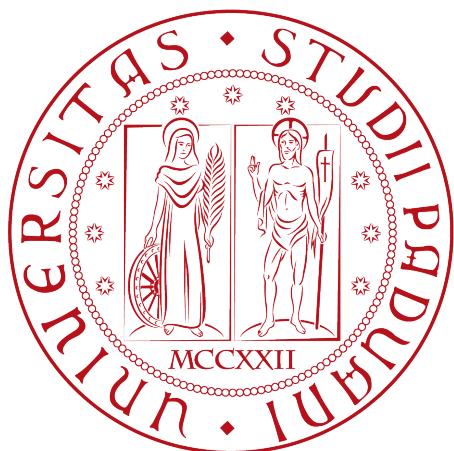


Università degli Studi di Padova

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.

Ringraziamenti

Innanzitutto, vorrei ringraziare il relatore della mia tesi, il Prof. Claudio Enrico Palazzi, per l'aiuto e il sostegno fornитоми durante la stesura del lavoro.

Un ringraziamento speciale va ad Alberto e in generale a tutto il team di WARDA S.r.l., che mi hanno permesso di vivere questa bella esperienza in un settore dell'informatica così interessante.

Desidero inoltre ringraziare con affetto i miei genitori e i miei zii che mi hanno aiutato e sostenuto durante il mio percorso di studi.

Infine, non posso non ringraziare tutti gli amici e compagni di corso che mi hanno sopportato per tutto questo tempo, sperando che continuino a farlo, perché senza di loro non sarei riuscito a raggiungere questo obiettivo.

Padova, Ottobre 2015

Giacomo Manzoli

Indice

1 Il contesto aziendale	1
1.1 L'azienda	1
1.1.1 WARDA - Work on what you see	1
1.2 Il progetto	2
2 Framework analizzati	3
2.1 Considerazioni generali	3
2.1.1 Differenze con le applicazioni ibride	4
2.2 Tabris.js	4
2.2.1 Come funziona	5
2.2.2 Pregi e difetti	5
2.2.3 Prototipo	5
2.3 NativeScript	6
2.3.1 Come funziona	6
2.3.2 Pregi e difetti	8
2.3.3 Prototipo	8
2.4 React Native	9
2.4.1 Come funziona	9
2.4.2 Pregi e difetti	10
2.4.3 Prototipo	10
2.5 Confronto finale	12
2.6 Framework scelto	12
3 Strumenti e tecnologie utilizzate	13
3.1 React Native	13
3.1.1 La sintassi JSX	14
3.1.2 Oggetti JavaScript per la definizione dello stile	15
3.1.3 JavaScript ES6 e ES7	15
3.1.4 Animazioni	16
3.1.5 Componenti esterni	16
3.1.6 Creazione di un progetto	17
3.1.7 Packager	17
3.1.8 Developer Menu	17
3.2 Flux	18
3.2.1 Sequenza delle azioni	19
3.2.2 Differenze con MVC	20
3.2.3 flux	20
3.3 Atom e Nuclide	20

3.4 Xcode	20
3.5 Google Chrome Dev Tools	21
4 Analisi dei Requisiti	23
4.1 Applicazione attuale	23
4.1.1 Pagina di visualizzazione della gallery	23
4.1.2 Pagina di dettaglio di un asset	27
4.2 Requisiti individuati	28
4.2.1 Requisiti Funzionali	28
4.2.2 Requisiti di Vincolo	30
4.3 Riepilogo requisiti	31
5 Progettazione	33
5.1 API REST di WARDA	33
5.1.1 Autenticazione	33
5.1.2 Gallery	33
5.2 Architettura generale dell'applicazione	37
5.2.1 Model	38
5.2.2 Utils	38
5.2.3 Stores	39
5.2.4 Actions	39
5.2.5 Pages	43
5.2.6 Components	44
5.2.7 Navigazione	45
6 Realizzazione	47
6.1 Dal prototipo alla gallery	47
6.2 Sistema di navigazione	49
6.3 Sistema dei filtri	50
6.4 Visualizzazione di dettaglio	51
6.5 Animazioni	51
6.6 Gestione degli errori	54
7 Conclusioni	57
7.1 Valutazione del risultato e di React Native	57
7.1.1 Requisiti soddisfatti	58
7.2 Aspetti critici e possibili estensioni	58
7.3 Conoscenze acquisite	59
Glossario	61
Riferimenti	63

Elenco delle figure

1.1	Logo di WARDÀ S.r.l.	1
2.1	Screenshot del prototipo realizzato con Tabris.js	6
2.2	Schema rappresentante il runtime di NativeScript	7
2.3	Architettura di NativeScript	8
2.4	Screenshot del prototipo realizzato con NativeScript	9
2.5	Screenshot del prototipo realizzato con React Native	11
3.1	Developer menu di React Native	18
3.2	Diagramma del pattern Flux	18
3.3	Funzionamento del pattern Flux	19
3.4	Tools di debug di Xcode	21
3.5	Debug di un'applicazione con Google Chrome	22
4.1	Screenshot della gallery dell'applicazione attuale	24
4.2	Screenshot della lista dei nodi dell'applicazione attuale	25
4.3	Dettaglio della griglia degli assets dell'applicazione attuale	25
4.4	Popover che mostra i dettagli di un asset	26
4.5	Screenshot della lista dei filtri dell'applicazione attuale	26
4.6	Pagina di dettaglio di un asset	27
4.7	Pagina di dettaglio di un asset con i dettagli visibili	28
4.8	Requisiti per importanza	31
4.9	Requisiti per tipologia	31
6.1	Immagine all'inizio dello swipe	53
6.2	Immagine durante lo swipe	53

7.1 Screenshot dell'applicazione realizzata	57
---	----

Elenco delle tabelle

2.1 Tabella comparativa dei framework analizzati	12
4.1 Requisiti Funzionali	30
4.2 Requisiti di Vincolo	31
4.3 Numero di requisiti per importanza	31
4.4 Numero di requisiti per tipologia	31

Elenco dei frammenti di codice

2.1 Esempio di creazione di un oggetto nativo con NativeScript	7
3.1 Esempio della sintassi JSX di React Native	14
3.2 Esempio della definizione dello stile di un componente di React Native	15
3.3 Utilizzo di LayoutAnimation	16
5.1 JSON Schema di GET /g/{galleryCode}/nodi/{args}	34
5.2 JSON Schema di GET /g/galleryCode/contents/args	35
5.3 Esempio dell'array da utilizzare per applicare dei filtri alla risorsa /g/galleryCode/contents/args	36
5.4 JSON Schema di GET /g/galleryCode/filtro/filtro/args	36
5.5 JSON Schema di GET /g/galleryCode/filtro/filtro/args	37
5.6 Action - load nodes	40
5.7 Action - load assets	40
5.8 Action - load more assets	41
5.9 Action - clear assets	41
5.10 Action - load asset details	41
5.11 Action - load filter items	42
5.12 Action - apply filter	42
5.13 Action - remove filter	42
5.14 Action - clear assets	42
5.15 Action - network error	43

6.1	Funzione che gestisce l'evento onScroll della griglia che visualizza gli assets	48
6.2	NodesStore - Caricamento dei nodi	49
6.3	WardaFetcher - Caricamento degli assets considerando i filtri	50
6.4	AssetDetailPage - Animazione della comparsa/scomparsa lista dei dettagli	51
6.5	AssetDetailPage - Spostamento dell'immagine allo swipe dell'utente . .	52
6.6	AssetDetailPage - Animazione dello swipe	53

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 descrivere 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 stata realizzata un'applicazione prototipo che visualizza una serie di immagini ottenute effettuando delle 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, una tecnica per ottimizzare il caricamento di grandi quantità di dati che prevede il caricamento iniziale di un limitato numero di elementi, per poi caricare gli elementi rimanenti man mano che l'utente prosegue nella visualizzazione dei dati.
- fluidità dell'applicazione, specialmente durante il caricamento dei dati;
- possibilità di personalizzare la barra di navigazione dell'applicazione.

2.1 Considerazioni generali

Le applicazioni realizzate con questa tipologia di framework hanno alla base lo stesso funzionamento: una [virtual machine](#) interpreta il codice JavaScript e utilizza un componente “*ponte*” per modificare l'interfaccia grafica dell'applicazione, la quale è realizzata con i componenti offerti dal [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 di questo tipo è composta da componenti grafici del sistema operativo, che vengono creati e composti 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 avvengono 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](#) che visualizza un'insieme di pagine web realizzate utilizzando HTML5, CSS3 e JavaScript in modo da replicare l'aspetto di un'applicazione nativa.

Trattandosi quindi di un'applicazione web è possibile utilizzare tutti i framework e le tecnologie disponibili nell'ambito web, come AngularJS o jQuery.

Inoltre, se è già disponibile un'applicazione web per desktop, la creazione di un'applicazione mobile ibrida risulta rapida in quanto è possibile riutilizzare sia parte del codice, sia le competenze legate allo sviluppo web già in possesso dai vari sviluppatori.

Questo approccio viene utilizzato da qualche anno e permette di creare applicazioni 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 riguardanti:

- **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](#) che l'utente può effettuare per interagire con le varie applicazioni. Queste applicazioni sfruttano un complesso sistema di riconoscimento delle gesture che non è ancora replicabile in ambito web e l'esperienza d'uso delle applicazioni ibride risente di questa limitazione.
- **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 sono presenti i problemi prestazionali legati al rendering dell'interfaccia grafica, il quale non viene bloccato dall'esecuzione del codice JavaScript.

Inoltre, 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 EclipseSource¹ nel Maggio 2014, che permette di controllare mediante JavaScript i componenti dell'interfaccia grafica nativa, sia di iOS, sia di Android.

¹<http://eclipsesource.com/en/home/>

2.2.1 Come funziona

Tabris.js funziona utilizzando come "ponte" una versione modificata di Cordova, la quale, grazie a dei plug-in sviluppati da EclipseSource, permette di interagire via JavaScript con i componenti nativi del sistema operativo.

Ognuno di questi plug-in incapsula un determinato componente dell'interfaccia grafica e fornisce delle API JavaScript per controllarlo.

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.

Il framework viene pubblicato come open source, tuttavia per accedere al codice sorgente è necessario acquistare una licenza, pertanto non è stato possibile analizzare più in dettaglio il funzionamento del framework.

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. È il framework che si occupa di eseguire tutte le operazioni specifiche per le varie piattaforme e di renderizzare gli opportuni componenti grafici.

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, definendo prima la dimensione e la posizione dei vari componenti, per poi organizzarli 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 online, infatti oltre alle risorse messe a disposizione da EclipseSource e alla documentazione ufficiale, non è stato possibile trovare altre informazioni riguardanti il framework.

In ogni caso, l'applicazione realizzata è molto fluida e l'esperienza d'uso è paragonabile a quella di un'applicazione nativa.

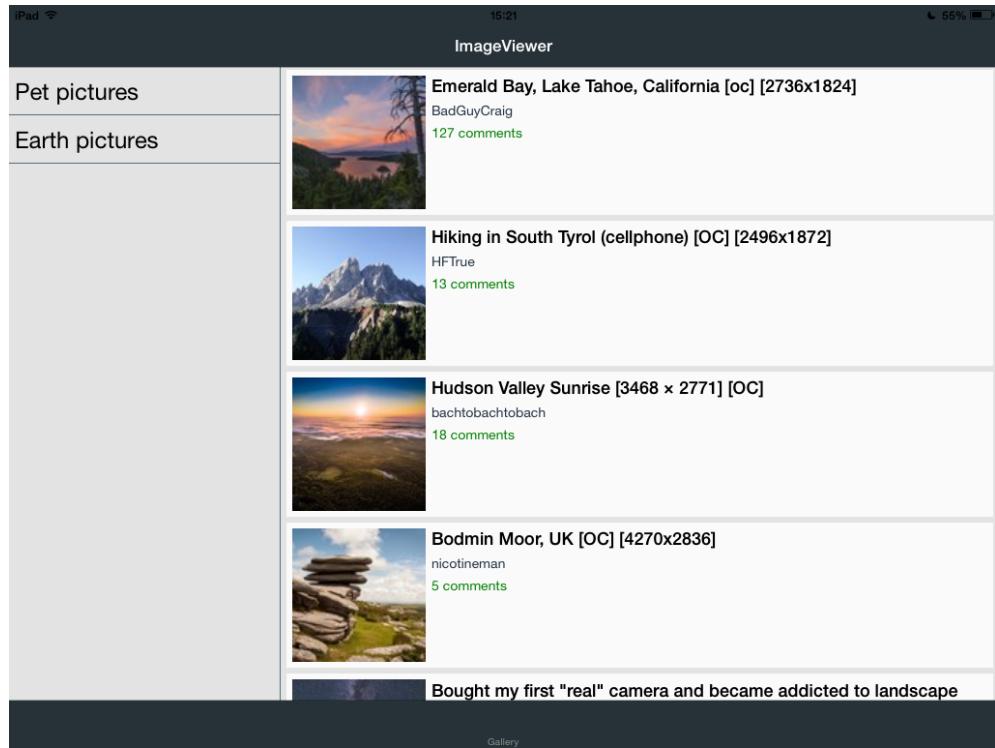


Figura 2.1: Screenshot del prototipo realizzato con Tabris.js

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 esegue del codice 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 determinato codice C++;
- l’iniezione di metadati che descrivono le API native.

In questo modo il runtime di NativeScript, ovvero la virtual machine modificata, può utilizzare i metadati per riconoscere le funzioni JavaScript che hanno una corrispondente funzione nativa, per poi richiedere l’esecuzione della funzione nativa mediante del codice C++.

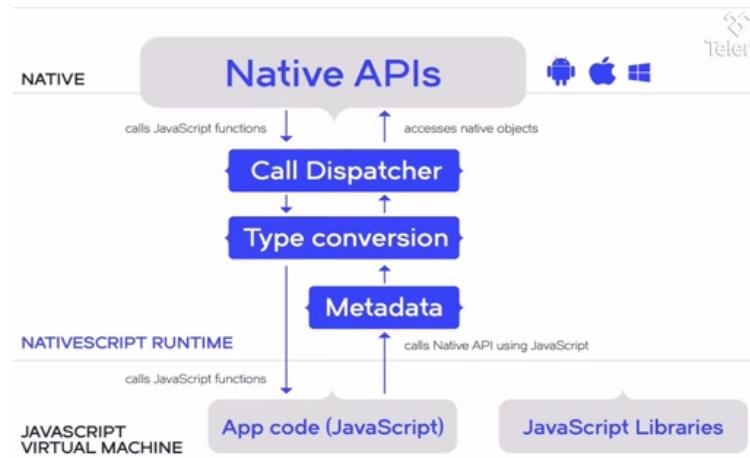


Figura 2.2: Schema rappresentante il runtime di NativeScript

Il runtime di NativeScript permette quindi di creare degli oggetti JavaScript che funzionano da **proxy** rispetto agli oggetti nativi della piattaforma e, quando l'oggetto proxy subisce delle modifiche, il runtime le riporta anche sull'oggetto nativo.

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 con NativeScript

All'esecuzione del JavaScript la virtual machine riconosce, grazie ai metadati 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 **UIAlertView** e memorizza un puntatore all'oggetto nativo in modo da poterlo recuperare in seguito per eseguire delle funzioni su di esso.

Alla fine, la virtual machine crea un oggetto JavaScript che funziona come proxy dell'oggetto nativo precedentemente creato e lo ritorna in modo che possa essere memorizzato e utilizzato come un normale oggetto JavaScript.

I metadati che vengono iniettati nella virtual machine e che descrivono tutte le funzioni offerte dalle API native della piattaforma, sono 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 potranno essere supportate appena queste vengono rilasciate.

Inoltre, questi metadati possono essere generati anche per tutte le librerie native di terze parti, rendendole disponibili in JavaScript.

Per permettere il riuso del codice NativeScript fornisce dei moduli che aggiungono un livello di astrazione ulteriore rispetto alle API native, che 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 dirette alle API native.

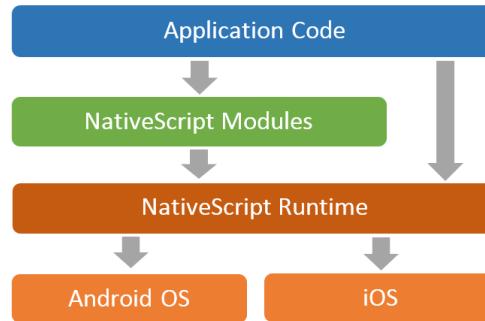


Figura 2.3: Architettura di NativeScript

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 in un'applicazione realizzata con l'SDK nativo.

Inoltre, l'interfaccia grafica di un'applicazione viene realizzata in modo dichiarativo mediante XML e CSS², in un modo analogo a quello utilizzato per le applicazioni web.

Tuttavia, le funzionalità offerte dal livello di astrazione sono limitate e di conseguenza per ottenere effetti particolari 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 fanno largo uso di codice specifico per iOS.

