



UNIVERSITÀ DI PARMA

Dipartimento di Ingegneria e Architettura

Corso di Laurea in Ingegneria Informatica, Elettronica e delle
Telecomunicazioni

Sviluppo di una Applicazione Mobile Multiplattaforma per la Ricerca e la Visualizzazione di Immagini

Development of a Cross-Platform Mobile Application for
Searching and Viewing Images

Relatore:

Prof. Michele Amoretti

Correlatore:

Ing. Alberto Dallaporta

Tesi di Laurea di:

Alex Spagni

Indice

Introduzione	1
1 Stato dell'arte	3
1.1 App Multipiattaforma	3
1.1.1 React Native	4
1.1.2 Expo	6
1.1.3 React Navigation	7
1.1.4 Redux	9
1.2 App Native	10
1.2.1 Kotlin	11
1.2.2 Swift	12
1.2.3 Swift vs Kotlin	14
1.3 App Native vs App Multipiattaforma	15
2 Specifica Funzionale del Sistema Realizzato	17
2.1 Obiettivi dell'Applicativo	17
2.2 Requisiti dell'Applicativo	18
2.2.1 Requisiti Necessari per il Funzionamento dell'Applicativo	18
2.2.2 Requisiti Funzionali	19
2.2.3 Requisiti non Funzionali	21
2.3 Casi d'Uso	21
3 Implementazione	29
3.1 Struttura iniziale di un'Applicazione in React Native	29

3.2	Navigazione all'interno dell'Applicazione	30
3.3	Stati all'Interno dell'Applicazione	33
3.4	Fetching delle Immagini	35
3.5	Presentazione delle Immagini	37
3.6	Ricerca Immagini per Nome Rover e per Data Solare	42
3.7	Immagini Persistenti	43
3.8	Ripristinare le Immagini Occultate	44
3.9	Procedura di Sign In e Sign Up	46
4	Risultati	50
4.1	Splash Screen	50
4.2	Sign In e Sign Up	51
4.3	Immagini Memorizzate nella Cache	53
4.4	Ricerca Immagini per Nome Rover	54
4.5	Ricercare Immagini per Data Solare	55
4.6	Dettagli delle Immagini	56
4.7	Filtri Photos e Hide All	57
4.8	Parametri di Ricerca Errati	58
4.9	Animazione Onde Gravitazionali	59
4.10	Istruzioni sui Metodi di Ricerca	60
4.11	Procedura di Logout	61
	Conclusioni	63
4.12	Obiettivi e Risultati Raggiunti	63
4.13	Conoscenze Acquisite	64
4.14	Sviluppi Futuri	65
	Bibliografia	66

Introduzione

Il mondo mobile fa sempre più parte della nostra vita e ogni giorno vengono inventate nuove applicazioni che permettono di semplificarla. Esistono app per poter ascoltare la musica, per poter eseguire delle transazioni bancarie ed altre destinate alla domotica, per fare alcuni esempi. Per questi motivi è stato particolarmente interessante svolgere il tirocinio presso l'azienda Novalab SRL con sede a Reggio nell'Emilia.

L'azienda nasce nel 2017 e gli ambiti di cui si occupa sono:

- Lo sviluppo mobile
- Lo sviluppo back-end
- Progetti su Blockchain

La maggior parte delle applicazioni sono state sviluppate attraverso l'uso del framework React Native: il che significa che sono multipliattaforma; mentre altre sono state sviluppate usando linguaggi nativi come Kotlin e Swift.

Per quanto riguarda lo sviluppo back-end, per la realizzazione di web services necessari al funzionamento dell'applicazione, viene utilizzato Node.js e quindi il linguaggio JavaScript. Dal 2017 l'azienda ha sviluppato applicazioni per diverse compagnie, tra cui: Roadhose, Casa.it, Cedacri e MoneyFarm. In particolare per Cedacri è stata realizzata un'applicazione di Home Banking che va a servire diverse banche utilizzando un'unica codebase.

L'esperienza di tirocinio si è concentrata sulla comprensione e sullo sviluppo di un'applicazione mobile. Affinchè fosse possibile raggiungere il mag-

gior numero di piattaforme, l'applicativo è stato realizzato usando come framework React Native.

La Tesi è strutturata nel seguente modo: nel **Capitolo 1** vengono descritte le varie tecnologie e linguaggi utilizzati per lo sviluppo dell'applicazione. Inoltre viene fatto un confronto fra lo sviluppo nativo e quello multipiattaforma descrivendo quando conviene seguire il primo rispetto al secondo e viceversa. Nel **Capitolo 2** vengono descritti i requisiti funzionali e non funzionali necessari per lo sviluppo dell'applicazione; sono poi illustrati i principali use Case in modo da spiegare come e perchè vengono eseguiti certi servizi. Nel **Capitolo 3** attraverso l'uso di frammenti di codice vengono descritte le librerie utilizzate per la realizzazione dell'applicazione e le principali funzioni all'interno di essa. Nel **Capitolo 4** viene illustrato il funzionamento dell'applicazione e i risultati ottenuti sulle diverse piattaforme su cui è stata installata, Android e iOS, descrivendo le principali differenze osservate.

La Tesi si conclude con un riepilogo dell'attività svolta e alcune considerazioni sui possibili sviluppi futuri.

Capitolo 1

Stato dell'arte

Al giorno d’oggi siamo costantemente influenzati dal mondo mobile: viene dunque spontaneo chiedersi quali sono le possibili strade da seguire per poter sviluppare un’app. Innanzitutto è necessario fare una distinzione tra app nativa e app multipiattaforma: le prime sono delle applicazioni software che sono state sviluppate per funzionare su uno specifico tipo di dispositivo o piattaforma; la seconda è un applicazione che può essere eseguita anche su sistemi operativi differenti ma usando sempre lo stesso codice. In particolare un’app nativa sviluppata per un dispositivo Android non funzionerà su dispositivi iOS e viceversa [1] [2].

1.1 App Multipiattaforma

React native è un Framework per lo sviluppo di app multipiattaforma che permette di sviluppare applicazioni sia per Android che iOS. Per poter utilizzare questo Framework esistono due principali “strumenti”: Expo [1] e React Native CLI [1]. Entrambi mettono a disposizione una serie di servizi per aiutare nello sviluppo degli applicazioni; in particolare Expo è un Framework che estende React Native

1.1.1 React Native

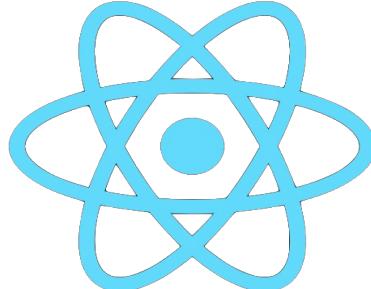


Figura 1.1

In React Native come in React, vengono costruiti dei componenti JSX, i quali combinano markup e JavaScript in un unico file. A differenza del web non si ha una separazione tra grafica, presentazione e controllo in file diversi, in quanto JSX privilegia la separazione dei “concern” rispetto alla separazione delle tecnologie.

Il ciclo di aggiornamento è lo stesso di React: quando le ”props” o gli ”state” cambiano, React Native esegue un nuovo rendering delle ”View”. La differenza principale tra React Native e React nel browser è che il primo sfrutta le librerie dell’interfaccia utente, della sua piattaforma ospitante, anzichè utilizzare il markup HTML e CSS.

React Native CLI permette di utilizzare React Native per sviluppare diversi tipi di applicazioni, tra cui:

- Prototipi
- Applicazioni multipiattaforma
- Applicazioni che non fanno largo uso di API native
- Applicazioni con una complessa User Interface (UI)
- Applicazioni per un determinato sistema operativo
- Applicazioni che non fanno uso intensivo di animazioni

Tra i vari vantaggi derivanti dall'utilizzo di questo “strumento”, bisogna sottolineare che:

- * Permette di includere moduli nativi scritti in Java, Kotlin e Object-C
- * Consente di ridurre i tempi di sviluppo, in quanto permette di riutilizzare gran parte del codice scritto per le applicazioni web
- * Mette a disposizione un gran numero di componenti pre-costruiti
- * Fornisce un interfaccia utente semplice
- * Permette di utilizzare plugin di terze parti
- * Consente di realizzare una migliore User Interface rispetto ad Expo
- * Presenta un architettura modulare la quale consente di semplificare lo sviluppo, il test e la manutenzione di programmi di grosse dimensioni

Mentre tra i vari svantaggi, si ha che:

- * Per poter effettuare i test è necessario collegare il dispositivo al Pc
- * Richiede Android Studio e XCode per l'esecuzione dei progetti
- * Per poter condividere la propria app è necessario condividere l'intero file .apk
- * La preparazione di un progetto richiedere tempo e non è così semplice come con Expo
- * Richiede una conoscenza preliminare della strutturazione delle cartelle in Android e iOS

Una volta installata un'applicazione sviluppata in React Native sul proprio dispositivo, questa non eseguirà il render nel browser DOM: andrà invece ad invocare le Object-C APIs, per eseguire il render su dispositivi IOS, e andrà ad invocare le Java APIs, per eseguire il render su dispositivi Android.

L'utilizzo di tali API è possibile grazie al "bridge", il quale fornisce a React un'interfaccia con gli elementi della UI nativa della piattaforma ospitante. In particolare i "React component" restituiscono il markup della loro funzione di rendering, che descrive il loro aspetto. Quando si utilizza React per lo sviluppo web, si traduce direttamente nel DOM del browser. Per React Native, il markup di un componente viene tradotto per adattarsi alla piattaforma host: una <View> potrebbe quindi diventare una UIView specifica per iOS [3] [1].

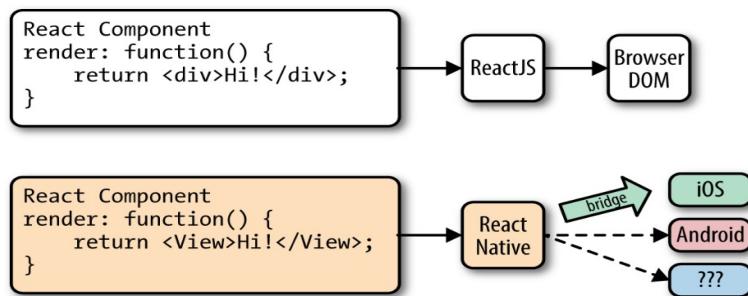


Figura 1.2

1.1.2 Expo

Expo è simile a React Native CLI, ma con uno svantaggio considerevole. In alcune applicazioni è necessario accedere alle API native che non sono disponibili su Javascript, per esempio le API per accedere ad Apple e Google play. In altri casi all'interno di un'applicazione si potrebbe voler riutilizzare le librerie scritte in Java, Object-C, Swift, Kotling o C++ che non sono state implementate in Javascript. I moduli nativi permettono di esporre delle istanze di classi Java, Object-C e C++ come oggetti JavaScript: in questo modo è possibile eseguire codice nativo in JavaScript.

Expo non permette di utilizzare i moduli nativi, quindi a meno che una certa libreria non sia già stata implementata in JavaScript, questa non potrà essere utilizzata in un'applicazione sviluppata tramite Expo. Un altro svantaggio derivante dall'utilizzo di questo strumento è che alcune device API non sono

supportate, come ad esempio il bluetooth.

A parte questo Expo presenta diversi vantaggi, tra cui [4] [1]:

- * Una migliore gestione dei link
- * Mette a disposizione una maggiore quantità di librerie standart, come ad esempio una libreria per le "Push Notification"
- * Offre una migliore "Development Experience": per poter testare un'applicazione, sviluppata tramite Expo, non è necessario avere un emulatore del dispositivo, su cui la si vuole testare, o collegare il proprio dispositivo al pc. Expo permette di testare le applicazioni tramite Wi-fi, collegando il dispositivo e il pc alla stessa rete LAN
- * Le applicazioni vengono aggiornate più velocemente rispetto a React Native CLI
- * Supporta "l'Over to air update", quindi un applicazione si aggiorna nel momento in cui quest'ultima si sta aprendo sul dispositivo su cui è installata
- * Non è necessario effettuare la build dell'applicazione per poterla testare
- * È possibile "espellere" il progetto in ExpoKit e integrare del codice nativo, continuando ad utilizzare alcune funzionalità di Expo, ma non tutte

1.1.3 React Navigation

Sia che si utilizzi Expo o React Native CLI, aspetti molto importanti nello sviluppo di un'app sono la presentazione e la navigazione tra i vari "Screen". Questi ultimi rappresentano la parte di User Interface con la quale l'utente interagirà. La navigazione tra i vari screen viene gestita da un "Navigator" ed il principale utilizzato sia da Expo che da React Native si chiama "React Navigation". Questa è una standalone library che permette di gestire lo spostamento tra i vari screen di cui l'applicazione è composta.

React Navigation fornisce principalmente quattro Navigator [5], ognuno dei quali consente ad un utente di passare da uno screen ad un altro:

- Native Stack Navigator: ogni volta che un utente effettua una transizione il nuovo screen viene messo in cima allo stack. Il Native Stack Navigator utilizza le API native UINavigationController per le applicazioni sviluppate per iOS e le API native Fragment per le applicazioni sviluppate per Android.

In questo modo la navigazione tra i vari screen avrà lo stesso comportamento e le stesse prestazioni di applicazioni che usano la navigazione nativa. Nonostante le prestazioni fornite da questo Navigator siano ottime, non è particolarmente "customizzabile" come lo Stack Navigator.

