare degli applicativi veri e propri, fruibili attraverso un browser, simili anche dal punto di vista della interfaccia utente, alle comuni applicazioni native.

Vantaggi

- **Portabilità**: le *mobile* web application sono applicazioni multipiattaforma che permettono il loro utilizzo su più dispositivi utilizzando tecnologie comuni e facilmente adottabili dagli sviluppatori. Questo implica generalmente tempi di sviluppo e costi di manutenzione ridotti.

- **Distribuzione**: un indubbio vantaggio risulta nel *deploy*, in quanto un utente può accedere ad una *mobile* web app, semplicemente inserendo un indirizzo web nel browser senza che l'applicazione sia sottoposta al processo di approvazione dello store. Questo determina un'ulteriore flessibilità, infatti, una *mobile* web application è aggiornabile, migliorabile ed espandibile autonomamente dall'amministratore e ogni evoluzione potrà essere fruibile da tutti gli utenti finali senza necessità di aggiornamenti lato utente consentendo a tutti di avere la stessa versione.

- **Spazio utilizzato**: mentre una native app è installata fisicamente sul dispositivo, una *mobile* web app consiste in un link verso un applicativo remoto ed ha il vantaggio di non incidere minimamente sulle capacità di *storage* del dispositivo e di essere sostanzialmente indipendente dalle capacità di calcolo dello stesso, infatti, la velocità di una *mobile* web app dipende dalla bontà della connessione alla rete e dalle performance del server remoto nell'offrire l'elaborazione richiesta dall'utente.

Svantaggi

- **Accesso ridotto alle funzionalità native del device**: non è possibile accedere completamente a tutte le funzionalità appartenenti al dispositivo come l'invio di notifiche push, l'utilizzo della bussola o dell'accelerometro.
- **User-Experience**: non tutti i browser effettuano il *rendering* nella stesso modo, quindi si potrebbero avere esperienze differenti in base ai dispositivi ed ai browser utilizzati. Inoltre, non è possibile utilizzare tutte le views standard come barre strumenti, bottoni o tabs, normalmente utilizzate nelle native app, ma solo quelle appartenenti alle interfacce utilizzate nelle classiche pagine web.
- **Mancanza dello store**: non potendo essere pubblicate in uno store, le *mobile* web app non usufruiscono dell'enorme visibilità che gli store offrono, oltre a ciò, non posso avvalersi della possibilità di vendita diretta che limita gli eventuali guadagni.
- **Prestazioni**: le *mobile* web apps non riescono ancora a supportare applicazioni CPU intensive o applicazioni contenenti grafica molto complessa come giochi 3D, anche se l'utilizzo delle *canvas* (elemento facente parte di HTML5) unito a Javascript hanno dato un forte impulso verso tale strada.

1.3.3 Hybrid Application

L'approccio ibrido appare come un giusto compromesso tra sviluppo web based e native app in quanto permette l'accesso e il controllo all'hardware sviluppando però l'essenza dell'applicazione in linguaggi come HTML5, CSS e Javascript. Una *hybrid application* è in sostanza un contenitore sviluppato nel linguaggio destinato ad una specifica piattaforma che però sfrutta, al suo interno, funzioni tipicamente legate al normale sviluppo web. Questo può essere realizzato attaverso vari framework, gratuiti o a pagamento, che forniscono elementi di UI standard e garantiscono compatibilità multidevice e multipiattaforma, prefiggendosi quindi l'obiettivo di sfruttare sia le potenzialità delle web app, che quelle delle native app. Una descrizione più approfondita di questa tipologia di sviluppo verrà trattata nel prossimo paragrafo.

1.4 Caratteristiche delle Hybrid Application

Lo sviluppo di una *hybrid application* consente di creare applicazioni che abbiano la maggior parte dello sviluppo inerente alla componente web, utilizzando i linguaggi HTML5, CSS, Javascript che garantiscono il cross-platform e sono quindi compatibili con le diverse piattaforme. L'involucro in un *wrapper* nativo, riscritto a seconda del sistema operativo con il quale si vuole commerciale l'app, permette di sfruttare e interagire con i componenti hardware come fotocamera, GPS e accelerometro. Inoltre le hybrid app richiedono il mantenimento di una sola base di codice, soddisfando requisiti di flessibilità, scalabilità e rendendo gli aggiornamenti più rapidi e agevoli, senza perdere il vantaggio di poter essere messe a disposizione degli utenti attraverso gli store ufficiali delle diverse piattaforme.

Per sostenere lo sviluppo di questo tipo di applicazioni si sono resi disponibili alcuni framework che rappresentano un collegamento tra API del dispositivo e codice HTML, Javascript. Questo determina un grande supporto per gli sviluppatori ma nello stesso tempo implica una dipendenza dai framework stessi, in quanto, per utilizzare una nuova funzionalità nativa introdotta per una determinata piattaforma, è necessario attendere l'aggiornamento del framework che si utilizza per l'implementazione dell'app, prima di poter sfruttare la nuova caratteristica. La memorizzazione dei dati e la possibilità di utilizzare le applicazioni in modalità offline, sono caratteristiche che giocano un ruolo fondamentale nell'ambito delle applicazioni. Per tale motivo le hybrid app utilizzano tecnologie di storage fornite da HTML5 e, nel caso in cui la connessione alla rete sia una parte necessaria all'applicazione, impiegano il caching per rendere possibile l'utilizzo, anche nel caso in cui l'utente non sia connesso ad Internet.

Le hybrid app, grazie al wrapper nativo, si prefiggono di offrire una *user-experience* quando più simile a quella di una native app. Nonostante questo, per quanto riguarda la parte grafica, il rendering è affidato alla *Web View* invece che alle API come nelle applicazioni native e allo stato attuale, nonostante gli enormi progressi, la velocità di rendering della Web View non è ancora paragonabile alla velocità di rendering delle UI delle applicazioni native, individuando quindi un divario sostanziale. Inoltre l'esecuzione di una parte di app sul web introduce un ulteriore livello tra l'utente e l'applicazione stessa determinando una reattività e velocità minore rispetto a quelle di una native app. A dispetto di ciò, le hybrid app stanno diventando

Figura 1.1: Confronto tra le caratteristiche delle *mobile* application

un ottimo compromesso per chi è interessato ad uno sviluppo di applicazioni che operino in logica multipiattaforma a ridotti costi di sviluppo e manutezione e se questa diffusione dovesse continuare, secondo un recente studio di Gartner, entro il 2016 si arriverà ad una situazione in cui oltre il 50% delle app saranno sviluppate in modalità ibrida.

Alcuni esempi di esperienze nell'uso delle hybrid app:

- **Facebook** è l'esempio per eccellenza che mette in evidenza i limiti di una hybrid app. Inizialmente è stata sviluppata l'applicazione in HTML5, interna ad una involucro nativo, così che l'utente potesse trovarla e scaricarla direttamente dagli store. In seguito Facebook decise di sviluppare un app nativa per ogni piattaforma: i problemi principali furono la velocità e la reattività. Le prestazione offerte da un'applicazioni ibrida, infatti, non erano sufficienti a garantire il livello di *user-experience* richiesto dagli utenti (esempio, navigare la *timeline* strisciando il dito sullo schermo doveva garantire una risposta quasi istantanea). Pertanto l'applicazione è stata riscritta interamente in codice nativo, passando da 2 a 4 stelle, come valutazione utente, in

sole 3 settimane. Prima di emulare l'esperienza e le scelte di Facebook scartando a priori l'approccio ibrido, bisogna, però, ricordare che i problemi di Facebook sono su una scala differente da qualsiasi altro scenario.

- **BBC** rappresenta un caso di successo nell'utilizzo delle hybrid app. BBC, infatti, concretizza la propria necessità di fornire un applicazione relativa allo sport con la realizzazione della hybrid app, BBC Sport. L'applicazione fornisce i contenuti presenti sul sito e l'aggiunta di funzionalità che sfruttano il wrapper nativo. BBC conferma, reduce dal successo della hybrid app *BBC Olympics*, questa tipologia di app come la soluzione alle proprie esigenze.

1.5 Obiettivi e organizzazione della tesi

A seguito delle premesse fatte, in relazione al mondo delle tecnologie in ambito *mobile* ed in particolare in riferimento alle diverse tipologie di applicazioni che si stanno sempre più largamente diffondendo, è interessante esaminare gli strumenti messi a disposizione degli sviluppatori per realizzare applicazioni *mobile*, in particolare quelle di tipo ibrido.

L'obiettivo di questo lavoro è chiarire ed approfondire il concetto di applicazione ibrida e porre in evidenza le differenze e i vantaggi che esse hanno, sia rispetto ad applicazioni native che ad applicazioni web. Questo tipo di soluzione, infatti, si sta imponendo nel mercato delle applicazioni *mobile*, assumendo un ruolo di grande rilevanza, chè è importante comprendere. Per questo si analizzano gli aspetti architetturali che definiscono un'applicazione ibrida e se ne approfondiscono le tecnologie di base con le quali è possibile il loro sviluppo.

Con tali presupposti, il lavoro proposto sarà organizzato nel modo seguente:

- Nel secondo capitolo verrà analizzata la tipica architettura propria delle applicazioni ibride, affrontando sia la parte riguardante il *front-end* che quella relativa all'interazione con la parte *back-end*. Sarà introdotto il concetto di "*bridge*", fondamentale per la realizzazione di questo particolare tipo di applicazioni ed infine verrà effettuato un rapido confonto architetturale ponendo da una parte le applicazioni ibride e dall'altra sia le applicazioni native che quelle web.

- Nel terzo capitolo verranno approfonditi gli strumenti di sviluppo che permettono la realizzazione di un applicazione ibrida, saranno quindi trattati i linguaggi standard web come HTML5, CSS3 e Javascript. Inoltre saranno posti in risalto i pattern utilizzati come modello di approccio nella realizzazione di questa tipologia di applicazioni.

- Nel quarto capitolo sarà presentata una panoramica sui framework di riferimento, che rappresentano un valido supporto allo sviluppo di applicazioni ibride ed in particolare verrà trattato più nel dettaglio il framework *PhoneGap* che risulta essere il più utilizzato per avvalersi dell'accesso alle API native del device.

- Nel capitolo conclusivo verrà fatto un quadro razionale su tutto ciò che si è analizzato, evidenziando aspetti negativi e positivi, valutando infine lo stato di questa tecnologia.

Capitolo 2

Architettura

Come già anticipato nel precedente capitolo, le applicazioni ibride hanno la caratteristica di essere indipendenti dalla piattaforma di utilizzo, permettendo così di sfruttare le caratteristiche hardware di qualsiasi dispositivo, a prescindere dal sistema operativo installato. Nonostante le applicazioni ibride siano sviluppate in linguaggi prettamente legati alle tecnologie web, i sofisticati *tools* di realizzazione garantiscono un *look and feel* strettamente vicino a quello offerto dalle applicazioni native. La progettazione di un'applicazione ibrida utilizza un sistema *client-server*, ovvero un'architettura di rete nella quale un dispositivo client si connette ad un server per la fruizione di un servizio o di particolari risorse. In particolare il client è un qualunque dispositivo (smarphone, personal computer, dispositivo *mobile* ecc.) che si connette alla rete Internet, e si occupa sia di trasmettere all'opportuno server le richieste di reperimento dati che derivano dalle azioni dell'utente, sia di ricevere dal server le informazioni richieste e visualizzarne il contenuto, gestendo in modo appropriato tutte le tipologie di informazioni pervenute e consentendo operazioni locali sulle stesse. Nel client è sufficiente che sia presente un componente per la visualizzazione dei documenti forniti dal server, e quindi che permetta di visualizzare normale testo, contenuti multimediali e che possa interpretare linguaggi *client side* come JavaScript. Il server, invece, è tipicamente un processo in esecuzione su un elaboratore, di norma sempre attivo. Ha il compito, più complesso, di attendere le richieste dei client ed inviare le relative risposte a seguito di opportune elaborazioni. Questo modello applicativo, quindi, rende possibile l'accesso di più client a funzioni residenti su un *Web Service*, utilizzando, per la comunicazione

tra le parti, il protocollo applicativo *HTTP*. Per la realizzazione di questo tipo di sistema è utilizzata l'architettura *n-tier*, che tratteremo in modo più approfondito nel paragrafo successivo. Nelle applicazioni ibride, la parte che risiede sul dispositivo *mobile* è suddivisa in due blocchi principali, uno dei quali è scritto in linguaggi standard per il web, come JS, CSS e HTML5, e rappresenta la logica dell'applicazione; l'altro blocco, invece, è scritto nel linguaggio nativo del dispositivo e rappresenta l'insieme di procedure messe a disposizione per accedere, controllare ed utilizzare le funzionalità di un determinato dispositivo, quali ad esempio l'accelerometro, il GPS o la fotocamera. Il collegamento tra queste due parti avviene grazie ad un *framework*, di cui tratteremo più in dettaglio nel quarto capitolo, che permette di creare un *bridge* (ponte) tra la parte nativa e la parte web.

La progettazione di un'applicazione *mobile* ibrida può essere affrontata con due approcci diversi:

- **Fat Client**: la logica dell'applicazione può essere integrata nella parte client-side come avviene in un'applicazione nativa, utilizzando un insieme di architetture di rete che permettono lo scambio di dati tra applicazione e server. Questo garantisce che alcune funzionalità restino indipendenti dal server, e permette all'utente di poter utilizzare l'applicazione anche in casi in cui non sia garantita la connessione. Se lo sviluppatore decide di utilizzare questo approccio, deve tenere in opportuna considerazione la capacità computazionale del dispositivo sul quale l'applicazione verrà installata.

