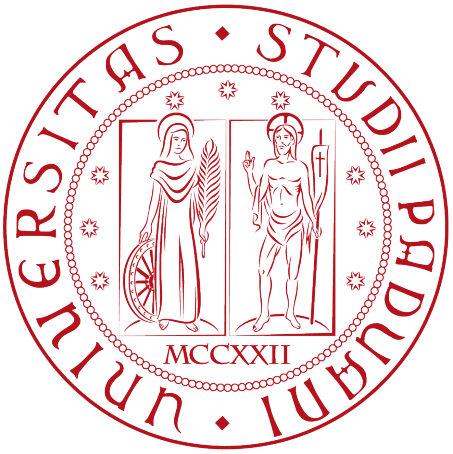
Università degli Studi di Padova

Dipartimento di Matematica

Corso di Laurea in Informatica



Sviluppo di applicazioni native per iOS in JavaScript

Tesi di laurea triennale

Relatore

Prof. Claudio Enrico Palazzi

Laureando

Giacomo Manzoli

Anno Accademico 2014-2015

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, dal laureando Giacomo Manzoli presso l'azienda WARDA S.r.l.. L'obiettivo di tale attività di stage è l'analisi dei vari framework disponibili per lo sviluppo di applicazioni native utilizzando il linguaggio JavaScript, al fine di creare un'applicazione nativa per iOS simile alla gallery sviluppata dall'azienda.

"Fuck it, i did it"

— Confucius

Ringraziamenti

Bla bla bla

Bla bla bla

Bla bla bla

Even more bla bla bla

...

Padova, Dec 2015 Giacomo Manzoli

Indice

Elenco delle figure

Elenco delle tabelle

Notes

Capitolo 1

Il contesto aziendale

1.1 L'azienda

WARDA S.r.l. è una startup avviata di recente dai due soci Marco Serpilli e David Bramini, con sede a Padova.

L'azienda è nata come spin-off di Visionest S.r.l, una società che si occupa di consulenza informatica e dello sviluppo di software gestionali dedicate alla aziende.

A differenza di Visionest, WARDA S.r.l. si focalizza sulle aziende che operano nel mercato della moda e del lusso, offrendo un sistema per la gestione delle risorse e dei processi aziendali, ottimizzato per le aziende del settore.

figura 1.1: Logo di WARDA S.r.l.

1.1.1 WARDA - Work on what you see

WARDA è un sistema software che si occupa di Digital Asset Management (DAM), cioè un sistema di gestione di dati digitali e informazioni di business, progettato per il mercato del lusso, fashion e retail.

In un sistema DAM, le immagini, i video ed i documenti sono sempre legati alle informazioni di prodotto, costituendo così i digital assets: un'immagine prodotto riporta le caratteristiche della scheda prodotto, un'immagine marketing le informazioni della campagna, un'immagine della vetrina i dati del negozio, del tema, dell'ambiente, e così via.

Memorizzando questi digital assets in un unico sistema centralizzato, WARDA permette di riutilizzare infinite volte le immagini, i video e i documenti durante le attività aziendali evitando così ogni duplicazione.

Il materiale digitale è sempre associato alla scheda prodotto gestita nel sistema informativo aziendale, rappresentando così l'unico "catalogo digitale" di tutti i beni/prodotti aziendali.

In questo modo WARDA diventa il centro dell'ecosistema digitale aziendale, coordinando e organizzato la raccolta e la condivisione dei digital assets attraverso processi ben definiti e istanziati secondo le prassi aziendali.

WARDA è disponibile sia come applicazione Web, sia come applicazione per iPad.

1.2 Il progetto

Lo scopo del progetto è quello di valutare se, mediante l'utilizzo di framework differenti, sia possibile migliorare l'esperienza d'uso del client per iPad di WARDA.

Secondo l'azienda la causa di alcuni problemi del client attuale derivano dal framework che è stato utilizzato per svilupparlo, infatti, l'applicazione attuale è un'applicazione di tipo ibrido, cioè composta da un'applicazione web ottimizzata per lo schermo dell'iPad e racchiusa in un'applicazione nativa utilizzando Cordova.

Di conseguenza, si ritiene che l'utilizzo di un'altra tipologia di framework che permette lo sviluppo di applicazioni native utilizzando JavaScript possa portare dei miglioramenti all'applicazione attuale.

Pertanto, la prima parte del progetto riguarda la ricerca e l'analisi di framework che permettono di sviluppare applicazioni native con JavaScript utilizzando un'interfaccia grafica nativa al posto di una pagina web.

La seconda parte invece, prevede lo sviluppo di un'applicazione analoga a al client per iPad attuale utilizzando uno dei framework individuati.