- Stack Navigator: a differenza del Native Stack Navigator questo è incredibilmente "customizzabile", poichè implementato in Javascript. Nonostante ciò non usando le primitive di navigazione native presenta prestazioni peggiori. Va comunque sottolineato che per la maggior parte delle applicazioni questa differenza di performance tra il Native Stack Navigator e lo Stack Navigator non è percepibile.
- Drawer Navigator: la differenza principale tra questo Navigator e gli altri è che quest'ultimo viene rappresentato con un'icona formata da tre linee orizzontali, posta in alto a sinistra della visualizzazione. Attraverso quest'ultima è possibile vedere tutti gli screen su cui si può navigare.
- Tab Navigator: questo Navigator viene rappresentato tramite una Tab-Bar posta nella parte bassa dello schermo che mostra gli screen verso cui si può navigare. Questo Navigator è uno dei più utilizzati nelle applicazioni basate su Tab-Bar.

1.1.4 Redux

Al giorno d'oggi le single-page application devono gestire sempre più stati, ed è per questo che nella maggior parte delle app si utilizza Redux. Questa è una libreria molto leggera che aiuta a sviluppare applicazioni che si comportano in modo coerente, che vengono eseguite in ambienti diversi (client e server) e che sono facili da testare. Si dice che Redux permetta di realizzare un “contenitore a stato” prevedibile, per le applicazioni JavaScript.

Questa libreria è completamente compatibile con React Native, attraverso l'utilizzo di React Redux, il quale rappresenta il livello di binding ufficiale di React UI per Redux. Questo livello consente ai ”React component” di leggere i dati da un archivio e di inviare azioni a quest'ultimo per aggiornarne lo stato.

In particolare, Redux:

- Permette di realizzare uno store che contiene lo stato globale
- Consente di emettere azioni che aggiornano lo stato dello store. Queste vengono notificate all'archivio nel momento in cui avviene un evento nell'applicazione
- Presenta delle funzioni riduttrici pure che esaminano le azioni e restituiscono uno stato immutabile aggiornato

Quando si parla di “stati” da gestire all'interno di un'app si fa riferimento ad esempio alle risposte di un server, a dati salvati nella cache o creati localmente e che non sono memorizzati in modo persistente. Redux è in grado di gestire tutti questi stati e permette di rendere la mutazione dello stato globale prevedibile attraverso l'imposizione di alcune restrizioni, ovvero quando e come lo stato può essere modificato.

Queste restrizioni possono essere riassunte nei tre principi cardine di Redux [6]:

1. Lo stato globale dell'applicazione è memorizzato in un albero di oggetti all'interno di un singolo store. In questo modo è possibile realizzare app universali, permettere un miglior debugging dell'applicazione e rendere lo stato globale persistente in modo da avere un ciclo di sviluppo più rapido.
2. Lo stato è Read Only: questo assicura che né le “viste” né le callback di rete scrivano direttamente sullo stato; esprimono piuttosto l'intenzione di trasformarlo.
Dato che tutte le modifiche sono centralizzate e avvengono una alla volta in un ordine rigoroso, non vi sono condizioni di concorrenza a cui dover prestare attenzione. Le azioni per aggiornare lo stato sono semplici oggetti JavaScript, quindi possono essere registrate, serializzate, memorizzate e successivamente riprodotte per scopi di debug o di test.
3. Lo stato viene aggiornato da funzioni pure che sono chiamate riduttori. Queste prendono lo stato precedente, un'azione e restituiscono lo stato successivo. In particolare i riduttori devono ritornare nuovi oggetti di stato e non modificare lo stato precedente.

1.2 App Native

Come già menzionato all'inizio di questo capitolo, le applicazioni native sono sviluppate per uno specifico dispositivo o piattaforma ed è quindi necessario un linguaggio di programmazione differente da un sistema operativo all'altro. iOS fa uso dei linguaggi Object-C e Swift, mentre Android sviluppa app native in Java e Kotlin.

1.2.1 Kotlin

Kotlin è un linguaggio sviluppato da JetBrains a partire dal 2010 e successivamente reso open source nel 2012. È definito come general purpose, free, open source e pragmatico. Pragmatico in quanto cerca di distinguersi dalla struttura chiusa e ben definita di Java al fine di permettere uno sviluppo più snello e veloce, grazie soprattutto al fatto che fornisce diversi “shortcut”.

Kotlin nel 2019 è diventato, secondo Google, il linguaggio più utilizzato per lo sviluppo di applicazioni Android, sostituendo Java [7]. Nonostante lo si utilizzi per lo sviluppo di applicazioni Android native, quest’ultimo permette anche la realizzazione di applicazioni per iOS e di applicazioni web, in quanto è possibile transpilare il codice scritto in Kotlin in codice JavaScript [8].

Kotlin consente di realizzare applicazioni seguendo il paradigma Object Oriented (OOP) o il paradigma della programmazione funzionale: nel primo caso si fa uso delle classi, dell’ereditarietà e del polimorfismo, proprio come in Java; nel secondo caso il programma si basa sulla valutazione di funzioni matematiche. In questo modo è possibile ottenere il meglio dei due mondi [9].

Un ambiente di esecuzione per le applicazioni scritte tramite questo linguaggio è la JVM (Java Virtual Machine): questa consente di utilizzare Kotlin in qualsiasi contesto in cui viene utilizzato Java e quindi di essere al 100% interoperabile con quest’ultimo. Il vantaggio di questo alto livello di interoperabilità è che si possono riutilizzare le librerie Java esistenti [10].

Kotlin e Java sono molto simili tra loro, ed entrambi possono essere utilizzati per lo sviluppo di applicazioni per Android. Nonostante ciò presentano alcune differenze [11]:

- Il codice scritto in Kotlin risulta essere più conciso di quello scritto in Java, in quanto riduce la quantità di ”BoilerPlate”.
- Un’applicazione sviluppata in Kotlin risulta essere più facile da leggere e presenta meno errori dovuti allo sviluppatore.

- Java consente di assegnare ad una variabile il valore "Null", il quale può generare un errore "NullPointerException" nel caso in cui la variabile venisse acceduta. Invece Kotlin applica la cosiddetta "Null Safety", cioè non permette di assegnare ad una variabile il valore "Null". In questo modo il codice risulta essere più stabile.
- Kotlin non è un linguaggio fortemente tipizzato come invece lo è Java. Infatti vengono utilizzate solo due parole chiave per la definizione delle variabili. Inoltre, solo nel momento in cui ad una variabile viene assegnato un valore, questa viene identificata come una stringa, un numero o un booleano. Lo stesso meccanismo ricorre anche in JavaScript.
- Una differenza fondamentale tra i due è che Java supporta solo la programmazione ad oggetti, mentre Kotlin oltre a quest'ultima supporta anche la programmazione funzionale.
- Java presenta un tempo di compilazione inferiore rispetto a Kotlin. Nel caso di piccole applicazioni si stima che Java sia il 15-20% più veloce rispetto al tempo necessario per compilare la stessa applicazione scritta in Kotlin. Se però consideriamo app di dimensioni maggiori, allora il tempo di compilazione è circa lo stesso.

Secondo alcuni studi solo i neofiti dello sviluppo Android continuano a sviluppare applicazioni in Java, poiché la maggior parte della documentazione e degli esempi sono in Java.

Altri studi dimostrano che il tempo medio impiegato da uno sviluppatore Java per imparare Kotlin è di poche ore. Dopo questa transizione si stima una riduzione del 40% del numero di linee di codice da Java a Kotlin [9].

1.2.2 Swift

Swift è un potente linguaggio di programmazione che permette di creare applicazioni sia per dispositivi mobili come cellulari, IPad e Apple Watch,

ma anche per dispositivi come Mac e Apple Tv. Essendo un linguaggio general-purpose, proprio come Kotlin, può essere usato anche per sviluppare applicazioni web e web service. Inoltre, è possibile realizzare applicazioni server che necessitano di "runtime security" e ingombro di memoria ridotto.

Swift è stato realizzato per essere un linguaggio veloce: utilizzando la tecnologia di compilazione LLVM il codice Swift viene trasformato in codice macchina ottimizzato che sfrutta al meglio l'hardware moderno.

Questo linguaggio di programmazione è il successore dei linguaggi C e Object-C, in quanto include sia primitive di basso livello come tipi, controllo di flusso e operatori ma fornisce anche funzionalità orientate agli oggetti come classi, protocolli e i tipi generici. Swift ha eliminato intere classi di codice "non sicuro": le variabili devono sempre essere inizializzate prima di essere utilizzate, gli indici degli array e gli interi vengono sempre controllati per verificare se vi sono errori di overflow [12].

Questo linguaggio presenta diverse caratteristiche, tra cui [13]:

- Supporta i tipi generici
- Le funzioni possono ritornare valori multipli e tuple
- Presenta un iterazione rapida e concisa su un intervallo o un insieme
- Presenta modelli di programmazione funzionale, ad esempio "map" e "filter"
- Presenta una gestione degli errori integrata

Tramite Swift la memoria viene gestita automaticamente utilizzando l'ARC (Automatic Reference Counting): esso permette di mantenere l'utilizzo della memoria al minimo e senza l'onere del Garbage Collection [14]. Si ricorda che in Java, il Garbage Collection si occupa di tenere traccia delle allocazioni di memoria utilizzate e le libera solo quando non sono più impiegate. Esso è un vero e proprio thread in esecuzione parallelamente al programma e anche se la sua priorità è minima, è un processo da considerare in termini di tempo di CPU.

Swift è open source e cross-platform, infatti può essere usato sia su piattaforme Apple che Linux. Presenta anche un package manager che è un unico strumento multipiattaforma per costruire, eseguire, testare e pacchettizzare le librerie e gli eseguibili Swift.

Tramite questo linguaggio è possibile creare una nuova applicazione oppure utilizzare il codice Swift per implementare nuove caratteristiche e funzionalità in applicazioni preesistenti. Il codice Swift coesiste con i file Object-C, esistenti nello stesso progetto, con pieno accesso alle API Object-C [12].

Anche se Swift è interoperabile con Object-C, esso presenta alcuni vantaggi rispetto a quest'ultimo [15]:

- * Secondo Apple Swift può essere 2.6 volte più veloce di Object-C
- * In Swift possono essere create delle variabili senza doverne definire prima il tipo
- * Non è necessario inserire i punti e virgola alla fine di ogni riga di codice
- * Il codice in Swift viene scritto più velocemente rispetto al codice scritto in Object-C, poichè Swift è stato progettato per essere "Developer-Friendly"

1.2.3 Swift vs Kotlin

Sia Kotlin che Swift sono linguaggi di programmazione che vengono usati per lo sviluppo di app native, ed entrambi sono multipiattaforma. Nonostante ciò presentano alcune differenze: [16]:

- Lo sviluppo in Kotlin non si limita ad un particolare IDE o OS. Infatti è possibile scrivere il proprio codice Kotlin in qualsiasi IDE o OS come VS Code, Atom, Windows, Linux e Mac.

Al contrario lo sviluppo in Swift è possibile solo all'interno di XCode perché solo attraverso esso si è possibile compilare il codice scritto tramite questo linguaggio. Questo IDE è disponibile solo su Mac, quindi non sarà possibile sviluppare app per IOS senza averne uno.

- Kotlin e Swift approcciano la gestione della memoria in modo differente. In particolare Kotlin, proprio come Java, affronta la gestione di quest'ultima dal punto di vista del Garbage Collection.
All'contrario Swift fa uso dell'ARC (Automatic Reference Counting): attraverso di esso le applicazioni sviluppate tendono ad essere più efficienti e prive di bug.
- Entrambi, all'interno di un progetto possono coesistere con i rispettivi predecessori. Swift è al 100% interoperabile con Object-C, mentre Kotlin con Java.

Una caratteristica comune ad entrambi è che permettono di ridurre la quantità di "BoilerPlate", necessaria sia nei progetti Java che in quelli Object-C.

1.3 App Native vs App Multiplattaforma

Quando si decide di sviluppare un'applicazione per un dispositivo mobile, come può essere un telefono, è necessario capire se si vuole sviluppare un'app nativa o un'app multiplattaforma. La scelta può essere fatta sulla base di cinque fattori: prestazioni, tempo di sviluppo, sicurezza, User Experience/Interface e stabilità [17].

Per fornire le massime prestazioni agli utenti, la scelta più saggia che si può fare è scegliere un approccio nativo, poiché le app native sono sviluppate tenendo conto dei requisiti specifici della piattaforma di riferimento. Esse sono compilate per una specifica serie di dispositivi ed eseguite per una precisa architettura. Ciò che consente alle app native di essere più efficienti è l'accesso ad API e componenti esclusivi, ottimizzati per diverse dimensioni di schermo e versioni di sistema. Questa tipologia di app può presentare dimensioni minori rispetto alle app multiplattaforma, consentendo di occupare meno memoria sul dispositivo.

Per quanto riguarda i tempi di sviluppo, le app multiplattaforma presentano uno produzione più rapida: grazie alla riusabilità del codice tra le piat-

taforme, un gruppo di sviluppatori non necessita di implementare progetti separati per sistemi operativi diversi.

Dal punto di vista della sicurezza, le app native sono più sicure rispetto alle app multipiattaforma, poiché sono dotate di molteplici funzioni di sicurezza incorporate. Per gli sviluppatori di applicazioni native è solitamente più facile implementare la crittografia dei file, il rilevamento intelligente delle frodi e altre funzioni di sicurezza attraverso le opportune librerie e risorse di ogni piattaforma. L'aggiornamento delle misure di sicurezza nelle app native richiede meno tempo rispetto alle app multipiattaforma. In queste ultime è più difficile prevedere quando i framework multipiattaforma saranno aggiornati.