²NativeScript implementa un sotto insieme limitato di CSS in quanto le istruzioni CSS devono essere applicate sui componenti nativi.

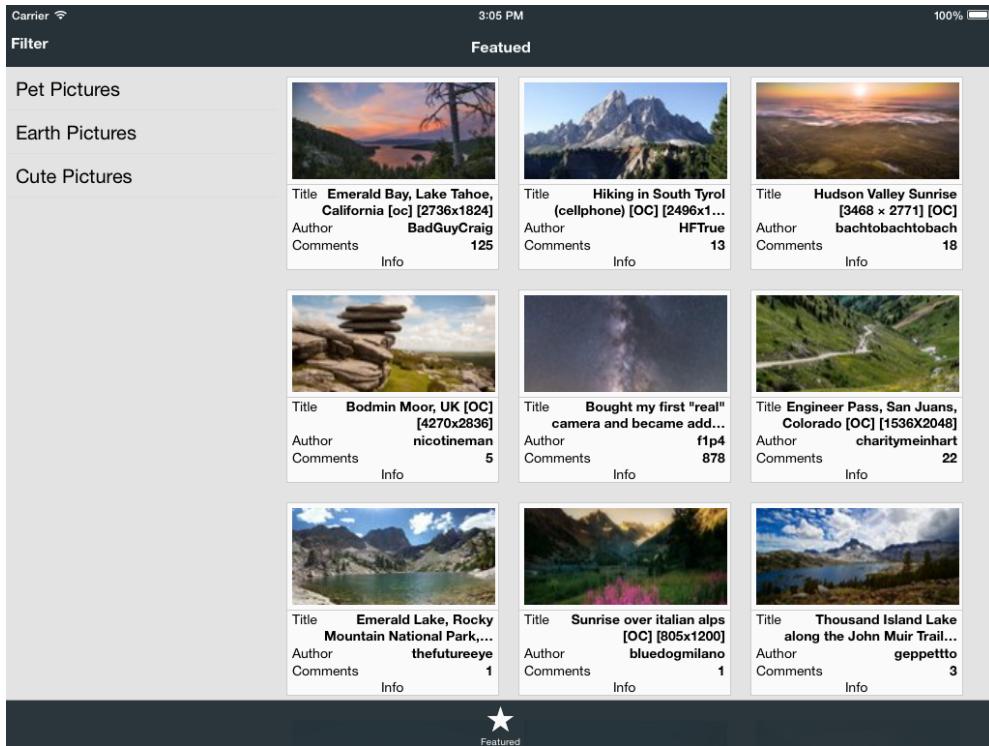


Figura 2.4: Screenshot del prototipo realizzato con NativeScript

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 React³.

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 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.

³Framework per lo sviluppo di applicazioni web pubblicato da Facebook <http://facebook.github.io/react/>.

La parte realizzata in JavaScript consiste in un livello di astrazione, organizzato in moduli, che nasconde l'interazione con le componenti native.

Tra questi moduli si trova la maggior parte dei componenti grafici necessari per la realizzazione di un'interfaccia grafica e per l'accesso ad alcune delle funzionalità del dispositivo, come le notifiche o i servizi di localizzazione.

È inoltre presente il modulo **NativeModules** che permette di interagire direttamente con il “*ponte*” in modo da utilizzare oggetti nativi personalizzati.

Quando viene avvata un'applicazione con React Native, viene eseguito del codice Obj-C che istanzia sia la virtual machine che andrà ad interpretare il JavaScript, sia il “*ponte*” tra la virutal machine e la piattaforma nativa.

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 asincrono 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, dando la possibilità 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, tuttavia questo processo prevede una buona conoscenza del linguaggio Obj-C e della piattaforma sottostante.

Tra gli altri pregi di React Native c'è la community di sviluppatori creatasi attorno al framework, infatti, nonostante 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 quanto il componente offerto dal framework non permette la presenza di pulsanti personalizzati nella barra di navigazione.

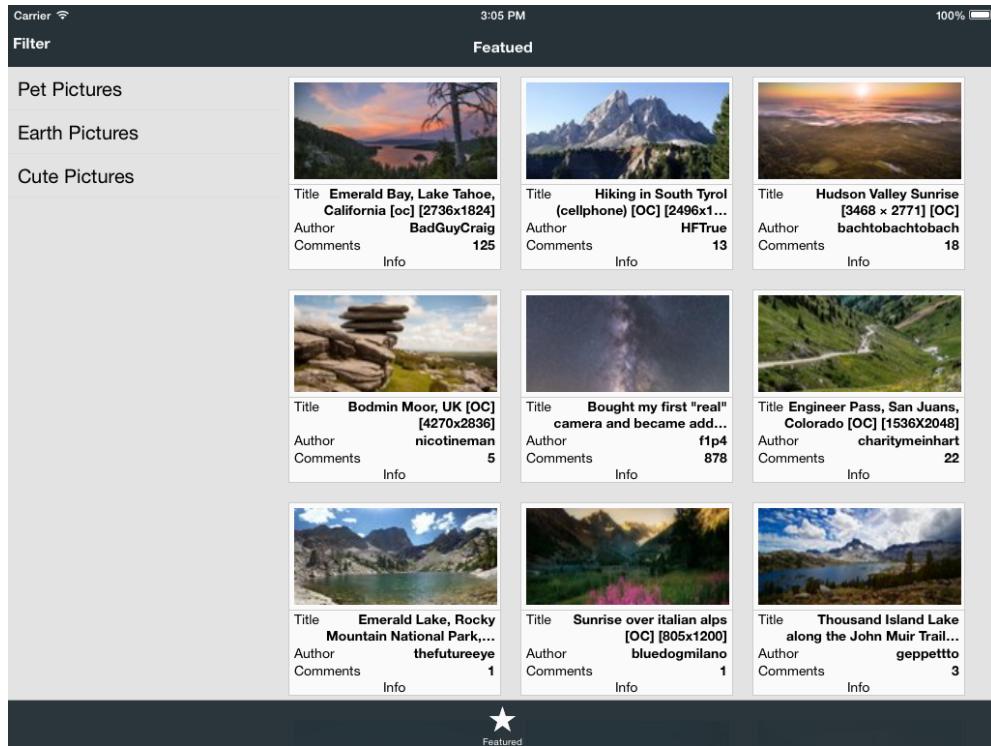


Figura 2.5: Screenshot del prototipo realizzato con React Native

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

2.5 Confronto finale

-	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 metadati 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 quanto 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 per la 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 attualmente utilizzando React Native per sviluppare alcune delle proprie applicazioni, 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.

Come anticipato nel precedente capitolo, si è scelto di utilizzare React Native come framework principale per lo sviluppo dell'applicazione, il che vincola la scelta di alcuni strumenti e tecnologie.

3.1 React Native

Trattandosi di un framework per la definizione di interfacce grafiche, React Native prevede di strutturare l'applicazione secondo componenti, ognuno dei quali viene definito combinando componenti standard offerti dalla libreria o altri componenti definiti dallo sviluppatore.

React Native offre una serie di componenti basilari che permettono di definire l'interfaccia grafica dell'applicazione e che vengono poi tradotti in componenti nativi. Questi componenti possono essere sia semplici come `View`, `Text` o `Image`, sia più complessi come `TabBarIOS` o `MapView`.

Ogni componente di un'applicazione realizzata con React Native ha sempre due proprietà:

- **state**: un oggetto che contiene le informazioni riguardanti lo stato del componente, la modifica di questo oggetto comporta il re-rendering dell'interfaccia grafica. Tipicamente viene utilizzato per memorizzare gli oggetti che contengono le informazioni da visualizzare sull'interfaccia grafica.
- **props**: un oggetto che contiene le informazioni che il componente riceve dal componente che lo contiene, spesso consistono in dati da visualizzare o in funzioni di callback da invocare al verificarsi di determinati eventi. Tipicamente i campi dati di questo oggetto vengono considerati immutabili in modo da evitare *side effect* indesiderati.

Questi due oggetti portano ad un pattern comune nella progettazione dei componenti detto *Smart & Dumb*, che prevede la divisione dei componenti in due categorie, quelli *smart* che svolgono il ruolo di *controller* dell'applicazione e quelli *dumb* sono più simili a dei template.

Tipicamente un componente *dumb* non ha un proprio stato e si limita a visualizzare i dati ricevuti mediante l'oggetto `props` o ad invocare delle callback al verificarsi di

determinati eventi. In questo modo si ottengono dei componenti generici, indipendenti dall'applicazione che possono essere testati e riutilizzati facilmente.

Un componente *smart* invece è tipicamente composto da più componenti, sia *dumb* che *smart*, ed è dotato di un proprio stato, contenente i dati da passare ai componenti figli. Inoltre, un componente di questo tipo contiene la definizione delle funzioni che si occupano della gestione degli eventi.

Riportando questo pattern in un'ottica [MVC](#), i componenti *dumb* funzionano come le *view* mentre i componenti *smart* funzionano da *view-controller*. Questo perché definiscono sia l'interfaccia grafica, utilizzando altri componenti, sia la logica applicativa.

Nel caso l'applicazione segua il design pattern [Flux](#) (§3.2), i componenti *smart* sono quelli che si occupano di recuperare lo stato dagli *stores* e di creare le varie *actions*.

La divisione in componenti dell'applicazione influenza anche la l'organizzazione del codice. Tipicamente viene definito un singolo componente per file di codice, il quale contiene tutto il codice del componente, sia quello riguardante la logica di funzionamento, sia quello riguardante la logica di layout. Questo è reso possibile dal fatto che le informazioni riguardanti lo stile sono definite come oggetti JavaScript e il layout viene definito utilizzando la sintassi JSX.

3.1.1 La sintassi JSX

La sintassi JSX permette di inserire all'interno del codice JavaScript alcuni pezzi di codice XML, che devono essere poi trasformati in JavaScript normale per poter essere eseguiti.

Il vantaggio offerto da questa sintassi è quello di poter definire in modo dichiarativo come i vari componenti dell'applicazione si compongono tra loro, semplificando così la definizione dell'interfaccia grafica.

```

1 render(){
2   return (
3     <View style={styles.container}>
4       <Text style={styles.welcome}>
5         Welcome to React Native!
6       </Text>
7       <Text style={styles.instructions}>
8         To get started, edit index.ios.js
9       </Text>
10      <Text style={styles.instructions}>
11        Press Cmd+R to reload,{'\n'}
12        Cmd+D or shake for dev menu
13      </Text>
14    </View>);
15 }
```

Codice 3.1: Esempio della sintassi JSX di React Native

Nell'esempio sopra riportato viene definito il layout di un componente e, grazie alla sintassi derivata dall'XML, è facile intuire da quali elementi è composto e come questi elementi sono combinati tra loro.

Nel caso di React Native la traduzione da JSX a JavaScript viene fatta dal *packager* prima della compilazione dell'applicazione (§3.1.7).

3.1.2 Oggetti JavaScript per la definizione dello stile

Con React Native la definizione dello stile dei componenti di un'applicazione viene effettuata utilizzando degli oggetti JavaScript che hanno dei campi dati simili alle proprietà dei CSS.

Questa scelta è stata effettuata perché il team di sviluppo di React Native ha dovuto implementare un sistema che trasformi le proprietà CSS in attributi dei componenti nativi ed ha ritenuto più comodo utilizzare degli oggetti JavaScript.

In questo modo vengono risolti alcuni problemi dei CSS, come la località dei nomi delle classi e la gestione delle variabili. Inoltre, non è più necessario riferirsi ad una determinata classe CSS utilizzando una stringa, in quanto basta usare il campo dati di un oggetto JavaScript, evidenziando così eventuali errori, come l'utilizzo di una classe inesistente.

Per creare uno di questi oggetti è necessario utilizzare delle apposite API messe a disposizione da React Native.

Queste API sono racchiuse all'interno del modulo `StyleSheet` e, mediante il metodo `create`, permettono di creare un oggetto che definisce lo stile di un componente a partire da un normale oggetto JavaScript.

```

1 var styles = StyleSheet.create({
2   container: {
3     flex: 1,
4     justifyContent: 'center',
5     alignItems: 'center',
6     backgroundColor: '#F5FCFF',
7   },
8   welcome: {
9     fontSize: 20,
10    textAlign: 'center',
11    margin: 10,
12  },
13  instructions: {
14    textAlign: 'center',
15    color: '#333333',
16    marginBottom: 5,
17  },
18});
```

Codice 3.2: Esempio della definizione dello stile di un componente di React Native

Le impostazioni delle stile sono analoghe a quelle offerte dai CSS ed includono alcune funzionalità che non sono ancora pienamente supportate nell'ambito web come il sistema di layout flexbox. Questo sistema prevede che un componente dell'applicazione possa andare a modificare le dimensione dei componenti che contiene, in modo da occupare al meglio lo spazio disponibile e di allineare in vari modi i componenti contenuti.

3.1.3 JavaScript ES6 e ES7

Il codice JavaScript prodotto utilizzando React Native segue lo standard ES5 dal momento che è lo standard supportato dalla versione attuale di JavaScriptCore.

Tuttavia è possibile utilizzare alcune funzionalità specifiche degli standard ES6 e ES7, come la destrutturazione degli oggetti e l'utilizzo delle classi, dal momento che il *packager* di React Native (§3.1.7), prima di compilare l'applicazione nativa, compila il codice JavaScript utilizzando Babel¹, un compilatore per JavaScript che trasforma la

¹<https://babeljs.io/>

sintassi ES6 e ES7 in modo che sia conforme allo standard ES5.

3.1.4 Animazioni

Le animazioni sono una delle caratteristiche principali dell'esperienza d'uso delle applicazioni mobile e la fluidità delle animazioni è uno dei fattori che differenzia le applicazioni native da quelle ibride.

Per la creazione delle animazioni React Native offre due moduli: **LayoutAnimation**, per le animazioni riguardanti le modifiche al layout dei componenti, e **Animated**, per definire animazioni personalizzate e specifiche per alcuni componenti.

Entrambi questi moduli sono realizzati completamente in JavaScript e quindi non usufruiscono delle funzionalità offerte dalla piattaforma nativa. Nonostante ciò la fluidità delle animazioni può essere paragonabile a quella di un'applicazione nativa.

Come già anticipato, il modulo **LayoutAnimation** permette di effettuare in modo animato le modifiche subite dal layout a causa dell'esecuzione delle funzione di rendering dei componenti. Questo, in combinazione con il sistema di layout flexbox, permette di ottenere animazioni come la comparsa o scomparsa di una barra laterale o il ridimensionamento animato dei componenti con una sola riga di codice.

```

1 onPress() {
2   LayoutAnimation.spring(); //Imposta l'esecuzione della prossima funzione di
                           //rendering in modo animato
3   this.setState({...}); //Modifica lo stato del componente in modo da
                       //causarne il re-rendering
4 }
```

Codice 3.3: Utilizzo di LayoutAnimation

Il modulo **Animated** permette invece di andare a definire animazioni più complesse, sfruttando dei componenti ad hoc il cui layout può essere definito utilizzando particolari valori che quando vengono modificati, producono un'animazione. Un tipico utilizzo di questa tipologia di animazioni è quello di spostare alcuni elementi grafici in base ad un **pan** effettuato dall'utente.

Le animazioni definite con questo modulo possono poi essere collegate tra loro, in modo da ottenere animazioni complesse come quelle legate al **pinch-to-zoom** su un'immagine.

3.1.5 Componenti esterni

React Native supporta la gestione dei moduli secondo lo standard CommonJS², questo permette di utilizzare npm per la gestione delle dipendenze con i componenti esterni.

Sfruttando le possibilità offerte da npm, la community di sviluppatori ha già iniziato a pubblicare alcuni componenti per le applicazioni di React Native, che possono essere facilmente integrati nella propria applicazione. Una raccolta di questi componenti può essere trovata sul sito React Parts³, il quale contiene componenti sia per React che per React Native.

²<http://requirejs.org/docs/commonjs.html>

³<https://react.parts/native-ios>

3.1.6 Creazione di un progetto

Per creare un'applicazione con React Native è necessario installare l'interfaccia a riga di comando che è disponibile come pacchetto npm con il nome di `react-native-cli`.

Una volta installata questa interfaccia è possibile creare un progetto con il comando `react-native init <NomeProgetto>`, il quale si occupa di creare una cartella contenente tutto il necessario per il funzionamento dell'applicazione.

Tra i vari file creati è possibile trovare il progetto di Xcode, il quale contiene tutto il codice Obj-C necessario all'esecuzione dell'applicazione.

Per avviare l'applicazione è sufficiente aprire il progetto, selezionare il simulatore o un dispositivo e premere il pulsante *Play*, dopo un po' verrà avviata l'applicazione.

3.1.7 Packager

Il *Packager* è un programma presente all'interno di React Native che permette di creare dei bundle JavaScript contenenti il codice dell'applicazione.

Quando richiesto, il *Packager* esegue la conversione da JSX in JavaScript, trasformando anche il JavaScript ES6 in JavaScript ES5 e successivamente crea un bundle, unendo tutti i file JavaScript necessari al funzionamento dell'applicazione.