- **Thin Client**: con questo approccio la maggior parte della logica dell'applicazione è implementata sul server, che si occupa di eseguire i vari script dinamici, effettuare eventuali letture dal database, ed inviare i risultati al client che dovrà solo visualizzarli. Tutto questo viene realizzato utilizzando il *wrapper* nativo esclusivamente come sottile *shell* (strato) sopra la Web View, con eventuale supporto di *caching* sul client, per migliorare le prestazioni dell'applicazione. Questa configurazione permette di gestire l'applicazione in modo centralizzato, ed è più adatta nel caso in cui le medesime informazioni debbano essere accessibili a tutti gli utenti.

La scelta dell'approccio da utilizzare dipende dalle esigenze di ciascuna applicazione, ed è in genere deciso a priori prima di iniziarne lo

sviluppo, in quanto una modifica apportata in seguito mediante aggiornamenti comporterebbe una serie di interventi architetturali di difficile implementazione.

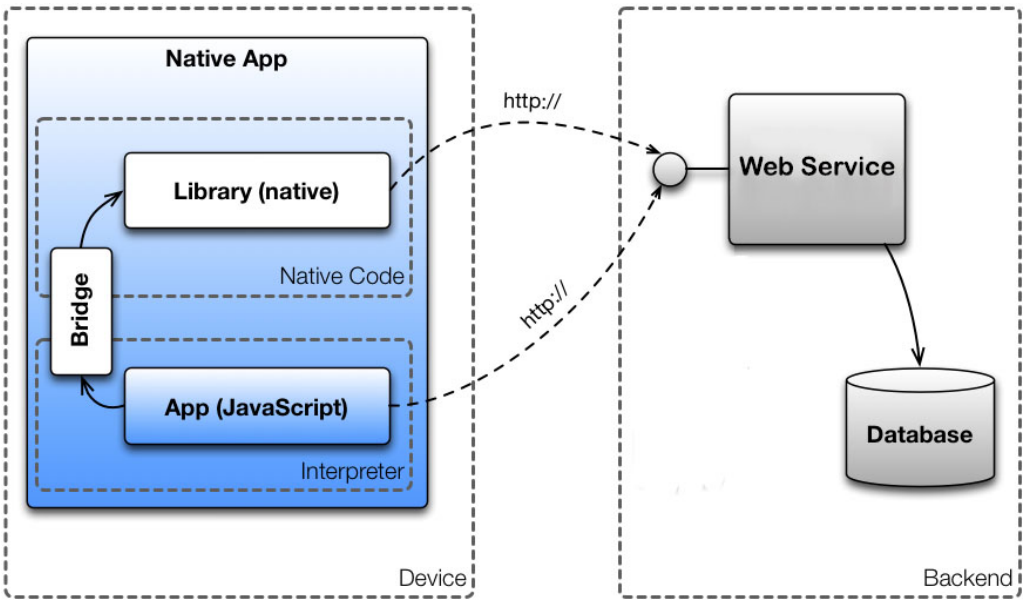

Figura 2.1: Hybrid App Architecture

2.1 N-Tier

L'architettura più comunemente usata nella progettazione di applicazioni ibride è caratterizzata dalla suddivisione in differenti *tier* applicativi delle parti che compongono l'applicazione web; questo modello è per l'appunto chiamato *N-tier*.

Nel caso di una applicazione *mobile* ci sono generalmente due approcci possibili per realizzare questa divisione:

- **2-tier architecture**: secondo questo approccio, il client richiede una risorsa e il server la fornisce direttamente, senza appoggiarsi ad un *provider* esterno per gestire la richiesta ed elaborare la risposta da inviare al client. Il server, quindi, si occupa di erogare il servizio in modo del tutto autonomo rispetto ad eventuali componenti esterne.

- **3-tier architecture**: secondo questo approccio, invece, il client si occupa esclusivamente della *UI* e dell'impaginazione dei contenuti, i quali vengono ottenuti connettendosi ad un server sul quale risiede la *business logic* dell'applicazione; a sua volta, il server svolge il proprio compito con l'ausilio di servizi e connessioni esterne, effettuando collegamenti al database per richiedere i dati necessari a soddisfare la richiesta.

Le applicazioni ibride sono tipicamente sviluppate con un modello *3-tier*, così composto:

1. **Presentation tier**: è il livello più alto, e rappresenta la parte di *user interface*, con la quale l'utente interagisce, e il *look and feel* percepito di conseguenza. Permette di acquisire dati e visualizzare risultati, mentre le richieste sono inviate al livello intermedio mediante l'interfaccia;

2. **Application tier**: svolge un ruolo da intermediario tra i restanti tier e si occupa delle elaborazioni dei dati in base alla cosiddetta *business logic*, cioè all'insieme delle regole per cui i dati sono considerati significativi e le loro relazioni consistenti. A seconda dell'approccio scelto, *fat client* o *thin client*, la business logic può in parte risiedere anche sul dispositivo oltre che sul Web Server. Le elaborazioni di questo tier producono i risultati richiesti dall'utente;

3. **Data tier**: è il tier che si occupa dello storage dei dati che vengono utilizzati e forniti all'applicazione. Rappresenta l'insieme dei servizi offerti da applicazioni indipendenti dal Web, come ad esempio un gestore di database.

L'architettura descritta è ritenuta la più vantaggiosa, in quanto permette una migliore suddivisione dei compiti tra i vari componenti funzionali dell'applicazione, facilitando non solo il processo di creazione ma anche quelli successivi di gestione ed estensione.

2.2 Architettura sul dispositivo

L'architettura che risiede sul dispositivo è organizzata in due macro blocchi, uno scritto in codice nativo e uno sviluppato secondo le tecnologie web.

Figura 2.2: Struttura composta da tre layer

Scendendo in dettaglio, è possibile individuare due ulteriori componenti indispensabili all'esecuzione dell'applicazione: il sistema operativo del dispositivo, e le API necessarie all'applicazione per accedere alle funzionalità tipiche di uno smartphone. Poiché le API rappresentano di fatto il ponte per la comunicazione tra sistema operativo e applicazione, è possibile raggruppare queste componenti in una struttura composta da tre livelli, come mostrato in figura 2.2.

2.2.1 Applicazione

Il primo livello analizzato è quello più alto, ovvero quello dell'applicazione vera e propria. Il grande vantaggio delle applicazioni ibride è quello di poter sfruttare l'efficienza dei linguaggi web per poter realizzare le funzionalità desiderate. Nello specifico, l'utilizzo del linguaggio HTML5 permette sia di strutturare che di visualizzare le pagine web, e consente inoltre di assolvere a compiti prettamente funzionali. L'impiego di CSS permette di gestire la formattazione e lo stile dell'applicazione, e di sfruttare le potenzialità delle *mediaqueries*. Infine, fondamentale risulta l'utilizzo del linguaggio Ja-

vaScript, che permette l'interazione con le API native e fornisce quindi un valido strumento di sviluppo della *User Interface* e della logica applicativa.

Una applicazione ibrida è realizzata con il *pattern* architetturale MVC, che contribuisce a migliorare sia l'esperienza utente che a rendere strutturalmente indipendenti moduli con funzionalità differenti, favorendo la qualità del software. Questo pattern, infatti, è basato sulla divisione dei compiti svolti dai diversi componenti dell'applicazione:

- **Model**: mette a disposizione i metodi per accedere ai dati utili dell'applicazione, e inoltre fornisce al Controller una rappresentazione univoca dei dati che l'utente ha richiesto;
- **View**: è il livello di presentazione dei dati. Fornisce l'interfaccia di interazione con l'applicazione, e permette di effettuare le richieste e visualizzarne gli esiti;
- **Controller**: dirige i flussi di interazione tra View e Model. Nello specifico, intercetta le richieste HTTP del client e traduce ogni singola richiesta in una specifica operazione per il Model; in seguito può eseguire lui stesso l'operazione oppure delegare il compito ad un'altra componente. Inoltre, seleziona la corretta vista da mostrare al client, ed inserisce eventualmente i risultati ottenuti. La sua funzione principale è quella di chiamare e coordinare le risorse per eseguire l'azione richiesta dall'utente.

Dal punto di vista della struttura, una applicazione ibrida è costituita da alcune componenti software che verranno illustare di seguito.

Wrapper Nativo

In ogni applicazione ibrida, l'involucro più esterno è costituito da ciò che viene chiamato *wrapper nativo*. Questa componente rappresenta la porzione di codice necessario affinchè l'applicazione venga scaricata, installata ed eseguita, in modo analogo alle applicazione native.

A seconda del sistema operativo preso in considerazione il wrapper può essere rappresentato da diversi elementi, ma questi ultimi sono sempre finalizzati a costruire un'applicazione il cui unico scopo è quello di istanziare una Web View. Per chiarire meglio il concetto, si prenda come esempio il sistema operativo Android; in questo caso il wrapper è composto da:

- il file AndroidManifest.xml;
- l'immagine da visualizzare nell'icona dell'applicazione;
- la Activity necessaria alla creazione e visualizzazione della Web View.

Grazie a questo contenitore, che rappresenta a tutti gli effetti una seppur minimale applicazione, è possibile sfruttare gli app store ufficiali per la distribuzione dell'applicazione ibrida.

Web View

La Web View è una componente che permette di caricare le pagine Web, ma che non include tutte le funzionalità fornite da un browser completo; sono escluse, ad esempio, i menù, le barre degli strumenti e le barre di scorrimento. Tuttavia, questa componente permette l'interazione tra il contenuto visualizzato e il codice dell'applicazione; grazie ciò, si hanno a disposizione strumenti specifici di interfaccia con il dispositivo, tipicamente richiesti in una applicazione *mobile*. La Web View, inoltre, è in grado di visualizzare contenuti Web sia in modalità online che in modalità offline. Le funzionalità messe a disposizione da Web View sono in realtà svolte dal motore di rendering chiamato *WebKit*, che verrà affrontato meglio in seguito. Il concetto di Web View rappresenta infatti solo la classe wrapper che le API di ogni piattaforma mettono a disposizione, per permettere agli sviluppatori di sfruttare le potenzialità di *WebKit*. La specifica implementazione di questa classe wrapper dipende ovviamente dal sistema operativo preso in considerazione, e può assumere anche nomi differenti (su iOS è chiamata *UIWebView*); ad ogni modo, rappresenta un elemento imprescindibile per lo sviluppo di un'applicazione ibrida.

WebKit

WebKit è un motore di rendering per browser web, ovvero un componente software che fornisce un insieme di classi che permettono di elaborare delle informazioni in ingresso, espresse tramite linguaggi di markup (come *HTML* e *XML*) e di formattazione (come *CSS*), restituendone una rappresentazione grafica. Questo componente integra funzionalità che permettono di interagire direttamente con gli eventi utente; infatti, essa implementa le caratteristiche principali di un classico browser, come ad esempio seguire

i link attivati dall'utente, gestire la lista di collegamenti *back/forward* e la cronologia delle pagine visitate. Le pagine possono essere interpretate sia sul dispositivo, sia sul Web.

WebKit nasce nel 2001, quando Apple fece un *branch* del progetto *KHTML* [8]; è sviluppato in *C++* ed è integrato nella maggioranza dei sistemi operativi *mobile* attraverso la componente Web View precedentemente descritta. Il progetto fu reso *open source* nel 2005, dando la possibilità ad ogni sviluppatore di modificarlo secondo le proprie esigenze.

La presenza di un *web engine* pressochè identico su ogni piattaforma ha favorito lo sviluppo e la diffusione delle applicazioni ibride, secondo l'architettura più comunemente usata.

2.2.2 API per l'interfacciamento e piattaforma *mobile*

Il secondo livello è costituito dalle API definite dallo sviluppatore, che permettono l'interazione tra le funzioni Javascript che definiscono la logica dell'applicazione e le funzionalità native del sistema operativo.

La necessità di un livello intermedio nell'architettura delle applicazioni ibride è dovuta al fatto che le varie piattaforme *mobile* presenti sul mercato sono sviluppate attraverso linguaggi di programmazione differenti; di conseguenza le API native, messe a disposizione degli sviluppatori dalle case costruttrici dei dispositivi, sono accessibili solo tramite i suddetti linguaggi, spesso proprietari. Ad aumentare ulteriormente l'eterogeneità dell'universo *mobile* vi sono i diversi modelli di programmazione implementati da ogni piattaforma, che caratterizzano fortemente l'approccio da seguire per lo sviluppo di applicazioni e, conseguentemente, per l'accesso alle funzionalità del dispositivo. Risulta ad esempio estremamente riduttivo affermare che un'applicazione Android si programma in Java, poichè lo sviluppo di un'applicazione richiede una conoscenza delle componenti software specifiche della singola piattaforma.

Riassumendo, questo livello risulta perciò di fondamentale importanza ai fini del funzionamento e della qualità di una applicazione ibrida.

L'obbiettivo delle API presenti in questo livello è quello di fornire da un lato un set di funzioni all'applicazione Web, richiamabili tramite Javascript, per l'accesso alle risorse del dispositivo che non vari a seconda della piattaforma ospite, dall'altro un'implementazione di tali funzioni in linguaggio

nativo, coerente con il modello di programmazione del sistema operativo in uso.

Mentre l'interfaccia pubblica di queste API, ovvero la loro definizione in Javascript, è unica, la specifica implementazione dovrà essere diversa per ogni piattaforma sulla quale si vuole portare l'applicazione. Le API dovranno quindi essere strutturate in modo da generare i medesimi effetti su ogni piattaforma; ad esempio, se si vuole far apparire una finestra per chiedere conferma di un'operazione, questa dovrà essere renderizzata con le stesse caratteristiche (dimensioni, messaggio riportato, titolo della finestra e testo dei pulsanti) indipendentemente dalle componenti software utilizzate specifiche di ogni sistema operativo.

Seguendo tale principio, le API dovranno anche garantire la restituzione di dati nel medesimo formato: ad esempio, se si volesse conoscere la propria posizione tramite coordinate GPS è plausibile pensare che diversi sistemi operativi codifichino questa informazione in formati diversi, come un array, una lista o anche un tipo appositamente definito, mentre il dato restituito alla funzione javascript chiamante dovrà essere sempre nello stesso formato.