Se l'obiettivo è quello di realizzare la miglior User Experience o User Interface per la propria applicazione, allora è necessario privilegiare un approccio nativo. Le applicazioni native offrono migliori funzionalità dell'interfaccia utente, poiché dispongono di librerie e componenti preimpostati e personalizzabili.

Se l'intento è quello di sviluppare un'applicazione stabile nel lungo tempo, allora bisogna adottare un approccio nativo. Dato che Android e IOS provengono rispettivamente da Google e Apple, è certo che queste aziende continueranno a supportare e migliorare i loro sistemi operativi mobili: questo significa che le app native beneficeranno di stabilità in termini di manutenzione e aggiornamenti. D'altra parte, dato che i framework multipiattaforma sono creati da aziende, organizzatori e comunità di sviluppatori di terze parti, c'è il rischio che l'aggiornamento o lo sviluppo di questi framework possa essere incoerente o interrotto.

Capitolo 2

Specifiche Funzionali del Sistema Realizzato

2.1 Obiettivi dell'Applicativo

Lo scopo principale dell'applicativo sviluppato è quello di fornire un servizio per la visualizzazione delle immagini scattate dai diversi Rover inviati su Marte nel corso degli ultimi anni. Queste immagini possono essere visualizzate sul proprio dispositivo mobile con sistema operativo Android o iOS, tramite una applicazione che può essere scaricata da un sito web online.

L'applicazione deve fornire una funzione di ricerca, tramite la quale si potranno ricercare immagini inserendo il nome di uno dei Rover oppure digitando la data solare in cui sono state scattate le immagini che si vogliono osservare. In particolare, i nomi dei Rover per i quali si potrà effettuare una ricerca sono: Curiosity, Spirit e Opportunity. Tra una ricerca e l'altra verrà mostrata una schermata di “loading” in modo da indicare all'utente che si sta eseguendo il “fetching” delle immagini.

Oltre a poter ricercare le immagini secondo diversi parametri, l'applicazione deve anche fornire la possibilità di nascondere alcune di queste in modo che non vengano visualizzate dall'utente in una ricerca futura. Nonostante le immagini nascoste non vengano presentate all'utente quando effettua una

ricerca, gli verrà sempre concessa la possibilità di ripristinarle. L’utente potrebbe non essere interessato ad un’intera categoria di immagini, quindi gli verrà fornita la possibilità di nascondere tutte le foto relative ad una sua ricerca.

Ogni volta che verrà chiusa l’applicazione, le ultime immagini visualizzate dovranno essere memorizzata in modo persistente così che possano essere mostrate all’utilizzatore quando riaprirà l’app.

Oltre a poter visualizzare le immagini in una lunga lista nell’Index Screen dell’applicazione, quest’ultima deve fornire anche la possibilità di visualizzare i dettagli di ognuna di esse.

Al primo avvio l’utente dovrà essere ridirezionato verso una schermata di login o di registrazione a seconda che sia in possesso o meno delle credenziali di accesso.

2.2 Requisiti dell’Applicativo

2.2.1 Requisiti Necessari per il Funzionamento dell’Applicativo

L’applicazione necessita di una connessione ad Internet in modo da collegarsi tramite il web ai server della Nasa e recuperare le immagini ricercate da ogni utente.

Le credenziali di ogni utente sono memorizzate in un database distribuito che memorizza i dati in documenti flessibili JSON-like. In particolare, per assicurare la sicurezza di ogni utilizzatore, le password di ognuno di essi vengono memorizzate solo dopo essere state cifrate. In questo modo anche se qualcuno dovesse penetrare nel database non riuscirebbe ad accedere ai vari account.

Il database viene consultato attraverso un server locale che espone delle API all’applicativo per far sì che ogni utente possa eseguire la registrazione

e il login. Affinché l'applicazione possa comunicare con il server locale, deve essere attiva un'applicazione Multiplattaforma chiamata Ngrok¹.

Per garantire l'efficienza dell'applicazione il numero di immagini recuperate, ad ogni richiesta GET ai server della Nasa, deve essere pari a 25. In questo modo si evita di sovraccaricare il client che deve elaborare le immagini e presentarle a schermo.

In Fig 2.1 si può osservare una rappresentazione grafica dell'interazione tra i vari sistemi:

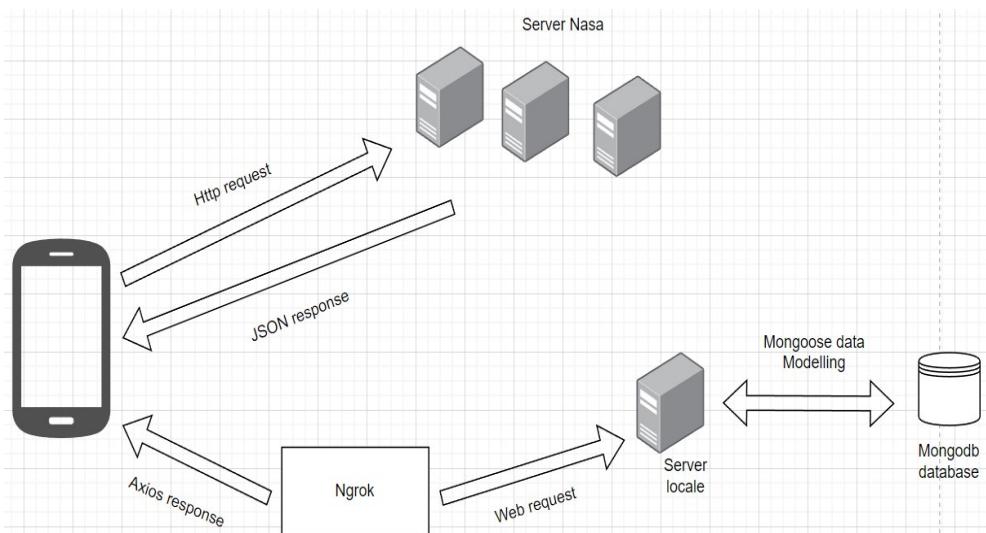


Figura 2.1: Comuncazone tra i vari sistemi costituenti l'applicazione

2.2.2 Requisiti Funzionali

Ad ogni requisito viene associato un nome mnemonico, una breve descrizione e un codice di priorità in modo tale da indicare quali requisiti devono essere implementati per primi e quindi definire una gerarchia di implementazione.

¹Ngrok consente di aprire una connessione diretta verso le API sviluppate in Express e fornisce un URL tramite il quale il dispositivo mobile può usufruirne. In particolare, espone le porte su cui i server locali sono in ascolto ad Internet.

Nome	Descrizione	Priorità
Sign in	L'utente deve poter accedere all'applicazione tramite le sue credenziali	Primario
Sign up	L'utente deve potersi registrare tramite la propria e-mail e password in modo da poter accedere all'applicazione	Primario
Search by name	L'utente deve poter ricercare le immagini tramite il nome di uno dei tre possibili Rover: Spirit, Opportunity e Curiosity	Primario
Search by date	L'utente deve poter ricercare le immagini che sono state scattate in una precisa data solare	Primario
Hide image	L'utente deve poter nascondere le immagini che gli vengono presentate	Primario
Restore images	L'utente deve poter ripristinare le immagini che ha nascosto	Secondario
Logout	L'utente deve poter uscire dal proprio profilo	Secondario
Hide all	L'utente deve poter nascondere nascondere tutte le ommagini relative ad una sua ricerca	Secondario
Privacy and Policy	L'utente deve poter conoscere quali condizioni e termini ha accettato scaricando l'applicazione	Secondario

2.2.3 Requisiti non Funzionali

In questa sezione vengono descritti i principali requisiti non funzionali necessari per il buon funzionamento dell'applicazione.

Nome	Descrizione
Prestazioni	L'applicativo deve essere in grado di presentare le immagini entro 6 secondi dall'avvio della ricerca
Portabilità	L'applicativo deve poter essere scaricabile sia su dispositivi iOS che Android
Manutenibilità	L'applicativo deve essere strutturato seguendo un architettura modulare in modo da facilitare la fase di test e di modifica

2.3 Casi d'Uso

In questa sezione vengono illustrati i principali casi d'uso.

Caso d'Uso U1: Sign In

- **Use case overview:** l'applicazione deve consentire agli utenti la possibilità di poter accedere al proprio account.
- **Actors:** end user, server locale.
- **Triggers:** si preme sul pulsante “Sign in”.
- **Pre-Condition:** l'utente non deve aver già effettuato l'accesso all'applicazione oppure si deve trovare nell'Index Screen dove può effettuare il logout.

- **Post-condition:** si viene indirizzati nell'Index Screen dell'applicazione.

- **Main Flow:**

1. L'utente ha appena terminato di scaricare l'applicazione e la apre, ritrovandosi come prima schermata lo "Splash screen". Una volta premuto il pulsante "Explore" viene reindirizzato sullo screen destinato al log in.
2. Arrivato allo screen di log in l'utente inserisce la propria e-mail e password e preme sul pulsante "Sign in".
3. Viene inviata una richiesta al server locale che va a confrontare le credenziali inserite in precedenza con quelle presenti sul database per verificare che coincidano. In caso affermativo l'utente viene indirizzato sull'Index Screen.

- **Alternative Flow:**

1. L'utente si trova nell'Index Screen e sta scorrendo tutta la lista di immagini a lui presentate. Una volta arrivato alla fine della lista preme sul pulsante di "logout".
2. L'utente si ritrova nello screen riservato al log in dove inserisce la propria mail e password e preme sul pulsante "Sign in".
3. Viene inviata una richiesta al server locale che va a confrontare le credenziali inserite in precedenza con quelle presenti sul database per verificare che coincidano. In caso affermativo l'utente viene indirizzato sull'Index Screen.

- **Exception Flow:**

1. Una volta inserite le proprie credenziali viene inviata una richiesta al server locale che va a confrontare queste ultime con quelle presenti sul database.

2. Le credenziali inserite dall'utente non sono corrette e gli viene presentato un messaggio di errore.

Caso d'Uso U2: Sign Up

- **Use case overview:** l'applicazione deve consentire agli utenti la possibilità di potersi registrare tramite la propria e-mail e password.
- **Actors:** end user, server locale.
- **Triggers:** si preme sul pulsante “Sign up”.
- **Pre-Condition:** l'utente non deve aver già effettuato l'accesso all'applicazione.
- **Post-condition:** si viene indirizzati nell'Index Screen dell'applicazione.
- **Main Flow:**
 1. L'utente ha appena terminato di scaricare l'applicazione e la apre, ritrovandosi come prima schermata lo “Splash screen”. Una volta premuto il pulsante ”Explore” viene reindirizzato sullo screen destinato al log in. Non avendo alcune credenziale valida l'utente deve premere sul pulsante ”Sign up” in modo da essere reindirizzato nello schermo dedicato alla registrazione.
 2. Arrivato allo screen di registrazione l'utente inserisce la propria e-mail e password e preme sul pulsante “Sign up”.
 3. Viene inviata una richiesta al server locale che va a confrontare che l'e-mail inserita dall'utente non sia già presente nel database. In caso affermativo l'utente viene indirizzato sull'Index Screen.
- **Exception Flow:**

1. Una volta inserite le proprie credenziali viene inviata una richiesta al server locale che va a confrontare che l'e-mail inserita dall'utente non sia già presente nel database.
2. Esiste già un utente registrato con quella e-mail, quindi il sistema restituisce un messaggio di errore.

Caso d'Uso U3: Ricerca Immagini tramite Nome Rover

- **Use case overview:** l'applicazione deve consentire agli utenti di ricercare le immagini in base al nome del Rover.
- **Actors:** End user, server Nasa.
- **Triggers:** si digita il nome di uno dei tre Rover nella Search Bar e si preme “fatto” sulla propria tastiera.
- **Pre-Condition:** l'utente deve aver effettuato l'accesso oppure essere in possesso di un JSON Web Token.
- **Post-condition:** vengono presentate delle nuove immagini o una modale che informa l'utente che nessuna immagine è stata trovata.
- **Main Flow:**
 1. L'utente si trova nella schermata principale e digita il nome del Rover che ha scattato le immagini richieste. Per avviare la ricerca si deve premere sul pulsante “All”.
 2. Viene effettuata una richiesta HTTP verso i server della Nasa i quali rispondono con un oggetto JSON che contiene tutte le foto che sono state trovate.
 3. Le immagini ottenute vengono inserite all'interno di una lista e vengono visualizzate a schermo.
- **Alternative Flow:**

1. L'utente si trova nella schermata principale e digita il nome del Rover che ha scattato le immagini le immagini richieste. Per avviare la ricerca si deve premere sul pulsante “All”.
2. Viene effettuata una richiesta HTTP verso i server della Nasa i quali rispondono con un oggetto JSON che contiene tutte le foto che sono state trovate.
3. Il nome del Rover inserito non è corretto. L'utente viene informato che non è stata trovata alcuna foto e gli viene presentata una lista di immagini vuota.

Caso d'Uso U4: Ricerca Immagini per Data Solare

- **Use case overview:** l'applicazione deve consentire agli utenti la possibilità di ricercare le immagini in base a una certa data solare.
- **Actors:** end user, server Nasa.
- **Triggers:** si preme sul pulsante “Search by date”.
- **Pre-Condition:** l'utente deve aver effettuato l'accesso oppure essere in possesso di un JSON Web Token.
- **Post-condition:** vengono presentate delle nuove immagini o una modale che informa l'utente che nessuna immagine è stata trovata.
- **Main Flow:**
 1. L'utente si trova nell'Index Screen e preme il pulsante “Data” in modo da spostarsi nello schermo dedicato alla ricerca per data solare. Deve quindi inserire giorno, mese e anno delle foto che vuole ricercare.
 2. Viene effettuata una richiesta HTTP verso i server della Nasa i quali rispondono con un oggetto JSON che contiene tutte le foto che sono state trovate.