Il bundle deve poi essere aggiunto al progetto Xcode in modo che sia disponibile sul dispositivo.

Durante lo sviluppo è inoltre possibile configurare l'applicazione in modo che il bundle con il codice JavaScript venga recuperato da un server presente nel computer con il quale si sta sviluppando.

Così facendo è possibile eseguire ogni volta il codice aggiornato senza dover reinstallare l'applicazione sul dispositivo o sul simulatore. Il progetto Xcode che viene creato dall'interfaccia a riga di comando è configurato in modo che avvi in modo automatico questo server.

3.1.8 Developer Menu

Durante lo sviluppo di un'applicazione con React Native è possibile accedere ad alcune funzionalità messe a disposizione dal framework per facilitare lo sviluppo delle applicazioni.

Queste funzionalità sono accessibili durante l'esecuzione dell'applicazione, indipendentemente dal fatto questa sia in esecuzione sul simulatore o su un dispositivo reale.

Per far comparire il menù quando l'applicazione viene eseguita sul simulatore è necessario premere i tasti `Cmd+D`, mentre se l'applicazione viene eseguita su un dispositivo reale è necessario scuotere il dispositivo.

Il menù è composto dalle voci:

- **Reload:** permette di ricaricare il bundle dell'applicazione.
- **Debug in ...:** permette di eseguire il debug dell'applicazione utilizzando Chrome o Safari, maggiori informazioni sono disponibili nella sezione §3.5.
- **Show FPS Monitor:** permette di visualizzare il numero di **FPS** dell'interfaccia grafica dell'applicazione.
- **Inspect Element:** permette di analizzare i vari componenti dell'interfaccia grafica, visualizzando, per ogni componente selezionato, i componenti che lo contengono, lo stile del componente e le dimensioni effettive del componente.

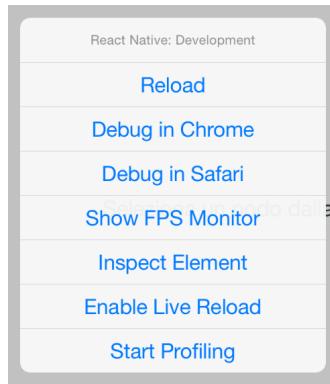


Figura 3.1: Developer menu di React Native

- **Enable Live Reload:** quando abilitato l'applicazione viene ricaricata in modo automatico ad ogni modifica subita dai file sorgenti.
- **Start Profiling:** permette di visualizzare gli stack delle chiamate durante l'esecuzione dell'applicazione, relativi sia alla parte Obj-C, sia alla parte JavaScript.

Questo menù viene rimosso automaticamente quando l'applicazione viene compilata per essere pubblicata nello store.

3.2 Flux

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

L'obiettivo di questo pattern è quello di organizzare la gestione dei dati dell'applicazione in modo che ci sia un flusso di dati unidirezionale sfruttando il sistema di composizione dei componenti di React.

Il flusso parte da degli oggetti *stores*, che contengono i dati dell'applicazione, questi dati vengono poi prelevati da alcuni componenti *smart* dell'applicazione, che a loro volta li forniscono ai componenti che li compongono.

Per modificare i dati presenti in uno *store* è necessario creare un oggetto *action* che, mediante il *dispatcher* dell'applicazione, viene ricevuto dai vari *stores*, i quali lo utilizzano per aggiornare i dati che contengono.

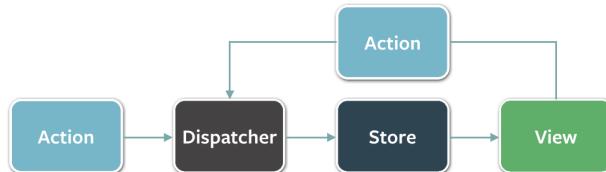


Figura 3.2: Diagramma del pattern Flux

Come anticipato, Flux prevede tre tipologie principali di componenti:

- **Stores:** sono dei singleton che contengono i dati dell'applicazione e che forniscono solamente dei metodi *getter* per recuperarli. Una volta creati, gli *stores* restano in attesa dell'esecuzione di un *action*, la quale può essere utilizzata per aggiornare i dati contenuti nello *store*.
- **Actions:** sono degli oggetti che contengono delle informazioni riguardante alle varie operazioni che possono eseguire dagli *stores* dell'applicazione. Tipicamente vengono create dai *view-controller* di React e contengono già i dati necessari agli *stores* per aggiornarsi. Nel caso di operazioni asincrone i *view-controller* creano l'azione che verrà comunicata al *dispatcher* solamente quando le istruzioni asincrone sono state completate.
- **Dispatcher:** oggetto che riceve un *action* e ne esegue il broadcast verso tutti gli *stores* dell'applicazione. Fornisce delle funzionalità che permettono ai vari *stores* di registrarsi e di specificare eventuali dipendenze verso altri *stores*, in modo che un determinato *store* venga aggiornato una volta completato l'aggiornamento degli *stores* da cui dipende, evitando così di ottenere uno stato inconsistente.

3.2.1 Sequenza delle azioni

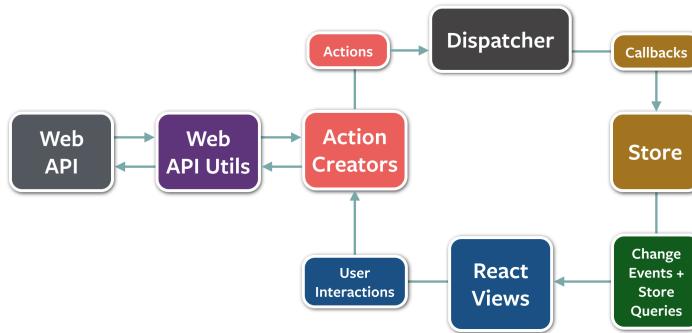


Figura 3.3: Funzionamento del pattern Flux

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 nel *model*. 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, questi vengono notificati in modo che possano recuperare i nuovi dati, che vengono poi passati ai componenti che compongono il *view-controller*.

La logica di base è quindi 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 degli *stores*, 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 e un 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.

3.2.3 flux

`flux` è un modulo pubblicato su npm da Facebook che fornisce delle classi che aiutano nell'implementazione del pattern Flux. Tra queste classi è presente l'implementazione completa di un *dispatcher* e una classe base per la creazione degli *stores* che si occupa di implementare tutta la parte relativa alla registrazione e pubblicazioni degli eventi legati all'aggiornamento dei dati.

3.3 Atom e Nuclide

Trattandosi di codice JavaScript è possibile utilizzare un qualsiasi editor di testo per sviluppare l'applicazione. Tuttavia viene consigliato l'utilizzo di Atom, un editor open source sviluppato di GitHub, che può essere personalizzato mediante dei pacchetti.

Tra i pacchetti disponibile per Atom c'è Nuclide⁴ una serie di pacchetti che aggiungono alcune funzionalità di supporto allo sviluppo con React Native, come l'auto-completamento delle keyword e l'evidenziazione della sintassi JSX.

3.4 Xcode

Nonostante il codice JavaScript possa essere scritto con qualsiasi editor di testo, è necessario utilizzare Xcode⁵ per compilare l'applicazione finale in modo da poterla

⁴<http://nuclide.io/>

⁵Xcode è disponibile solamente per Mac OS X, tuttavia è possibile sviluppare applicazioni con React Native su qualsiasi sistema operativo grazie ad Exponent <http://exponentjs.com/>. Il

installare sul simulatore di iOS o su un dispositivo Apple.

Xcode rende disponibili un serie di tools per lo sviluppo delle applicazioni native che possono essere riutilizzati durante lo sviluppo di un'applicazione con React Native. Ad esempio durante l'esecuzione dell'applicazione sul simulatore di iOS è possibile controllare il consumo della memoria, il traffico dati e l'utilizzo della CPU.

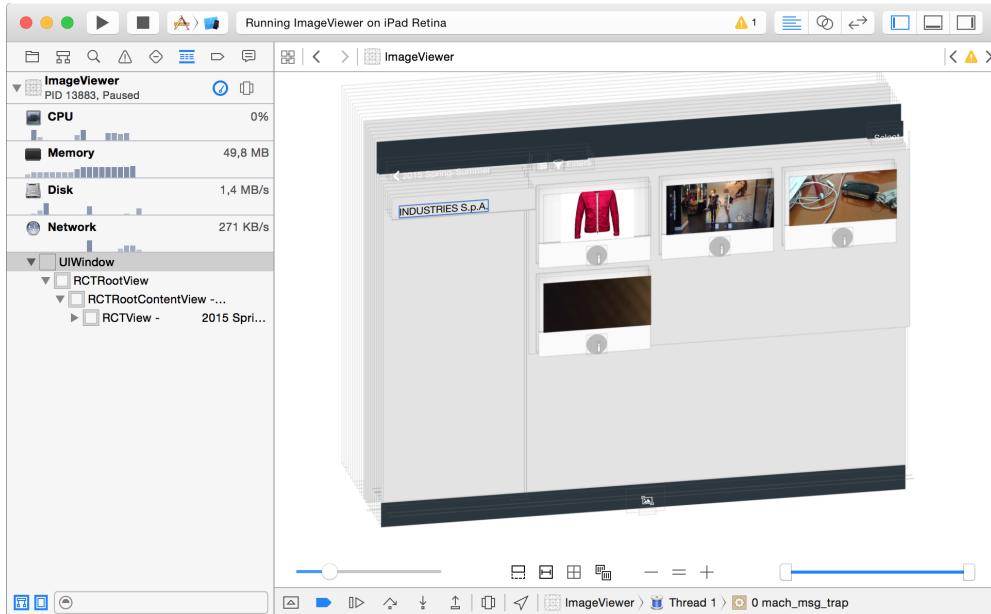


Figura 3.4: Tools di debug di Xcode

3.5 Google Chrome Dev Tools

Durante lo sviluppo di un'applicazione con React Native è possibile utilizzare gli strumenti di sviluppo offerti da Google Chrome per effettuare il debug del codice JavaScript.

In particolare è possibile utilizzare il debugger di Chrome per inserire dei break point nel codice dell'applicazione ed effettuare l'esecuzione passo passo del codice, oppure è possibile stampare sulla console di Chrome mediante l'istruzione `console.log`.

Al momento non è possibile utilizzare tutti i tools in quanto la modalità debug di React Native utilizza una virtual machine diversa per eseguire il JavaScript.

Infatti, durante l'esecuzione normale il JavaScript viene interpretato da JavaScriptCore, mentre, durante il debug, viene utilizzata la virtual machine V8 presente all'interno di Chrome, il quale comunica con l'applicazione nativa mediante WebSocket.

In questo modo Chrome riesce a controllare l'esecuzione del JavaScript, ma non riesce ad accedere alle funzionalità che vengono eseguite dai componenti nativi, come l'utilizzo delle risorse di rete o la gestione della memoria.

Per avviare il debug è sufficiente preme **Cmd+D** e selezionare l'opzione *Debug in Chrome*

funzionamento di questo servizio non è stato approfondito in quanto è stato possibile utilizzare Xcode.

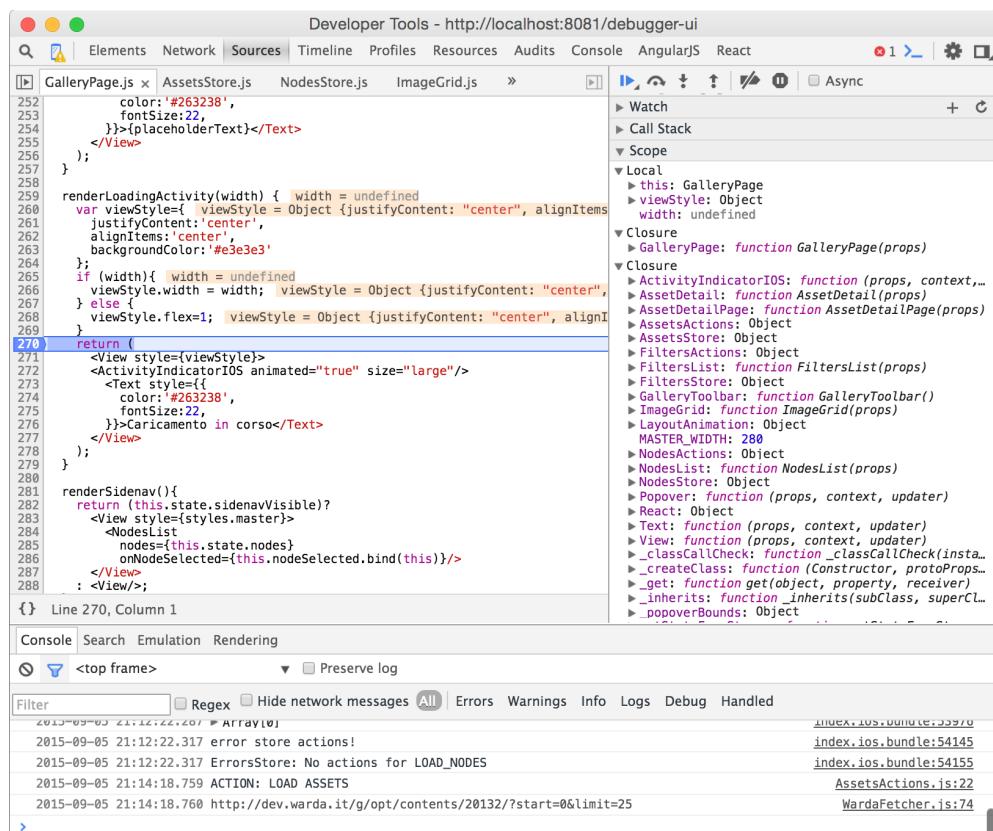


Figura 3.5: Debug di un'applicazione con Google Chrome

Capitolo 4

Analisi dei Requisiti

Questo capitolo contiene i requisiti dell'applicazione che sono stati individuati sia discutendo con il tutor aziendale, sia analizzando l'attuale client per iPad di WARDA.

Dal momento che lo scopo dello stage è quello di riprodurre l'applicazione attuale non è stato necessario effettuare un'analisi dei requisiti a partire dai casi d'uso, in quanto i requisiti possono essere estrapolati direttamente dall'applicazione attuale.

Il capitolo inizia quindi con la descrizione delle caratteristiche d'interesse dell'applicazione attuale per poi elencare i requisiti individuati.

4.1 Applicazione attuale

L'applicazione attuale per iPad di WARDA permette di visualizzare il contenuto di una gallery che si trovata sul server principale dell'applicazione.

Oltre alla visualizzazione della gallery è possibile anche creare nuovi asset, recuperando un'immagine dalla libreria interna del dispositivo o dalla fotocamera e accedere alla funzionalità collaborativa offerte dalla piattaforma WARDA.

Per il progetto dello stage, l'azienda è interessata solamente alla componente gallery dell'applicazione, in quanto si tratta della parte della applicazione che richiede la maggior quantità di risorse e che attualmente soffre di alcuni problemi prestazionali.

La struttura dati alla base di una gallery realizzata con WARDA è un albero, composto da vari nodi, ognuno dei quali contiene un'insieme di assets e dei possibili filtri.

Per WARDA un'asset è un'immagine a cui vengono associati dei metadati che la descrivono, questi metadati possono poi essere utilizzati per filtrare gli assets presenti in un nodo, in modo da permettere all'utente di visualizzare solamente gli assets con determinate caratteristiche.

4.1.1 Pagina di visualizzazione della gallery

La pagina dell'applicazione che visualizza la gallery è composta da tre parti principali:

- una lista che visualizza i nodi figli del nodo corrente;
- una griglia che visualizza gli asset presenti nel nodo corrente;
- una lista di filtri che visualizza i filtri che possono essere applicati sugli assets contenuti nel nodo corrente.

Ognuno di questi componenti sarà descritto nelle seguenti sotto sezioni.