Per poter utilizzare le API da Javascript è necessario effettuare una mappatura delle funzioni definite in linguaggio nativo con le rispettive definizioni in javascript; la specifica implementazione dipende dalla piattaforma utilizzata e viene generalmente definita *bridge*. La struttura di questo collegamento tra javascript e codice nativo deve essere strutturato in modo tale da da poter gestire qualsiasi chiamata alle API, siano quest'ultime sincrone o asincrone [14].

API Sincrone

Le API sincrone forniscono un set di metodi così come intesi nella accezione classica del paradigma ad oggetti. Secondo tale paradigma il chiamante, rappresentato dal codice Javascript, rimane bloccato fino a quando la funzione nativa chiamata non avrà completato la sua esecuzione, ed eventualmente avrà restituito un valore di ritorno.

La particolarità di questo tipo di funzioni è che, come visto in precedenza, l'intera operazione viene mediata da un elemento intermedio che permette la comunicazione tra i due diversi linguaggi, come rappresentato in figura 2.3.

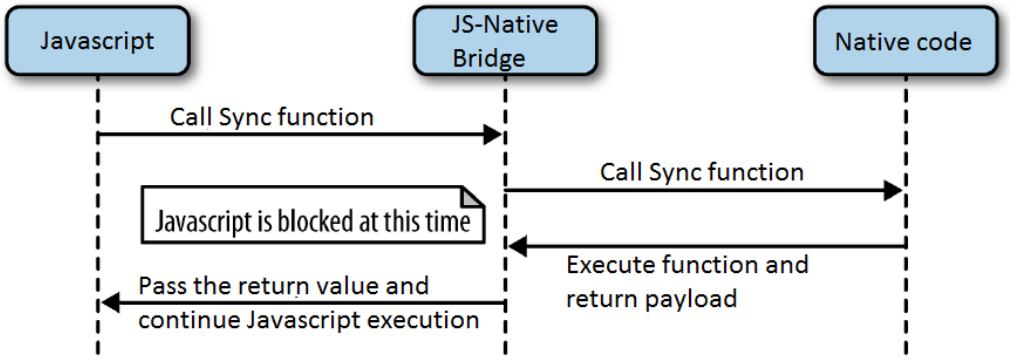

Figura 2.3: Modello di interazione delle API Sincrone

API Asincrone

Il modello asincrono prevede che l'esecuzione del codice javascript chiamante prosegua indipendentemente dallo stato di avanzamento della funzione richiamata. Questo implica la presenza di due o più flussi d'esecuzione contemporanei, e la restituzione del risultato dell'operazione tramite un modello basato su funzioni di *callback*. Uno dei principali problemi consiste, pertanto, nel riuscire a veicolare il risultato prodotto dall'esecuzione di una funzione nativa all'oggetto Javascript corretto. Per ovviare a questa problematica si può sviluppare un proprio *framework*, oppure utilizzarne uno preesistente, appositamente studiato per le funzioni asincrone in Javascript, come ad esempio *Deferred Object* presente in *jQuery*. Un'ulteriore difficoltà deriva dalla possibilità di dover gestire più callbacks allo stesso tempo; anche in questo caso i vari framework già esistenti propongo semplici soluzioni al problema. Viene riportato in figura 2.4 un esempio di utilizzo di API asincrone.

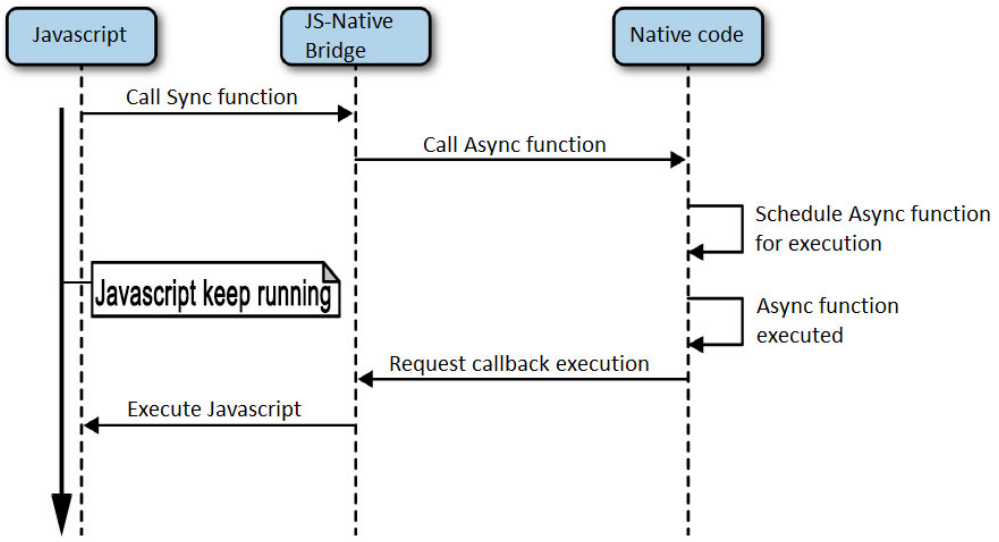

Figura 2.4: Modello di interazione delle API Asincrone

2.2.3 Interazione con la parte backend

Analizzando la parte del flusso applicativo che risiede all'esterno del dispositivo *mobile*, possiamo riconoscere alcune importanti componenti che permettono l'interazione, la comunicazione e lo scambio di risorse con il device. Fondamentale è la presenza di un *Web Service*, che determina i servizi e le funzionalità alle quali può accedere l'applicazione.

Un *Web Service* è un sistema software in grado di mettersi al servizio delle applicazioni, ed è progettato per poter consentire un'interoperabilità tra diversi dispositivi attraverso un'unica rete, quindi in un contesto distribuito. Un Web Service è in grado di offrire, insieme alla descrizione delle sue caratteristiche, un'interfaccia standard che permette alle applicazioni di usufruire dei servizi disponibili. Uno dei compiti del Web Service è interfacciarsi con uno o più *database* per il recupero dei dati richiesti dall'applicazione durante la sua esecuzione. Il protocollo di base che permette la comunicazione tra Web Service e l'applicazione è l'HTTP. HTTP ha le caratteristiche di funzionare su un meccanismo client/server e di essere un protocollo "*stateless*", ovvero senza memoria, che permette sia la ricerca che il recupero dell'informazione in maniera veloce, e permette quindi di seguire

i rimandi ipertestuali. HTTP ha quindi la particolarità di effettuare una nuova connessione al server ad ogni richiesta ricevuta che viene chiusa al termine del trasferimento dell'oggetto richiesto; questo garantisce una migliore efficienza nell'ambito web in quanto, essendo presenti numerosi 'link' ospitati su diversi server ai quali l'utente può accedere, mantiene attive solo le connessioni necessarie,chiudendo quelle non più utilizzate. Oltre al protocollo HTTP, un Web Service utilizza diversi standard web e può essere progettato con diversi stili architetturali che permettono la comunicazione con l'applicazione, tra i più importanti vi sono SOAP e REST.

SOAP

SOAP (*Simple Object Access Protocol*) è uno specifico protocollo di comunicazione che permette di scambiare informazioni in ambiente distribuito ed è basato essenzialmente su chiamate di procedura remota ed ha un comportamento tipicamente asincrono, con eventuale sincronismo imposto dal protocollo di trasporto impiegato. SOAP può operare su diversi protocolli di rete anche se HTTP rimane il più comunemente utilizzato.

Un messaggio SOAP è basato su XML ed è caratterizzato da una struttura *Header-Body*, solitamente l'header contiene le informazioni che possono servire al destinatario del messaggio, mentre il body contiene le informazioni da spedire.Come scritto in precedenza un Web Service ha un'interfaccia descritta da un formato standard creato per essere leggibile da qualsiasi dispositivo. Questo formato è il WSDL, *Web Service Description Language*, un formato XML utilizzato per descrivere servizi sulla rete, che viene utilizzato per permettere ai client che vogliono utilizzare il Web Service di conoscere tutte le caratteristiche del Web Service stesso. Questa è un'ulteriore evidenza del tentativo di adattare al Web l'approccio di interoperabilità basato su chiamate remote.

REST

REST (*Representational State Transfer*) è uno stile architetturale per sistemi software distribuiti che permette la manipolazione delle risorse per mezzo dei metodi *GET*, *POST*, *PUT* e *DELETE* del protocollo HTTP. Un concetto fondamentale in REST è quello di risorsa, ovvero una qualunque entità che possa essere indirizzabile tramite Web, in particolare univocamente individuata da un identificatore globale (*URI*), ed accessibile e trasferibile

tra client e server. REST utilizza il protocollo HTTP che, essendo "*stateless*", implica che ogni richiesta debba essere totalmente indipendente dalle precedenti; in tal modo il messaggio dovrà contenere tutte le informazioni necessarie affinché il server possa processarlo e fornire una risposta corretta. REST non prevede esplicitamente nessuna modalità per descrivere come interagire con una risorsa, infatti le operazioni sono implicite nel protocollo HTTP. Qualcosa di analogo a WSDL è WADL, *Web Application Definition Language*, un'applicazione XML per definire risorse, operazioni ed eccezioni previsti da un Web Service di tipo REST.

Anche se l'approccio di questi stili architetturali è il medesimo, ovvero l'utilizzo del Web come piattaforma di elaborazione, la sostanziale differenza deriva dal fatto che mentre SOAP evidenzia il concetto di servizio ed è stato realizzato esplicitamente per utilizzare HTTP esclusivamente come protocollo di "trasporto", REST pone l'attenzione sul concetto di risorsa e sfrutta appieno le potenzialità, la semantica e le funzionalità del protocollo applicativo HTTP. Inoltre è importante precisare che REST rappresenta lo stile architetturale di riferimento per la realizzazione delle applicazioni in ambito *mobile*, questo motivato dal fatto che lo stack di protocolli utilizzato da SOAP risulta eccessivamente pesante per le caratterics tiche hardware dei dispositivi *mobile*.

Modello Sincrono e Asincrono

Il protocollo HTTP utilizzato per la comunicazione tra applicazione e Web Service è tipicamente sincrono, in quanto utilizzando un meccanismo client-server, il client dopo l'invio della richiesta rimane in attesa, aspettando la relativa risposta. Queste comunicazioni sono appunto dette sincrone e sono caratterizzate da un'applicazione client che invia una richiesta verso un Web Service e blocca il proprio *thread* chiamante fino alla ricezione della risposta. Pertanto, l'applicazione client può continuare la propria esecuzione solo dopo aver ricevuto una risposta o un *time-out*. Per questo motivo, le comunicazioni sincrone sono anche definite bloccanti e possono risultare limitanti a seconda dell'applicazione che si vuole realizzare.

Le comunicazioni asincrone, al contrario, adottano un approccio noto come "*Fire and Forget*", ovvero, l'applicazione client invia una richiesta e poi continua con la propria elaborazione ignorandone lo stato di avanzamento. Pertanto, in questo caso il *thread* che ha avviato la richiesta non

viene bloccato in attesa che il Web Service risponda alla richiesta. La logica conseguenza è che le applicazioni presentano un elevato grado di agilità dell'applicazione eliminando i tempi di attesa. Un possibile approccio per la realizzazione di chiamate asincrone è l'utilizzo di un componente fondamentale della tecnologia di sviluppo Ajax, l'elemento *XMLHttpRequest*. *XMLHttpRequest* permettere la realizzazione di chiamate asincrone, consentendo all'utente di svolgere diversi compiti contemporaneamente all'interno di una stessa pagina web senza dover attendere la risposta dal server e i tempi di refresh. L'utilizzo di questo elemento permette quindi all'utente di effettuare diverse richieste simultanee al server, che saranno completamente indipendenti e potranno essere eseguite attraverso l'utilizzo di Javascript. La rappresentazione dei due modelli di comunicazione è mostrata in figura 2.5.

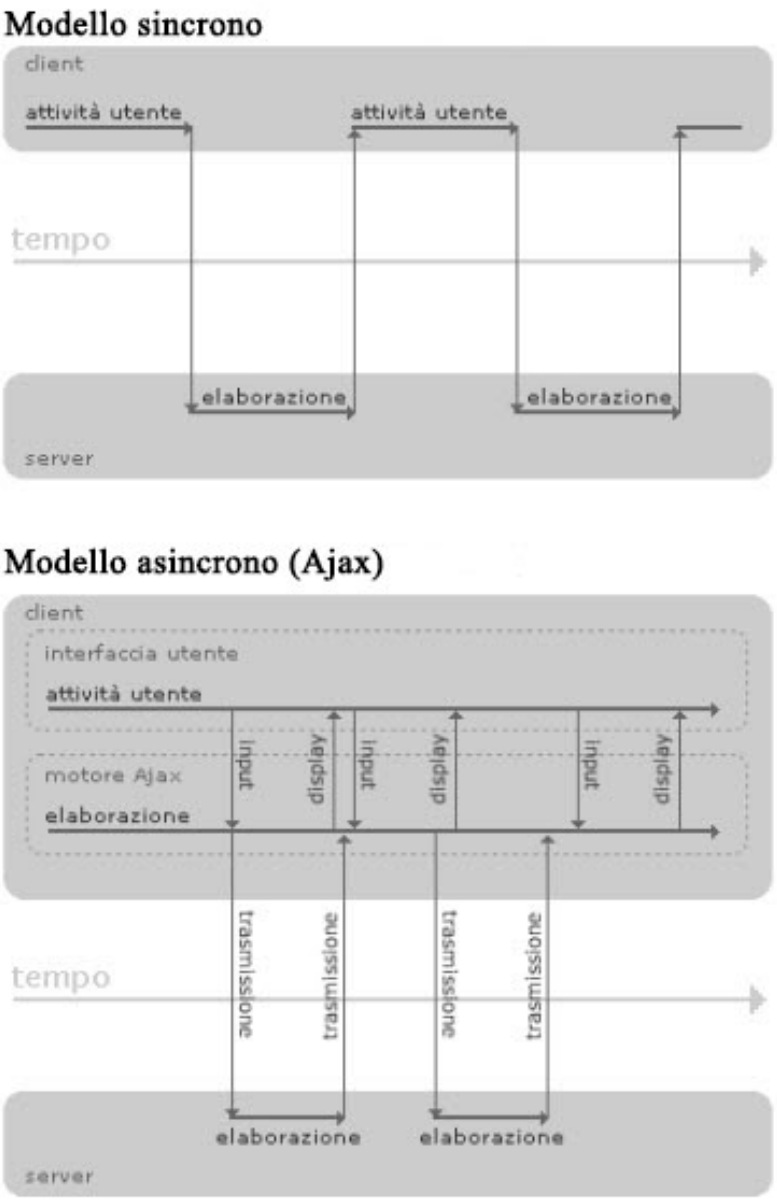

Figura 2.5: Esempio di modello sincrono e asincrono

2.3 Confronto tra diverse architetture

Per meglio comprendere il concetto che vi è alla base delle applicazioni ibride, può risultare interessante effettuare un confronto con le altre due tipologie di applicazioni esistenti in ambito *mobile*.