3. Le immagini ottenute vengono inserite all'interno di una lista e vengono visualizzate a schermo.

- **Alternative Flow:**

1. L'utente si trova nell'Index Screen e preme il pulsante "Data" in modo da spostarsi nello schermo dedicato alla ricerca per data solare. Deve quindi inserire giorno, mese e anno delle foto che vuole ricercare.
2. Viene effettuata una richiesta HTTP verso i server della Nasa i quali rispondono con un oggetto JSON che contiene tutte le foto che sono state trovate.
3. Nella data solare inserita dall'utente non è stata scattata alcuna foto. Si viene quindi informati che non sono state trovate immagini e quindi non viene presentata alcuna foto.

Caso d'Uso U5: Nascondere un'Immagine

- **Use case overview:** l'applicazione deve consentire agli utenti la possibilità di nascondere le immagini mostrate in modo che non vengano visualizzate in futuro.
- **Actors:** end user.
- **Triggers:** si preme sul pulsante "Hide this image".
- **Pre-Condition:** l'utente deve aver effettuato l'accesso oppure essere in possesso di un JSON Web Token.
- **Post-condition:** l'immagine selezionata verrà nascosta.
- **Main Flow:**

1. L'utente si trova nell'Index Screen e premendo su un'immagine accede ai dettagli di quest'ultima.

2. Premendo sul tasto “Hide this image” viene nascosta l’immagine in questione e viene notificato all’utente un messaggio di conferma. A questo punto si viene reindirizzati sull’Index Screen.

- **Alternative Flow:**

1. L’utente si trova nell’Index Screen e premendo su un’immagine accede ai dettagli di quest’ultima.
2. L’utente decide di non nascondere l’immagine in questione e ritorna sull’Index Screen premendo l’icona con la “X”.

Caso d’Uso U6: Ripristinare le Immagini Nascoste Precedentemente

- **Use case overview:** l’applicazione deve consentire agli utenti la possibilità di vedere anche le immagini nascoste in precedenza.
- **Actors:** end user.
- **Triggers:** si preme sul pulsante “Photos”.
- **Pre-Condition:** l’utente deve aver effettuato l’accesso oppure essere in possesso di un JSON Web Token.
- **Post-condition:** in base ai parametri di ricerca impostati dall’utente verranno mostrate tutte le immagini, anche quelle nascoste.
- **Main Flow:**

1. L’utente si trova nell’Index Screen e preme il pulsante “Photos” il quale passa dal colore grigio ad azzurro in modo da indicare che è stato abilitato.

2. Oltre alle immagini che l’utente stava già visualizzando, vengono mostrate anche quelle che aveva nascosto in precedenza. In particolare vengono ripristinate solo le foto nascoste che soddisfano i criteri di ricerca dell’utente.

- **Alternative Flow:**

1. L’utente si trova nell’Index Screen e preme il pulsante “Photos” il quale passa dal colore azzurro a grigio in modo da indicare che è stato disabilitato.
2. Tutte le immagini che l’utente aveva nascosto in precedenza vengono nuovamente occultate. Ora l’utente vede solo le foto che soddisfano i suoi parametri di ricerca e che non sono state nascoste.

Capitolo 3

Implementazione

Successivamente alla definizione dei requisiti funzionali e non funzionali dell'applicazione e alla descrizione dei diversi casi d'uso che coinvolgono l'utente e altri sistemi esterni, si passa all'implementazione vera e propria dell'applicativo.

3.1 Struttura iniziale di un'Applicazione in React Native

Come strumento per lo sviluppo dell'applicazione è stato usato Expo, il quale permette di creare un nuovo progetto tramite l'utilizzo di questo comando all'interno del proprio terminale:

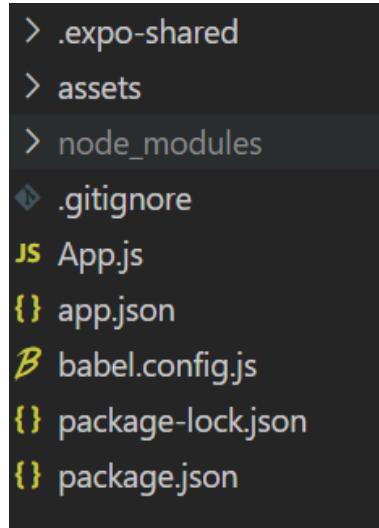
- `npx create-expo-app my-app`

Quando si vuole sviluppare un'applicazione in React Native è richiesto di seguire una “gerarchia delle cartelle” gestita da Expo. Quando si utilizza il comando sopra indicato per la creazione di un nuovo progetto, queste sono le cartelle e i file che vengono creati:

- La cartella “node modules” contiene tutte le dipendenze e i package necessari al funzionamento dell'applicazione. Ogni volta che viene in-

stallato un nuovo package realizzato da terze parti, lo si ritrova in tale cartella.

- Il file “package.json” permette di tenere traccia delle varie dipendenze, indicando quale versione di ogni package deve essere scaricata.
- Il file “App.js” contiene il primo componente che viene mostrato ogni volta che viene aperta l’applicazione.
- La cartella “assets” contiene le immagini che verranno utilizzate all’interno dell’applicazione.



```
> .expo-shared
> assets
> node_modules
↳ .gitignore
JS App.js
{} app.json
B babel.config.js
{} package-lock.json
{} package.json
```

A screenshot of a terminal window showing the directory structure of a React Native project. The files listed are: .expo-shared, assets, node_modules, .gitignore, App.js (highlighted in yellow), app.json, babel.config.js, package-lock.json, and package.json.

Figura 3.1

3.2 Navigazione all’interno dell’Applicazione

Ogni applicazione è composta da diversi screen ed è necessario permettere all’utente di transitare tra di essi. Per svolgere questa operazione è stata usata la libreria *React Navigation*, la quale fornisce diversi componenti e funzioni per la gestione della navigazione all’interno dell’app. In particolare all’interno dell’applicativo ho utilizzati [18] [19]:

- **NavigationContainer:** è un componente responsabile della gestione dello stato dell'applicazione, del collegamento del top-level navigator all'ambiente dell'applicazione e dell'integrazione con la piattaforma specifica su cui verrà installata. Solitamente viene utilizzato una sola volta all'interno dell'applicativo e viene inserito nel file App.js/App.tsx. Lo scopo principale di questo componente è la gestione della navigazione tra i vari navigator che verranno inseriti all'interno dell'applicazione. Per poterlo utilizzare è necessario importarlo dalla libreria *react-navigation*.
- **Stack Navigator:** fornisce all'applicazione la possibilità di transitare attraverso gli screen e gestire la storia della navigazione. Ogni volta che l'utente transita da uno screen ad un altro viene fatta un'operazione di "push" e fa sì che il nuovo screen venga messo in cima allo stack. Invece, ogni volta che l'utente vuole tornare alla schermata precedente, viene fatta un'operazione di "pop" e lo screen in cima allo stack viene rimosso. Per poterlo utilizzare è necessario importare la funzione "createStackNavigator", la quale restituisce un oggetto che contiene due proprietà: *Screen* e *Navigator*. Entrambi sono dei "React component" che vengono utilizzati per configurare il navigator. Il *Navigator* permette di definire la route iniziale, ossia lo screen da mostrare per primo ogni volta che viene invocato lo Stack Navigator; questo componente dovrà contenere i vari *Screen* i quali consentono di definire i vari schermi verso cui l'utente può transitare. Lo *Screen* presenta 3 prop:
 - * *Name:* il nome della route.
 - * *Component:* il JSX component da mostrare.
 - * *Option:* un oggetto che definisce alcune proprietà con cui lo schermo dovrà essere presentato.

Tramite l'utilizzo della libreria (*React Navigation*) è possibile annidare più navigator in modo da definire una gerarchia. L'importante è che in cima ad

essa si abbia il NavigationContainer.

```
<NavigationContainer ref={navigationContainerRef}>
  <Stack.Navigator initialRouteName="loading">
    <Stack.Screen
      name="MainStackNavigator"
      component={MainStackNavigator}
      options={{ headerShown: false }}
    />
    <Stack.Screen
      name="ShowScreen"
      component={ShowScreen}
      options={({ props: any }) => ({
        headerShown: false,
        cardStyleInterpolator:
          props?.route?.params?.slide == "right"
            ? invertedForHorizontalIOS
            : CardStyleInterpolators.forHorizontalIOS,
      })}
    />

    <Stack.Screen
      name="SigningStackNavigator"
      component={SigningStackNavigator}
      options={{ headerShown: false, presentation: "modal" }}
    />
    <Stack.Screen
      name="loading"
      component={LoadingScreen}
      options={{ headerShown: false, presentation: "modal" }}
    />

    <Stack.Screen
      name="ImagesLoading"
      component={ImagesLoading}
      options={{
        headerShown: false,
        presentation: "modal",
        headerMode: "screen",
        headerStyle: {
          height: 80, // Specify the height of your custom header
        },
      }}
    />
  </Stack.Navigator>
</NavigationContainer>
```

Figura 3.2: Frammento di codice che gestisce la navigazione tra gli schermi

nella Fig 3.2, viene mostrato un frammento di codice è stato preso dal file App.tsx . Come si può vedere il NavigationContainer è il componente in cima alla gerarchia e va ad avvolgere i vari Stack Navigator:

1. Il primo è quello immediatamente successivo al NavigationContainer e viene utilizzato per gestire la transizione verso:
 - * Gli altri due Stack Navigator
 - * Lo schermo di loading iniziale
 - * Lo schermo per la visualizzazione delle immagini memorizzate in cache
 - * Lo screen che mostra i dettagli di un'immagine
2. Il Secondo si chiama MainStackNavigator e gestisce la navigazione tra l'index Screen e il Search Screen.
3. Il terzo si chiama SigningStackNavigator e gestisce la navigazione tra lo screen di sign in e quello di sign up.

3.3 Stati all'Interno dell'Applicazione

All'interno dell'applicazione vi sono diversi stati che vengono continuamente aggiornati; per tenere traccia di ognuno di essi sono stati utilizzati le librerie *redux* e *react-redux*. Attraverso di esse si va a creare uno *store* che contiene l'intero stato dell'applicazione come un semplice oggetto JavaScript.

Per poter modificare gli stati vengono utilizzate delle funzioni pure chiamate *reducers* che specificano come cambia lo stato in risposta ad una *action*. I *reducers* prendono come argomento un'azione con un payload e restituiscono un nuovo stato basato sull'azione passata. Essendo funzioni pure non modificano i dati dell'oggetto che viene passato e non eseguono alcun effetto collaterale nell'applicazione; dato lo stesso stesso oggetto dovrebbero produrre sempre lo stesso risultato. Le *action* devono avere una proprietà “*type*” per indicare il tipo di azione da eseguire. Esse rappresentano l'unica fonte di informazioni per lo *store* [20].

Sia le *action* che i *reducer* vengono definiti come delle arrow function. La *action* rappresentata nella Fig 3.3 viene utilizzata per indicare che l'array

```
type LibrariesAddActionTypeRefreshMars = {
  type: typeof LIBRARIES_ADD;
  payload: imageType[];
};

export const addElementsToLibrariesMarsRefreshing = (
  array: imageType[]
): LibrariesAddActionTypeRefreshMars | undefined => {
  if (array.length) {
    return {
      type: LIBRARIES_ADD,
      payload: array,
    };
  } else {
    return {
      type: LIBRARIES_ADD,
      payload: [],
    };
  }
};
```

Figura 3.3: Esempio di *action*

```
export const getImagesReducer = (
  state = initialStateRover,
  action: AllLibrariesAction
) => {
  switch (action.type) {
    case LIBRARIES_ADD:
      return action.payload;

    case LIBRARIES_ADD_MARS:
      const imagesToRender = imagesFilterDuplicate(
        action.payload as imageType[],
        state
      );
      return [...state, ...imagesToRender as imageType[]];
    case LIBRARIES_RESET:
      return [];

    default:
      return state;
  }
};
```

Figura 3.4: Esempio di *reducer*

contenente le immagini che devono essere visualizzate a schermo, deve essere aggiornato. Un *action* ritorna sempre un oggetto che contiene due proprietà ovvero *type* e *payload*: La prima viene utilizzata per indicare al *reducer* quale azione deve essere eseguita, mentre la seconda rappresenta il contenuto con cui deve essere aggiornato lo stato all'interno dello store.

Il *reducer* rappresentato nella Fig 3.4, riceve sempre due parametri, che sono *state* e *action*: il primo rappresenta lo stato memorizzato nello store e viene sempre passato da quest'ultimo ogni volta che il *reducer* viene invocato; l'*action* rappresenta l'azione che il *reducer* deve eseguire. In base all'azione passata si entra in uno dei diversi “case” dello switch.

Supponendo che venga eseguita l'*action* mostrata nella Fig 3.3, si entrerebbe nel primo “case” dello switch e si andrebbe a sostituire lo stato attuale con il *payload* fornito dall'*action*.

Le *action* passate ad ogni *reducer* sono degli oggetti JavaScript inviati attraverso il metodo nativo di React, chiamato *dispatch()* che prende come parametro la *action* stessa. Per poter invocare questo metodo è necessario utilizzare un hook, chiamato “*useDispatch*”.