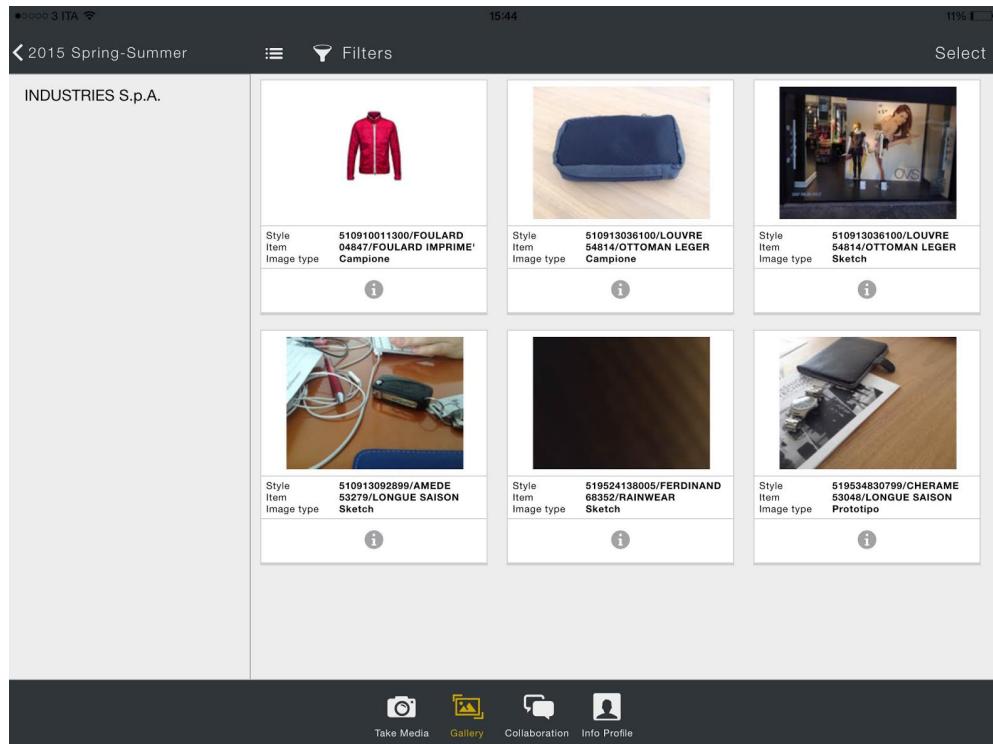


Figura 4.1: Screenshot della gallery dell'applicazione attuale

Lista dei nodi

La lista dei figli del nodo corrente visualizza il titolo di ogni nodo e, quando l'utente seleziona un nodo dalla lista, questa viene aggiornata in modo che visualizzi i nodi figli del nodo selezionato. La selezione di un nodo dalla lista comporta anche l'aggiornamento della griglia degli assets, la quale andrà a visualizzare gli assets contenuti nel nodo selezionato.

Durante il caricamento dei dati della lista viene visualizzato un indicatore di attività per fornire all'utente un feedback riguardo l'operazione di caricamento dei dati in corso.

Sopra la lista dei nodi è presente un pulsante che permette di tornare al nodo padre del nodo correntemente visualizzato.

Questo pulsante è caratterizzato da una freccia verso sinistra e dal titolo del nodo correntemente visualizzato, nel caso il nodo corrente sia il nodo radice della gallery, il pulsante non deve essere visibile.



Figura 4.2: Screenshot della lista dei nodi dell'applicazione attuale

Griglia degli assets

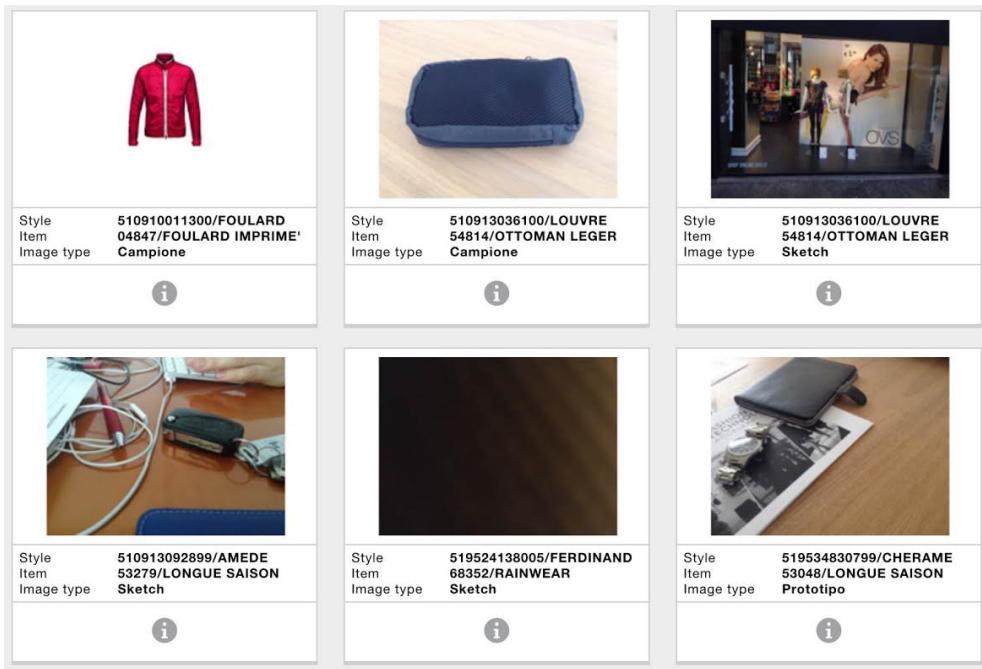


Figura 4.3: Dettaglio della griglia degli assets dell'applicazione attuale

La griglia degli assets rappresenta il componente principale dell'applicazione, questa griglia permette di visualizzare, per ogni asset contenuto nel nodo corrente, un'immagine di anteprima e un pulsante che permette all'utente di visualizzare i dettagli dell'asset mediante un [popover](#).

Se l'utente esegue un [tap](#) sull'immagine di un asset, viene visualizzata una pagina dell'applicazione contenente i dettagli dell'asset e un'immagine ingrandita, maggiori informazioni riguardo questa pagina sono disponibili nella sezione §4.1.2.

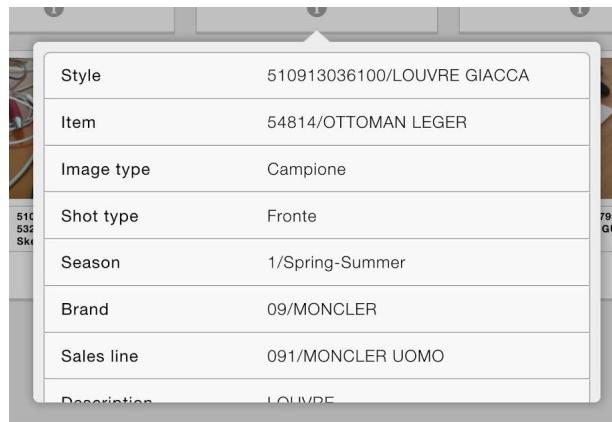


Figura 4.4: Popover che mostra i dettagli di un asset

Per motivi prestazionali, la griglia non carica subito tutti gli assets contenuti nel nodo corrente ma si limita a visualizzare solo i primi 25 assets, i successivi assets contenuti nel nodo vengono caricati man mano che l'utente prosegue nella visualizzazione della griglia, secondo il sistema dello scroll infinito.

Lista dei filtri

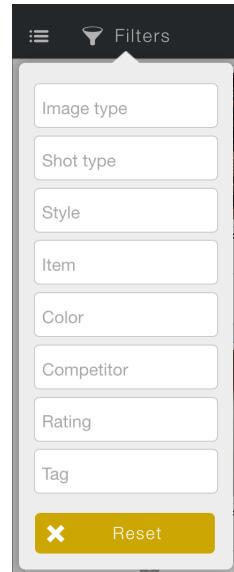


Figura 4.5: Screenshot della lista dei filtri dell'applicazione attuale

La lista dei filtri compare come popover quando l'utente esegue un tap sul pulsante “Filters” presente nella barra di navigazione.

Per ogni filtro presente nel nodo corrente, viene visualizzata una lista dei possibili valori che possono essere assegnati al filtro ed una casella di testo che permette all'utente di filtrare i valori presenti nella lista.

4.1.2 Pagina di dettaglio di un asset

Questa pagina contiene l'immagine ingrandita di un asset e una lista con tutti i dettagli dell'asset.

Mediante un apposito pulsante l'utente può nascondere o rendere visibile la lista dei dettagli, in modo da lasciare più spazio all'immagine.

Sull'immagine l'utente può eseguire alcune gesture:

- **swipe** verso sinistra, per visualizzare l'asset successivo contenuto nel nodo visualizzato dalla gallery;
- swipe verso destra, per visualizzare l'asset precedente contenuto nel nodo visualizzato dalla gallery;
- pinch-to-zoom, per ingrandire ulteriormente l'immagine.

Infine, nella barra di navigazione della pagina è presente un pulsante che permette all'utente di tornare alla pagina con la visualizzazione a griglia.

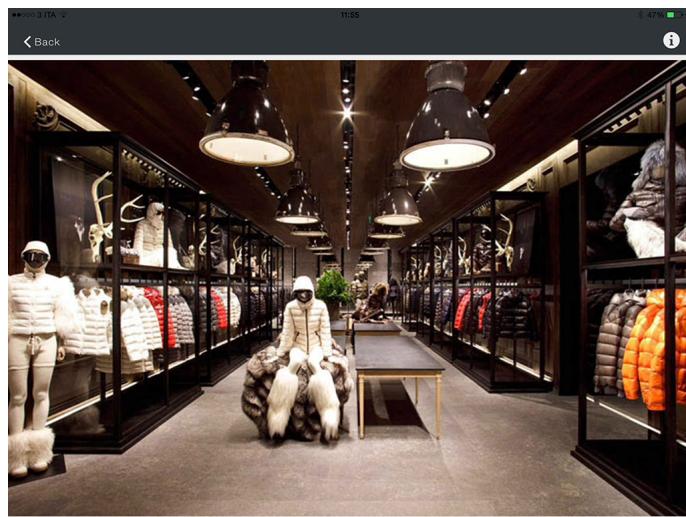


Figura 4.6: Pagina di dettaglio di un asset

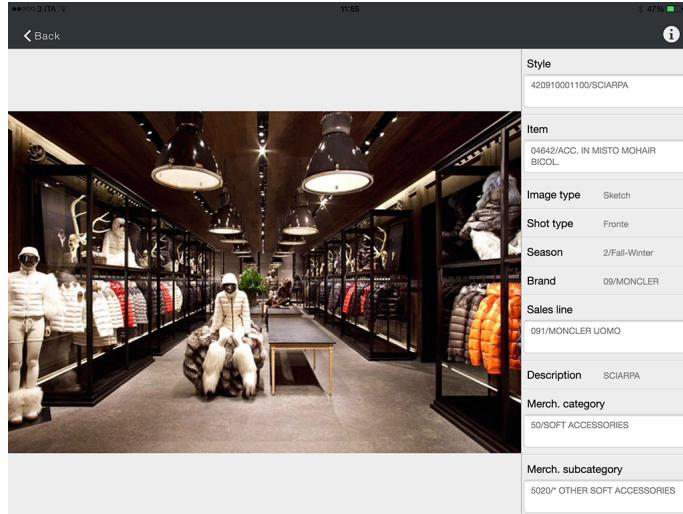


Figura 4.7: Pagina di dettaglio di un asset con i dettagli visibili

4.2 Requisiti individuati

I requisiti individuati dall’analisi dell’applicazione attuale e dalle discussioni con il tutor aziendale sono stati catalogati secondo il codice:

$$R[T]/I/[C]$$

dove:

- **Tipo:** specifica la tipologia del requisito e può assumere i seguenti valori:
 - **F - funzionale**, cioè che determina una funzionalità dell’applicazione;
 - **V - vincolo**, che riguarda un vincolo che il prodotto deve rispettare.
- **Importanza:** specifica l’importanza del requisito e può assumere i seguenti valori:
 - **O - obbligatorio**, il requisito corrisponde ad un obiettivo minimo del piano di stage e deve essere soddisfatto per garantire il funzionamento minimo dell’applicazione;
 - **D - desiderabile**, il requisito corrisponde ad un obiettivo massimo del piano di stage e deve essere soddisfatto per garantire il funzionamento dell’applicazione;
 - **F - facoltativo**, indica che il requisito fornisce del valore aggiunto all’applicazione e non era stato previsto nel piano di stage.
- **Codice:** rappresenta un codice che identifica il requisito all’interno di una gerarchia. Questo codice è definito in modo che il requisito $RTIx.y$ sia un requisito che va a definire con un grado maggiore di dettaglio alcuni degli aspetti del requisito $RTIx$.

4.2.1 Requisiti Funzionali

Id Requisito	Descrizione
RFO1	L'utente deve poter visualizzare una gallery dell'applicazione WARDA a partire dal nodo radice della gallery
RFO1.1	L'utente deve poter visualizzare la lista dei nodi contenuti nel nodo correntemente visualizzato
RFD1.1.1	Durante il caricamento della lista dei nodi, deve essere presente un indicatore di attività per evidenziare l'attività in corso
RFD1.2	L'utente deve poter rendere visibile la lista dei nodi contenuti nel nodo corrente
RFF1.2.1	La comparsa della lista dei nodi deve avvenire in modo animato
RFD1.3	L'utente deve poter nascondere la lista dei nodi dei nodi contenuti nel nodo corrente
RFF1.3.1	La scomparsa della lista dei nodi deve avvenire in modo animato
RFO1.4	L'utente deve poter spostarsi tra i nodi presenti in una gallery
RFO1.4.1	L'utente deve poter selezionare un nodo contenuto nel nodo corrente per visualizzarne i contenuti
RFO1.4.2	L'utente deve poter ritornare al nodo precedente visualizzato
RFF1.4.3	Il cambiamento degli elementi presente nella lista dei nodi deve essere animato
RFO1.4.4	Lo spostamento da un nodo all'altro deve comportare l'aggiornamento della lista dei nodi figli e della lista degli assets
RFO1.5	L'utente deve poter visualizzare la lista degli assets contenuti nel nodo correntemente visualizzato
RFD1.5.1	Durante il caricamento degli elementi della lista, deve essere presente un indicatore di attività per evidenziare l'attività in corso
RFO1.5.2	La lista contenente gli assets deve avere un layout a griglia
RFO1.5.3	La lista deve visualizzare un numero limitato di assets ed essere dotata di un sistema di scroll infinito
RFO1.5.3.1	Una volta che l'utente visualizza tutti gli assets contenuti della griglia, se presenti, devono essere caricati ulteriori assets
RFO1.5.3.2	Durante il caricamento degli ulteriori assets deve essere presente un indicatore di attività
RFO1.5.4.	Gli elementi della lista degli assets devono essere composti da un'immagine di anteprima dell'asset e da un pulsante "Info"
RFO1.5.4.1	Quando l'utente esegue un tap sull'immagine di anteprima di un asset, deve essere visualizzata la pagina di dettaglio dell'asset
RFO1.5.4.2	Quando l'utente esegue un tap sul pulsante Info di un asset, deve essere visualizzato un popover contenente le informazioni di dettaglio dell'asset
RFO1.6	L'utente deve poter visualizzare la lista dei filtri disponibili per il nodo correntemente visualizzato
RFO1.6.1	Per ogni filtro disponibile, l'utente deve poter visualizzare tutti i valori che può assumere il filtro
RFD1.6.1.1	L'utente deve poter cercare un determinato valore all'interno della lista dei valori che può assumere il filtro
RFO1.6.1.2	L'utente deve poter selezionare un valore per il filtro da applicare
RFO1.6.2	L'utente deve poter applicare più filtri contemporaneamente

Id Requisito	Descrizione
RFO1.6.3	L'utente deve poter rimuovere un filtro
RFO1.6.4	L'utente deve poter rimuovere tutti i filtri applicati
RFD1.6.5	L'utente deve poter visualizzare il numero di filtri applicati
RFO1.6.6	I filtri devono rimanere attivi anche se l'utente si sposta su un altro nodo
RFD1.6.7	La lista dei filtri deve essere visualizzata come un pop-up
RFO2	L'utente deve poter visualizzare una pagina contenente i dettagli di un asset
RFO2.1	L'utente deve poter tornare alla pagina contenente la gallery
RFO2.2	L'utente deve poter visualizzare un'immagine ingrandita dell'asset
RFD2.2.1	Durante il caricamento dell'immagine, deve essere presente un indicatore di attività che visualizzi la percentuale di caricamento
RFF2.2.2	L'utente deve poter effettuare il pinch-to-zoom sull'immagine
RFO2.2.3	L'utente deve poter effettuare uno swipe da destra verso sinistra sull'immagine, per visualizzare in dettaglio l'asset successivo secondo l'ordine del contenuto della gallery
RFF2.2.3.1	Allo swipe deve essere associata un'animazione che sposti l'immagine da destra verso sinistra seguendo il movimento effettuato dall'utente
RFF2.2.3.2	Nel caso non sia presente un'asset successivo da visualizzare, l'animazione dello swipe deve essere interrotta
RFO2.2.4	L'utente deve poter effettuare uno swipe da sinistra verso destra sull'immagine, per visualizzare in dettaglio l'asset precedente secondo l'ordine del contenuto della gallery
RFF2.2.4.1	Allo swipe deve essere associata un'animazione che sposti l'immagine da sinistra verso destro seguendo il movimento effettuato dall'utente
RFF2.2.4.2	Nel caso non sia presente un'asset precedente da visualizzare, l'animazione dello swipe deve essere interrotta
RDO2.3	L'utente deve poter visualizzare una lista contenente i dettagli dell'asset visualizzato
RFD2.4	L'utente deve poter rendere visibile la lista contenente i dettagli dell'asset visualizzato
RFF2.4.1	La comparsa della lista contenente i dettagli deve avvenire in modo animato
RFD2.5	L'utente deve poter nascondere la lista contenente i dettagli dell'asset visualizzato
RFF2.4.1	La scomparsa della lista contenente i dettagli deve avvenire in modo animato
RFF3	L'utente deve visualizzare un messaggio d'errore nel caso l'applicazione non riesca a connettersi con il server

Tabella 4.1: Requisiti Funzionali

4.2.2 Requisiti di Vincolo

Id Requisito	Descrizione
RVD1	L'applicazione deve essere dotata di un file di configurazione che permette di impostare: l'indirizzo del server a cui connettersi, l'id della gallery da visualizzare, l'username e password con i dati da utilizzare per effettuare l'accesso e il numero di assets da visualizzare in una singola pagina
RVO2	L'applicazione deve essere realizzata con React Native
RVO3	L'applicazione deve essere compatibile con iPad di seconda generazione
RVD4	L'interfaccia grafica dell'applicazione deve essere fluida e non bloccarsi

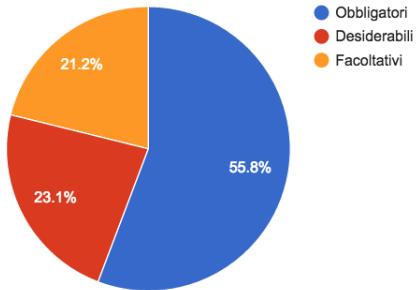
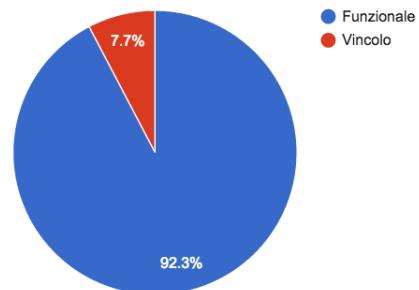
Tabella 4.2: Requisiti di Vincolo

4.3 Riepilogo requisiti

In totale sono stati individuati 52 requisiti, ripartiti tra le varie tipologie secondo quanto riportato nelle seguenti tabelle.