Capitolo 2

Framework analizzati

Questo capitolo inizia con una descrizione generale della tipologia di framework analizzati, per poi andare a descrive più in dettaglio il funzionamento dei singoli framewrok, seguito da una sintesi dei pregi e difetti e un breve resoconto riguardo il prototipo realizzato. Infatti, per valutare al meglio ogni framework è stato realizzata un'applicazione prototipo che visualizza una gallery di immagini ottenuta mediante chiamate ad API REST.
La struttura del prototipo è stata scelta in modo tale da verificare le seguenti caratteristiche:

- possibilità di disporre le immagini in una griglia;
- possibilità di implementare lo scroll infinito, cioè il caricamento di ulteriori dati una volta visualizzati tutti quelli disponibili;
- fluidità dello scroll, specialmente durante il caricamento dei dati;
- possibilità di personalizzare la barra di navigazione dell'applicazione.

Valutare se inserire le informazioni riguardo l'esecuzione di un'applicazione

2.1 Considerazioni generali

Le applicazioni realizzate con questa tipologia di framework hanno alla base lo stesso funzionamento: una macchina virtuale interpreta il codice JavaScript e mediante un "*ponte*" viene modifica l'interfaccia grafica dell'applicazione, che è realizzata con i componenti offerti dall'SDK nativo. Ognuno dei framework analizzati utilizza un "*ponte*" diverso, il cui funzionamento verrà descritto più in dettaglio nell'apposita sezione.

Un'altra caratteristica di questa tipologia di framework è l'assenza del DOM. Infatti, l'interfaccia di un'applicazione è composta da componenti grafici del sistema operativo che vengono creati e composti a durante l'esecuzione dell'applicazione e non da elementi HTML come nelle applicazioni ibride.

Nel complesso si ottengono due grossi vantaggi:

- L'esecuzione del codice JavaScript e il rendering dell'interfaccia grafica avviene su due thread distinti, rendendo l'applicazione più fluida;

- Utilizzando i componenti grafici nativi si ottiene un esperienza utente più simile a quella che si ottiene con un'applicazione realizzata in Obj-C/Java.

2.1.1 Differenze con le applicazioni ibride

Un'applicazione ibrida consiste in un'applicazione nativa composta da una WebView[1] che visualizza un'insieme di pagine web realizzate utilizzando HTML5, JavaScript e CSS3, che replicano l'aspetto di un'applicazione nativa.

Trattandosi quindi di un'applicazione web è possibile utilizzare tutti i framework disponibili per il mondo web, come Angular, jQuery, Ionic, ecc. Inoltre, se è già disponibile un'applicazione web per desktop, la creazione di un'applicazione mobile ibrida risulta rapida.

Questo approccio viene utilizzato da qualche anno e permette di creare applicazione che possono accedere ad alcune funzionalità hardware del dispositivo come il giroscopio o la fotocamera e che possono essere commercializzate nei vari store online.

Per supportare questo processo di sviluppo sono stati sviluppati dei framework come Cordova/PhoneGap che si occupano di gestire la WebView e di fornire un sistema di plug-in per accedere alle funzionalità native.

Questa tipologia di applicazioni ha però delle limitazioni:

- **Prestazioni:** trattandosi di una pagina web renderizzata all'interno di un browser, non è possibile sfruttare al massimo le potenzialità della piattaforma sottostante, come il multi-threading. Questo comporta che il codice JavaScript e il rendering dell'interfaccia grafica vengano eseguiti nello stesso thread, ottenendo così un'interfaccia poco fluida.

- **Esperienza d'uso:** una delle caratteristiche principali delle applicazioni native sono le gesture, l'utente è abituato ad interagire con le applicazioni sviluppate in linguaggio nativo che sfruttano un sistema complesso di riconoscimento delle gesture e che non è replicabile in ambito web.

- **Funzionalità:** non tutte le funzionalità che può sfruttare un'applicazione nativa sono disponibili in un'applicazione ibrida.

Il funzionamento dei framework analizzati in questo capitolo permette di risolvere i problemi principali delle applicazioni ibride, in quanto sfruttando i componenti nativi, non è presente il problema dell'interfaccia grafica che viene renderizzata nello stesso thread che si occupa dell'esecuzione del JavaScript. Sempre per il fatto che vengono utilizzati componenti nativi, è possibile sfruttare lo stesso sistema di riconoscimento delle gesture e le stesse animazioni, rendendo l'esperienza d'uso più simile a quella offerta da un'applicazione realizzata con l'SDK nativo.

2.2 Tabris.js

Framework pubblicato da EclipseSource[2] nel Maggio 2014, che permette di controllare mediante JavaScript i componenti dell'interfaccia grafica nativa, sia di iOS, sia di Android.

[1] Componente delle applicazioni native che consiste in un browser web con funzionalità ridotte e che permette di visualizzare pagine web scaricate da internet oppure memorizzate in locale.