Oltre a poter memorizzare gli stati all'interno dello store è anche possibile accedere ai dati contenuti in essi attraverso un altro hook, chiamato “*useSelector*”. Gli state memorizzati nello store sono:

1. dataRover: un oggetto JavaScript che contiene le informazioni riguardanti la ricerca fatta dall'utente.
2. images: un array che contiene tutte le immagini che, una volta filtrate, devono essere visualizzate a schermo.
3. imagesHide: un array che contiene tutte le immagini che sono state occultate dall'utente.
4. loading: è una variabile booleana che indica quando deve essere presentata all'utente un'animazione.
5. search: è una variabile booleana che permette di avviare una procedura di ricerca.
6. sign: viene utilizzato per memorizzare il JWT che è stato fornito all'utente in fase di registrazione o login.

```
const rootReducer = combineReducers({
  dataRover: roverDataReducer,
  images: getImagesReducer,
  imagesHide: getImagesHided,
  loading: LoadingReducer,
  search: makeASearch,
  sing: signReducer,
});
```

Figura 3.5: A sinistra lo state memorizzato nello store e a destra il rispettivo reducer

3.4 Fetching delle Immagini

Ogni volta che l'utente effettua una nuova ricerca digitando il nome di un Rover oppure la data solare in cui sono state scattate le immagini, viene

effettuata una richiesta HTTP ai server della Nasa; in particolare, viene effettuata una GET request [21].

Per effettuare richieste HTTP è stato utilizzato Axios, un Client HTTP che ritorna una *Promise*, la quale rappresenta l'eventuale completamento (o fallimento) di un'operazione asincrona e il suo conseguente valore [22].

```
export const NasaApi = axios.create({
    //url di base a cui vado a fare la richiesta HTTP
    baseURL: 'https://api.nasa.gov'
});
```

Figura 3.6: Base URL

```
const DAY:string='3';
const MONTH:string='6';
const YEAR:string='2016';
export const getImageMars = async (roverName:string,page:number,day=DAY,month=MONTH,year=YEAR): Promise<marsObject[]> => {
    try {
        //quello che vado a fare qui è dire che i dati che sono in result dovrà essere un array di tipo marsObject
        const response = await NasaApi.get<{ photos: marsObject[] }>(`mars-photos/api/v1/rovers/${roverName}/photos?
            earth_date=${year}-${month}-${day}&page=${page}&api_key=XU7atD6DNBzEbPlouVUOfIwb7004dBJwzvrF7JMX`);
        return response.data.photos;
    } catch (err) {
        console.log("no data");
        return [];
        //throw err;
    }
};
```

Figura 3.7: Esempio di richiesta GET

Per poter eseguire una GET request finalizzata al recupero delle immagini, come nella Fig 3.7, è necessario fornire al metodo “get” dell’Axios instance (creata nella Fig 3.6) L’URL a cui deve essere effettuata la richiesta e i parametri per ricercare le immagini; in particolare questi ultimi sono:

- * Il nome del Rover (required)
- * La data solare in cui sono state scattate le foto (required)
- * Il numero di pagina a cui si vuole accedere (optional)

Si evidenzia che la data solare viene impostata ad un valore iniziale in modo da poter presentare all’utente un certo tipo di immagini nel momento in cui apre l’applicazione per la prima volta.

Il metodo GET utilizzato ritorna una “Axios Response”, la quale è un oggetto che contiene diverse proprietà tra cui “data”. Attraverso essa è possibile accedere alla risposta fornita dai server della Nasa in formato JSON.

3.5 Presentazione delle Immagini

Per visualizzare le immagini che sono state recuperate dai server della Nasa, viene utilizzato un Core Component chiamato `FlatList`, reso disponibile da React Native. Questo componente presenta diverse *prop*, in particolare nel mio Applicativo sono state usate le seguenti [23]:

- `data`: a questa *prop* deve sempre essere passata un array; in questo caso l’array di immagini che devono essere visualizzate a schermo.
- `renderItem`: a questa *prop* deve essere passata una funzione o un function component che verrà eseguito per ogni elemento dell’array che è stato fornito alla prop `data`. Nel caso di questa applicazione a `renderItem` viene passato un “Function Component” chiamato *PhotoComponent* il quale permette di visualizzare ognin immagine e il suo id.
- `keyExtractor`: questa *prop* viene utilizzata per estrarre una chiave univoca per ogni elemento della lista. In questo modo è possibile riodinare gli elementi se alcuni di essi venissero eliminati o ne venissero aggiunti altri. In questo applicativo ad ogni elemento della lista viene associato come chiave univoca l’id di ogni immagine.
- `listFooterComponent`: questa *prop* permette di visualizzare un React Element/Component alla fine della lista; in questo applicativo le viene passato un React Element chiamato *FooterComponent* che permette di visualizzare due link: Privacy and Policy e Log Out.

- onEndReached: a questa *prop* viene passata una funzione invocata ogni volta che viene raggiunta la fine della lista. In questo applicativo, una volta raggiunta l'ultima immagine recuperata dalla ricerca precedente, viene effettuata una nuova richiesta ai server della Nasa, utilizzando gli stessi parametri di ricerca, recuperando la pagina di immagini successiva. Quest'ultima potrebbe o meno contenere delle immagini: nel caso in cui ve ne siano, verranno inserite in coda all'array visualizzate a schermo; in caso contrario le immagini visibili all'utente rimarranno le stesse.

```
<FlatList
  style={styles.FlatListStyle}
  ref={flatListRef}
  data={images.filter((element) => {
    if (element.hide == false) {
      return element;
    }
  })}
  keyExtractor={(item) => item.image.id}
  renderItem={({ item }) => (
    <View style={styles.container}>
      <PhotoComponent object={item.image} />
      <View
        style={{
          marginTop: 10,
          borderBottomColor: "#323436",
          borderBottomWidth: 2,
          width: 360,
        }}
      />
    </View>
  )}
  onEndReached={() => [
    const nextPage = roverData.page_number + 1;
  ]}
}
```

Ogni volta che l'utente chiude l'applicazione e poi la riapre, gli vengono subito mostrate delle immagini secondo dei parametri di ricerca predefiniti. Per andare a recuperare queste immagini viene utilizzato un hook di React chiamato useEffect, al quale vengono assegnate due dipendenze: Search e allButtonColor. La prima dipendenza è uno state che viene aggiornato ogni volta che un utente vuole effettuare una nuova ricerca di immagini; il secondo

```

        addImageFromMarsToList(
          roverData.rover_name,
          newPage,
          roverData.earth_day,
          roverData.earth_month,
          roverData.earth_year
        );
      }
    }
  }

  onEndReachedThreshold={0.5}
  ListFooterComponent={
    images.filter((element) => {
      if (element.hide == false) {
        return element;
      }
    }).length ? (
      <FooterComponent />
    ) : null
  }
/>

```

Figura 3.8: Esempio di uso della Flat List

è uno state che viene aggiornato ogni volta che l’utente preme sul pulsante “all”, esprimendo in questo modo l’intenzione di voler eseguire una ricerca.

Lo useEffect mostrato nella Fig 3.9 non viene eseguito solo quando l’utente apre l’applicazione per la prima volta, ma anche ogni volta che effettua una nuova ricerca di immagini. Tutte le volte che viene eseguito il frammento di codice nella Fig 3.9 viene aggiornato lo state *dataRover*, andando a modificare il numero di pagina da recuperare. Inoltre viene invocata una funzione chiamata ReplaceImageFromMarsList(), la quale va a recuperare le nuove immagini da mostrare all’utente.

ReplaceImageFromMarsList()

Come si può vedere nella Fig 3.10, questa funzione richiede che le vengono passati alcuni parametri:

```
useEffect(() => {
  if (allButtonColor === "#2E8AF6" && loading) [
    dispatch({
      type: LIBRARIES_PAGE_NUMBER,
      payload: { ...roverData, page_number: 1 },
    });
    replaceImageFromMarsToList(
      roverData.rover_name,
      1,
      roverData.earth_day,
      roverData.earth_month,
      roverData.earth_year
    );
  ]
}, [search, allButtonColor]);
```

Figura 3.9: UseEffect per il caricamento delle immagini

- Il nome del Rover per cui deve essere eseguita la ricerca.
- La pagina che contentiene le immagini da recuperare: questa funzione viene eseguita ogni volta che l'utente vuole ricercare nuove immagini, quindi sarà necessario recuperare sempre la prima pagina.
- Gli ultimi tre parametri corrispondono al giorno, mese e anno terrestre in cui sono state scattate le foto.

Questa funzione va ad invocare un metodo, chiamato getImageMars(), il quale va ad eseguire una GET request per recuperare le nuove immagini; una volta ottenute, si controlla che l'utente abbia premuto il tasto “Photos” e quindi voglia visualizzare o meno le immagini nascoste. Nel primo caso si va ad aggiornare lo state “images” con tutte le foto recuperate; nel secondo caso viene aggiornato lo state ”images” solo con le immagini che l'utente non ha occultato in precedenza.

Potrebbe accadere che l'utente fornisca dei parametri di ricerca errati e in tal caso verrà mostrato un messaggio di errore.

Le immagini recuperate vengono mostrate a schermo solo dopo che lo state “loading” verrà posto a “false”: questo avviene dopo quattro secondi tramite l’utilizzo di una funzione asincrona, chiamata setTimeout.

```
const replaceImageFromMarsToList = async (
  roverName: string,
  page: number,
  day?: string,
  month?: string,
  year?: string
) => {
  try {
    const results = await getImageMars(roverName, page, day, month, year);
    if (photosButtonColor != "#2E8AF6") {
      const imagesToRender = imagesFilterHideImage(results, hides);
      dispatch(addElementsToLibrariesMarsRefreshing(imagesToRender));
    } else {
      const imagesToRender = results.map((element) => {
        return {
          image: element,
          hide: false,
        };
      });
      dispatch(addElementsToLibrariesMarsRefreshing(imagesToRender));
    }

    const ErrorMessage = () => {
      if (results.length == 0) [
        navigationContainerRef.current?.navigate("InfoScreenImageNotFound");
      ]
    };

    setTimeout(() => dispatch(setLoadingReducer(false)), 4000);
    setTimeout(() => ErrorMessage(), 4000);
  } catch {}
  dispatch({
    type: LIBRARIES_PAGE_NUMBER,
    payload: { ...roverData, page_number: page },
  });
  setTimeout(() => setAllButtonColor("#727477"), 4000);
};
```

Figura 3.10: Funzione replaceImageFromMarsList

3.6 Ricerca Immagini per Nome Rover e per Data Solare

Per poter ricercare delle immagini tramite il nome del Rover, viene fornita una search-bar il cui componente corrispondente è mostrato nella Fig 3.11. Quando l'utente abilita la ricerca premendo sul pulsante “All” e poi digita nella Search-bar il nome di uno dei Rover, vengono emesse tre azioni:

- Viene modificato lo state che mantiene le informazioni sui parametri di ricerca forniti dall'utente, aggiornando quindi il campo RoverName.
- Viene posto a true lo state “loading”, in modo da mostrare un'animazione che informi l'utente dell'avvio della ricerca.
- Viene modificato lo state “search”, in modo che venga eseguito lo useEffect mostrato in Fig 3.9.



Figura 3.11: Search-bar

Per poter ricercare delle immagini in base alla data solare vengono forniti all'utente tre campi “input text”, nei quali deve inserire giorno, mese e anno in cui sono state scattate. Quando viene premuto il tasto “Search by date” avviene un aggiornamento degli stati simile a quello della Fig 3.11.

L'unica differenza riguarda l'aggiornamento dello stato “dataRover”: invece di modificare il campo roverName vengono aggiornati i campi earth-day, earth-month, earth-year.

3.7 Immagini Persistenti

L'applicazione oltre a recuperare delle immagini in base ai parametri di ricerca inseriti dall'utente, deve far sì che vengano visualizzate solo le foto che non occultate in precedenza. Inoltre, le ultime immagini visualizzate dall'utente, devono essere presentate a quest'ultimo quando andrà a riaprire l'applicazione. Per soddisfare entrambe le richieste è necessario rendere gli “state” persistenti all'interno dello store fornito da Redux: ciò significa che quando l'applicazione viene chiusa i dati all'interno dello store non vengono persi.

Per ottenere questo risultato è stata usata una libreria, chiamata *react-redux*; essa richiede che venga utilizzato un altro storage (AsyncStorage), che non sia quello di Redux, per memorizzare i dati che si vogliono mantenere in modo persistente.

```
import { persistStore, persistReducer } from "redux-persist";
import AsyncStorage from "@react-native-async-storage/async-storage";
import reducers from "./reducers";
import { createStore } from "redux";
const persistConfig = {
  key: "root",
  storage: AsyncStorage,
  whitelist: ["images", "imagesHide"],
};
const persistedReducer = persistReducer(persistConfig, reducers);
|
export default () => {
  let store = createStore(persistedReducer);
  let persistor = persistStore(store);
  return { store, persistor };
};
```

Figura 3.12: Creazione dello store persistente

Come si può osservare nella Fig 3.12, react-redux richiede che vengano indicati quali stati dello store debbano essere mantenuti in modo persistente e quali no; in questo applicativo solo gli state “images” e “imagesHide” vengono mantenuti in questo modo. È necessario poi creare lo store utilizzando la configurazione persistConfig, fornita da react-redux.