Importanza	#
Obbligatori	29
Desiderabili	12
Facoltativi	11
Totale	52

Tipologia	#
Funzionali	48
Vincolo	4
Totale	52

Tabella 4.3: Numero di requisiti per importanza**Tabella 4.4:** Numero di requisiti per tipologia**Figura 4.8:** Requisiti per importanza**Figura 4.9:** Requisiti per tipologia

Capitolo 5

Progettazione

Questo capitolo inizia con una descrizione delle API REST messe a disposizione dalla piattaforma WARDA, per poi andare a descrivere i vari componenti dell'applicazione realizzata.

5.1 API REST di WARDA

Le API REST di WARDA rendono disponibili tutte le funzionalità dell'applicazione, tuttavia per la realizzazione dell'applicazione saranno utilizzate solamente quelle relative all'autenticazione e alla navigazione della gallery.

5.1.1 Autenticazione

Le API di WARDA offrono due risorse diverse per effettuare il login, `/authenticate` per le applicazioni e `/login` per le pagine HTML. Nel caso dell'applicazione sviluppata deve essere utilizzata la prima risorsa che deve essere richiesta come POST e con un corpo contenente i campi dati `username` e `password`, codificati come form-data.

Nel caso che l'autenticazione sia andata a buon fine, il server risponde con un header HTTP 200, mentre se l'autenticazione fallisce risponde con un header 401.

5.1.2 Gallery

Le API di WARDA offrono varie risorse che permettono di navigare la gallery, di applicare dei filtri, di caricare nuovi assets o di eliminarne uno già presente. Gli URL delle risorse richiedono sempre due parametri, `galleryCode` e `args`, il primo specifica l'id della gallery alla quale si vuole accedere e il secondo specifica il percorso a partire dal nodo radice della gallery. Ad esempio l'URL `/g/opt/nodi/20151/` corrisponde ai nodi figli del nodo `20151` che è il figlio del nodo radice della gallery `opt`.

Nel realizzare l'applicazione sono state utilizzate saranno utilizzate solamente alcune delle risorse disponibili, le quali verranno descritte nelle seguenti sotto sezioni.

GET `/g/{galleryCode}/nodi/{args}`

Questa risorsa permette di ottenere l'elenco dei nodi figli del nodo specificato dal parametro `args`.

Ad una richiesta di questo tipo il server può rispondere con un header 200, nel caso la risorsa sia presente o con un 404, nel caso in un cui non sia stato trovato alcun nodo corrispondente.

Una caratteristica di questa risorsa è che può essere richiesta senza fornire il parametro `args`, in questo caso il server fornisce come risposta la lista dei nodi figli del nodo radice della gallery.

Quando la richiesta va a buon fine, il corpo della risposta è definito secondo il seguente schema:

```

1 [ 
2   {
3     "text": string,
4     "urlContentDataStore": string,
5     "urlChildrenStore": string,
6     "disabled": boolean,
7     "leaf": boolean,
8     "expanded": boolean,
9     "filters": [
10       {
11         "name": string,
12         "label": string,
13         "url": string,
14         "position": int,
15         "type": string
16       }
17     ],
18     "permissions": [
19       {
20         "name": string,
21         "code": string
22       }
23     ],
24     "objectId": int,
25     "galleryCode": string,
26     "allowDrag": boolean,
27     "allowDrop": boolean
28   }
29 ]

```

Codice 5.1: JSON Schema di GET /g/{galleryCode}/nodi/{args}

L'applicazione per iPad necessita solo di alcuni dei campi dati ricevuti in risposta:

- **text:** titolo del nodo;
- **urlContentDataStore:** URL della risorsa REST che permette di scaricare la lista degli assets contenuti nel nodo;
- **urlChildrenStore:** URL della risorsa REST che permette di scaricare la lista dei nodi figlio del nodo;
- **filters:** array dei filtri applicabili al nodo. Di ogni oggetto filtro vengono utilizzati i campi:
 - **name:** nome che identifica il filtro;
 - **label:** etichetta del filtro da visualizzare;
 - **url:** URL della risorsa REST che permette di scaricare la lista dei possibili valori che il filtro può assumere.

GET /g/{galleryCode}/contents/{args}

Questa risorsa fornisce gli assets contenuti all'interno del nodo `args` della gallery `galleryCode`.

Ad una richiesta di questo tipo il server può rispondere con un header 200, nel caso la risorsa sia presente o con un 404, nel caso in un cui non sia stato trovato alcun nodo corrispondente.

Nel caso la richiesta vada a buon fine, il server risponde con del JSON conforme al seguente schema:

```

1 {
2   "success":boolean,
3   "totalCount":int,
4   "size":int,
5   "items": [
6     {
7       "id": int,
8       "description": string,
9       "galleryRadix": string,
10      "name": string,
11      "mimeType": string,
12      "size": int,
13      "created": int,
14      "cmisDocId": int,
15      "url": string,
16      "thumbnailUrl": string,
17      "viewerUrl": string,
18      "details": {
19        "foto": string,
20        "digital": string,
21        "products": string
22      },
23      "linkedAssets": {
24        "careLabel": string,
25        "swatch": string,
26        "seeothers": string
27      },
28      "renditions": {
29        "s": string,
30        "xl": string,
31        "xs": string,
32        "l": string,
33        "m": string
34      },
35      "allowDrop": boolean
36    }
37  ]
38 }
```

Codice 5.2: JSON Schema di GET /g/galleryCode/contents/args

Dei dati ricevuti vengono usati:

- **totalCount**: numero di assets contenuti nel nodo;
- **size**: numero di assets presenti nel campo `items`;
- **items**: array con gli assets contenuti nel nodo `args`. Di ogni asset vengono usati solamente alcuni campi dati:
 - **details.products**: URL della risorsa contenente i dettagli dell'asset;

- **renditions**: oggetto contenente gli URL per scaricare l’immagine dell’asset in varie risoluzioni.

A questa risorsa possono essere forniti ulteriori parametri che permettono di effettuare una paginazione dei risultati e di applicare dei filtri.

I parametri che possono essere forniti sono:

- **start**: indice del record dal quale deve iniziare la pagina;
- **limit**: numero che specifica la dimensione massima di una pagina;
- **filter**: array di oggetti contenenti il nome del filtro da applicare e il valore da utilizzare. Non essendo possibile inserire oggetti JavaScript all’interno di un URL è necessario convertire l’array in stringa e poi effettuare un’opportuna codifica della stringa, in modo che possa essere inserita all’interno di un URL.

Ad esempio, per recuperare 25 assets contenuti nel nodo **20141** della gallery **opt** a partire dal 75esimo asset e applicando i filtri sotto riportati è necessario effettuare una richiesta GET all’URL:

```
/g/opt/contents/20141/?start=75&limit=25&filter=%5B%7B%22property%22:%22color%22,%22value%22:%22003%22%7D,%7B%22property%22:%22shotType%22,%22value%22:%223%22%7D%5D
```

```

1 [
2   {
3     "property": "color",
4     "value": "003",
5   },
6   {
7     "property": "shotType",
8     "value": "3",
9   }
10 ]
```

Codice 5.3: Esempio dell’array da utilizzare per applicare dei filtri alla risorsa /g/galleryCode/contents/args

GET /g/{galleryCode}/filtro/{filtro}/{args}

Questa risorsa fornisce i possibili valori che può assumere il filtro **filtro** quando viene applicato ai contenuti del nodo **args** della gallery **galleryCode**.

Ad una richiesta di questo tipo il server può rispondere con un header 200, nel caso la risorsa sia presente o con un 404, nel caso in cui non sia stato trovato alcun nodo corrispondente.

Nel caso la richiesta vada a buon fine, il server risponde con del JSON conforme al seguente schema:

```

1 [
2   {
3     "id": string,
4     "description": string,
5   }
6 ]
```

Codice 5.4: JSON Schema di GET /g/galleryCode/filtro/filtro/args

L'array ricevuto in risposta contiene tutti i valori che può assumere il filtro, sotto forma di oggetti con i due campi dati:

- **id**: identificativo del valore;
- **description**: descrizione del valore che deve essere mostrata all'utente.

All'URL delle risorsa può essere passato lo stesso parametro **filter** dell'URL `/g/{galleryCode}/contents/{args}`, in questo modo i risultati ottenuti in risposta dal server vengono limitati tenendo conto dei filtri che sono già stati applicati.

GET /g/{galleryCode}/details/{assetId}/products

Questa risorsa fornisce i dettagli relativi prodotto identificato da **assetId**.

Ad una richiesta di questo tipo il server può rispondere con un header 200, nel caso la risorsa sia presente o con un 404, nel caso in un cui non sia stato trovato alcun nodo corrispondente.

I dettagli ricevuti in risposta dal server sono conformi al seguente schema:

```

1 [ 
2 { 
3   "name": string,
4   "value": string,
5   "label": string,
6   "groupField": string,
7   "selected": boolean,
8   "entityId": int
9 }
10 ]

```

Codice 5.5: JSON Schema di GET /g/galleryCode/filtro/filtro/args

L'array ricevuto in risposta contiene tutte le informazioni relative al prodotto, organizzate in oggetti, ognuno dei quali rappresenta un particolare dato. I campi dati degli oggetti ricevuti, utilizzati dall'applicazione, sono:

- **label**: etichetta del campo dati;
- **value**: valore del campo dati.

5.2 Architettura generale dell'applicazione

La progettazione dell'architettura dell'applicazione è stata effettuata con un approccio *top-down*, individuando prima le componenti di alto livello, per poi definire le componenti più specifiche, tenendo sempre come punto di riferimento l'applicazione il client per iPad attuale.

In particolare, la progettazione è partita dall'interfaccia grafica, individuando prima le pagine che l'utente visualizza, per poi passare ai componenti delle pagine ed infine al come organizzare la logica applicativa.

Per progettare l'architettura dell'applicazione sono stati seguiti i due pattern tipici di un'applicazione realizzata con React, in particolare il flusso delle informazioni segue il pattern Flux, mentre le varie componenti sono state divise secondo il pattern *Smart & Dumb*.

Considerando che l'applicazione verrà poi sviluppata in JavaScript ES6, l'architettura è stata scomposta in moduli, in modo che questi possano essere compatibili con lo standard CommonJS. In particolare un modulo può essere:

- *un componente dell'interfaccia grafica*, in questo caso si tratta di una classe, che deriva dalla classe **Component** di React Native;
- *un componente del pattern Flux*, in questo caso si tratta di un singolo oggetto che viene esportato direttamente come modulo CommonJS.
- *un componente che fornisce funzioni d'utilità*, in questo caso si tratta sempre di un oggetto che viene esportato direttamente come modulo CommonJS.

Questi moduli verranno presentati nelle successive sezioni, organizzati per funzionalità. Tra questi moduli non è presente il *dispatcher* del pattern Flux, dal momento che come è possibile utilizzare come *dispatcher* quello fornito da **flux** (§3.2.3).

5.2.1 Model

Come *model* dell'applicazione vengono usati gli oggetti ricevuti in risposta dalle chiamate alle API.

I tipi di oggetti che possono essere ricevuti in risposta sono:

- **Node**: oggetti contenuti nell'array ricevuto in risposta da

`/g/{galleryCode}/nodi/{args};`

- **Filter**: oggetti contenuti nell'array **filters** di un oggetto **Node**;

- **FilterItem**: oggetti contenuti nell'array ricevuto in risposta da

`/g/{galleryCode}/filtro/{filtro}/{args};`

- **Asset**: oggetti contenuti nell'array **items** della risposta ricevuta da

`/g/{galleryCode}/contents/{args};`

- **AssetDetail**: oggetti contenuti nell'array ricevuto in risposta da

`/g/{galleryCode}/filtro/{filtro}/{args}.`

Dal momento che l'applicazione verrà realizzata in JavaScript, non è necessario definire delle classi specifiche per ogni tipo di oggetti, in quanto è possibile utilizzare direttamente il tipo **object**.

5.2.2 Utils

Questa categoria di moduli fornisce delle funzioni di utilità globali.

L'unico modulo appartenente a questa categoria è **WardaFetcher** che espone dei metodi che permettono di effettuare le chiamate alle API REST. Inoltre, **WardaFetcher** si occupa anche di gestire la sessione, effettuando in modo automatico il login e mantenendone lo stato.

I metodi di questo oggetto che lavorano in modo asincrono utilizzano gli oggetti **Promise** presenti nello standard ES6 di JavaScript, questo perché l'utilizzo delle promesse permette di ottenere del codice più leggibile.

5.2.3 Stores

Gli *stores* sono i componenti del pattern Flux che si occupano di gestire i dati con i quali lavora l'applicazione.

Sono stati definiti 4 *stores*, ognuno con lo scopo di gestire varie tipologie di dati:

- **NodesStore:** si occupa di tenere in memoria la lista dei nodi figli del nodo corrente e di tenere traccia del percorso effettuato dall'utente all'interno della gerarchia nella gallery. Questo avviene mediante un'array contenente oggetti **Node** nel quale vengono memorizzati i nodi che ha visitato l'utente. In questo modo è sempre disponibile l'informazione riguardo al nodo corrente, che è l'ultimo elemento inserito nell'array, ed è possibile tornare al nodo padre, che è il penultimo elemento dell'array.
- **AssetsStore:** si occupa di tenere in memoria l'array degli assets contenuti nel nodo correntemente visualizzato e, se l'utente ha selezionato un asset, contiene anche l'array con le informazioni riguardanti l'asset selezionato.
- **FiltersStore:** si occupa di tenere in memoria l'array con i filtri disponibili per il nodo corrente, l'array con i possibili valori del filtro selezionato dall'utente e l'array contenente i filtri che sono stati applicati. Questo *store* deve essere aggiornato dopo **NodesStore**, in quanto, per come sono definite le API REST di WARDA, le informazioni riguardanti i filtri disponibili sono inserite all'interno di un nodo e, di conseguenza, per caricare i filtri disponibili è necessario che sia disponibile il nodo corrente.
- **ErrorsStore:** si occupa di memorizzare le informazioni relative agli errori che si sono verificati durante l'applicazione. Al momento l'unico errore che si può verificare è l'impossibilità di comunicare con il server.

5.2.4 Actions

Nel pattern Flux, le *actions* sono degli oggetti che il *dispatcher* invia agli *stores* per modificare i dati in essi contenuti, questi oggetti hanno un campo dati **actionType** di tipo stringa, che rappresenta il nome dell'azione e altri campi dati contenenti i dati che gli *stores* devono utilizzare per aggiornarsi.

Per ognuno degli *stores* precedentemente riportati è stato quindi definito un modulo contenente tutte le *actions* che il corrispettivo *store* può eseguire.

Ognuno di questi moduli contiene un oggetto che espone dei metodi che creano una determinata *action* e ne richiedono il *dispatch* al *dispatcher*.