[2] http://eclipsesource.com/en/home/

2.2.1 Come funziona

Tabris.js utlizza come "*ponte*" una versione modificata di Cordova e dei plug-in personalizzati per incapsulare i componenti grafici offerti dai vari SDK nativi.

Trattandosi di un framework derivato da Cordova è possibile utilizzare i plug-in di Cordova già esistenti per aggiungere nuove funzionalità, oltre a quelle offerte framework, come per esempio l'utilizzo della fotocamera.

L'unica condizione per il corretto funzionamento dei plug-in esterni è che non dipendano dal DOM, dal momento che il DOM non è presente durante l'esecuzione di un'applicazione.

2.2.2 Pregi e difetti

Uno dei pregi di Tabris.js è quello che il codice JavaScript scritto è indipendente dalla piattaforma, questo permette di utilizzare lo stesso codice sorgente e lo stesso layout sia per iOS sia per Android, si occupa il framework di eseguire tutte le operazioni specifiche per le varie piattaforme.

Un altro pregio deriva dall'estensibilità, è infatti possibile utilizzare sia dei plug-in di Cordova, sia i moduli disponibili su npm, con la condizione che questi non dipendano dal DOM.

Le criticità di Tabris.js riguardano per lo più il layout che deve essere fatto in modo imperativo, prima definendone posizione e dimensione, e poi combinandoli tra loro in modo gerarchico.

Sempre per quanto riguarda il layout, la personalizzazione è limitata in quanto risulta complesso se non impossibile definire dei componenti grafici composti o con layout particolari, come la visualizzazione a griglia.

Infine, il framework non impone né suggerisce alcun pattern architetturale da adottare, lasciando completa libertà al programmatore, con il rischio che il codice sorgente dell'applicazione diventi complesso e difficile da manutenere.

2.2.3 Prototipo

Nel realizzare l'applicazione prototipo sono state riscontrate varie problematiche in particolare riguardanti il layout. Ad esempio, non è stato possibile disporre gli elementi in una griglia e la personalizzazione della barra di navigazione si è rilevata essere molto limitata.

Un altro problema emerso durante la realizzazione del prototipo è stata la scarsa disponibilità di materiale on line, infatti oltre alle risorse messe a disposizione dai creatori e alla documentazione ufficiale, non è stato possibile trovare altre informazioni riguardanti il framework. Tuttavia, l'applicazione realizzata è molto fluida e l'esperienza d'uso è paragonabile a quella di un'applicazione nativa.

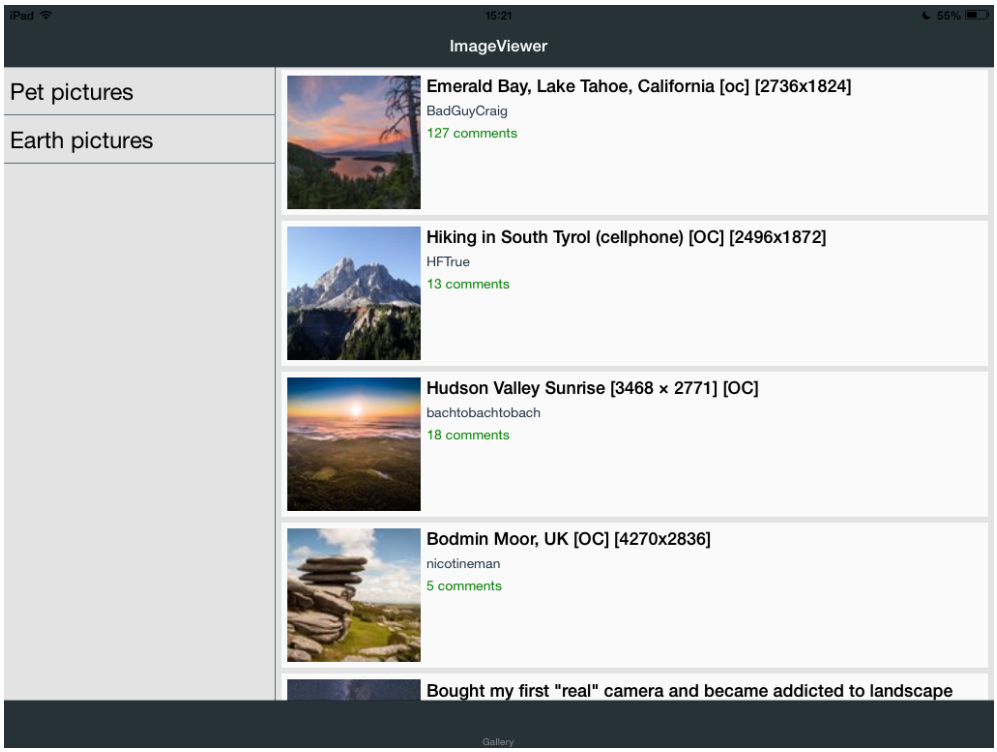


figura 2.1: Screenshot del prototipo realizzato con Tabris.js

2.3 NativeScript

Framework rilasciato da Telerik nel Maggio 2015 che permette di realizzare applicazioni mobile native sia per iOS che per Android rendendo possibile utilizzare tutte le API native mediante JavaScript.