Per completare la configurazione dello store persistente si deve “avvolgere” il NavigationContainer, che a sua volta “avvolge” l’intera applicazione, come mostrato nella Fig 3.8.

```
<Provider store={store}>
  <PersistGate loading={null} persistor={persistor}>
    <NavigationContainer ref={navigationContainerRef}>
      ...
    </NavigationContainer>
  </PersistGate>
</Provider>
```

Figura 3.13: Uso dello store persistente

3.8 Ripristinare le Immagini Occultate

Una volta occultate, le immagini non vengono più visualizzate a meno che non sia l’utente a deciderlo. È stato realizzato un pulsante, chiamato “Photos”, che permette di ripristinare le immagini nascoste.

Nella schermata principale si hanno diversi pulsanti con uno stile simile, tra cui “Photos”: ho deciso quindi di realizzare un “Custom component” in modo da poterlo riutilizzare per tutti i pulsanti. Come si può osservare nella Fig 3.14, a FilterButtonComponent (il “Custom component”) vengono passate cinque prop: tre per configurarne lo stile e due per configurarne il funzionamento.

Ogni volta che il pulsante Photos viene premuto, il suo colore cambia da azzurro a grigio e viceversa: questo avviene grazie alla prop “setColor”. Il cambio di colore indica una precisa azione del filtro:

- Blu: vengono visualizzate anche le immagini nascoste. L’array “images” contiene sia le foto da visualizzare che quelle nascoste, e vengono visualizzate solo quelle che possiedono la proprietà “hide”= false; affinchè tutte le immagini vengano mostrate si deve settare la proprietà “hide” di ogni oggetto all’interno dell’array al valore “false”. Per poter aggiornare la FlatList è richiesto l’aggiornamento dello stato “images” all’interno dello store.
- Grigio: vengono visualizzate solo le immagini che non sono state nascoste in precedenza. Per ritornare alla condizione in cui solo le immagini non occultate vengono visualizzate, viene invocata la funzione dontShowImagesHide(); questa va a ripristinare l’array “images”, andando a settare correttamente la proprietà “hide” di ogni oggetto all’interno di esso. Per poter aggiornare la FlatList è richiesto l’aggiornamento dello stato “images” all’interno dello store.

```
<FilterButtonComponent
  color={photosButtonColor}
  setColor={(newColor) => setPhotosButtonColor(newColor)}
  buttonName="Photos"
  onPressButton={() => [
    if (photosButtonColor == "#727477") {
      const newArray = images.map(element) => {
        return { image: element.image, hide: false };
      };
      dispatch(addElementsToLibrariesMarsRefreshing(newArray));
    } else {
      const newArray = dontShowImagesHide(images, hides);
      dispatch(addElementsToLibrariesMarsRefreshing(newArray));
    }
  ]}
  buttonWidth={50}
  buttonHeight={34}
/>
```

Figura 3.14: Filtro Photos

3.9 Procedura di Sign In e Sign Up

Ogni utente prima di poter accedere all'applicazione deve disporre di un JWT (JSON Web Token); per poterlo ottenere è necessario fornire e-mail e password nella schermata di login o di registrazione, a seconda che le credenziali siano già registrate nel database o meno. Sia la procedura di *sign in* che quella di *sign up* richiedono che venga contattato il server locale. In particolare, nella prima procedura viene contattata la route localhost3000/signin, mentre nella seconda viene contattata la route localhost3000/signup.

```
const signIn = async ({ email, password }: signType) => {
  try {
    const response = await expressApi.post("/signin", { email, password });
    await AsyncStorage.setItem("token", response.data.token);
    dispatch(setLoadingReducer(true));
    dispatch(setSearchReducer(!search));
    dispatch(addToken(response.data.token));
    navigationContainerRef.current?.navigate("MainStackNavigator");
  } catch (err: any) {
    console.log(err.message);
    dispatchaddError("Something is gone wrong with Sign In");
  }
};
```

Figura 3.15: Funzione di sign in

Come si può notare nella Fig 3.15 l'utente invia un oggetto contenente la propria e-mail e password utilizzato il metodo post della libreria Axios; il quale invierà l'oggetto in formato JSON.

Se la procedura di *sign in* va andare a buon fine, il server risponde con il JWT dell'utente, il quale viene salvato all'interno dell'“`asyncStorage`”; così facendo quando l'utente riapre l'applicazione non deve nuovamente inserire le proprie credenziali, ma viene automaticamente loggato all'interno dell'applicazione.

Se invece la procedura di *sign in* non va a buon fine, viene mostrato un errore all'utente in modo che capisca che le proprie credenziali non sono corrette.

La procedura di *sign up* è identica tranne per la route che l'applicazione contatta.

Vediamo ora nel dettaglio la route di *sign in* e *sign up* all'interno del server.

Sign Up

Come si può vedere nella Fig 3.16, la prima operazione svolta dalla route di *sign up* consiste nell'andare ad estrarre le credenziali fornite dall'utente; una volta ottenute viene creato e poi salvato un nuovo “user” all'interno del database tramite il metodo “*save*”.

Se il salvataggio avviene correttamente, si genera un JWB che viene inserito nel body della risposta inviata dal server all'applicazione.

```
router.post('/signup', async (req, res) => {
  const { email, password } = req.body;

  try {
    const user = new User({ email, password });
    await user.save();

    const token = jwt.sign({ userId: user._id }, 'MY_SECRET_KEY');
    console.log(token);
    res.send({ token });
  } catch (err) {

    return res.status(422).send(err.message);
  }
});
```

Figura 3.16: Route di sign up

Prima di effettuare il salvataggio dell'utente nel database viene eseguita una operazione di pre-saving come mostrato nella Fig 3.17. In questa operazione viene cifrata la password fornita dall'utente. Per eseguire la cifratura vengono usati due metodi della libreria bcrypt; ossia genSalt e hash: in particolare, il secondo permette di eseguire l'hash della password. Quest'ultima viene poi sostituita a quella inserita dall'utente in modo che venga inserita nel database.

```

userSchema.pre('save', function(next) {
  const user = this; // this si riferisce all'utente che stiamo cercando di salvare
  if (!user.isModified('password')) {
    return next();
  }

  bcrypt.genSalt(10, (err, salt) => {
    if (err) {
      return next(err);
    }

    bcrypt.hash(user.password, salt, (err, hash) => {
      if (err) {
        return next(err);
      }
      user.password = hash;
      next();
    });
  });
});

```

Figura 3.17: Funzione di pre-saving

Sign In

```

router.post('/signin', async (req, res) => {
  const { email, password } = req.body;

  if (!email || !password) {
    return res.status(422).send({ error: 'Must provide email and password' });
  }

  const user = await User.findOne({ email });
  if (!user) {
    return res.status(422).send({ error: 'Invalid password or email' });
  }

  try {
    await user.comparePassword(password);
    const token = jwt.sign({ userId: user._id }, 'MY_SECRET_KEY');
    res.send({ token });
  } catch (err) {
    return res.status(422).send({ error: 'Invalid password or email' });
  }
});

```

Figura 3.18: Route di sign in

Come si può vedere nella Fig 3.18, per prima cosa si accede alle credenziali inviate dall'utente e si controlla che siano state fornite sia un'email che una password. In seguito, attraverso l'email fornita, viene recuperata la password cifrata presente nel database e tramite la funzione comparePassword() si confrontano le due password. Questa funzione fa uso del metodo compare del modulo bcrypt per andare a verificare che la password fornita dall'utente

e quella cifrata presente sul database siano uguali. Se la verifica da esito positivo, viene generato un JWT tramite il metodo sign del modulo “jsonwebtoken”. Il token generato viene poi inserito nel body della risposta per l’utente.

Capitolo 4

Risultati

In questo capitolo viene dimostrato il funzionamento dell'applicazione su un Samsung s9 e un Iphone 13, in modo da visualizzare le piccole differenze di funzionamento e grafica sui due dispositivi.

4.1 Splash Screen

Ogni volta che l'utente apre l'applicazione viene presentato lo Splash Screen mostrato nelle Fig 4.1 e 4.2 . Osservando attentamente le due figure non si notano differenze significative, se non che la dimensione del testo nell'iPhone 13 è notevolmente più piccola. Inoltre, nel dispositivo Apple il testo è posto più in basso nello schermo: ciò è dovuto ad una dimensione maggiore del dispositivo fisico.

Cliccando sul pulsante “Explore” l'utente potrebbe essere reindirizzato su due screen alternativi: “Sign in” o “Images Cache” .



Figura 4.1: Android Splash Screen

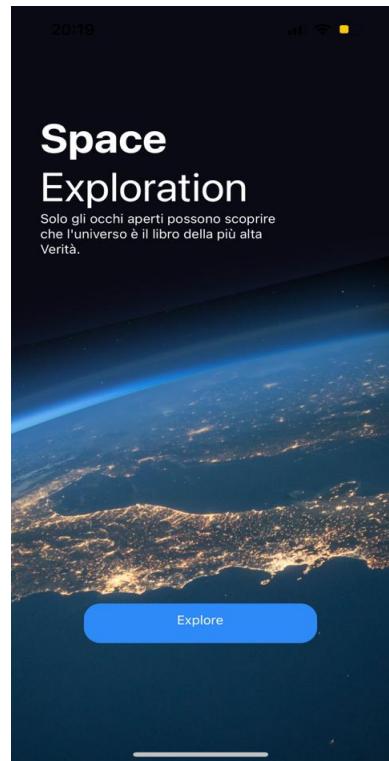


Figura 4.2: iPhone Splash Screen

4.2 Sign In e Sign Up

Se l'utente non ha mai effettuato l'accesso all'applicazione, dallo splash screen viene reindirizzato allo schermo di "Sign in", mostrato nella Fig 4.3 (4.4 per iOS), dove può accedere al suo profilo con le proprie credenziali. Nel caso in cui l'utente non abbia già un profilo, cliccando sul pulsante "Sign up" viene portato nella schermata di registrazione mostrata nella Fig 4.5 (4.6 per iOS); in quest'ultima inserendo la propria e-mail e password può registrarsi ed accedere alla schermata principale. Nel caso in cui l'utente cerchi di accedere con delle credenziali non valide oppure voglia registrarsi con una e-mail già presente nel database, viene presentato un messaggio di errore.

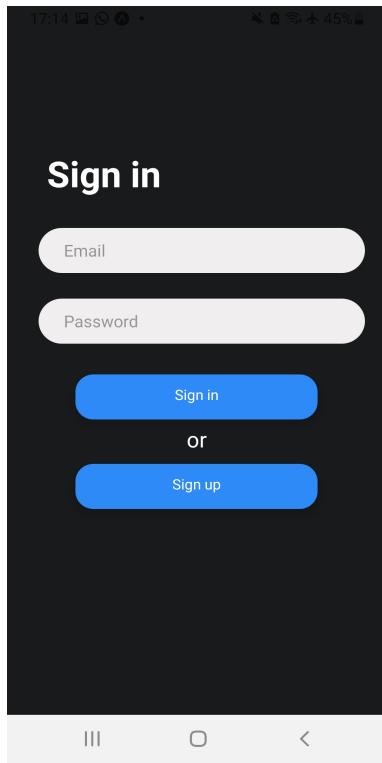


Figura 4.3: Android schermo Sign In

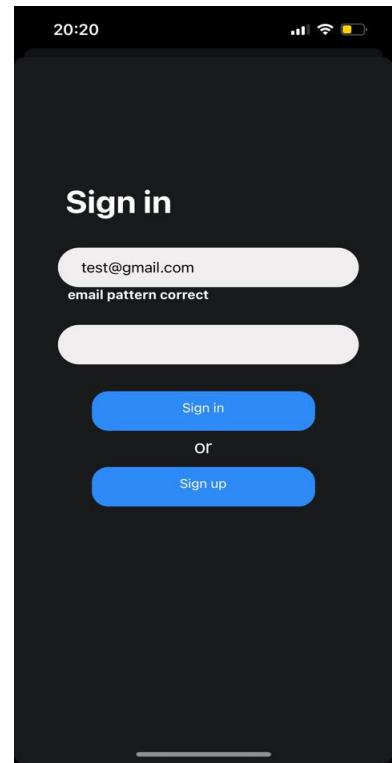


Figura 4.4: iPhone schermo Sign In

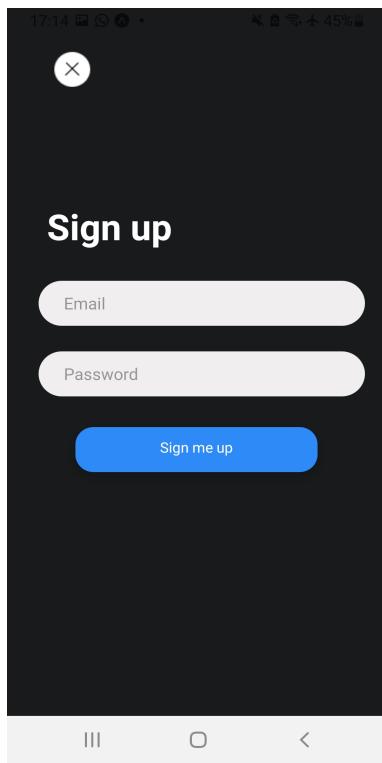


Figura 4.5: Android schermo Sign Up

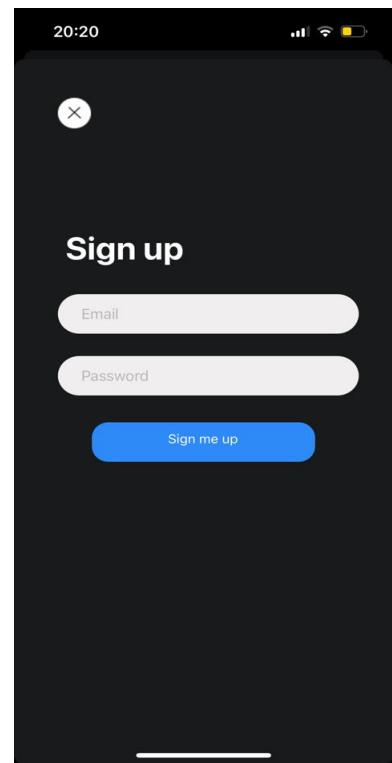


Figura 4.6: iPhone schermo Sign Up

Una volta acceduto al proprio profilo, l'utente viene reindirizzato nella schermata principale, nella quale vengono visualizzate di volta in volta le immagini ricercate.