Per identificare le *actions* sono stati definiti dei moduli che contengono delle costanti di tipo stringa che identificano le *actions* che riguardano un determinato di *store*. Ad esempio, le *actions* contenute in **NodesActions** sono relative ai dati di **NodesStore** e vengo identificate dalle costanti presenti in **NodesConstants**

Nonostante ad uno *store* corrisponda un modulo contenente le possibili *actions*, il pattern Flux prevede che tutti gli *stores* vengono notificati di tutte le *actions* e quindi l'esecuzione di una di esse può comportare la modifica di più di uno *store*.

Le seguenti sezioni descrivono i moduli che contengono le possibili *actions*, elencando per ogni modulo i metodi che vengono esposti e il relativo oggetto JavaScript che rappresenta l'*action* creata.

NodesActions

Questo modulo contiene le *actions* che riguardano la gestione dei nodi e l'unica *action* disponibile è quella associata al metodo `loadNodes(url)` che, dato l'URL di un nodo, permette di richiedere il caricamento dei nodi figli.

Questo metodo utilizza `WardFetcher` per scaricare i nodi figli e, nel caso il download vada a buon fine, invia al `dispatcher` un *action* contenente le seguenti informazioni:

```

1 {
2   "actionType": string, //NodesConstants.LOAD_NODES
3   "parentUrl": string, //Url ricevuto come parametro dell'azione
4   "nodes": Array<Nodes> //Array ricevuto in risposta dal server
5 }
```

Codice 5.6: Action - load nodes

Quando `NodesStore` riceve questo oggetto, deve caricare al suo interno i dati contenuti nell'oggetto in modo che questi siano disponibili all'interno dell'applicazione. Nel caso `NodesStore` contenga già dei nodi, questi devono essere sovrascritti utilizzando i nodi contenuti nell'*action*.

Inoltre, quando `FiltersStore` riceve questo oggetto, deve richiedere a `NodesStore` le informazioni relative al nodo corrente in modo da aggiornare la lista dei filtri disponibili.

AssetsActions

Questo modulo contiene i metodi che creano le azioni riguardanti la gestione degli assets, in particolare è possibile caricare degli assets, aggiungere ulteriori assets, eliminare tutti gli assets caricati oppure caricare i dettagli di un determinato asset.

L'oggetto esposto dal modulo contiene i seguenti metodi:

- **loadAssets(contentUrl,filters)**

Il metodo recupera gli assets dall'URL `contentUrl`, applicando i filtri presenti nell'array `filters`, che deve essere strutturato secondo quanto indicato nel codice 5.3.

Una volta ottenuti i dati richiede il `dispatch` dell'oggetto:

```

1 {
2   "actionType": string, //AssetsConstants.LOAD_ASSETS
3   "assets": Array<Asset>, //Array contenente gli assets da caricare
4   "totalCount": int //Numero di assets presenti nel nodo che li contiene
5 }
```

Codice 5.7: Action - load assets

Quando `AssetsStore` riceve questo oggetto deve caricare al suo interno i dati contenuti nell'oggetto in modo che questi siano disponibili all'interno dell'applicazione. Nel caso siano già presenti dei dati, questi devono essere sovrascritti.

- **loadMoreAssets(contentUrl, start, filters)**

Il metodo recupera gli assets dall'URL `contentUrl` a partire dall'asset di indice `start`, applicando i filtri presenti nell'array `filters`.

Una volta ottenuti i dati richiede il *dispatch* dell'oggetto:

```

1 {
2   "actionType": string, //AssetsConstants.LOAD_MORE_ASSETS
3   "assets": Array<Asset>, //Array contenente gli assets da aggiungere in
4   coda agli assets attuali
}
```

Codice 5.8: Action - load more assets

Quando **AssetsStore** riceve questo oggetto deve caricare al suo interno i dati contenuti nell'oggetto in modo che questi siano disponibili all'interno dell'applicazione. I dati devono essere inseriti in coda a quelli già presenti, senza sovrascrivere nulla.

- **clearAssets()**

Il metodo richiede il *dispatch* dell'oggetto:

```

1 {
2   "actionType": string //AssetsConstants.CLEAR_ASSETS
3 }
```

Codice 5.9: Action - clear assets

Quando **AssetsStore** riceve questo oggetto deve cancellare tutti i dati che contiene.

- **loadAssetDetail(asset)**

Il metodo recupera i dettagli dell'asset ricevuto come parametro, utilizzando l'URL presente nel campo dati `asset.details.products`. Una volta ottenuti i dati richiede il *dispatch* dell'oggetto:

```

1 {
2   "actionType": string, //AssetsConstants.LOAD_ASSET_DETAILS
3   "assetDetails": Array<AssetDetail> //Array contenente i dettagli dell'
4   asset
}
```

Codice 5.10: Action - load asset details

Quando **AssetsStore** riceve questo oggetto deve caricare al suo interno i dati contenuti nell'oggetto in modo che questi siano disponibili all'interno dell'applicazione. Nel caso siano già presenti dei dati, questi devono essere sovrascritti.

FiltersActions

Questo modulo contiene i metodi che creano le azioni riguardanti il sistema di gestione dei filtri.

L'oggetto esposto dal modulo contiene i seguenti metodi:

- **loadFilterItems(filter, appliedFilters)**

Il metodo recupera la lista dei possibili valori dell'oggetto `filter` ricevuto come parametro a partire dall'URL presente nel campo dati `filter.url`. Alla richiesta

dei dati viene aggiunta l'informazione riguardante i filtri che sono correntemente applicati, in quanto la lista dei valori che può assumere un filtro dipende dai filtri che sono stati applicati.

Una volta ottenuti i dati richiede il *dispatch* dell'oggetto:

```

1 {
2   "actionType": string, //FiltersConstants.LOAD_FILTER_ITEMS
3   "filterItems": Array<FilterItem> //Array con i possibili valori
4 }
```

Codice 5.11: Action - load filter items

Quando **FiltersStore** riceve questo oggetto, deve caricare al suo interno i dati contenuti nell'oggetto in modo che questi siano disponibili all'interno dell'applicazione. Nel caso **FiltersStore** contenga già i valori di un filtro, questi devono essere sovrascritti.

- **applyFilter(filterData)**

Il metodo richiede il *dispatch* dell'oggetto:

```

1 {
2   "actionType": string, //FiltersConstants.APPLY_FILTER
3   "filterData": {"filter": Filter, "filterItem": FilterItem} //coincide
4     con l'oggetto ricevuto come parametro
```

Codice 5.12: Action - apply filter

Quando **FiltersStore** riceve questo oggetto, deve applicare il filtro contenuto nel campo dati **filterData**.

- **removeFilter(filter)**

Il metodo richiede il *dispatch* dell'oggetto:

```

1 {
2   "actionType": string, //FiltersConstants.REMOVE_FILTER
3   "filter": Filter //Filtro da rimuovere, coincide con l'oggetto ricevuto
4     come parametro
```

Codice 5.13: Action - remove filter

Quando **FiltersStore** riceve questo oggetto, deve rimuovere dai filtri applicati il filtro contenuto nel campo dati **filter**.

- **resetAppliedFilters()**

Il metodo richiede il *dispatch* dell'oggetto:

```

1 {
2   "actionType": string //FiltersConstants.RESET_APPLIED_FILTERS
3 }
```

Codice 5.14: Action - clear assets

Quando **FiltersStore** riceve questo oggetto, deve cancellare tutte le informazioni relative ai filtri che sono stati applicati.

ErrorsActions

Questo modulo contiene le *actions* che riguardano la gestione degli errori e l'unica *action* disponibile è quella associata al metodo `networkError()` che segnala un problema di connessione con il server.

La segnalazione dell'errore avviene effettuando il *dispatch* dell'oggetto:

```

1 {  
2   "actionType": string //ErrorsConstants.NETWORK_ERROR  
3 }
```

Codice 5.15: Action - network error

Quando `ErrorsStore` riceve questo oggetto, deve aggiornare il messaggio d'errore in modo che contenga una descrizione dell'errore che sia comprensibile all'utente dell'applicazione.

5.2.5 Pages

Questi moduli appartengono alla categoria dei componenti *smart* dell'applicazione, in quanto si occupano di recuperare i dati dagli *stores*, per poi passarli agli altri componenti in modo che vengano visualizzati dall'utente.

Ognuno di questi moduli rappresenta una singola schermata dell'applicazione e si occupa di definire dei gestori degli eventi per le azioni che può compiere l'utente, questi gestori vengono poi passati ai componenti che compongono la pagina e tipicamente richiedono l'esecuzione di un *action*.

GalleryPage

Questa pagina è il componente principale dell'applicazione in quanto si occupa di recuperare la maggior parte dei dati dagli *stores* e visualizzarli mediante i componenti che la compongono. Inoltre, è la prima pagina che visualizza l'utente quando accede all'applicazione.

Le responsabilità di questa pagina riguardano principalmente la gestione degli eventi legati alla navigazione della gallery, alla visualizzazione dei dettagli della gallery e alla gestione dei filtri.

La pagina è composta da vari componenti ai quali fornisce i dati da visualizzare e le funzioni da invocare per gestire gli eventi causati dall'utente. Questi componenti sono:

- **GalleryToolbar**: per visualizzare la toolbar nella parte superiore dello schermo;
- **AssetsGrid**: per visualizzare gli assets contenuti nel nodo corrente;
- **NodesList**: per visualizzare i figli del nodo corrente;
- **FiltersList**: per visualizzare la lista dei disponibili e applicati;
- **AssetsDetail**: per visualizzare i dettagli di un asset selezionato dalla griglia.

AssetDetailPage

Questa pagina viene utilizzata per permettere all'utente di visualizzare i dettagli di un asset, mostrando un'immagine più grande e una barra laterale con i dettagli. Viene

inoltre fornita la possibilità all’utente di spostarsi all’asset precedente o successivo mediante uno swipe.

La pagina è composta dai seguenti componenti:

- **AssetDetailToolbar**: per visualizzare la toolbar nella parte superiore dello schermo;
- **NetworkImage**: per visualizzare l’immagine ingrandita e un indicatore di caricamento durante il caricamento dell’immagine;
- **AssetsDetail**: per visualizzare i dettagli dell’asset corrente.

5.2.6 Components

I moduli che appartengono a questa categoria rappresentano alcune componenti grafiche dell’applicazione e rientrano nella categoria *dumb*, in quanto definiscono sono come vengono visualizzati i dati e sono privi di logica applicativa.

Asset

Questo componente rappresenta un elemento della griglia degli asset ed è composto da un’immagine e un pulsante per la visualizzazione delle informazioni di dettaglio dell’asset.

Il componente deve essere in grado di rilevare quando l’utente esegue il tap sull’immagine o sul pulsante ed invocare l’apposito gestore dell’evento ricevuto dal componente padre.

AssetsGrid

Questo componente rappresenta la griglia degli assets che viene visualizzata nella pagina principale dell’applicazione.

Visualizza gli assets che riceve dal componente padre utilizzando vari componenti **Asset**, inoltre, si occupa di implementare la funzionalità dello scroll infinito, controllando, quando l’utente effettua uno scroll, se è necessario caricare ulteriori dati e nel caso sia necessario, richiede il caricamento invocando l’apposita funzione ricevuta dal componente padre.

AssetDetail

Questo componente permette di visualizzare un’array di oggetti **AssetDetail**, sotto forma di lista.

Trattandosi di un componente generico, questo viene usato sia da **GalleryPage**, sia da **AssetDetailPage**, utilizzando però uno stile grafico diverso.

NodesList

Questo componente permette di visualizzare un’array di oggetti **Node** e di rilevare quando l’utente seleziona una voce della lista. Alla selezione di un elemento viene invocata l’apposita funzione ricevuta dal componente padre.

Viene utilizzato da **GalleryPage** per visualizzare i nodi figli del nodo visualizzato, in modo da permettere all’utente di navigare la gerarchia dei nodi presenti nella gallery.

FiltersList

Questo componente rappresenta la lista dei filtri disponibili per il nodo visualizzato.

Nel caso l'utente selezioni un filtro dalla lista, la riga selezionata viene espansa in modo che sia visibile una casella di testo e la lista dei possibili valori che il filtro può assumere. La casella di testo permette all'utente di filtrare i valori presenti nella lista e la selezione di uno dei valori comporta l'applicazione del filtro, mediante l'invocazione di un'apposita funzione ricevuta dal componente padre.

Nel caso ci siano uno o più filtri applicati, le righe della lista corrispondenti a tali figli contengono anche il valore selezionato per quel determinato filtro.

NetworkImage

Questo componente permette di visualizzare un'immagine a partire da un URL.

Essendo necessario scaricare i dati dell'immagine da internet, questo componente durante il download visualizza un indicatore di attività e la percentuale di caricamento.

GalleryToolbar

Questo componente rappresenta la toolbar per la pagina che visualizza la gallery, ed è composto da tre pulsanti:

- un pulsante indietro, che visualizza il titolo del nodo corrente e che permette all'utente di tornare la nodo padre del nodo visualizzato;
- un pulsante che permette all'utente di visualizzare o nascondere la lista dei nodi figli del nodo corrente;
- un pulsante che permette all'utente di visualizzare la lista dei filtri disponibili. Se sono stati applicati dei filtri, il testo del pulsante deve riportare anche il numero di filtri applicati.

Dal momento che si è definita la toolbar come un componente *dumb*, è necessario che tutte le informazioni vengano fornite dal componente che contiene la toolbar, che nello specifico è **GalleryPage**.

AssetDetailToolbar

Questo componente rappresenta la toolbar per la pagina di che visualizza i dettagli di un asset.

È composta composta da un pulsante indietro che permette all'utente di tornare alla gallery e da un pulsante che permette di visualizzare o nascondere la lista contenente i dettagli dell'asset visualizzato.

5.2.7 Navigazione

La navigazione tra le pagine dell'applicazione avviene mediante due componenti. Il primo è una *tabbar*, una barra posta nella parte bassa dello schermo che visualizza un numero limitato di pulsanti, ad ognuno dei quali viene associata una *tab* (pagina) da visualizzare.

La *tabbar* contiene solamente una sola *tab* per la gallery ed è stata inserita solamente per replicare l'aspetto grafico dell'applicazione già esistente.

All'interno della *tab* della gallery è presente un *router*, un componente che permette di navigare tra più pagine che funziona come uno stack, in quanto per passare da una pagina ad un'altra viene effettuato il *push* di un componente sopra lo stack, mentre per tornare alla pagina precedente viene effettuato il *pop* del componente che si trova al primo posto dello stack.

Questo *router* permette di passare dalla pagina di visualizzazione della gallery alla pagina con i dettagli di un asset, per poi permettere all'utente di tornare indietro.

Capitolo 6

Realizzazione

Questo capitolo contiene la descrizione delle attività svolte e dei problemi incontrati durante lo sviluppo dell'applicazione, la quale è stata sviluppata a partire dal prototipo realizzato durante lo studio di React Native.

L'attività di codifica è stata organizzata in modo da riuscire a produrre delle versioni intermedie dell'applicazione, da utilizzare per effettuare delle demo all'interno dell'azienda e per valutare la necessità di alcune modifiche.

6.1 Dal prototipo alla gallery

Prima di iniziare lo sviluppo di nuove funzionalità sul prototipo è stato prima necessario adattarlo all'architettura progettata, modificano alcuni componenti.

La versione del prototipo che è stata utilizzata implementava già il pattern Flux mediante il modulo npm `flux` ed era organizzata con due *stores*, uno che gestiva le immagini e uno che si occupava di gestire le fonti disponibili per le immagini.

Come prima cosa sono stati modificati gli *stores* e le relative *actions* secondo quanto progettato, trasformando lo *store* delle immagini in quello degli assets e lo *store* delle fonti in quello dei nodi.

Inoltre, l'oggetto del prototipo che si occupava del recuperare i dati è stato modificato in modo che interroghi le API di WARDA e che gestisca l'autenticazione.

L'implementazione dell'autenticazione è stata semplice in quanto è bastato aggiungere una variabile booleana che specifica se l'autenticazione è già stata effettuata. Nel caso questa non sia già stata effettuata, viene prima richiesta l'autenticazione e poi viene effettuata la chiamata alle API vera e propria.

Non è stato necessario gestire cookie o quant'altro dal momento che l'oggetto `fetch` offerto da React Native per effettuare chiamate HTTP funziona come proxy del componente nativo di iOS che si occupa di gestire il traffico di rete e questo componente nativo gestisce in modo automatico i cookies.

Per quanto riguarda l'interfaccia grafica è stato necessario modificare la griglia in modo che visualizzare oggetti di tipo `Asset` e che implementasse lo scroll infinito.

La griglia degli assets è stata implementata utilizzando il componente `ListView` di React Native, il quale rende disponibile l'evento `onEndReached`, che viene sollevato ogni volta che l'utente raggiunge la fine della lista e che permette implementare lo scroll infinito.

Tuttavia questa implementazione aveva due problemi:

- il caricamento dei nuovi dati iniziava quanto l'utente arrivava alla fine delle liste, quando poteva essere anticipato, in modo da migliorare ulteriormente l'esperienza d'uso;
- in alcuni casi l'evento veniva sollevato troppe volte il che portava a caricare più volte lo stesso blocco di dati.