2.3.1 Come funziona

NativeScript utilizza come "*ponte*" una virtual machine appositamente modificata che all'occorrenza usa il C++ per invocare le funzioni scritte in linguaggio nativo (Obj-C, Java). Questa virtual machine deriva da V8 se l'applicazione viene eseguita su Android o da JavaScriptCore nel caso l'applicazione venga eseguita su iOS.

Le modiche subite dalla virtual machine riguardano:

- la possibilità di intercettare l'esecuzione di una funzione JavaScript ed eseguire in risposta del codice C++ personalizzato;
- l'iniezione di meta dati che descrivono le API native;

In questo modo il runtime di NativeScript può utilizzare i meta dati per riconoscere le funzioni JavaScript che richiedono l'esecuzione di codice nativo ed eseguire il corrispettivo codice nativo invocando le funzione con delle istruzioni scritte in C++.

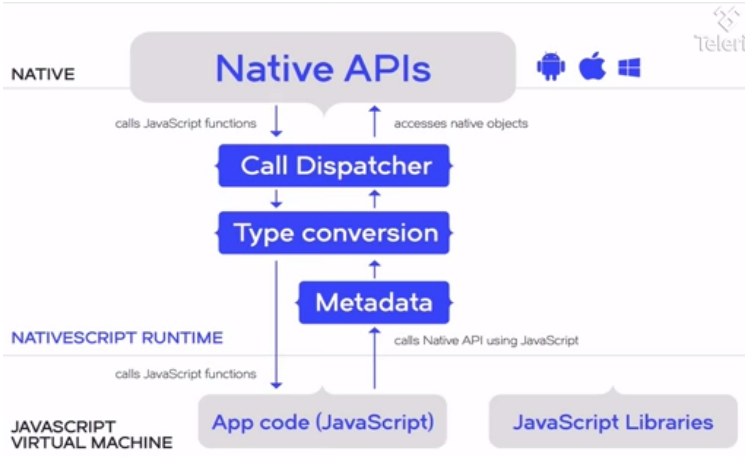


figura 2.2: Schema rappresentante il runtime di NativeScript

Di conseguenza il runtime di NativeScript permette di creare degli oggetti JavaScript che funzionano da proxy rispetto ad oggetti nativi, specifici della piattaforma. Un esempio del funzionamento è dato dal seguente codice che su iOS crea un oggetto di tipo `UIAlertView`:

```
1 var alert = new UIAlertView();
```

codice 2.1: Esempio di creazione di un oggetto nativo

All'esecuzione del JavaScript la virtual machine riconosce, grazie ai meta dati iniettati, che la funzione JavaScript deve essere eseguita come una funzione nativa.

Di conseguenza, utilizzando del codice C++, invoca la corrispondente funzione Obj-C che in questo caso istanzia un oggetto UIAltertView e salva un puntatore all'oggetto nativo in modo da poter eseguire delle funzioni su di esso.

Alla fine crea un oggetto JavaScript che funziona come un proxy dell'oggetto nativo precedentemente creato e lo ritorna in modo che possa essere memorizzato come oggetto locale.

I meta dati che vengono iniettati nella virtual machine e descrivono tutte le funzioni offerte dalle API native della piattaforma e vengono ricavati durante il processo di compilazione dell'applicazione utilizzando la proprietà di riflessione dei linguaggi di programmazione.

Il vantaggio di questa implementazione è che tutte le API native sono invocabili da JavaScript e anche le future versioni delle API saranno supportate subito, così come possono essere supportate librerie di terze parti scritte in Java/Obj-C.

Per permettere il riuso del codice NativeScript fornisce dei moduli che aggiungo un livello di astrazione ulteriore rispetto alle API native, questo livello contiene sia componenti grafici sia funzionalità comuni ad entrambe le piattaforme come l'accesso al filesystem o alla rete. Questo livello di astrazione è opzionale ed è sempre possibile effettuare chiamate alle API native.

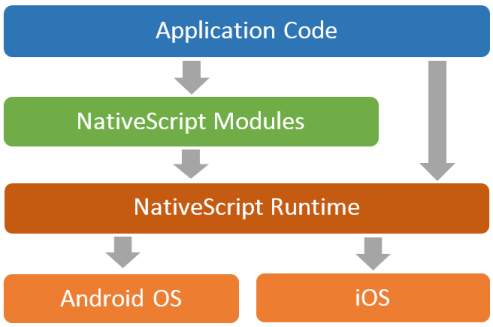


figura 2.3: Architettura di NativeScript

2.3.2 Pregi e difetti

Il pregio principale di NativeScript è che rende disponibili ad un'applicazione nativa realizzata in JavaScript tutte le funzionalità che possono essere presenti su un applicazione realizzata con l'SDK nativo.