4.3 Immagini Memorizzate nella Cache

Se l'utente ha già effettuato in precedenza l'accesso all'applicazione, nel proprio dispositivo è già stato memorizzato un JWT; essendo già in possesso di questo oggetto l'utente viene reindirizzato dallo Splash Screen allo screen destinato alle immagini memorizzate nella Cache. Tale schermo è mostrato nella Fig 4.7 (4.8 per iPhone) e mostra le ultime immagini che l'utente ha visualizzato prima di chiudere l'applicazione.

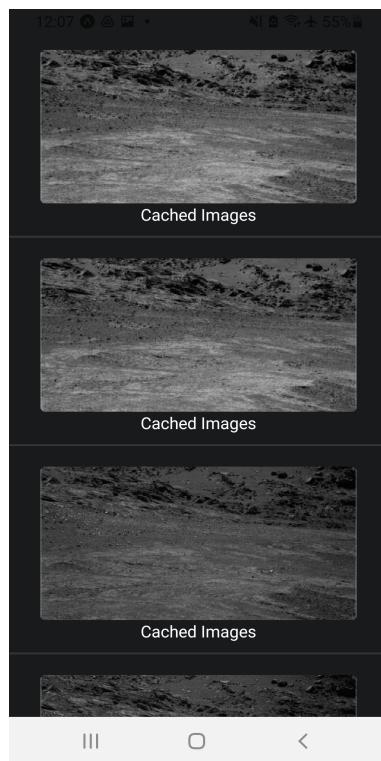


Figura 4.7: Android schermo per immagini memorizzate in cache



Figura 4.8: iPhone schermo per immagini memorizzate in cache

Dalle Fig 4.7 e 4.8 si può evincere che l'unica differenza tra Android e iPhone è una maggiore distanza dall'header dello schermo.

4.4 Ricerca Immagini per Nome Rover

Per poter ricercare le immagini tramite il nome del Rover, l'utente deve:

1. Assicurarsi che il pulsante “All” non sia stato premuto, quindi sia di colore grigio.
2. Digitare uno dei tre nomi possibili nella Search-Bar: Curiosity, Opportunity e Spirit.
3. Cliccare sul pulsante “All” avviando la ricerca.



Figura 4.9: Android ricerca per nome

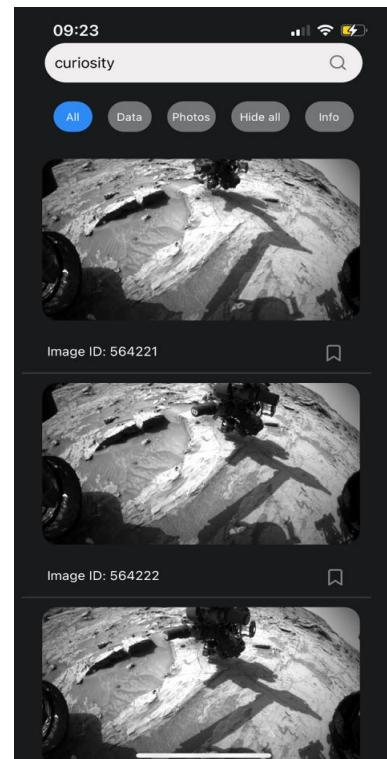


Figura 4.10: iPhone ricerca per nome

Dalla Fig 4.9 e 4.10 non si nota nessuna differenza tra i due dispositivi: per la realizzazione di questi screen è stato infatti utilizzato uno stile di presentazione chiamato “FlexBox”, il quale si adatta ai vari dispositivi su cui è utilizzato.

4.5 Ricercare Immagini per Data Solare

Per ricercare immagini per data solare l’utente deve premere il pulsante “data” in modo da essere reindirizzato nello screen mostrato nella Fig 4.11 (4.12 per iPhone). In questa schermata si possono inserire giorno, mese e anno in cui sono state scattate le immagini e premere su “Search by date” per iniziare la ricerca.

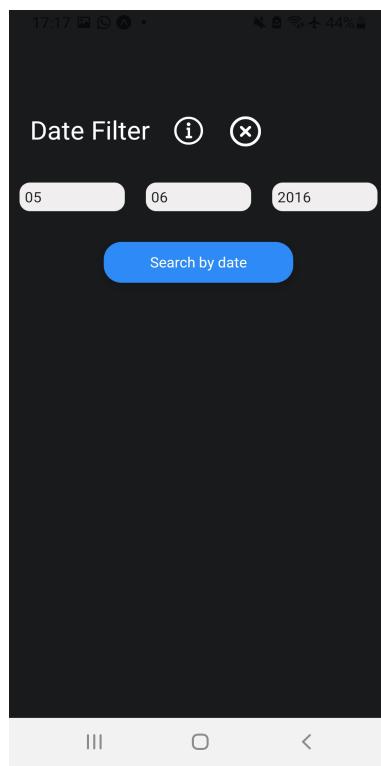


Figura 4.11: Android ricerca per data solare

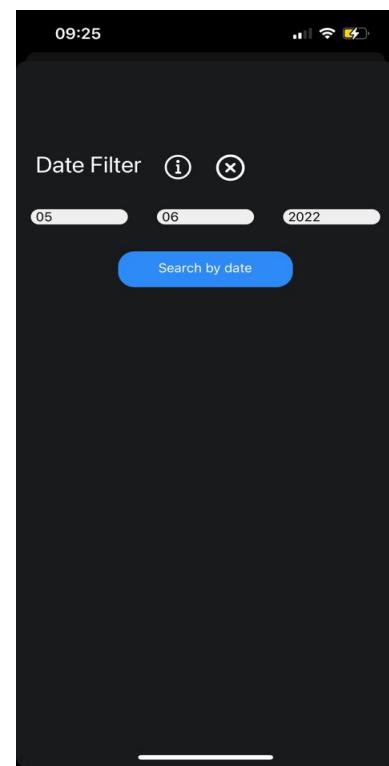


Figura 4.12: iPhone ricerca per data solare

Osservando attentamente le Fig 4.11 e 4.12 si possono notare due principali differenze tra i due dispositivi:

- I campi dove l’utente può digitare il testo sono diversi, in quanto il componente TextInput di React Native viene reindirizzato in un componente nativo diverso a seconda della piattaforma di destinazione [24].
- Il pulsante “Search by date” è diverso per lo stesso motivo descritto sopra.

4.6 Dettagli delle Immagini

Oltre a poter ricercare delle immagini è possibile visualizzare i dettagli di esse, come: l’ID del Rover, il nome del Rover e la camera che ha scattato la foto. Per poter accedere ai dettagli di ogni immagine è necessario premere su una di esse, in modo da essere reindirizzati nello screen mostrato nella Fig 4.13 (4.14 per iOS). In questo screen oltre a poter vedere i dettagli di una immagine è anche possibile nascondere la stessa premendo sul pulsante “Hide this image”. Una volta occultata, l’immagine non viene più mostrata nella schermata principale a meno che l’utente non prema il pulsante “Photos”.

Osservando le Fig 4.13 e 4.14 possiamo notare alcune differenze tra i due dispositivi:

1. Il pulsante “Hide this image” viene presentato in modo diverso poichè il componente Button di React Native viene reindirizzato differentemente a seconda della piattaforma di destinazione [24].
2. L’icona rappresentata con una “X” si trova in due posizioni diverse sui dispositivi; in iOS è molto vicino all’Header dello schermo, mentre in Android si trova più in basso. Questa differenza di locazione è dovuta al fatto che l’icona è stata posizionata utilizzando come dimensione i pixel dello schermo e quindi non è responsive rispetto alle dimensioni di ogni dispositivo.

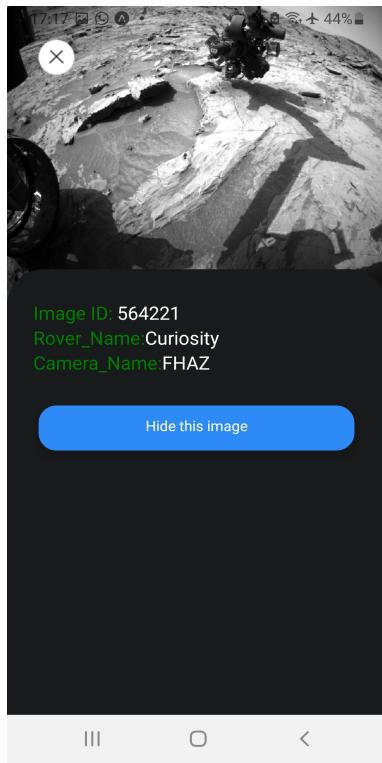


Figura 4.13: Android dettagli immagine

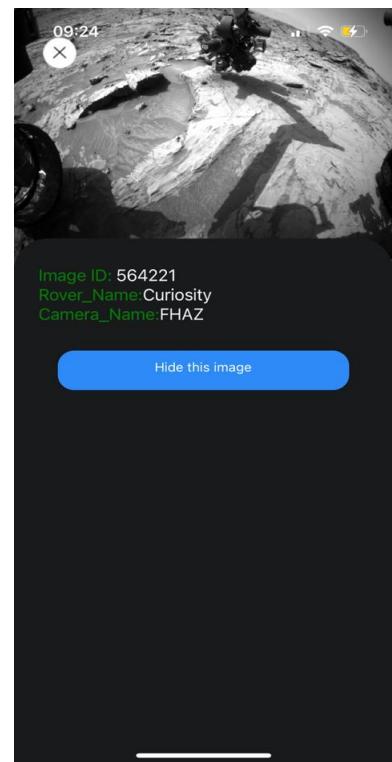


Figura 4.14: iPhone dettagli immagine

4.7 Filtri Photos e Hide All

Una volta nascoste le immagini, queste non vengono mostrate all'utente quando effettua una ricerca: possono però essere ripristinate tramite l'utilizzo del filtro “Photos”. Nella Fig 4.15 si può notare che l'immagine con ID=564234 non è visibile prima di premere il pulsante “Photos”; questa viene visualizzata dopo averlo premuto.

All'utente viene fornita la possibilità di nascondere tutti gli elementi di una ricerca premendo il pulsante “Hide All”.



Figura 4.15: Lista delle immagini prima di usare il filtro Photos

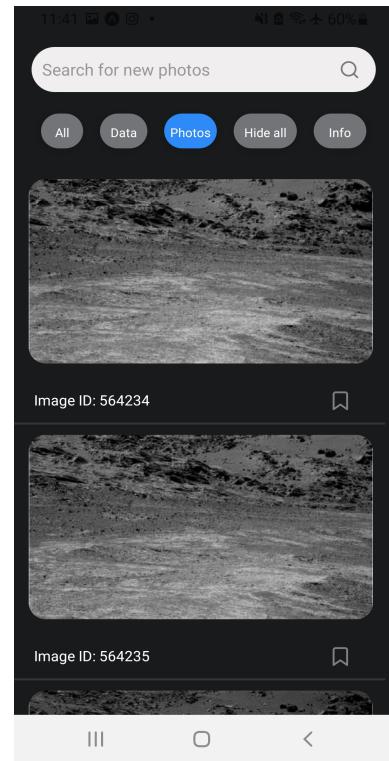


Figura 4.16: Lista delle immagini dopo l'uso del filtro Photos

4.8 Parametri di Ricerca Errati

Quando l'utente effettua una nuova ricerca di immagini, potrebbe inserire dei parametri di ricerca errati come ad esempio il nome del Rover errato e la data solare in formato errato; in tal caso nessuna immagine viene recuperata. In queste situazioni l'applicativo si occupa di informare l'utente del fatto che nessuna immagine è stata trovata e viene visualizzata a schermo una lista vuota.

Osservando le Fig 4.17 e 4.18 possiamo notare che le visualizzazione delle modali non cambia tra Android e iOS; l'unica differenza percepibile nei due dispositivi consiste in un testo più marcato in Android.

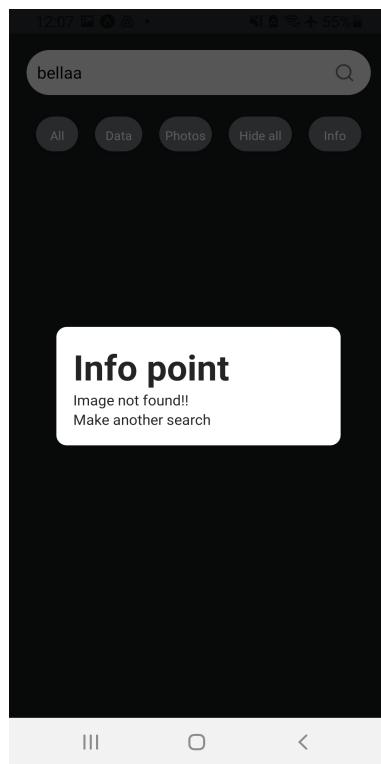


Figura 4.17: Android ricerca errata



Figura 4.18: iPhone ricerca errata

4.9 Animazione Onde Gravitazionali

Ogni volta che viene eseguita una nuova ricerca, dopo aver premuto il pulsante “All”, viene avviata una animazione. In questo modo l’utente ha una migliore UX (User Experience) in quanto viene informato che la ricerca è iniziata e l’animazione ricorda le onde gravitazionali, quindi lo spazio.