Mentre nel primo capitolo è stato effettuato un paragone basato più sulle funzionalità e sugli aspetti positivi e negativi che i diversi approcci garantiscono, in questo paragrafo si effettuerà un confronto basato sulle differenze tra le architetture, confrontando le applicazioni ibride prima con quelle native e successivamente con quelle web.

2.3.1 Confronto con applicazioni native

Analizzando l'architettura tipica di un'applicazione nativa, si possono distinguere immediatamente due differenze sostanziali rispetto alla soluzione ibrida.

In primo luogo le applicazioni native non fanno uso dei linguaggi standard per il Web, ma solo del codice nativo relativo alla piattaforma utilizzata. Questa caratteristica permette di eliminare l'utilizzo di una Web View sia per quanto riguarda l'esecuzione della *business logic*, in quanto non più espressa tramite Javascript, sia per quanto riguarda la visualizzazione della *user-interface*, che verrà implementata tramite l'utilizzo di apposite componenti *software* presenti nelle API del sistema operativo preso in considerazione.

L'altra differenza fondamentale è rappresentata dalla mancanza del livello costituito dalle API implementate dallo sviluppatore per avere la possibilità di sfruttare le funzionalità native del sistema operativo, in quanto se ne ha già il pieno controllo. Proprio l'assenza di questa componente intermedia, che permette l'accesso diretto all'*hardware* del dispositivo, garantisce una *user-experience* ottimale in grado di ridurre i tempi di reattività e migliorare in modo evidente le prestazioni grafiche.

Per quanto riguarda l'interazione con la parte di *backend*, il volume delle informazioni scambiate è generalmente inferiore, in quanto l'esecuzione dell'applicazione è effettuata in locale e la comunicazione con un Web Service è utilizzata esclusivamente per la sincronizzazione di dati dinamici che necessitano di aggiornamenti. L'architettura propria di un'applicazione nativa è mostrata in figura 2.6

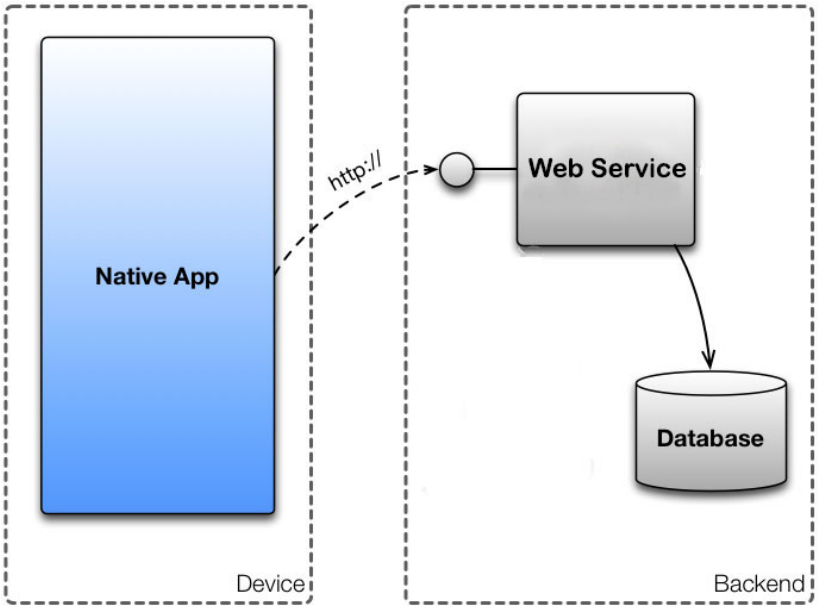

Figura 2.6: Native App Architecture

2.3.2 Confronto con applicazioni web

Nel caso delle applicazioni web, a differenza delle applicazioni ibride, l'intera *business logic* è situata su un Web Server. In questo caso il browser web installato sul dispositivo assume il ruolo di client dell'applicazione, rendendo perciò non più necessaria la presenza del wrapper nativo e consentendo l'accesso unicamente attraverso un indirizzo *URL*. Questo determina, per il dispositivo, uno sgravo completo dal carico computazionale ma nello stesso tempo comporta una totale dipendenza dalla rete per il complessivo funzionamento dell'applicazione.

Mentre l'applicazione appare completamente diversa da un punto di vista della struttura, intesa come componenti *software* utilizzati, essa rimane pressochè invariata per quanto riguarda il codice applicativo che continua ad esser espresso tramite HTML, CSS e Javascript. L'architettura propria di un'applicazione web è mostrata in figura 2.7

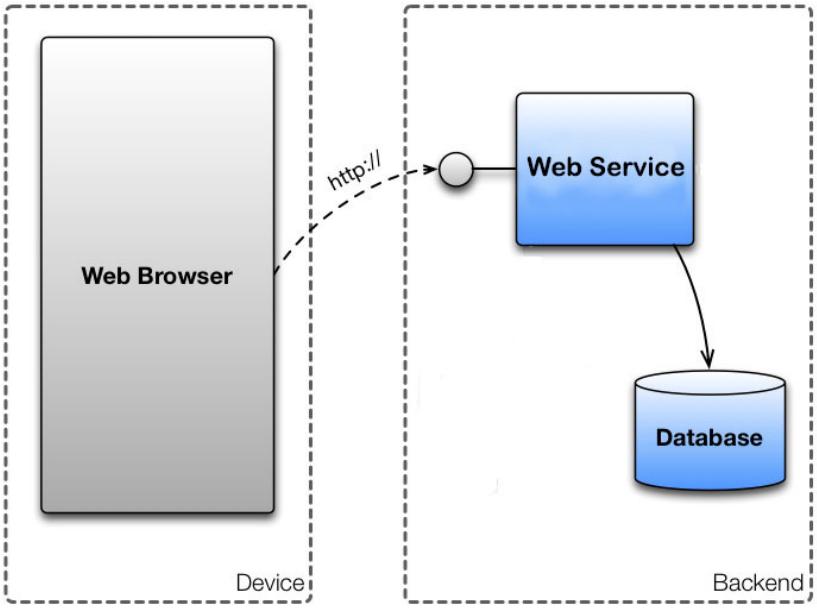

Figura 2.7: Web App Architecture

Capitolo 3

Approcci per lo sviluppo

In relazione all'architettura delle applicazioni ibride, descritta nel precedente capitolo, si pongono ora in evidenza gli strumenti di sviluppo atti a realizzare sia le funzionalità e le caratteristiche dell'interfaccia utente sia la logica dell'applicazione.

3.1 HTML5

HTML è nato nel 1990 grazie a uno scienziato, Tim Berners-Leecome, che sviluppò un linguaggio di *markup* in grado di strutturare pagine web [3]. Il suo utilizzo consiste nell'inserimento nel testo di marcatori, detti *tag*, costituiti da parentesi angolate, che permettono l'impaginazione e la gestione dei contenuti di una pagina web, per esempio specificandone la struttura grafica o le dimensioni. La Web View elabora ed interpreta il codice sorgente della pagina HTML così realizzata, riconoscendone i *tag* e la loro funzione, al fine di generare la visualizzazione per l'utente della pagina desiderata.

La comparsa della multimedialità sul palcoscenico di Internet, dove le pagine web non erano più formate esclusivamente da testo e immagini statiche, portò il processo di specifica, tutt'ora in corso, verso uno sviluppo più orientato alle applicazioni web che ai documenti, fino a giungere ai giorni nostri con l'eccellente risultato ottenuto da HTML5.

Alcuni elementi fondamentali, introdotti con l'ultima versione, sono descritti in seguito, in quanto essenziali per lo sviluppo di un'applicazione ibrida.

3.1.1 Canvas

Canvas rappresenta un'estensione dell'HTML, inizialmente sviluppata da Apple per ampliare le funzionalità di *MacOs X WebKit*. Questa *feature* permette di disegnare dinamicamente elementi grafici di tipo *bitmap*, creando uno spazio all'interno di una pagina web, al quale il codice Javascript può accedere con diversi set di API. Gli usi possono spaziare tra grafici, animazioni, giochi e composizioni di immagini.

Per creare un Canvas è sufficiente aggiungere il codice seguente alla pagina HTML :

```
<canvas id="myCanvas" width="578" height="200"></canvas>
```

Il *tag* ¡canvas¿ crea un contenitore che necessita di un linguaggio di scripting di supporto, come Javascript, per funzionare correttamente e sfruttare totalmanete le sue potenzialità. La seguente funzione Javascript, ad esempio, consente la creazione di un cerchio :

```
function circle(){
    var canvas = document.getElementById('myCanvas');
    var context = canvas.getContext('2d');
    var centerX = canvas.width / 2;
    var centerY = canvas.height / 2;
    var radius = 70;

    context.beginPath();
    context.arc(centerX, centerY, radius, 0, 2 * Math.PI,
      false);
    context.fillStyle = 'green';
    context.fill();
    context.lineWidth = 5;
    context.strokeStyle = '#003300';
    context.stroke();
}
```

Oltre a elementi di tipo bitmap, HTML5 mette a disposizione anche il formato SVG, acronimo di *Scalable Vector Graphics*, che fornisce uno standard per la rappresentazione di immagini vettoriali. Rispetto alle immagini bitmap si evidenzia una sostanziale differenza, in quanto, mentre su Canvas è necessario un approccio attraverso istruzioni JavaScript che pilotano la creazione del disegno, con il formato SVG ci si trova di fronte ad una

descrizione dell'immagine attraverso il linguaggio XML, come nell'esempio seguente, che rappresenta un rettangolo rosso:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
   version="1.1"
   width="200"
   height="100">
<rect fill="red" x="10" y="10" width="180" height="80" />
</svg>
```

L'uso della sintassi XML fa sì che il documento possa essere trattato con molteplici strumenti, anche più evoluti, oppure con designer dedicati alle immagini vettoriali, come Adobe Illustrator, Corel Draw o Open Office.

3.1.2 Offline e Storage

L'HTML5 si prefigge l'obiettivo di mettere a disposizione strumenti che permettano la creazione di applicazioni ibride con funzionalità avanzate, che consentano non solo la consultazione di documenti ma che realizzino funzionalità più complesse. A questo scopo, vengono rese disponibili alcune tecnologie, descritte in seguito, che rendono possibile una maggior efficienza e completezza di questa tipologia di applicazioni.

Local Storage

L'utilizzo del *Local Storage* rappresenta una grande risorsa per le applicazioni ibride, in quanto offre l'opportunità di effettuare il salvataggio di dati in locale e in modo permanente, consentendone l'accesso anche se non connessi alla rete. Questo tipo di memorizzazione, caratterizzata da un accesso esclusivamente locale e da una disponibilità di memoria almeno di 5 Mb, sostituisce l'utilizzo dei *cookies* che permettono, al contrario, un accesso anche da parte del server con capacità di salvataggio minime.

Il meccanismo su cui si basa il *Local Storage* è una struttura chiave-valore, il cui utilizzo è riportato nel seguente esempio:

```
//salvataggio di un valore associato ad una chiave
localStorage.setItem('key', 'value');
```

Inoltre *Local Storage* rende disponibile la proprietà *length*, che consente di recuperare il valore di tutte le chiavi memorizzate attraverso indi-

ci sequenziali permettendo, con un semplice ciclo, di scorrere tutti i dati salvati.

```
//recupero della lista di valori, a partire dalle chiavi
for (i=0; i<=localStorage.length-1; i++) {
  key = localStorage.key(i);
  val = localStorage.getItem(key);
}
```

Questo meccanismo non è il solo messo a disposizione da HTML5 per effettuare il salvataggio dei dati in locale: le API che mettono a disposizione l'interfaccia *Storage* sono implementate anche da *SessionStorage*, una soluzione analoga a quella precedentemente descritta, con la differenza sostanziale rappresentata dalla memorizzazione dei dati in modo non permanente. Infatti, i dati, sono dipendenti dalla singola sessione, la cui chiusura ne determinerà la perdita.

Application Cache

Questa funzionalità permette l'uso dell'applicazione anche nel caso in cui non ci sia una connessione disponibile, consentendo di salvare in un'apposita *cache* i files necessari. Il funzionamento base dell'*Application Cache* è piuttosto semplice; si crea un file *manifest* formato da tre sezioni:

- **CACHE**: nel quale si specifica la lista dei file che devono essere memorizzati e mostrati nel caso di un utilizzo offline;
- **NETWORK**: in questa sezione vengono elencati i file che necessitano di una connessione e per i quali il salvataggio nella *cache* non determinerebbe l'uso corretto in modalità offline. L'eventuale utilizzo del simbolo "*"indica che tutti i file non specificati nella sezione CACHE necessitano di connessione alla rete per essere utilizzati;
- **FALLBACK**: viene specificato cosa visualizzare quando si tenta l'accesso ad un file non memorizzato nella *cache* durante l'uso offline dell'applicazione.