Inoltre, l'interfaccia grafica di un'applicazione viene realizzata in modo dichiarativo mediante XML e CSS[3], in un modo analogo a quello utilizzato per le applicazioni web.

Tuttavia, le funzionalità offerte dal livello di astrazione sono limitate e di conseguenza è necessario utilizzare le API native. Questo comporta la diminuzione del codice riusabile su più piattaforme e un notevole aumento della complessità, allontanandosi così dallo scopo principale dell'utilizzo del JavaScript per lo sviluppo di applicazioni native, che è quello di sviluppare in modo semplice e senza utilizzare le API native.

2.3.3 Prototipo

Sfruttando quasi solamente i componenti generici offerti da NativeScript è stato possibile ottenere un layout simile a quello dell'applicazione attuale.

Tuttavia per realizzare la visualizzazione a griglia delle immagini è stato necessario utilizzato un componente `Repeater` all'interno di una `ScrollView`, questa implementazione risulta poco efficiente e poco fluida, in quanto non essendo mappata su un componente nativo non gode delle stesse ottimizzazioni.

Sono state individuate soluzioni alternative sfruttando librerie native di terze parti, che non sono state adottate in quanto ritenute troppo complesse dal momento che facevano uso largo uso di codice specifico per iOS.

[3]NativeScript implementa un sotto insieme limitato di CSS in quando le istruzioni CSS devono essere applicate sui componenti nativi.

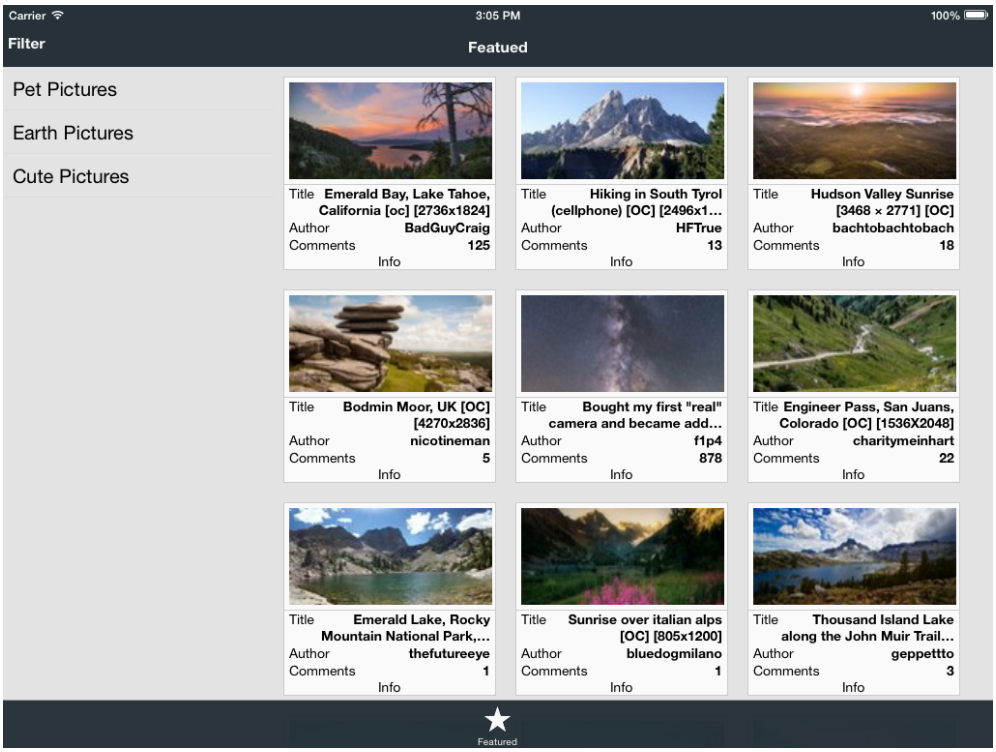


figura 2.4: Screenshot del prototipo realizzato con NativeScript

2.4 React Native

Framework sviluppato da Facebook come progetto interno per la realizzazione di applicazioni native per iOS sfruttando il JavaScript e con un funzionamento analogo a quello di React[4].

React Native è stato successivamente rilasciato come progetto open source nel Marzo 2015.

2.4.1 Come funziona

React Native è composto da una parte scritta in Obj-C e un'altra parte scritta in JavaScript. La parte realizzata in Obj-C comprende:

- una serie di classi che definiscono il "*ponte*" che permette di alla virtual machine di invocare codice nativo;
- un'insieme di macro che permettono alle classi Obj-C di esportare dei metodi in modo che questi possano essere invocati dal "*ponte*";
- un'insieme di classi che derivano dai componenti nativi di uso comune e che utilizzano le macro per esportare alcune funzionalità.