Questi problemi sono stati risolti implementando lo scroll infinito utilizzando l'evento `onScroll` della `ListView`, evento che viene sollevato ogni volta che l'utente esegue uno scroll.

Alla funzione che gestisce l'evento viene passato come parametro un oggetto con tutte le informazioni relative all'evento che si è verificato, in questo modo è stato possibile richiedere il caricamento di ulteriori dati ad una distanza personalizzata dalla fine ed effettuare maggiori controlli in modo da evitare che la stessa porzione di dati venisse caricata più volte.

```

1 //AssetsGrid.js
2 function onScroll(args) {
3   var scrollY = args.nativeEvent.contentOffset.y;
4   var height = args.nativeEvent.contentSize.height;
5   var visibleHeight = args.nativeEvent.layoutMeasurement.height;
6
7   if (scrollY > height - 2 * visibleHeight){
8     //Se mancano meno di due schermate alla fine della lista, richiedo il
9     //caricamento di ulteriori dati
10    //Per evitare che il caricamento venga effettuato troppe volte controllo
11    //che i dati siano cambiati rispetto all'ultima volta che ho effettuato il
12    //caricamento.
13    if (_dataChanged){ //Variabile che viene settata a true quando il
14      //componente riceve dei dati nuovi
15      _dataChanged = false;
16      var downloadStarted = this.props.onLoadMoreData();
17      var temp = this.props.assets.concat([{loading:true}]); //Visualizza un'
18      //indicatore di caricamento
19      this.setState({
20        downloading: downloadStarted,
21        dataSource: dataSource.cloneWithRows(temp)
22      });
23    }
24  }
25}

```

Codice 6.1: Funzione che gestisce l'evento onScroll della griglia che visualizza gli assets

Una volta completata la visualizzazione a griglia è stato implementato il popover con l'anteprima dei dettagli, il quale è stato realizzato utilizzando il componente `react-native-popover`¹ presente in npm. Questo componente permette di visualizzare un popover contenente altri componenti di React Native ed è stato utilizzato per contenere la lista dei dettagli di un asset.

Al termine di questa prima fase l'applicazione è in grado di visualizzare il contenuto di un nodo e di visualizzare la lista dei nodi disponibili, anche se non è ancora possibile navigare tra i vari nodi.

¹<https://github.com/jeanregisser/react-native-popover>

6.2 Sistema di navigazione

Successivamente è stato sviluppato il sistema di navigazione completo, il quale fornisce la possibilità all'utente di scendere o risalire lungo la gerarchia dei nodi.

La maggior parte delle modifiche effettuate in questa fase riguardano **NodesStores** in quanto è stato necessario implementare un sistema che tenga traccia del percorso effettuato dall'utente lungo la gerarchia della gallery.

Questo sistema è inoltre vincolato dal funzionamento delle API di WARDA, le quali, dato l'id di un nodo, permettono solamente di ottenere le informazioni riguardanti i nodi figli o gli assets in esso contenuti.

È stato quindi necessario implementare un sistema che, quando viene richiesto il caricamento dei nodi a partire da un determinato URL, sia in grado di recuperare un oggetto **Node** contenente le informazioni relative al nodo padre dei nodi di cui è richiesto il caricamento. Inoltre, questo sistema deve essere in grado di tenere traccia del percorso che l'utente ha effettuato durante la navigazione della gallery.

Per far funzionare il tutto è stato necessario modificare la funzione che carica i nodi all'interno dello *store* in modo che, prima di caricare i nuovi nodi, cerchi tra gli oggetti **Node** presenti all'interno dello *store* l'oggetto rappresentante il nodo padre dei nuovi nodi utilizzando l'URL dal quale sono stati scaricati i nuovi dati. L'oggetto viene poi inserito in coda ad un'array contenente tutti i nodi appartenenti al ramo della gallery che l'utente sta visualizzando.

Nel caso che l'utente stia tornando all nodo precedente, anziché effettuare l'inserimento dell'oggetto nodo, viene rimosso l'ultimo elemento dell'array.

```
1 //NodesStore.js
2
3 // _nodes --> Array con i nodi correntemente visualizzati
4 // _nodeHierarchy --> Array con i nodi visualizzati dall'utente, è
5 // inizializzato come array vuoto
6 function loadNodes(parentUrl, nodes) {
7
8     if (_nodesHierarchy.length === 0){
9         //E' il primo caricamento dei dati, la gerarchia dei nodi inizia da un
10        nodo "finto" che funziona da radice
11        //in quanto le API di WARDA non forniscono informazioni riguardanti il
12        nodo radice di una gallery
13        _nodesHierarchy.push({isRoot:true, text: '', urlContentDataStore: '',
14        urlChildrenStore:''});
15    } else {
16        //Ultimo nodo del ramo della gerarchia dei nodi che l'utente ha
17        visualizzato
18        var lastNode = _nodesHierarchy[_nodesHierarchy.length -1];
19
20        if (lastNode.urlChildrenStore.length < parentUrl.length){
21            //Se l'url dell'ultimo nodo è più corto (più alto in gerarchia) dell'url
22            //del nodo corrente, vuol dire che l'utente sta scendendo lungo il ramo
23            //Il nodo corrente (quello di cui sto caricando i figli) è il nodo che
24            ha urlContentDataStore == parentUrl e che si trova correntemente in
25            memoria
26            var currentNode = _nodes.filter((item) => item.urlChildrenStore ===
27            parentUrl)[0];
28            //Aggiungo il nodo corrente in coda
29            _nodesHierarchy.push(currentNode);
30        } else {
31            //L'url dell'ultimo nodo è più lungo dell'url del nodo corrente, vuol
32            dire che sto risalendo lungo il ramo
33            //Faccio un pop per togliere l'ultimo elemento dell'array
```

```

24     _nodesHierarchy.pop();
25   }
26 }
//Aggiornamento dei nodi
28 _nodes = nodes;
29 }

```

Codice 6.2: NodesStore - Caricamento dei nodi

Questa versione dell'applicazione rispecchia gli obiettivi minimi previsti dal piano di stage ed è risultata particolarmente fluida, considerando che non è stata effettuata alcun tipo di ottimizzazione per quanto riguarda la gestione della memoria.

6.3 Sistema dei filtri

Una volta completati gli obiettivi minimi è stato implementato il sistema dei filtri per il contenuto di un nodo.

La struttura di questo sistema risulta particolarmente complessa, in quanto i valori che può assumere un filtro dipendono sia dal nodo sul quale viene applicato il filtro, sia dai filtri che sono già stati applicati.

Fortunatamente le API di WARDA forniscono un livello di astrazione tale da rendere l'implementazione lato client semplice.

Infatti, per inserire il sistema di filtri all'interno dell'applicazione è bastato implementare **FiltersStore** per tenere traccia dei filtri applicati e modificare alcuni metodi di **AssetsActions** e di **WardaFetcher** in modo che prendessero un ulteriore parametro con i filtri da applicare.

```

1 //WardaFetcher.js
2 function fetchContents(contentUrl, start, filters){
3   //Funzione che scarica gli assets contenuti in un nodo
4
5   var filtersString = '';
6   //Gestione dei casi limite
7   if (!contentUrl){ contentUrl = ''; }
8   if (!start){ start = 0; }
9
10  if (filters && filters.length > 0){
11    //Conversione dell'array con i filtri da applicare in stringa
12    filtersString = JSON.stringify(filters);
13    //Codifica della stringa
14    filtersString = '&filter=' + encodeURIComponent(filtersString);
15  }
16  //Template string, una nuova tipologia di stringhe presente nello standard
17  //ES6 di JavaScript, le variabili presenti all'interno del blocco ${ }
18  //vengono sostituite con il loro valore
19  var url = '${WARDA_URL+contentUrl}?start=${start}&limit=${Config.PAGE_SIZE}${
20    filtersString}';
21
22  //La richiesta effettiva dei dati viene effettuata in modo autenticato
23  return authenticatedFetch(url)
24    .then((response) => response.json());
25}

```

Codice 6.3: WardaFetcher - Caricamento degli assets considerando i filtri

Per quanto riguarda l'implementazione grafica è stato riutilizzato il componente **react-native-popover**.

6.4 Visualizzazione di dettaglio

Durante questa fase è stato implementato il componente `AssetDetailPage` che viene visualizzato quanto l'utente effettua un tap sull'immagine di asset nella visualizzazione a griglia.

Una caratteristica di questa pagina è che quando l'utente esegue uno swipe sull'immagine dell'asset, devono essere visualizzati i dettagli dell'asset precedente o successivo in base alla direzione dello swipe, ottenendo così una visualizzazione della gallery a carosello.

L'ordine di visualizzazione degli assets nel carosello deve seguire l'ordine della griglia di `GalleryPage`, inoltre, dal momento che la visualizzazione a griglia può contenere solamente alcuni degli assets disponibili, è necessario che anche dalla visualizzazione a carosello sia possibile effettuare il download di ulteriori assets.

Nell'implementare questo sistema si sono visti i alcuni dei vantaggi dell'architettura Flux, in quanto è bastato alimentare `AssetDetailPage` con i dati presenti in `AssetsStore` e aggiungere due metodi a quest'ultimo, i quali permettono di recuperare l'asset precedente o successivo a partire da un determinato asset.

Per la gestione del caricamento di ulteriori assets viene utilizzata la stessa *action* che utilizza `GalleryPage` e, dal momento che i dati presenti in `NodesStore` sono comuni ad entrambe le pagine, quando `AssetDetailPage` richiede il caricamento di ulteriori dati, questi diventano subito disponibili anche per `GalleryPage` e viceversa, riducendo così il numero di download necessari.

Una volta ultimato il carosello è stato implementato il componente `NetworkImage` in modo che l'utente ricevesse un feedback riguardo il caricamento dell'immagine. Questo si è reso necessario dal momento che l'immagine visualizzata in questa pagina è ad alta risoluzione e il più delle volte il download richiede una quantità di tempo significativa.

L'implementazione di questo componente ha sofferto per un po' di tempo di alcuni problemi che derivavano da un bug del componente `Image` di React Native. La versione `0.9.0` del framework ha poi risolto questi bug, rendendo così possibile l'implementazione finale di `NetworkImage`.

6.5 Animazioni

Una volta soddisfatti i requisiti obbligatori e desiderabili si è passati all'introduzione di alcune animazioni, sfruttando le API offerte da React Native, che hanno permesso di ottenere animazioni fluide con delle prestazioni paragonabili a quelle delle animazioni native.

Le prime animazioni introdotte riguardano la comparsa e la scomparsa della lista dei nodi nella visualizzazione della gallery e della lista dei dettagli di un asset nella visualizzazione di dettaglio.

Queste animazioni sono state ottenute utilizzando le API `LayoutAnimation` che permettono di renderizzare l'interfaccia grafica in modo che le modifiche risultino animate.

```

1 //AssetDetailPage.js
2 function onDetailButtonPress() {
3   LayoutAnimation.easeInEaseOut(); //Imposta l'animazione
4   this.setState({detailsVisible: !this.state.detailsVisible}); //Modifica lo
      stato del componente causandone il re-rendering
5 }
```

Codice 6.4: AssetDetailPage - Animazione della comparsa/scomparsa lista dei dettagli

Come si può notare dall'esempio sopra riportato l'implementazione di queste animazioni risulta estremamente semplice.

Una volta inserite tutte le animazioni necessarie alla modifica del layout è stato animato lo swipe della visualizzazione a carosello di `AssetDetailPage`.

Questa animazione ha richiesto l'utilizzo delle API `Animated` che permettono di definire animazioni più complesse.

Per funzionare, queste API utilizzano dei particolari componenti grafici, il cui stile dipende da determinati valori che possono essere modificati in vario modo. La modifica di questi valori comporta quindi la modifica dello stile del componente in modo animato.

Ad esempio, il seguente codice implementa l'animazione dell'immagine in modo che l'immagine segua lo swipe dell'utente.

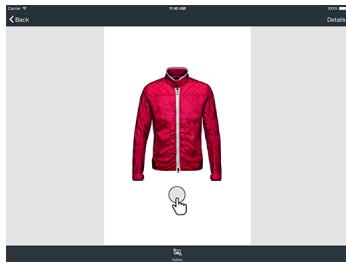
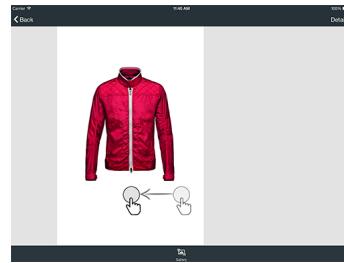
```

1 //AssetDetailPage.js - Costruttore
2 ...
3 this.state.panX = new Animated.Value(0); //Variabile che rappresenta lo
   spostamento dell'immagine
4 this.state.swipePanResponder = PanResponder.create({ //Oggetto che si occupa
   di rilevare le gesture dell'utente
5 ...
6 onPanResponderMove: Animated.event([null, {dx: this.state.panX}]), //All'
   evento onPanResponderMove, che viene sollevato quando l'utente esegue un
   pan (equivalente del drag'n'drop nei dispositivi touchscreen) viene
   collegata la variabile panX, in modo che il valore della variabile venga
   modificato e che la modifica venga effettuata in modo animato
7 ...
8 });
9
10 //funzione di rendering dell'immagine
11 return (
12   <Animated.View
13     {...this.state.swipePanResponder.panHandlers} //Il gestore degli eventi
       viene collegato alla View
14     style={..., [
15       transform: [{translateX: this.state.panX}, ], //Il valore dello
         spostamento viene associato allo stile della View, in particolare alla
         traslazione sull'asse X
16     ]}
17     //Codice che visualizza l'immagine
18   </Animated.View>
19 );

```

Codice 6.5: AssetDetailPage - Spostamento dell'immagine allo swipe dell'utente

L'effetto prodotto dal quel codice è il seguente:

**Figura 6.1:** Immagine all'inizio dello swipe**Figura 6.2:** Immagine durante lo swipe

Tuttavia, il codice sopra riportato sposta l'immagine in base al pan effettuato dall'utente e non esegue nessuna animazione quando lo swipe viene completato e viene visualizzata un'altra immagine.

Infatti, nelle applicazioni native, una volta che l'utente ha completato uno swipe, l'immagine corrente viene mandata *“fuori dallo schermo”* in modo animato, per poi visualizzare la nuova immagine.

Mentre se l'utente non completa lo swipe, l'immagine viene fatta ritornare alla posizione iniziale, sempre in modo animato.

Per implementare ciò, è stato necessario modificare il gestore dello swipe, in modo che esegua le animazioni dell'immagine quando l'utente completa o annulla la gestione.

Il codice utilizzato per effettuare le altre animazioni è il seguente:

```

1 //AssetDetailPage.js - Costruttore
2 this.state.swipePanResponder = PanResponder.create({
3   ...
4   onPanResponderRelease: (e, gestureState) => { //Funzione che viene invocata
5     //quando l'utente termina la gestione
6     var swipeSize = 150; //Dimensione dello swipe in pixel
7
8     //gestureState.dx rappresenta lo spostamento sull'asse X effettuato dall'
9     //utente
10    if (Math.abs(gestureState.dx)>swipeSize){
11      //L'utente ha completato lo swipe
12      var toValue;
13
14      if (gestureState.dx > 0) {
15        toValue = 1000; //Valore sufficientemente grande in modo che l'
16        //immagine venga renderizzata fuori dallo schermo
17      } else {
18        toValue = -1000;
19      }
20
21      //Mando l'immagine offscreen in modo animato, la direzione dipende
22      //dalla direzione della gestione (segno di gestureState.dx)
23      Animated.spring(this.state.panX, {
24        toValue,
25        velocity: gestureState.vx,
26        tension: 10,
27        friction: 3,
28      }).start(); //Avvio dell'animazione
29
30      this.state.panX.removeAllListeners();
31      var id = this.state.panX.addListener(function({value}){
32        //Aggiungo un listener all'esecuzione dell'animazione in modo di
33        //riuscire a capire quando l'immagine è finita fuori dallo schermo
34        if (Math.abs(value) > 400) { //L'immagine è fuori dallo schermo
35          this.state.panX.removeListener(id);
36        }
37      });
38    }
39  }
40}

```

```
31     var loading = (value > 0)? this.showPreviousAsset() : this.
32 showNextAsset();
33     if (loading){
34         //C'è una nuova immagine da visualizzare
35         this.state.panX.setValue(-toValue); //Posiziona l'immagine dalla
36         parte opposta dello schermo
37         Animated.spring(this.state.panX, { //Animazione che fa "entrare"
38             la nuova immagine dalla parte opposta dello schermo
39             toValue:0,
40             velocity: gestureState.vx,
41             tension:1,
42             friction:6,
43         }).start();
44     } else {
45         //Non c'è nessun immagine da visualizzare, viene effettuata l'
46         animazione che riporta l'immagine alla posizione iniziale
47         Animated.spring(this.state.panX, {
48             toValue:0,
49             velocity: gestureState.vx,
50             tension:1,
51             friction: 4
52         }).start();
53     }
54 }.bind(this));
55 } else {
56     //L'utente non ha completato lo swipe, l'immagine deve tornare alla
57     posizione iniziale in modo animato
58     Animated.spring(this.state.panX, {
59         toValue:0,
60         velocity: gestureState.vx,
61     }).start();
62 }
63 });
64 }
```

Codice 6.6: AssetDetailPage - Animazione dello swipe

Sfruttando le stesse API si è inoltre provato da implementare il pinch-to-zoom, per permettere all'utente di ingrandire l'immagine.