```
CACHE MANIFEST
CACHE:
  index.html
```

```
/images/logo.png
/css/styles.css
/js/main.js

NETWORK:
*

FALLBACK:
/ /offline.html
```

Infine, ogni pagina presente nella sezione CACHE dovrà contenere al suo interno il *tag* <html>così specificato:

```
<html manifest="offline.manifest">
```

Uno dei vantaggi nell'uso di questa tecnologia, oltre all'incremento di velocità di fruizione, è la possibilità di salvare pagine e risorse anche nel caso non siano state visitate dall'utente.

3.1.3 Geolocation

Con il termine *geolocation* si descrivono quei meccanismi che permettono di stabilire la posizione fisica di un individuo sul globo terrestre con la possibilità opzionale di condividere tale informazione. Le *Geolocation* API, messe a disposizione da HTML5, consentono ad una applicazione di ottenere la posizione geografica dell'utente che la sta utilizzando. Se in passato ci si serviva dell'indirizzo IP per determinare la posizione di un device, le API di geolocalizzazione sono noncuranti riguardo la sorgente utilizzata, che può essere quindi un segnale GPS, una triangolazione basata sulle celle UMTS/GSM oppure altre possibili tecniche; ovviamente da ogni tecnica utilizzata dipende l'accuratezza della posizione. Le *Geolocation* API, quindi, forniscono una struttura dati atta a contenere vari dati geospaziali e impostare alcune funzioni di accesso ai dati stessi.

La richiesta di localizzazione geografica dev'essere accettata e autorizzata dall'utente, che può decidere o meno di comunicare i propri dati; solo dopo l'approvazione della comunicazione verrà eseguita una funzione di *callback* che darà la possibilità di accesso e di manipolazione dei dati ricevuti.

Le *Geolocation* API, consentono di acquisire i dati con due modalità:

- lettura singola: viene utilizzato il metodo *getCurrentPosition()* passando come parametri un *handler* per gestire la ricezione dei dati ed un

altro per gestire eventuali errori. In questo caso il il dato riguardante la posizione viene fornito un'unica volta;

- campionamento continuo: si utilizza il metodo *watchPosition()*, con gli stessi parametri della modalità precedente, che monitora la posizione corrente iniziando un processo asincrono di controllo, occupandosi di aggiornare i valori ritornati dal metodo ogni qualvolta la posizione cambi. Questa funzione può essere fermata con il metodo *clearWatch()*.

L'utilizzo di queste specifiche è illustrato nel seguente esempio:

```
function onDeviceReady() {
  navigator.geolocation.getCurrentPosition(onSuccess,
    onError);
}
```

Dopo aver acquisito il valore di ritorno si possono visualizzare i dati relativi alla posizione:

```
function OnSuccess(position) {
  alert("Latitude: " + position.coords.latitude
    + ", Longitude: " + position.coords.longitude);
}
```

3.1.4 Web Workers

L'uso dei *Web Workers* consente l'esecuzione di porzioni di codice Javascript in modo asincrono, senza che questi gravino sulle prestazioni dell'interfaccia grafica, in modo che l'utente possa continuare l'utilizzo dell'applicazione anche durante l'esecuzione di *script* particolarmente pesanti. I *Web Workers* sfruttano le capacità *multicore*, ormai presenti in tutti gli smartphone di ultima generazione, permettendo la creazione di nuovi *threads* con i quali la pagina può comunicare attraverso semplici metodi.

```
/*costruttore che specifica lo script da eseguire
* sul Worker */
var myWorker = new Worker("count.js")
```

Il *Worker* e la pagina potranno comunicare attraverso due modalità:

- *postmessage()*: è utilizzato per l'invio di un messaggio e permette di trasmettere un parametro come una stringa, un oggetto o un numero;
- *onmessage*: è un *event-listener* per la ricezione di un messaggio inviato con la funzione *postMessage()*.

Nel seguente esempio, supponiamo di avere un file "count.js"in cui si definisce un contatore che invia il suo valore e si incrementa in base ad uno specifico intervallo di tempo:

```
var i=0;
function timedCount(){
  i=i+1;
  postMessage(i);
  setTimeout("timedCount()",500);
}
timedCount();
```

Mentre nella pagina HTML viene definito il seguente codice, che permette di inizializzare un *Web Worker* il cui compito è quello di eseguire il file "*count.js*"e restituirne i risultati:

```
<output id="result"></output>
var myWorker;
//viene inizializzato il Web Worker
function startWorker(){
  if(typeof(myWorker)=="undefined"){
    myWorker=new Worker("count.js");
  }
/*funzione che permette di catturare l'evento inviato da
* postMessage() presente nel file "count.js" e di restituirne
* i risultati */
  myWorker.onmessage = function (event) {
    document.getElementById("result").innerHTML=event.data;
  };
}

//funzione che permette di fermare il Web Worker
function stopWorker(){
  myWorker.terminate();
}
```

3.1.5 Server-Sent Events

La *Server-Sent Events* API è una libreria che permette di delegare alla Web View il monitoraggio di una sorgente di dati e di ottenere notifiche *push* dal Web Service ad ogni nuovo evento. Si avvale del protocollo standard HTTP per la comunicazione e permette esclusivamente lo scambio di dati in modo unidirezionale, dal Web Service all'applicazione, rappresentando un'ottima alternativa ai classici meccanismi di *polling*.

Nell'esempio di seguito si può chiarire il funzionamento di questo strumento. In particolare si ipotizza di avere il seguente file "*server_sent.asp*"sul server con il quale l'applicazione comunica:

```
//funzione che comunica data e ora
<%
Response.ContentType="text/event-stream"
Response.Expires=-1
Response.Write("data: The server time is:" & now())
Response.Flush()
%>
```

Viene creato l'oggetto *EventSource*, specificando la pagina che dovrà inviare gli aggiornamenti, ogni volta che sarà ricevuto un aggiornamento questo sarà catturato dal metodo *onmessage* che ne mostrerà il risultato.

```
<div id="result"></div>
function data_server(){
  var source=new EventSource("server_sent.asp");
  source.onmessage=function(event){
    document.getElementById("result").innerHTML
      +=event.data + "<br>";
  };
}
```

3.2 CSS3

Il *Cascading Style Sheet* o CSS è un linguaggio che definisce lo stile, la formattazione e l'aspetto dei contenuti presenti in un documento scritto in un linguaggio di markup, per questo è spesso associato a pagine web HTML ma può essere applicato anche ad ogni tipo di documento XHTML e XML [2].

Sfruttando i fogli di stile è possibile modificare la struttura di diverse pagine agendo su un unico file esterno che ne determina l'aspetto, consentendo di definire i font, i colori, le immagini di sfondo, il layout, il posizionamento di elementi sulla pagina e permettendo di mantenere separato il contenuto dalla presentazione. Di seguito analizziamo gli aspetti più interessanti che permettono di creare un'interfaccia grafica nelle applicazione ibride che consenta una *user-experience* simile a quella nativa.

3.2.1 Transition

Il modulo CSS3 *Transition* mette a disposizione degli sviluppatori una serie di tecniche e proprietà che consentono di realizzare effetti di transizione, modificando i valori CSS e specificando la durata dell'effetto di transizione, senza aver bisogno di script esterni. Un'effetto che permettono di gestire è la transazione tra la *view* in entrata e la *view* in uscita dallo schermo. Nelle applicazioni native questa è una tipica funzione che permette di prevenire l'effetto "schermo-bianco"che si avrebbe nel momento in cui una pagina viene tolta dalla memoria ed una nuova viene caricata. Questo può essere realizzato, grazie a questa *feature*, anche nella applicazioni ibride, come nell'esempio risportato di seguito:

```
#view {
-webkit-transition-duration: 800ms;
-webkit-transform: translate3d(0,600px,0);
position: absolute; left: 0; top: 0;
}
```

Tramite funzioni in Javascript, dopo opportuni eventi, si avvia la transition invocando:

```
function showView() {
  x$("#view").css({
    "-webkit-transform":"translate3d(0,0,0)",
    "-webkit-transition-timing-function":"ease-in"
  });
}

function hideView() {
  x$("#view").css({
    "-webkit-transform":"translate3d(0,600px,0)",
```

```
    "-webkit-transition-timing-function":"ease-out"
  });
}
```

3.2.2 Animation

Il modulo *Animation*, facente parte di CSS3, risponde alla necessità di realizzare animazioni, su dispositivi che non hanno la potenza di calcolo e l'autonomia di un computer tradizionale e per i quali il consumo di risorse deve necessariamente ridursi al minimo. Si tratta di qualcosa di più complesso e avanzato rispetto alle transizioni perché entra in gioco l'idea di animazione sviluppata su una sorta di timeline a partire da una serie di *keyframe*, che permettono la realizzazione di animazioni senza l'utilizzo di Javascript o Flash. Le animazioni CSS3 si basano sull'introduzione di una nuova direttiva, chiamata *@keyframes*, nella quale si dovranno specificare le caratteristiche fondamentali dell'animazione, prima di definire le proprietà CSS della stessa. Un esempio di possibile implementazione che permette di creare un'animazione di un quadrato che cambia sia colore che posizione, è riportato in seguito:

```
/* attributi che permettono di specificare caratteristiche
* dell'elemento di animazione e dell'animazione stessa */
div
{
width:100px;
height:100px;
background:red;
position:relative;
-webkit-animation-name: myAnimation;
-webkit-animation-duration: 5s;
-webkit-animation-timing-function: linear;
-webkit-animation-delay: 2s;
-webkit-animation-iteration-count: infinite;
-webkit-animation-direction: alternate;
-webkit-animation-play-state: running;
}

//permette di specificare come deve avvenire l'animazione
@-webkit-keyframes myAnimation
```

```
{
0%   {background:red; left:0px; top:0px;}
25%  {background:yellow; left:200px; top:0px;}
50%  {background:blue; left:200px; top:200px;}
75%  {background:green; left:0px; top:200px;}
100% {background:red; left:0px; top:0px;}
}
```

3.2.3 Media Queries

La diffusione di dispositivi con proprietà sempre più eterogenee ha richiesto lo sviluppo di uno strumento che potesse permettere di creare fogli di stile specifici in base sia ai metodi di fruizione dei contenuti sia alle diverse caratteristiche dei vari dispositivi : la *media query*. Questa funzionalità ci permette di adattare lo stile CSS alle più svariate risoluzioni e personalizzare la resa dell'interfaccia a seconda del dispositivo sul quale si sta visualizzando la pagina. L'utilizzo di questo strumento, consiste nella dichiarazione di un tipo di *media* nel quale aggiungere uno o più argomenti che consentono di applicare le regole del foglio di stile solo se si verificano determinate condizioni.

Con il termine "*media*"vengono indicati tutti i tipi di dispositivi che permettono una forma di visualizzazione, come schermi, proiettori e stampanti o anche dispositivi meno comuni come supporti braille. In sostanza una *media query* è un'espressione logica, con la possibilità di utilizzo di operatori come *and*, *not* e *only*, che può essere soddisfatta o meno. Si immagina di voler definire delle regole di visualizzazione come dimensione del carattere, colore dello sfondo e larghezza della pagina, a seconda del dispositivo e delle dimensioni dello schermo :

```
@media screen and (device-width: 960px)
  and (device-height: 640px) {
    body {
        font-size: 18px;
        width: 960px;
        background-color: #ff33cc;
    }
}
@media all and (min-width: 480px) and (max-width: 1024px) {
```

```
  body {
    font-size: 14px;
    width:1000px;
    background-color: #ffff33;
  }
}
@media screen and (orientation: landscape) {
  body {
    font-size: 12px;
    background-color: #ff0000;
  }
}
```

Queste funzionalità permettono di realizzare una tecnica chiamata Responsive Web Design, che consente la realizzazione di pagine web, che adattino in automatico il proprio *layout* al supporto utilizzato, fornendo di conseguenza una visualizzazione ottimale su qualsiasi dispositivo.

3.3 Javascript

Nel 1995, *Netscape* decise di dotare il proprio browser web di un linguaggio che permettesse l'interazione con oggetti istanziati in una pagina web. Fu sviluppato, da Brendan Eich, della *Netscape Communications*, un linguaggio di scripting orientato agli oggetti che originariamente fu chiamato *Mocha*, successivamente *LiveScript*. Infine fu rinominato in Javascript, quando Netscape incluse nel browser *Netscape Navigator* il supporto per la tecnologia Java, con la quale Javascript non ha nessuna relazione se non una somiglianza sintattica [5]. Attraverso il paradigma *object oriented*, Javascript, permette di istanziare in una pagina web, oggetti in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi. Il linguaggio Javascript è interpretato da un componente lato-client, nel caso specifico di applicazione ibride, dalla Web View e può essere integrato in una pagina HTML senza avere la necessità di nessuna compilazione. Questo linguaggio si occupa di tutta la parte di logica applicativa presente in un'applicazione ibrida, permettendo l'inserimento di contenuto eseguibile che consente di compiere azioni e di interagire con l'utente, tali funzionalità sarebbero normalmente non realizzabili con l'HTML statico.

3.3.1 AJAX

Come già anticipato nel capitolo precedente, Ajax è una tecnica di sviluppo software *cross-platform* che permette uno scambio di dati in *background* tra server e client senza interferire con il comportamento della pagina e senza un esplicito *refresh* della stessa da parte dell'utente. L'oggetto specifico alla base di questa funzionalità è *XMLHttpRequest*, che permette l'esecuzione di richieste asincrone con le quali rimane in contatto grazie ad un gestore di eventi. Questa funzione di "*callback*" viene invocata ogni volta che avviene un cambiamento di stato come invio di una risposta o termine della ricezione. Il metodo che permette la creazione di una connessione di richiesta asincrona è *open()* che accetta come parametri: il metodo di invio o ricezione dei dati (GET o POST), l'URL della risorsa che si richiede, il tipo di richiesta se asincrona (true) o sincrona (false), username e password nel caso la risorsa sia protetta [1].

L'utilizzo di questo oggetto può essere riassunto in tre semplici passaggi:

- Impostazione dell'oggetto per eseguire una richiesta;
- Invio della richiesta e della quale viene supervisionato lo stato, utilizzando le proprietà a disposizione;
- Completamento della richiesta ed estrazione delle informazioni di ritorno dall'oggetto *XMLHttpRequest*.

Nell'esempio di seguito si chiarisce meglio l'utilizzo di questa funzionalità:

```
//inizializzazione dell'elemento XMLHttpRequest
var myRequest = new XMLHttpRequest();
//monitoraggio dei cambiamenti di stato
myRequest.onreadystatechange = function() {
  alertContents(myRequest);
};
/*apertura della connessione specificando la modalità
* asincrona */
myRequest.open("GET","demo_get.asp",true);
//invio della richiesta
myRequest.send();
```

3.3.2 DOM e gestione eventi

Il DOM (*Document Object Model*) è un'API, indipendente da qualsiasi piattaforma, che rappresenta lo standard W3C per descrivere la struttura di un documento HTML (e XML), con il quale gli sviluppatori di pagine web possono accedere e manipolare tutti gli elementi della pagina stessa. E' importante specificare che di seguito verrà trattato DOM in relazione al linguaggio Javascript, in quanto rappresenta uno dei modi per accedervi. Il DOM rappresenta una generica pagina web come un albero, considerando un nodo per ogni tag del documento e per ogni nodo tanti nodi-figlio quanti sono i tag nidificati all'interno dello stesso. Tramite il DOM sarà possibile accedere e manipolare ogni nodo, aggiungerne di nuovi dinamicamente ed eliminarne altri già presenti.

Per capire più precisamente il funzionamento di questo strumento è utile introdurre l'oggetto *document*, che rappresenta l'insieme di tutti gli elementi presenti in una pagina e che ci permette di accedere agli elementi stessi attraverso due metodi fondamentali :

- *getElementById()*: questo metodo permette di recuperare l'elemento caratterizzato univocamente dal valore del proprio attributo ID. In particolare restituisce un riferimento all'elemento in questione. La sintassi è:

  ```
  document.getElementById(ID_elemento)
  ```

- *getElementsByTagName()*: permette di recuperare l'insieme degli elementi caratterizzati dallo stesso tag. In particolare il valore di ritorno è rappresentato da un array di tutti gli elementi del tag considerato, nell'ordine in cui compaiono nel codice della pagina. La sintassi è:

  ```
  document.getElementsByTagName(nome_TAG)
  ```

Il fondamentale utilizzo dell'API DOM, riguarda la gestione degli eventi, che rappresentano una proprietà estremamente importante di un'applicazione *mobile*. Le due modalità più semplici di gestione degli eventi sono:

- attraverso markup: ciò consiste nello specificare la funzione da eseguire direttamente nella dichiarazione dell'oggetto del quale si vuole intercettare l'evento, come nell'esempio di seguito:

```
<input id="btnHello" type="button" value="Hello"
  onclick="alert('Hello World!');" />
```

- attraverso codice Javascript: è possibile associare ad un evento di uno specifico oggetto l'esecuzione di specifica funzione, come nell'esempio seguente:

```
function Hello(){
    alert("Hello World!");
}
document.getElementById("btnHello").onclick = Hello;
```

Considerando l'importanza e la complessità che la gestione degli eventi può assumere nella creazione di una applicazione *mobile* ed in particolare in una applicazione ibrida, è bene adottare un pattern che permetta di organizzare il codice in modo più ordinato ed efficiente. Di seguito verranno illustrati alcuni dei pattern più utilizzati per la gestione di eventi in ambito *mobile*.

Event-Listener

Il meccanismo che permette di effettuare un'azione in seguito ad un evento, consiste nel registrare un ascoltatore per quell'evento specifico. Tale ascoltatore di occuperà di intercettare l'evento ed eseguire la funzione associata. Il pattern che permette questa realizzazione è detto *event-listener* che, come da specifica W3C, è il modo corretto per registrare un listener. Questo approccio infatti permette di aggiungere diversi *handler* per singolo evento, consente il funzionamento con qualunque elemento del DOM, non solo con gli elementi HTML e conferisce un controllo migliore di ciò che accade all'attivazione di un listener.

Per chiarire meglio il funzionamento degli eventi DOM riportiamo di seguito un esempio in cui si voglia eseguire del codice dopo l'utente abbia premuto un bottone, l'evento "ascoltato"è *click* :

```
document.getElementById("btnHello").addEventListener("click",
  function () {alert("Hello World!");},
  false);
```

Event Pooling

Il pattern Event-Listener per il suo funzionamento richiede una dipendenza tra oggetto e osservatore, questo può comportare un problema nel caso in cui si debbano gestire numerosi eventi e altrettanti riferimenti. L'*Event Pooling* è una semplice variante del pattern Event-Listener che utilizza una componente software come intermediario tra osservatore e oggetto, che permette di coordinare gli eventi e le funzioni ad essi associate. Nel caso in cui ci siano un certo numero di osservatori e di oggetti generatori di eventi, i primi dovranno registrarsi all'*Event Pool* specificando l'evento che vogliono intercettare, mentre gli oggetti dovranno comunicare all'Event Pool che un determinato evento si è verificato. Pertanto sia l'osservatore che il soggetto necessitano esclusivamente di un riferimento all'Event Pool.

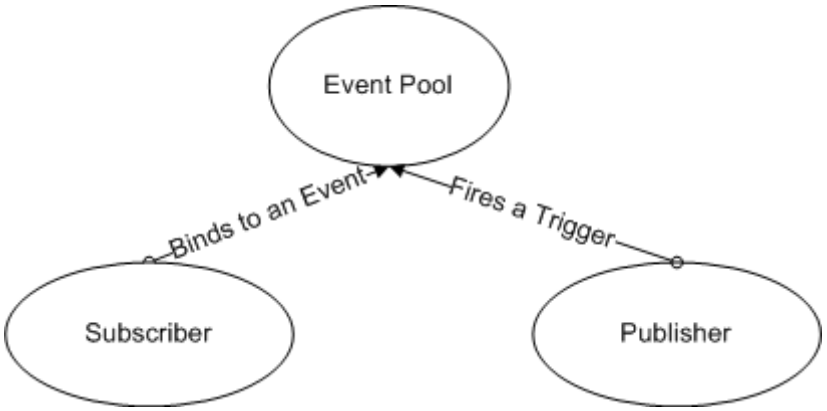

Figura 3.1: Modello di funzionamento del pattern Event Pooling

Per chiarire meglio il concetto si riporta un esempio, realizzato tramite JQuery, dell'utilizzo di questo pattern:

```
/* si definisce la funzione da eseguire dopo aver
* intercettato l'evento */
function HelloWorld(){
    alert("Hello World!");
}

//si registra l'osservatore per l'evento 'HELLO'
$(document).bind('HELLO', function() {
```

```
    HelloWorld();
});

//si definisce l'oggetto che genera l'evento 'HELLO'
  <input id="btnHello" type="button" value="Hello"
    onclick="javascript:$(document).trigger('HELLO');" />
```

3.4 Model-View-Controller

Dopo aver illustrato i molteplici strumenti che permettono di sviluppare le applicazione ibride, possiamo chiarire i ruoli che esse hanno nel concetto di pattern Model-View-Controller introdotto precedentemente. Gli elementi HTML rappresentano gli oggetti principali all'interno di una pagina web, le proprietà CSS possono essere pensate come gli attributi di rendering degli oggetti, mentre JavaScript svolge la funzione di rapportarsi all'utente, rendendo dinamiche struttura e proprietà degli oggetti.

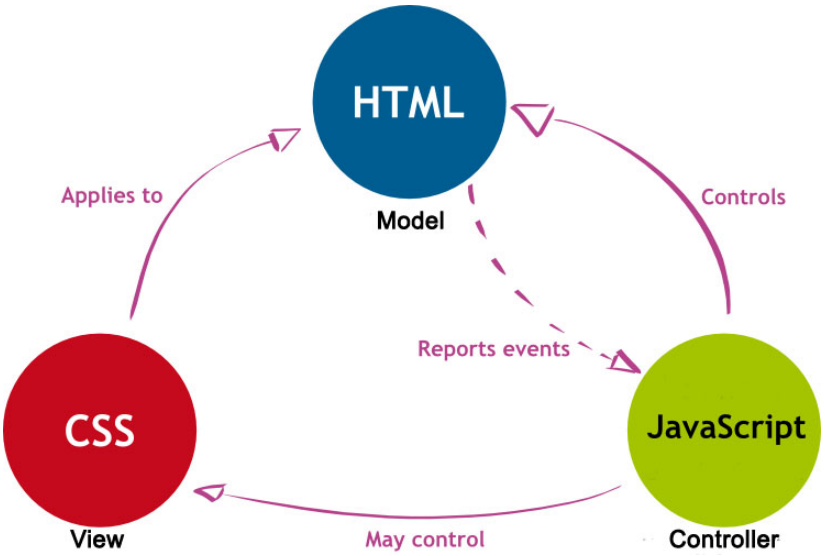

Figura 3.2: Modello MVC

Per meglio chiarire quanto detto si riporta l'esempio di seguito:

```
HTML (Model)
<div id="test"></div>
CSS (View)
.empty {
    background: yellow;
}

JavaScript (Controller)
var test = document.getElementById('test');