La parte realizzata in JavaScript consiste in un livello di astrazione, organizzato in moduli, che nascondere l'interazione con le componenti native, viene inoltre fornito

[4]Framework per lo sviluppo di applicazioni web pubblicato da Facebook.

il modulo `NativeModules` che permette di interagire direttamente con il "*ponte*" in modo da utilizzare oggetti nativi personalizzati.

Quando viene avvita un'applicazione con React Native, viene eseguito del codice Obj-C che istanzia la virtual machine che andrà ad interpretare il JavaScript e il"*ponte*" tra la virutal machine e il codice nativo.

Durante l'esecuzione del codice JavaScript è possibile richiedere l'esecuzione di codice nativo usando il modulo `NativeModules`, questo modulo fornisce all'oggetto "*ponte*" le informazioni necessarie per permettergli di identificare la porzione di codice nativo da eseguire. Per poter essere eseguito il codice nativo deve utilizzare le macro fornite da React Native, altrimenti il "*ponte*" non riesce ad identificare il metodo da invocare.

Al fine di ottenere prestazioni migliori, la virtual machine che esegue il JavaScript viene eseguita su un thread diverso rispetto a quello che si occupa dell'esecuzione del codice nativo e del rendering dell'interfaccia grafica.

Inoltre, la comunicazione tra questi due thread viene gestita in modo asicrono ed eseguita in blocchi, in questo modo è possibile ridurre le comunicazioni tra i due thread ed evitare che l'interfaccia grafica si blocchi durante l'esecuzione del codice JavaScript.

2.4.2 Pregi e difetti

React Native fornisce un buon livello di astrazione rispetto la piattaforma nativa, rendendo possibile ad uno sviluppatore web di realizzare un'applicazione nativa completa senza conoscere nulla riguardo il funzionamento della piattaforma sotto stante.

A causa di questa astrazione non tutte le funzionalità native sono disponibili, tuttavia è possibile adattare classi Obj-C già esistenti mediante le macro messe a disposizione dal framework, anche se questo processo prevede una buona conoscenza del linguaggio Obj-C.

Tra gli altri pregi di React Native c'è la community di sviluppatori creatasi attorno a framework, infatti, nonstante si tratti di un framework pubblicato recentemente, si è già creata una community numerosa ed è già possibile trovare dei moduli open source che estendono le funzionalità base del framework.

Infine il flusso di lavoro per lo sviluppo di un'applicazione con React Native risulta molto veloce, in quanto grazie all'utilizzo dei WebSocket, non è necessario eseguire la build dell'applicazione ad ogni modifica del codice sorgente, portando un notevole risparmio di tempo.

2.4.3 Prototipo

Il prototipo realizzato è stato in grado di soddisfare la maggior parte delle caratteristiche ricercate, infatti è stato possibile ottenere una visualizzazione a griglia, dotata di scroll infinito e fluido.

Sono stati invece incontrati dei problemi riguardanti la personalizzazione della barra di navigazione, in quando il componente offerto dal framework non permette la presenza di pulsanti personalizzati nella barra di navigazione.

Tuttavia sono state individuate alcune soluzioni che prevedono l'utilizzo di moduli open source che forniscono una barra di navigazione maggiormente personalizzabile.