Tuttavia il pinch-to-zoom è una gestione complessa che combina più gestioni:

- pinch, per regolare lo zoom;
 - pan, per spostare l'immagine una volta zoomata;
 - swipe, per cambiare l'immagine.

È stata effettuata un'implementazione parziale di questa gestione che è risultata poco performante e pertanto si è deciso, in accordo con il tutor aziendale, di non proseguire lo sviluppo di tale funzionalità.

6.6 Gestione degli errori

Nell'ultima fase si è stata implementata la gestione degli errori di connessione.

Le versioni precedenti dell'applicazione, infatti, non gestivano gli errori di comunicazione con il server, e nel caso se ne verificasse uno, i vari indicatori di attività rimanevano attivi fino al riavvio dell'applicazione.

È stato quindi implementato `ErrorsStore` in modo che fosse possibile memorizzare i vari messaggi d'errore, inoltre, sono state modificati tutti i metodi che creano delle azioni, in modo che, se la promessa ritornata dai metodi di `WardarFetcher` fallisce, anziché eseguire il *dispatch* normale dell'azione, richiedano ad `ErrorsActions` la creazione di un errore di rete, mediante il metodo `networkError()`.

La visualizzazione dei messaggi d'errore è stata affidata al componente di navigazione principale dell'applicazione, in modo quando si verifichi un errore venga renderizzato il messaggio al posto della *tabbar*.

È stato inoltre modificato il componente `NetworkImage` in modo che nel caso si verifichi un errore durante il caricamento dell'immagine venga visualizzato un messaggio d'errore al posto di un'immagine grigia.

Capitolo 7

Conclusioni

In questo capitolo finale vengono tratte le conclusioni riguardo alle attività svolte durante il periodi di stage.

7.1 Valutazione del risultato e di React Native

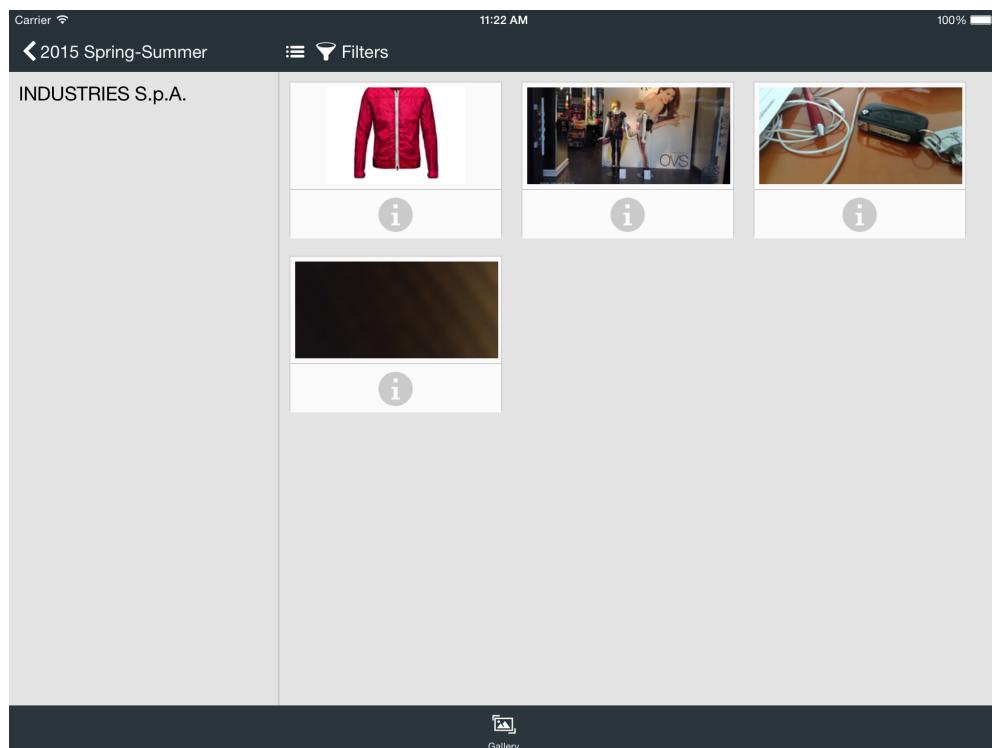


Figura 7.1: Screenshot dell'applicazione realizzata

L'applicazione sviluppata soddisfa in pieno le aspettative dell'azienda e risulta più fluida dell'applicazione attuale, dimostrando che l'utilizzo di un framework diverso da quello attuale permette di ottenere prestazioni migliori.

Considerando inoltre che l'applicazione sviluppata non utilizza particolari ottimizzazioni si ritiene che ci sia un ulteriore margine di miglioramento.

Questo è stato reso possibile da React Native, il quale ha permesso di riuscire a sviluppare un'applicazione nativa come se fosse una normale applicazione web, garantendo una velocità di apprendimento notevole.

React Native non è ancora perfetto, infatti, durante lo sviluppo dell'applicazione si sono riscontrati alcuni bug all'interno del framework, dovuti principalmente al fatto che si tratta di un framework ancora giovane. In ogni caso, la maggior parte di questi bug sono stati man mano risolti dalle release avvenute durante il periodo di sviluppo dell'applicazione, segno che gli sviluppatori di React Native sono attenti ai problemi segnalati dalla community.

Considerando anche il workflow di sviluppo che offre il framework e il tool di supporto presenti, la valutazione di React Native non può essere altro che positiva.

Inoltre, la roadmap di React Native prevede la pubblicazione della versione per Android e ulteriori ottimizzazioni e funzionalità, il tutto supportato da Facebook che sta attualmente utilizzando React Native per alcune delle sue applicazioni che sono pubblicate nell'App Store di iOS e nel Google Play Store.

7.1.1 Requisiti soddisfatti

L'applicazione soddisfa tutti i requisiti obbligatori e desiderabili individuati, mentre dei requisiti facoltativi non viene soddisfatto il requisito **RFF2.2.2 - L'utente deve poter effettuare il pinch-to-zoom sull'immagine**.

In totale sono stati soddisfatti 47 su 48 requisiti funzionali, mentre sono stati soddisfatti tutti e 4 i requisiti di vincolo.

Per misurare la fluidità dell'applicazione è stato utilizzato il FPS Monitor offerto da React Native, il quale ha rilevato una media di 60 FPS durante l'utilizzo dell'applicazione.

Durante il caricamento di ulteriori dati nella visualizzazione a griglia si è registrata la maggior parte delle volte un leggero calo degli FPS che da 60 sono scendono a 54-55. Solo una volta l'applicazione è scesa a 34 FPS per un breve periodo di tempo.

Considerando che il minimo numero di FPS necessari perché un'interfaccia grafica risulti fluida all'occhio umano è di 30 FPS, il risultato ottenuto è più che accettabile.

7.2 Aspetti critici e possibili estensioni

Gli unici problemi incontrati con l'utilizzo di React Native riguardano alcuni bug, ma come è già stato detto, questi sono dovuti al fatto che si tratta di un framework nuovo ed in ogni caso, i bug segnalati il più delle volte vengono risolti dalla release successiva.

Per quanto riguarda l'applicazione, l'aspetto grafico è stato lasciato in secondo piano e quindi può essere migliorato, specialmente per quanto riguarda l'aspetto dei pulsanti e delle icone.

Questa scelta è stata fatta dal momento che lo scopo principale dell'applicazione è quello di valutare se l'utilizzo di un framework diverso possa portare alla realizzazione di un'applicazione migliore rispetto al client per iPad attuale.

Per lo stesso motivo non sono stati sviluppati test d'unità automatizzati in quanto è stato ritenuto più interessante provare ad implementare funzionalità aggiuntive offerte dal framework come le animazioni, piuttosto che realizzare i test.

Sempre riguardo l'applicazione sviluppata, è stata implementata solamente la visualizzazione della gallery, trascurando tutte le altre funzionalità offerte dal client

attuale, come la pagina di autenticazione, la creazione di nuovi assets e la possibilità di utilizzare la parte collaborativa del sistema WARDA, funzionalità che non sono state analizzate e implementate per motivi di tempo.

7.3 Conoscenze acquisite

Durante le attività di stage sono state acquisite competenze sia nello sviluppo di applicazioni mobile, sia in altri settori ad esso correlati.

Principalmente è stato studiato come è possibile sviluppare applicazioni mobile utilizzando JavaScript, sia sotto forma di applicazioni ibride, sia come applicazioni native, analizzando le varie strategie utilizzate per astrarre le API native dei dispositivi mobile.

Inoltre, i framework analizzati durante il periodo di stage erano totalmente sconosciuti, mentre al termine dello stage sono state acquisite delle conoscenze basilari riguardo i vari framework. In particolar modo per quanto riguarda React Native è stata raggiunta una buona padronanza del framework, specialmente nello sviluppo di un'applicazione secondo i pattern tipici, che erano anch'essi sconosciuti prima dell'inizio dello stage.

Sempre legato allo sviluppo, si sono acquisite delle competenze riguardo ai tools offerti da Xcode per il debug di applicazioni native, che prima non erano mai stati utilizzati.

Sono state inoltre rafforzate le competenze riguardo il linguaggio JavaScript, che era già noto prima dello stage, ma del quale non ero a conoscenza dello standard ES6 e la sintassi JSX.

In secondo luogo sono state acquisite delle nozioni riguardo alcuni aspetti del mondo aziendale, in particolare si è potuto osservare come un'azienda valuta le caratteristiche di un framework per stimare i benefici e i costi derivanti dall'introduzione di tale framework nel processo di sviluppo.

Inoltre è stato possibile osservare come l'esperienza utente e i feedback forniti dai clienti influiscano sulla progettazione di un'interfaccia grafica e portino ad adottare determinate strategie per massimizzare le prestazioni al fine di soddisfare le loro esigenze.

Infine, utilizzando le API REST di WARDA è stato possibile osservare come viene utilizzata l'architettura REST a livello aziendale e come creare un'applicazione che utilizzi tali API.

Glossario

API Indica un’insieme di procedure rese disponibili al programmatore allo scopo di ottenere un’astrazione della complessità della piattaforma sottostante, che può essere sia hardware che software. [3](#)

Cordova Apache Cordova è un framework open source per la realizzazione di applicazioni ibride che offre delle API che permettono di accedere via JavaScript ad alcune funzionalità native del dispositivo, come l’accelerometro o la fotocamera. [2](#)

DOM Il DOM o *Document Object Model* è lo standard del W3C per la rappresentazione ad oggetti di documenti strutturati, come le pagine HTML. [3](#)

FPS *fotogrammi per secondo*, è un’unità di misura utilizzata per indicare la frequenza con la quale vengono generati dei fotogrammi. [17](#)

Gestures Combinazione di movimenti dell’utente effettuati con le dita su un dispositivo touch-screen, che vengono riconosciuti da un’applicazione. [4](#)

JavaScriptCore JavaScriptCore è un motore JavaScript open source sviluppato da Apple, attualmente incluso in Safari e Safari Mobile. [6](#)

MVC Pattern architettonale che prevede la separazione tra la logica di gestione dei dati e come questi dati vengono presentati. Il pattern prevede la divisione dell’architettura in tre parti: **Model**: si occupa della gestione dei dati; **View**: si occupa di visualizzare i dati presenti nel model; **Controller**: si occupa di aggiornare il model in base alle operazioni che l’utente compie sulla view. [14](#)

npm Acronimo di Node Package Manager, è un sistema di gestione delle dipendenze per le applicazioni JavaScript che permette di installare librerie di terze parti mediante un’interfaccia a riga di comando. [5](#)

Obj-C Abbreviazione di Objective-C, un linguaggio di programmazione orientato agli oggetti derivato dal C. Questo linguaggio è stato scelto da Apple come strumento di sviluppo per le applicazioni iOS e Mac OS X. [4](#)

Pan Gestione che consiste in un tocco prolungato dello schermo da parte dell’utente. Durante l’esecuzione della gestione, l’utente può trascinare il punto di contatto in un modo simile al *drag’n’drop* effettuato con il mouse. [16](#)

PhoneGap Framework che permette la realizzazione di applicazioni mobile ibride e multi piattaforma utilizzando HTML, CSS e JavaScript. Questo framework è stato pubblicato da Adobe e utilizza Apache Cordova per interagire con le funzionalità native offerte dai vari sistemi operativi. [4](#)

Pinch-to-zoom È la gestione che viene utilizzata per eseguire lo zoom su un elemento dell'interfaccia grafica, tipicamente un'immagine o una pagina web. Questa gestione consiste nel toccare lo schermo con due dita e allontanarle o avvicinarle tra loro, senza mai staccarle dallo schermo. Nel caso le due dita vengano allontanate viene eseguito uno zoom del contenuto mentre nel caso contrario viene rimpicciolito. [16](#)

Popover Un popover è un componente delle interfacce grafiche simile ad un pop-up, che compare quanto l'utente seleziona un elemento. A differenza di un pop-up che compare al centro dello schermo, un popover compare vicino al pulsante che l'ha reso visibile ed è collegato ad esse mediante una freccia. [25](#)

Proxy Il proxy è un design pattern individuato dalla *Gang of Four* che prevede l'utilizzo di un classe o oggetto come interfaccia per qualche altro oggetto. Un esempio di utilizzo di questo pattern è dato da Java RMI, che mediante l'utilizzo di oggetti remoti, nasconde la complessità legata al fatto che l'oggetto vero e proprio sul quale viene invocato il metodo si trova su un computer diverso. Nel caso di NativeScript, viene utilizzato un oggetto JavaScript come interfaccia di un oggetto nativo (che può essere sia un oggetto Obj-C, sia Java) in modo che l'oggetto nativo possa essere utilizzato via JavaScript. [7](#)

Rendering Termine inglese che indica l'insieme di attività da svolgere per la rappresentazione grafica di un elemento, nel caso specifico dell'interfaccia grafica di un'applicazione. [3](#)

REST Riferisce ad un insieme di principi di architetture di rete, i quali delineano come le risorse sono definite e indirizzate. Il termine è spesso usato nel senso di descrivere ogni semplice interfaccia che trasmette dati su HTTP. [3](#)

Riflessione In informatica, è la capacità di un programma di analizzare, durante la sua esecuzione, le classi che lo compongono, ricavando così informazioni sulla struttura del proprio codice sorgente. [7](#)

SDK Acronimo di *Software Development Kit*, insieme di strumenti per lo sviluppo e la documentazione di software. [3](#)

Singleton Il singleton è un design pattern individuato dalla *Gang of Four* che ha lo scopo di garantire che venga creata una sola istanza di una determinata classe, e di fornire un punto di accesso globale a tale istanza. Nel progetto questo pattern viene implementato sfruttando i moduli CommonJS, creando l'istanza di un oggetto, per poi esportarla come modulo. In questo modo l'oggetto viene creato solo una volta e risulta accessibile a tutta l'applicazione in quanto è un normale modulo CommonJS. [19](#)

Swipe È un particolare tipo di pan, effettuato in modo rapido e in una singola direzione, tipicamente da destra verso sinistra o viceversa. [27](#)

Tap Gesture che consiste in un singolo tocco dello schermo da parte dell'utente, è l'equivalente di un click del mouse. [25](#)

V8 V8 è un motore JavaScript open source sviluppato da Google, attualmente incluso in Google Chrome. [6](#)

Virtual machine Software che simula delle risorse hardware e che utilizza queste risorse per eseguire determinate applicazione, in modo che queste possano utilizzare le risorse simulate. Le virtual machine hanno vari utilizzi, in questo caso vengono utilizzate per interpretare il codice JavaScript. [3](#)

WebView Componente grafico offerto dalle API native, sia di iOS, sia di Android, che permette la visualizzazione di pagine HTML. [4](#)

Bibliografia

Apache Cordova vs. Tabris.js. URL: <http://eclipsesource.com/blogs/2015/03/02/apache-cordova-vs-tabris-js/>.

Atom. URL: <https://atom.io>.

Flux. URL: <http://facebook.github.io/flux/>.

How NativeScript Works. URL: <http://developer.telerik.com/featured/nativescript-works/>.

NativeScript. URL: <https://www.nativescript.org/>.

Nuclide. URL: <http://nuclide.io>.

React Native. URL: <https://facebook.github.io/react-native/>.

React Native: Smart and Dumb components. URL: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0.

React Parts. URL: <https://react.parts/native-ios>.

Tabris.js. URL: <https://tabrisjs.com/>.

Use React Native. URL: <http://www.reactnative.com/>.