if(!test.hasChildNodes) {
    test.className = 'empty';
}
```

3.4.1 Single-page application

Uno degli approcci possibili nello sviluppo di un'applicazione ibrida coerente con il pattern MVC è la realizzazione di una SPA (*Single Page Application*). Le SPA sono applicazioni web in cui tutta l'esperienza utente è contenuta all'interno di una singola pagina web, senza la necessità di dover richiedere un ulteriore caricamento di dati durante la navigazione. Tutto il codice necessario al caricamento iniziale dell'applicazione, in questo approccio di sviluppo, è estratto da dati locali o da dati recuperati a richiesta da un Web Service. L'idea di base di una SPA è che, indipendentemente da come gli utenti interagiscono con l'applicazione, la pagina non venga mai ricaricata o non sia controllata/gestita da un'altra pagina al di fuori di quella attuale. Le SPA in genere contrastano con le classiche applicazioni multi-pagina in cui i cambiamenti di pagina sono regolari e nelle quali viene richiesto di recuperare nuovi contenuti dal server, ricaricando la pagina, per soddisfare le richieste degli utenti. Il problema dato dall'approccio classico multi-pagina è che la navigazione richiesta dall'utente non è fluida in quanto c'è un evidente passaggio da una pagina a quella successiva che richiede l'attesa dell'utente, dovuta al fatto che la nuova pagina e tutti i suoi contenuti dovranno essere caricati.

Le SPA mirano a risolvere questo problema, eliminando la necessità di ricaricamento; la tecnica più diffusa in tal senso è AJAX, che delega al client, oltre al rendering dell'HTML, aspetti come la gestione dei dati o la

logica di business. Le applicazioni a singola pagina, migliorano notevolmente l'esperienza utente, rendendo molto più veloce la fruizione dei contenuti e non solo. Tuttavia non mancano anche alcuni svantaggi: con tale approccio, infatti, si incrementa notevolmente la quantità di codice *client-side* in Javascript e di conseguenza, la manutenzione evolutiva dello stesso diventa più onerosa.

Capitolo 4

Framework

Per realizzare un'applicazione ibrida lo sviluppatore dovrebbe servirsi degli strumenti che sono stati elencati e descritti nel capitolo precedente, questo naturalmente potrebbe comportare un investimento considerevole di tempo e lo scontro con alcuni inevitabili ostacoli. Di conseguenza, per semplificare e fornire un sostegno allo sviluppo di questo tipo di applicazioni, è possibile sfruttare i framework, ovvero, strutture di supporto che mettono a disposizione una collezione di librerie utilizzabili con uno o più linguaggi di programmazione e in molti casi, anche alcuni strumenti di sviluppo, come un IDE o un debugger. I framework, quindi, agevolano lo sviluppo fornendo una struttura su cui un'applicazione cross-platform può essere organizzata e progettata.

4.1 Framework per la User Interface

I framework che verranno illustrati di seguito, sono specifici nella realizzazione dell'interfaccia grafica, permettendo allo sviluppatore di non dover creare tutti gli oggetti grafici tramite HTML5 e CSS, ma effettuando dei semplici *drag and drop* dei template presenti. Questi framework infatti, possiedono delle collezioni di elementi già realizzati, i quali possono essere "trascinati"nell'applicazione che si sta sviluppando e in seguito sarà unicamente il framework stesso ad occuparsi della traduzione dell'elemento in codice. Tali framework non offrono l'accesso ai sensori fisici dei device, ma il loro utilizzo può essere combinato con framework che lo permettono creando un applicazione ibrida completa. Prendiamo ora in analisi, riferendoci a

questa categoria, due framework che emergono tra gli altri: JQuery Mobile e SenchaTouch.

4.1.1 JQuery Mobile

JQueryMobile è uno dei più utilizzati tra i framework di questo genere, è basato su un'insieme di plug-in JQuery e widget che mirano a fornire una multipiattaforma API per lo sviluppo di un'interfaccia utente completa [4]. Nonostante gli sviluppatori di JQuery Mobile puntino alla realizzazione di un framework in grado di creare un'unica applicazione fruibile da un largo numero di dispositivi, essi sono consapevoli dell'impossibilità di ricreare esattamente la medesima esperienza di un'applicazione nativa. Proprio per questo motivo hanno scelto un approccio basato su *progressive enhancement* ovvero scrivendo il codice inizialmente nella sua forma più essenziale e compatibile, aggiungendo arricchimenti progressivi (enhancements) che vengono riconosciuti ed interpretati di volta in volta solo dai device più recenti, che a loro volta vengono raggruppati in classi con diversi livelli di supporto. La caratteristica principale di JQuery Mobile è la possibilità di creare applicazioni basilari grazie al semplice utilizzo dell'HTML5. Questo approccio ha dunque il vantaggio di non richiedere tecnologie particolari, come il linguaggio JavaScript, per rendere un contenuto navigabile.

JQuery Mobile si avvale della struttura semantica delle pagine HTML5 e degli attributi *data* per definire le varie parti dell'interfaccia. Alla richiesta di una pagina ed al suo caricamento, la libreria utilizzerà questa struttura per aggiungere tag e associare gli eventi e le interazioni ai componenti dell'applicazione. Il framework include un sistema di navigazione Ajax che comprende transizioni animate delle pagine e un set di base di widget UI come finestre di dialogo, barre degli strumenti, pulsanti con le icone, elementi per la realizzazione di form e altro ancora.

Una caratteristica principale di questo framework è quella di poter creare più pagine in un unico file. Questo è possibile grazie all'uso del tag *<div data-role="page"id="home">* dove *data-role="page"* sta appunto a dichiarare alla Web View che ciò che è racchiuso in questo tag è da considerarsi una pagina indipendente dal resto del contenuto del file. Qualora volessimo quindi creare più pagine basterà ripetere questo tag più volte. Attraverso la sua semplicità e il suo meccanismo di switch tra le pagine, si può dire

che JQuery Mobile è particolarmente adatto ad applicazioni con contenuti statici.

4.1.2 SenchaTouch

Sencha Touch è un framework Javascript, nato dall'evoluzione del potente e popolarissimo ExtJS, che tramite una sintassi ed una logica, in qualche modo ispirate al mondo Swing di Java, consente di descrivere interfacce attraverso istruzioni JavaScript e tecniche di templating [7]. Il framework si prende poi in carico l'onere di generare il documento HTML finale e nel farlo applica al descrittore di interfaccia un tema che può ricalcare quello delle applicazioni native di un particolare sistema operativo *mobile* o può essere creato dallo sviluppatore. *Sencha Touch* è un framework flessibile ed ottimizzato per i sistemi mobili e gli schermi multitouch; le animazioni di transizione tra i vari pannelli che compongono la nostra applicazione sono gestite in modo da consentire, con poco sforzo, di sviluppare interfacce complesse che ricalcano molto fedelmente quelle delle applicazioni native. Sencha Touch adotta il paradigma MVC come fondamenta sulle quali costruire le applicazioni, infatti nella creazione di una applicazione con questo framework sono sempre presenti tre cartelle "js/models", "js/views", "js/-controllers", ognuna delle quali contiene il codice JavaScript responsabile delle funzioni previste dal pattern MVC.

Pensando ai device per cui verranno sviluppate le applicazioni, il metodo di input primario é sicuramente il "tocco": per questo motivo é fondamentale che vengano riconosciuti, senza errori, la maggior parte dei movimenti, normalmente utilizzati per interagire con le applicazioni, a cui l'utente é abituato. Quindi, oltre ai classici *touchstart* e *touchend*, vengono aggiunti dal framework una serie di eventi personalizzati come il tocco, il doppio tocco, il *pinch* e la rotazione. Questi permettono interazioni presenti finora solo nelle applicazioni native.

4.2 Framework per l'accesso alle API del dispositivo: PhoneGap

Al contrario dei framework appena descritti, ce ne sono altri che permettono un accesso diretto alle API native del dispositivo e quindi consentono di

utilizzare le funzionalità hardware di base di un device. Tra questi si impone, sicuramente, il framework PhoneGap che, di seguito, sarà trattato in modo più approfondito.

PhoneGap è un framework opensource, sviluppato dalla Nitobi Software, che permette di realizzare applicazioni ibride [19]. L'idea fondamentale di questo progetto può essere riassunta dallo slogan "*Write once, port everywhere*". PhoneGap, infatti, si propone di fornire un supporto che consenta allo sviluppatore, anche sprovvisto di conoscenze specifiche riguardanti le diverse piattafome, di realizzare un'applicazione cross-platform e di concentrarsi sulla realizzazione della sola parte web. PhoneGap rappresenta un valido strumento che permette di costruire applicazioni in HTML, JavaScript e CSS, capaci di accedere alle funzionalità di base, normalmente non accessibili da una semplice applicazione web, in piattaforme come iPhone, Android, Windows Phone, Symbian, Blackberry e Tizen. PhoneGap realizza la funzione di *bridge*, precedentemente anticipata, tra l'applicazione e le funzionalità hardware del dispositivo *mobile* fornendo una modalità "standard", realizzata tramite l'utilizzo di semplici invocazioni JavaScript, per invocare le API native in maniera indipendente dal tipo di piattaforma sottostante. Il *porting* verso le varie piattaforme viene conseguentemente realizzato, installando gli ambienti di sviluppo relativi e compilando l'applicazione ibrida realizzata. I requisiti sono quindi di installare le SDK, dove richiesto dalla piattaforma, e gli strumenti per consentire la compilazione delle applicazioni.

Per capire meglio come un'applicazione ibrida, sviluppata con PhoneGap, possa adattarsi a qualunque tipologia di piattaforma, è utile vedere come questo framework sia strutturato sia a livello architetturale che a livello di strumenti messi a disposizione dello sviluppatore.

4.2.1 Architettura

Il vantaggio fondamentale della realizzazione di applicazione ibride con PhoneGap consiste nel realizzare un'applicazione che si "adatta"a qualunque piattaforma tra quelle supportate. PhoneGap deve, quindi, interfacciarsi sia con l'applicazione che con ciascun sistema operativo della piattaforma sottostante.

Tutto ciò viene realizzato usando il linguaggio nativo della piattaforma che permette l'accesso alle risorse hardware e software, mettendo a dispo-

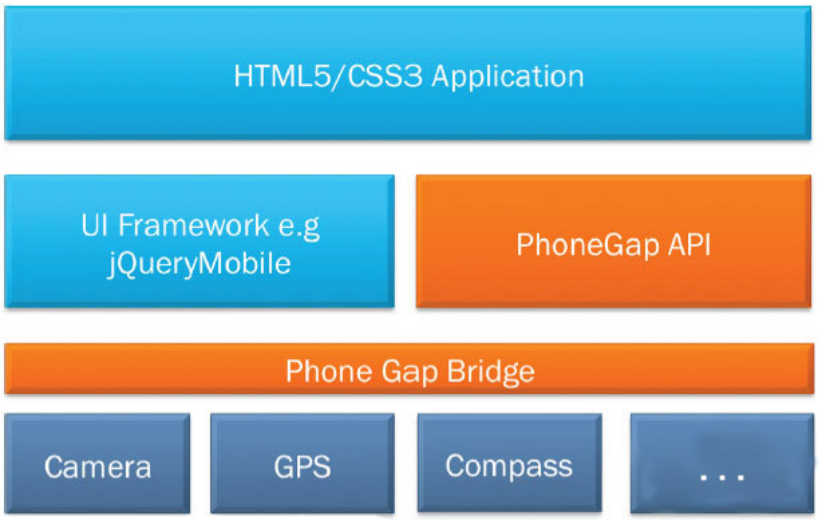

Figura 4.1: Architettura di un'applicazione PhoneGap

sizione queste funzionalità allo sviluppatore attraverso il linguaggio JavaScript, così da renderle facilmente utilizzabili dall'applicazione come fossero tradizionali metodi di libreria. In base alla tipologia di piattaforma con la quale dovrà interfacciarsi, l'implementazione di collegamento, fornita dallo stesso framework, sarà di conseguenza sviluppata in Objective C per iPhone, in Java per Android, in C++/C# per Windows Phone e così via.

PhoneGap, quindi, fornisce una soluzione già implementata, atta a permettere la comunicazione tra API native e applicazione, dell'architettura che è stata descritta nel capitolo 2. Il risultato sarà un pacchetto composto di due elementi principali con differenti responsabilità che però cooperano tra loro per fornire delle funzioni a valore aggiunto. Una prima componente si occupa di dialogare direttamente con il dispositivo, interfacciandosi con le API native e assicurandosi l'accesso alle funzionalità proprie del device, mentre una seconda offrirà l'interfaccia verso l'utente e permetterà, quindi, l'utilizzo delle componenti native [17]. L'uso di JavaScript e di Ajax consente inoltre di realizzare applicazioni complete in grado di avere comportamenti complessi, in cui il client rappresenta un elemento fondamentale.

4.2.2 PhoneGap API

La caratteristica più rilevante di PhoneGap consiste nel fornire l'accesso a funzionalità native, presentandole all'utente nello stesso modo in cui verrebbero presentate dalle applicazioni native. Le funzionalità native piu importanti sono quelle che riguardano sia il recupero che l'utilizzo dei dati provenienti dai sensori presenti nel dispositivo. Un sensore è un elemento hardware, presente su un device, che permette di recuperare informazioni riguardanti l'ambiente circostante in cui si trova il dispositivo. Le informazioni relative ai sensori possono essere recuperate con due modalità:

- in modo asincrono, invocando:

  ```
  navigator.sensor.getCurrentVariable(successCallback,
    errorCallback, options)
  ```

  La funzione interroga il sensore e i dati restituiti vengono passati come primo parametro alla funzione di callback;

- periodicamente, invocando:

  ```
  navigator.sensor.watchVariable(successCallback,
    errorCallback, options)
  ```

  La funzione chiama periodicamente *getCurrentVariable()*, passando ogni volta il risultato al *successCallback*. L'intervallo con il quale effettuare l'interrogazione dev'essere specificato nel parametro *option*. La funzione ritorna una varibale *watchId*, che può essere utilizzata per terminare il polling chiamando la seguente funzione: *navigator.sensor.clearWatch(watchId)*.

Nell'utilizzare i metodi per richiamare le informazioni provenienti dai sensori bisogna sempre tenere in considerazione che non tutti i device hanno a disposizione gli stessi sensori e quindi non possono rilevare determinati dati.

In seguito sarà fatta una panoramica di alcune funzionalità alle quali PhoneGap permette l'accesso, per maggiori dettagli si faccia riferimento al sito ufficiale [6].

CAMERA

L'oggetto *camera* permette di accedere all'applicazione nativa della fotocamera presente sul device. L'API Camera permette di recuperare un'immagine dal dispositivo, nelle modalità specificate nel parametro *cameraOption*, eseguendo la seguente funzione:

```
navigator.camera.getPicture(onCameraSuccess, onCameraError,
    cameraOption);
```

Come in altre API di PhoneGap, che verranno analizzate in seguito, per eseguire una funzione, in questo caso *getPicture()*, dev'essere specificata una funzione che verrà eseguita nel caso la chiamata venga effettuata correttamente e una funzione nel caso in cui, invece, si presenti un errore.

Di seguito viene riportato un'esempio per chiarire meglio l'uso di questa funzionalità:

```
function capturePhoto() {
/*Recupera un'immagine usando la fotocamera integrata nel
* device e ritorna un file URI della foto */
navigator.camera.getPicture(onSuccess, onFail, {quality:50,
  destinationType: Camera.DestinationType.FILE_URI });
}

function onSuccess(imageURI) {
    var image = document.getElementById('myImage');
    image.src = imageURI;
}

function onFail(message) {
    alert('Failed because: ' + message);
}
```

Per l'acquisizione di un'immagine utilizzando l'applicazione fotocamera di default, che permette all'utente di scattare la foto, la proprietà *Camera.sourceType* dovrà essere impostata con *Camera.PictureSourceType.CAMERA*, nel caso si voglia utlizzare un'immagine già presente nel photoalbum del device l'impostazione sarà *Camera.SourceType.SAVEDPHOTOALBUM* o *Camera.SourceType.PHOTOLIBRARY*, che sono essenzialmente la stessa cosa. Una volta che l'utente ha scattato la fotografia, si può settare la proprietà

destinationType, che indica il formato del valore di ritorno delle informazioni riguardante l'immagine, con *Camera.DestinationType.FILE_URI* nel caso si voglia come come valore di ritorno l'URI della foto nel filesystem del device oppure con *Camera.DestinationType.DATA_URL* nel caso si voglia una stringa base64 che rappresenta il contenuto dell'immagine.

GEOLOCATION

L'oggetto *geolocation* consente l'accesso ai dati di localizzazione basati su sensori GPS del dispositivo o sui segnali di rete. Per l'utilizzo di questa API il device deve essere dotato di una qualsiasi funzionalità di geolocalizzazione che permetta di derminare la sua posizione sulla superficie terrestre. Per accedere ai dati sono messe a disposizione due modalità:

- Viene comunicata la posizione corrente del dispositivo restituendo come parametro della callback un oggetto *Position* con la funzione seguente:

```
navigator.geolocation.getCurrentPosition(
  onGeolocationSuccess,ongeolocationError);
```

di seguito è riportato un esempio con l'utilizzo di questa modalità:

```
/*onSuccess Callback: questo metodo accetta un oggetto
* textit{Position} che contiene le coordinate GPS */
var onSuccess = function(position) {
  alert('Latitude:'+ position.coords.latitude + '\n' +
        'Longitude:'+ position.coords.longitude + '\n'+
        'Altitude:'+ position.coords.altitude + '\n' +
        'Accuracy:'+position.coords.accuracy + '\n' +
        'Altitude Accuracy:'
          + position.coords.altitudeAccuracy+'\n' +
        'Heading:'+ position.coords.heading+ '\n' +
        'Speed:'+ position.coords.speed + '\n' +
        'Timestamp:'+ position.timestamp + '\n');
};