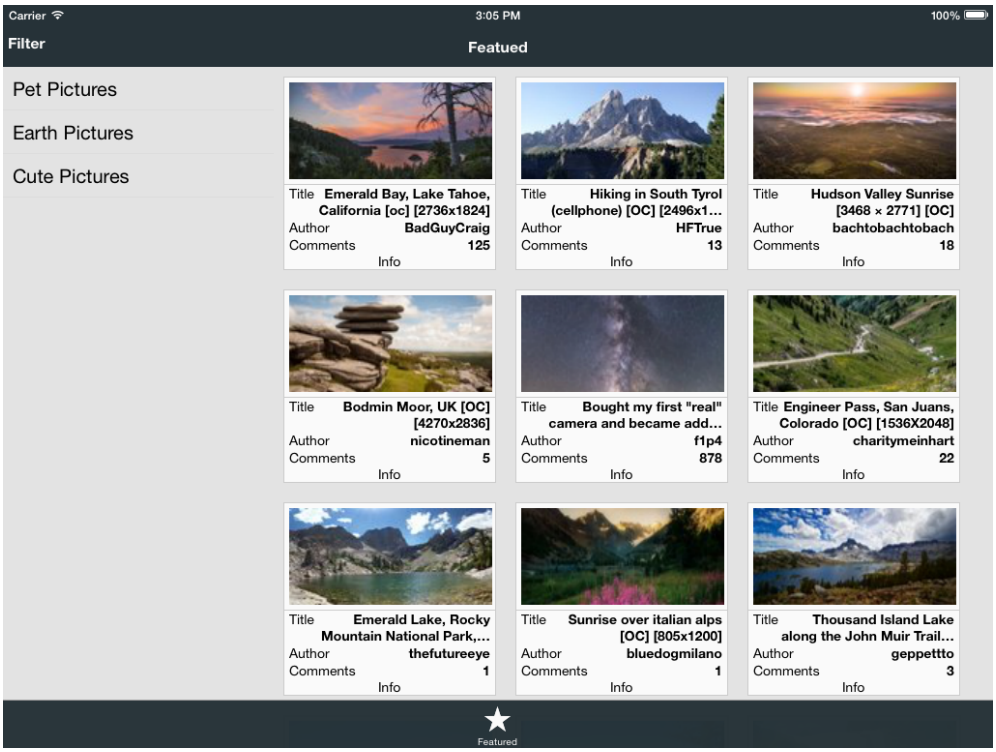


figura 2.5: Screenshot del prototipo realizzato con React Native

2.5 Confronto finale

Trovare un nome migliore

-	Tabris.js	NativeScript	React Native
Funzionamento	Utilizza dei plug-in di Cordova per rendere accessibili i componenti grafici nativi via JavaScript	Utilizza una VM modificata e dei meta dati relativi alle API native per renderne possibile l'utilizzo via JavaScript	Utilizza delle classi Obj-C per creare un ponte tra la VM che esegue il JavaScript e le componenti native
Possibilità di personalizzazione	Limitata a quando offerto dal framework	Come se l'applicazione fosse scritta in linguaggio nativo	Limitata a quanto offerto dal framework
Estensibilità	Mediante plug-in di Cordova	Mediante librerie native di terze parti	Estendendo librerie native di terze parti con le macro offerte dal framework
Piattaforme supportate	iOS e Android	iOS e Android	iOS

tabella 2.1: Tabella comparativa dei framework analizzati

2.6 Framework scelto

Dopo aver esaminato e confrontato i tre framework individuati si è scelto di utilizzare React Native nella seconda parte dello stage.

Questo perché React Native si è dimostrato un framework molto potente anche se non offre l'accesso completo alle API native come fa NativeScript. Inoltre, la diffusione già ampia del framework e il supporto da parte di Facebook, che sta usando React Native per sviluppare le proprie applicazione, fornisce al framework un'ampia possibilità di crescita, mediante l'estensione di nuove funzionalità o il supporto di ulteriori sistemi operativi, come Android.

Per quanto riguarda NativeScript, è stato scartato principalmente perché non raggiunge l'obiettivo di nascondere la complessità dello sviluppo di un'applicazione nativa con Obj-C o Swift. Infatti, l'obiettivo che si vuole raggiungere sviluppando un'applicazione nativa in JavaScript è quello di riutilizzare il più possibile le competenze derivate dallo sviluppo web senza dover imparare tutti i dettagli dello sviluppo con l'SDK nativo.

Infine, Tabris.js è stato scartato perché è risultato troppo limitato e non è stato in grado di soddisfare alcune delle caratteristiche ricercate.

Capitolo 3

Strumenti e tecnologie utilizzate

Il contenuto di questo capitolo contiene una descrizione più dettagliata delle tecnologie e degli strumenti utilizzati per sviluppare l'applicativo oggetto dello stage.

3.1 React Native

Cosa scrivere?

Come anticipato nel precedente capitolo, si è scelto di utilizzare React Native come framework principale per lo sviluppo dell'applicazione.

L'applicazione è composta da componenti (con pattern smart & Dumb –> ogni componente ha uno stato –> rendering

3.1.1 La sintassi JSX

Parlare di come viene usata per definire il layout dei componenti

3.1.2 Componenti esterni

todo: react-native-popover e react-native-cookies

3.2 Flux

Flux è un pattern architetturale per le applicazioni sviluppate con React e React Native proposto da Facebook.

Questo pattern sfrutta il sistema di composizione delle view di React costruendo un flusso di dati unidirezionale, che a partire da un oggetto *store* diventano lo stato di un oggetto *view-controller*, che a sua volta li fornisce ai propri componenti.

L'unico modo per modificare i dati presenti in uno *store* è mediante un *action*, quando un *view-controller* vuole modificare i dati a seguito di un evento scatenato dall'utente, crea un *action* che, mediante un *dispatcher*, viene ricevuto dai vari *store* dell'applicazione.