//onError Callback: riceve un oggettoPositionError
function onError(error) {
    alert('code: '    + error.code    + '\n' +
```

```
            'message: ' + error.message + '\n');
}

navigator.geolocation.getCurrentPosition(onSuccess,
   onError);
```

- Viene resituita la posizione corrente del dispositivo ogni volta che si rileva un cambiamento di posizione, il codice è il seguente:

```
navigator.geolocation.watchPosition(
  onGeolocationSuccess, onGeolocationError);
```

di seguito è riportato un esempio con l'utilizzo di questa modalità:

```
/*onSuccess Callback: questo metodo accetta un oggetto
* Position che contiene le coordinate GPS */
function onSuccess(position) {
    var element = document.getElementById(
      'geolocation');
    element.innerHTML =
      'Latitude:' + position.coords.latitude + '<br>' +
      'Longitude: '+ position.coords.longitude + '<br>'
        + '<hr />' + element.innerHTML;
}

//onError Callback: riceve un oggettoPositionError
function onError(error) {
    alert('code: '    + error.code    + '\n' +
          'message: ' + error.message + '\n');
}

/*Opzioni: lancia un errore se non si riceve un update
* ogni 30 secondi */
var watchID = navigator.geolocation.watchPosition(
  onSuccess, onError, { timeout: 30000 });
```

L'oggetto *Position* ritorna all'utente attraverso la funzione di callback ed è formato dalle seguenti proprieta:

- *coords*: Un insieme di coordinate geografiche (Coordinates);

- *timestamp*: momento in cui sono state rilevate le coordinate in millisecondi (DOM-TimeStamp).

L'oggetto Coordinates è costituito da un insieme di proprietà che descrivono le coordinate geografiche:

- *latitude*: Latitudine in gradi (Numero);
- *longitude*: Longitudine in gradi (Numero);
- *altitude*: Altezza della posizione in metri sopra l'elissoide (Numero);
- *accuracy*: Livello di precisione delle coordinate di latitudine e longitudine in metri (Numero);
- *altitudeAccuracy*: Livello di precisione della coordinata altitudine in metri (Numero);
- *heading*: Senso di marcia, specificata in gradi conteggiati in senso orario rispetto alla direzione nord (Numero).

Occorre precisare, infine, che HTML5 fornisce un supporto analogo per la Geolocation, le applicazioni PhoneGap possono direttamente utilizzare questo per la rilevazione.

ACCELEROMETER

L'accelerometro permette ad un'applicazione PhoneGap di determinare l'orientamento di un dispositivo in uno spazio tridimensionale, usando le coordinate X, Y e Z.

Le API Accelerometer consentono all'applicazione di rilevare i dati relativi all'orientamento del device utilizzando due metodi:

- ```
  navigator.accelerometer.getCurrentAcceleration(
    onAccelSuccess,onAccelFailure);
  ```

- ```
  navigator.accelerometer.watchAcceleration(
    onAccelSuccess, onAccelFailure, accelOptions);
  ```

Il primo permette una sola rilevazione, mentre per il secondo è possibile specificare l'intervallo di tempo con il quale rilevare i dati. Di seguito un esempio dell'utilizzo di questa funzionalità utilizzando il primo metodo:

```
function onSuccess(acceleration) {
    alert('Acceleration X: ' + acceleration.x + '\n' +
          'Acceleration Y: ' + acceleration.y + '\n' +
          'Acceleration Z: ' + acceleration.z + '\n' +
          'Timestamp: ' + acceleration.timestamp + '\n');
};

function onError() {
    alert('onError!');
};

navigator.accelerometer.getCurrentAcceleration(onSuccess,
    onError);
```

L'oggetto *acceleration* ottenuto come valore di ritorno nel caso la chiamata sia stata eseguita con successo è composto dalle seguenti proprietà:

- x: Posizione sull'asse x (Numero);
- y: Posizione sull'asse y (Numero);
- z: Posizione sull'asse z (Numero);
- *timestamp*: momento in cui sono stati rilevati i dati in millisecondi (DOMTimeStamp).

Dopo aver ottenuto i dati, rilevati in istanti diversi, questi possono essere utilizzati per ricavare il movimento e l'accelerazione del device.

CAPTURE

L'API Capture consente ad un'applicazione di recuperare audio, video e immagini in serie, inter- facciandosi con le applicazioni native di sistema. Le immagini, come precedentemente detto, possono essere catturate grazie alle API Camera che mettono a disposizione il metodo *getPicture()*, la sostanziale differenza che introducono le API Capture è rappresentata dallo standard con cui si presentano sia che si tratti di recuperare un'immagine scattata che di registrare un video o un audio.

L'implementazione di queste API si basa sulle API Media Capture della W3C e viene utilizzata eseguendo i seguenti funzioni:

- per recuperare immagini scattate:

```
navigator.device.capture.captureImage(onCaptureSuccess,
onCaptureError, captureOptions);
```

- per registrare audio:

```
navigator.device.capture.captureAudio(onCaptureSuccess,
onCaptureError, captureOptions);
```

- per registrare video:

```
navigator.device.capture.captureVideo(onCaptureSuccess,
onCaptureError, captureOptions);
```

Queste fuzioni hanno numerose opzioni ed oggetti che permettono di configurarne il comportamento. Per ogni ulteriore dettaglio si rimanda alla documentazione ufficiale.

COMPASS

Questa funzionalità permette di individuare la direzione verso la quale è rivolto un dispositivo in un piano bidimensionale. Gli smartphone più moderni sono dotati di una bussola integrata, di conseguenza l'API può interrogare questo chip che restituirà un valore, espresso in gradi, compreso tra 0 e 359.99. L'API Compass si comporta in modo analogo all'API Accelerometer rilevando l'orientamento del device con due funzioni, che si differenziano a seconda che il dato sia fornito una sola volta o sia fornito con una determinata frequenza:

- `navigator.compass.getCurrentHeading(successFunction, errorFunction);`

- `navigator.compass.watchHeading(onHeadingSuccess, onHeadingError, watchOptions);`

Di seguito un esempio che mostri l'utilizzo di questa funzionalità:

```
function onSuccess(heading) {
    alert('Heading: ' + heading.magneticHeading);
};

function onError(error) {
    alert('CompassError: ' + error.code);
};

navigator.compass.getCurrentHeading(onSuccess, onError);
```

L'oggetto *CompassHeading* ottenuto come valore di ritorno ha le seguenti proprietà:

- *magneticHeading*: La direzione in un singolo momento di tempo espressa in gradi da 0-359.99 (Numero);
- *trueHeading*: La direzione relativa al Polo Nord geografico espressa in gradi da 0-359.99 in un singolo momento. Un valore negativo indica che il valore non èuò essere determinato (Numero);
- *headingAccuracy*: La deviazione in gradi tra la direzione riportata e il valore di trueHeading (Numero);
- *timestamp*: Il momento in cui la direzione è stata determinata (millisecondi).

Capitolo 5

Conclusioni

Il lavoro svolto nell'ambito di questa Tesi ha esposto una trattazione esaustiva sulla progettazione e sullo sviluppo di applicazioni ibride. Coerentemente con gli obiettivi espressi nel capitolo di introduzione, si è posta l'attenzione sull'architettura e le tecnologie di base per la realizzazione di questa tipologia di applicazioni.

Nella prima parte della trattazione si è messa in evidenza la parte architetturale tipica di un'applicazione ibrida *mobile*, precisando la presenza di un wrapper nativo che offre agli sviluppatori la possibilità di accedere alle funzionalità proprie di uno *smart device* e di poter usufruire dei relativi *app store* per la distrubuzione dell'applicazione. Si è analizzato il ruolo fondamentale del motore di rendering *WebKit* e del concetto di Web View responsabili sia dell'esecuzione che della visualizzazione dell'applicazione, realizzata con linguaggi standard web. Mantenendo una visione quanto più generale possibile è stato illustrato il principio alla base della parte di interfacciamento alle API native del dispositivo, qualità per la quale le applicazioni ibride sono riuscite ad affermarsi nel mercato delle applicazioni *mobile*.

Successivamente si sono introdotte ed analizzate le tecnologie di base per lo sviluppo dell'applicazione, quali HTML5, CSS3 e Javascript. Queste tecnologie mettono a disposizione molteplici elementi che permettono la realizzazione di *features* avanzate tipiche di linguaggi nativi quali programmazione *multithread* e l'applicazione del pattern architetturale MVC.

Infine si è voluta porre l'attenzione sulla realizzazione di queste applica-

zioni con un approccio che includesse il supporto di alcuni framework, atti sia allo sviluppo delle interfacce grafiche sia in grado di mettere a disposizione librerie che permettano l'utilizzo delle funzionalità native di un device, sollevando, quindi, lo sviluppatore dal dover implementare API proprie per potersi interfacciare con il sistema operativo.

Dopo quanto trattato si può affermare che le applicazioni ibride rappresentino un valido compromesso, tra le soluzioni native e quelle puramente basate su web, in grado di soddisfare l'esigenza di un'applicazione *cross-platform* senza che questa richieda la conoscenze degli specifici modelli di programmazione e dei linguaggi utilizzati. L'utilizzo di tecnologie largamente diffuse e conosciute come quelle web permette, anche a chi non ha familiarità con lo sviluppo in ambito *mobile*, di realizzare valide applicazioni di varie tipologie. Questo assicura, inoltre, un impiego di tempo notevolmente inferiore sia per quanto riguarda lo sviluppo che la manutenzione di un'applicazione ibrida, in contrasto con le tempistiche riguardanti la tipologia nativa. Oltre a ciò, è importante evidenziare che, la presenza di un layer intermedio, da una parte ha un ruolo fondamentale in quanto permette l'interfacciamento alle funzionalità native, dall'altra rappresenta un limite in quanto le interazioni tra applicazione e device non sono dirette, di conseguenza questo determina una perdita di reattività nelle operazioni eseguite. Spostando l'analisi ad un livello più tecnico si sono riscontrate alcune criticità riguardo al modello di esecuzione di Javascript che, essendo stato originariamente progettato per un diverso utilizzo, porta ad un rapido aumento della complessità del codice in relazione al crescere delle funzionalità implementate, questo a causa dell'uso intensivo di *callback*, utilizzate sia per la gestione degli eventi dell'interfaccia grafica sia per le interazioni asincrone con il sistema operativo o il sistema di back-end. In contrasto con questa critica, un aspetto sicuramente interessante, che innalza il livello di maturità di Javascript, è rappresentato dall'introduzione dei Web Worker, che consentono un esecuzione, basata sul modello *multi-thread*, che permette lo svolgimento di operazioni computazionalmente onerose senza bloccare l'interfaccia grafica. In seguito all'analisi fatta si deduce che la diffusione di questo tipo di applicazioni è rallentata esclusivamente da una tecnologia di supporto che in questo particolare contesto, nonostante gli aspetti recentemente introdotti, non è ancora giunta ad un livello di maturità tale da colmare il *gap* in termini di *performance* e *user-experience* che le separa dalle applicazioni native.

Ringraziamenti

Ringrazio la mia famiglia, che mi ha sostenuto nel raggiungimento di questo traguardo.

Ringrazio Mattia, per essermi sempre stato accanto, dandomi la forza di superare qualsiasi ostacolo.

Ringrazio Micol, Michela, Sara e Alessandra per avermi sempre incoraggiato e per avermi strappato un sorriso anche nei momenti più difficili.

Ringrazio i miei compagni di università che hanno reso speciali questi anni.

Ringrazio Gozzino, che mi ha fatto vivere un'esperienza unica, grazie alla quale ho potuto affrontare le giornate con entusiasmo.

Bibliografia

[1] Ajax history & information. http://www.xmluk.org/ajax-history-and-information.htm.

[2] Cascading style sheets. http://www.w3.org/Style/CSS/.

[3] Html. http://en.wikipedia.org/wiki/HTML.

[4] Jquerymobile. http://jquerymobile.com/.

[5] Mocha, livescript, javascript. http://computer-programming-languages.knoji.com/mocha-livescript-javascript-2/.

[6] Phonegap. http://phonegap.com/.

[7] Senchatouch. http://www.sencha.com//.

[8] Webkit. http://en.wikipedia.org/wiki/WebKit.

[9] W. Alex. Get smart. In *Communication of ACM*, 2009.

[10] F. Amabile. Inizia l'era biomediatica - il censis su italiani e comunicazione, 2012. http://www.lastampa.it/2012/10/03/italia/cronache/inizia-l-era-biomediatica-il-censis-su-italiani-e-comunicazione-nMBuGl3qjNnp9ee23SdMuK/pagina.html.

[11] Apple. ios. http://www.apple.com/it/ios/.

[12] F. Censis. I media siamo noi, l'inizio dell'era biomediatica. In *Decimo rapporto Censis/Ucsi sulla comunicazione*, 2012.

[13] E. Franceschini. E' un mondo di cellulari, sono oltre 4 miliardi, 2009. `http://www.repubblica.it/2007/08/sezioni/tecnologia/cellulari/mezzo-mondo-cell/mezzo-mondo-cell.html`.

[14] N. Gok and N. Khanna. In *Building Hybrid Android Apps with Java and Javascript*. O'Reilly, 2013.

[15] Google. Android. `http://www.android.com/`.

[16] L. James A., J. Anthony D., and R. Franklin. Smarter phones. In *Pervasive Computing*. IEEE CS, 2009.

[17] A. Lunny. In *PhoneGap - Beginner's Guide*, 2011.

[18] L. Ramon, R. Kevin, and S. Michael. Idc worldwide mobile phone tracker, 2013. `http://www.idc.com/getdoc.jsp?containerId=prUS24257413`.

[19] J. M. Wargo. Building cross-platform mobile apps. In *PhoneGap Essential*, 2012.

[20] Windows. Windows phone. `http://www.windowsphone.com/it-it/`.