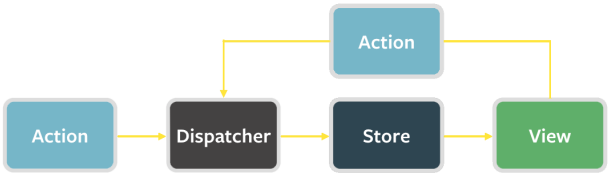


figura 3.1: Diagramma del pattern Flux

Come anticipato, Flux prevede tre tipologie principali di componenti:

- **Stores:** sono dei *singleton* che contengono i dati dell'applicazione, forniscono solamente dei metodi *getter*, per interagire con i dati contenuti è necessario usare le *actions*.
- **Actions:** sono degli oggetti che contengono delle informazioni riguardante alle varie operazioni che possono eseguite dagli *stores* dell'applicazione. Tipicamente vengono create dai *view-controller* di React e contengono già i dati necessari agli *store* per aggiornarsi, nel caso di operazioni asincrone i *view-controller* creano l'azione che verrà comunicata al *dispatcher* solamente quando sono stati caricati i dati.
- **Dispatcher:** oggetto che riceve un *action* e ne esegue il broadcast verso tutti gli *stores* registrati dell'applicazione. Fornisce delle funzionalità che permettono ai vari *stores* di registrarsi e di specificare eventuali dipendenze verso altri *stores* dell'applicazione in modo che l'aggiornamento di un determinato *store* venga effettuato una volta completato l'aggiornamento degli *stores* da cui dipende, evitando così di ottenere uno stato inconsistente.

3.2.1 Sequenza delle azioni

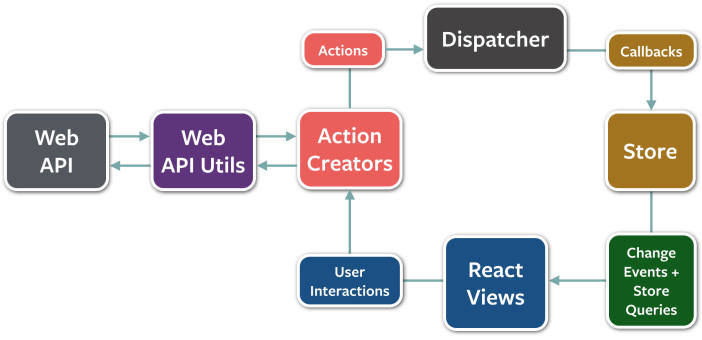


figura 3.2: Funzionamento del pattern Flux

1. L'utente esegue un'azione sulla view.
2. Il gestore dell'evento crea un *action* e la comunica al *dispatcher*.

3. Il *dispatcher* manda a tutti gli *stores* registrati l'oggetto *action* ricevuto.
4. Ogni *store* esamina l'oggetto *action* e se necessario si aggiorna.
5. Gli *stores* che hanno subito modifiche emettono un evento per comunicare ai componenti React in ascolto che si devono aggiornare.
6. I componenti React richiedono agli *stores* i dati per aggiornarsi.

3.2.2 Differenze con MVC

Nonostante Flux ed MVC possano sembrare due pattern totalmente diversi, in realtà Flux è una variante del MVC classico con delle modifiche che lo adattano al funzionamento di React e React Native.

Infatti, con MVC, i *controllers* interagiscono con il *model* e le *view* dell'applicazione visualizzano i dati presenti al suo interno. Quando il *model* viene modificato, le *view* vengono notificate e recuperano i dati aggiornati dal model.

Mentre con Flux, i *view-controller* aggiornano il model, definito dagli *stores*, in un modo più strutturato utilizzando le *actions* e, una volta che l'aggiornamento degli *store* è completato, i *view-controller* vengono notificati in modo che possano recuperare i nuovi dati.

Considerando che per React e React Native una *view* e il relativo *controller* sono lo stesso oggetto diventa chiaro che la logica di base è la stessa, l'unica differenza è come viene effettuato l'aggiornamento dei dati, che con Flux deve passare attraverso delle *actions*.

Questo vincolo imposto dall'utilizzo delle *actions* permette di circoscrivere la logica di aggiornamento del model all'interno del model stesso, limitando la complessità dell'applicazione, che nel caso di grandi applicazioni può diventare ingestibile.

Un altro vantaggio che viene dall'adozione di Flux con React riguarda l'aggiornamento dell'interfaccia grafica a seguito di una modifica dei dati, in quanto sia React che Flux ragionano a stati: con React l'interfaccia grafica visualizza uno stato dell'applicazione, il cambiamento dello stato comporta il re-rendering dell'interfaccia, mentre con Flux l'insieme degli *stores* rappresenta lo stato dell'applicazione e l'esecuzione di un'azione comporta il cambiamento dello stato.

Di conseguenza è possibile collegare direttamente lo stato definito dagli *stores*, con lo stato dei componenti grafici, limitando il numero di operazioni intermedie.

3.3 Flow (?)

Riferimenti bibliografici

Siti Web consultati