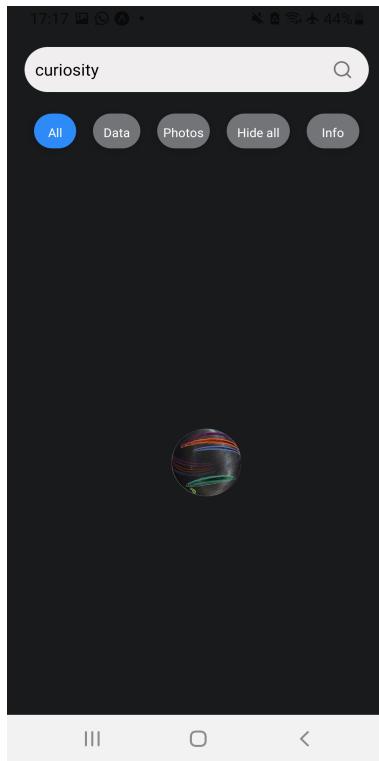


Figura 4.19: Android animazione onde gravitazionali

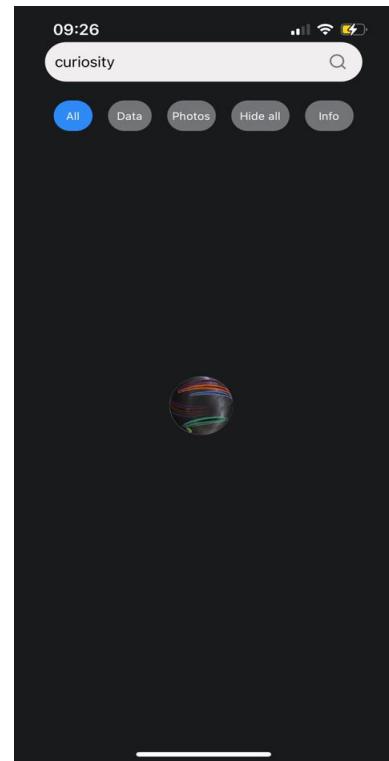


Figura 4.20: iPhone animazione onde gravitazionali

4.10 Istruzioni sui Metodi di Ricerca

Affinchè l'utente possa utilizzare correttamente l'applicazione, nello screen principale e in quello dedicato alla ricerca per data solare, sono state inserite delle icone “info-point”. Queste ultime se premute mostrano delle modali che istruiscono l'utente sul corretto uso dell'applicativo. Come già detto in precedenza la presentazione visiva delle modali non cambia tra iOS e Android.

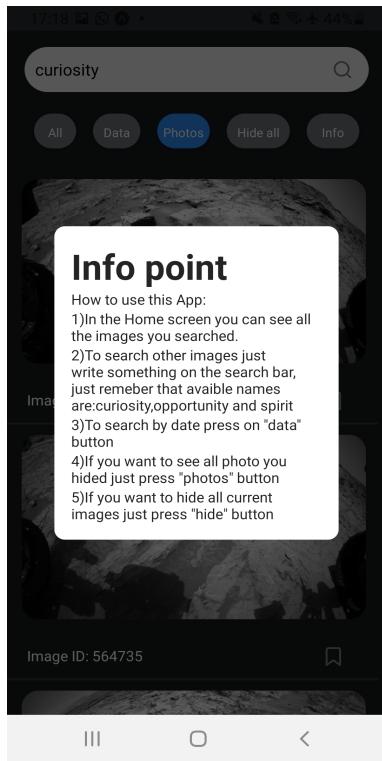


Figura 4.21: Android info-point

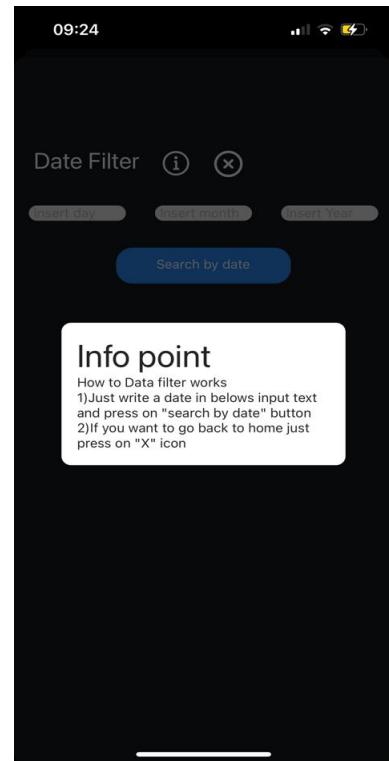


Figura 4.22: iPhone info-point

4.11 Procedura di Logout

Per effettuare il logout dal proprio profilo, l’utente deve scorrere le immagini fino alla fine della lista e premere sul pulsante “Log out”. Una volta fatto ciò si viene reindirizzati alla schermata di *sign in* o *sign up*.

Nelle Fig 4.23 e 4.24 si può vedere che in Android si ha una migliore User Experience, in quanto il pulsante di logout non si sovrappone ad altri componenti nativi del dispositivo: ad esempio sull’iPhone la barra orizzontale in fondo allo schermo si sovrappone al pulsante di Log out

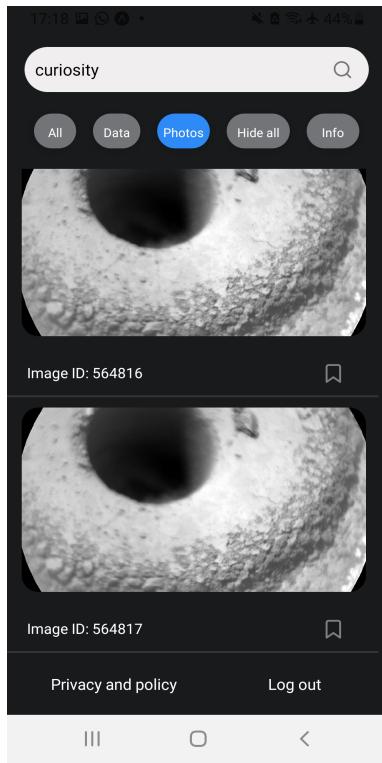


Figura 4.23: Android pulsante di logout

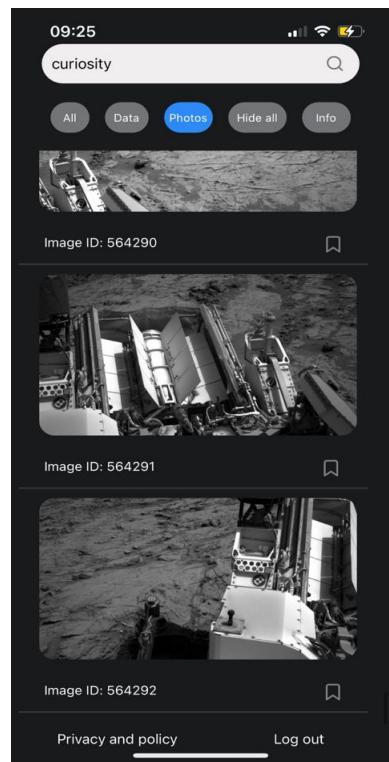


Figura 4.24: iPhone pulsante di logout

Conclusioni

In questo capitolo vengono discussi gli obiettivi, i risultati e le conoscenze acquisite al termine del tirocinio ed eventuali sviluppi futuri dell'applicazione.

4.12 Obiettivi e Risultati Raggiunti

Gli obiettivi principali di questo progetto di tirocinio sono stati:

1. Lo sviluppo di un'applicazione multipiattaforma, quindi di un applicativo che possa funzionare su diversi sistemi operativi.
2. L'uso del protocollo HTTP per la comunicazione con i server della Nasa.
3. L'utilizzo del formato JSON per la trasmissione e l'estrapolazione di dati.
4. La gestione della navigazione tra i vari screen dell'applicazione.
5. La comprensione e l'utilizzo dell'architettura modulare.

Tutti questi obiettivi non solo sono stati raggiunti con successo, ma è stato aggiunto anche uno strato di autenticazione e registrazione che non era richiesto dal progetto. Come si può dedurre dal Capitolo 4, il primo obiettivo è stato raggiunto in pieno: l'applicazione svolge le stesse funzioni sia su Android che iOS. Nonostante ciò, essendo stato utilizzato lo stesso codice per entrambe le piattaforme, vi saranno sempre delle differenze per quanto riguarda l'aspetto grafico. L'applicazione non presenta cali di performance, il che dimostra che lo sviluppo di applicazioni multipiattaforma è una valida

alternativa allo sviluppo nativo. Si sottolinea che questo applicativo non fa uso di funzioni native del dispositivo come GPS e notifiche Push.

Sia il secondo che il terzo obiettivo sono stati raggiunti con successo: dal Capitolo 3 si può infatti osservare che ad ogni ricerca vengono caricate delle nuove immagini, e questo è il risultato di diverse richieste HTTP. Inoltre l'uso del formato JSON per la trasmissione dei dati è stato cruciale: in effetti le immagini restituite dai server della Nasa sono proprio in questo formato. Tramite JSON è stato possibile comunicare facilmente al server locale le credenziali inserite da ogni utente e in seguito estrarre il JWT dalle risposte ottenute.

Durante l'utilizzo dell'applicazione, la navigazione tra i vari screen che la compongono risulta essere fluida e garantisce una buona User Experience. Nonostante ciò, essendo l'applicazione multipiattaforma, la navigazione può essere diversa tra i vari sistemi operativi a causa delle funzioni di navigazione native di ogni dispositivo.

Oltre a poter testare l'applicazione in locale ne è stata fatta anche la build in modo da poterla distribuire sugli store in un futuro, a seguito di ulteriori migliorie. La build è stata caricata sul sito di Expo ed è possibile scaricarne l'apk.

4.13 Conoscenze Acquisite

Terminato il progetto si può dire di aver approfondito ed acquisito competenze con nuovi strumenti e tecnologie, in particolare:

- JavaScript: l'utilizzo di JavaScript al di fuori dello sviluppo web è stato particolarmente utile per capire come utilizzare questo linguaggio sia per lo sviluppo lato server, tramite Node.js, sia per lo sviluppo dell'intera applicazione tramite l'utilizzo del framework React Native.
- Expo: il suo utilizzo ha permesso di iniziare lo sviluppo dell'applicazione fin da subito e lasciare la gestione di eventuali altre attività a servizi

di terze parti. Questo strumento è una perfetta scelta per l'introduzione allo sviluppo mobile.

- Node.js: attraverso la realizzazione di questo applicativo è stato possibile approfondire l'utilizzo di JavaScript nello sviluppo lato server, e comprendere meglio l'importanza delle API.
- Mongodb: questo database NoSQL è stata una scelta necessaria, in quanto il server locale e l'applicazione comunicano attraverso il formato JSON. Nonostante ciò, è stato molto interessante e utile imparare la logica di comunicazione e memorizzazione di un database NoSQL.

4.14 Sviluppi Futuri

Nonostante l'applicazione soddisfi tutti i requisiti di progetto, si potrebbe pensare di aggiungere altre funzionalità come:

- La possibilità di ricercare le immagini in base alla camera utilizzata per scattare le foto.
- La possibilità di ricercare le immagini per data “Marziana”.

Un altro possibile miglioramento potrebbe consistere nel memorizzare le ultime immagini visualizzate dall'utente sul database, piuttosto che mantenerle in modo persistente sul dispositivo: in questo modo si risparmierebbe memoria. Per garantire una migliore User Experience si potrebbe creare una classifica delle immagini più visualizzate: in questo modo ogni utente potrebbe dare la propria opinione sulle diverse immagini. In base a questa classifica si potrebbe decidere di non mostrare le immagini meno popolari.

Bibliografia

- [1] fulcrum. React native cli vs expo. <https://fulcrum.rocks/blog/react-native-init-vs-expo#:~:text=to%20get%20started.-,What%20is%20Expo,testing%20of%20React%20Native%20app>, 2022.
- [2] Corael srl. Differenza tra app nativa e app ibrida. https://corael.it/qual-e-la-differenza-tra-app-nativa-e-app-ibrida/?gclid=Cj0KCQjwg02XBhCaARIsANrW2X2GyVHY_r_m3okI4e8Kdy-KShS7Ftj0sZf30GWkEAD5f4R7_KFGfL4aAlIiEALw_wcB, 2021.
- [3] Bonnie Eisenman. *Learning React Native*, chapter Working with React Native. O'Reilly Media, 2016.
- [4] Meta. React native documentation. <https://reactnative.dev/docs/native-modules-intro>, 2022.
- [5] Expo, Software Mansion, and Callstack. React navigation documentation. <https://reactnavigation.org/docs/getting-started/>, <https://reactnavigation.org/docs/stack-navigator>, 2022.
- [6] Dan Abramov and the Redux documentation authors. Redux and react redux documentation. <https://redux.js.org/introduction/getting-started>, <https://redux.js.org/understanding/thinking-in-redux/three-principles>, <https://redux.js.org/understanding/thinking-in-redux/motivation>,

- <https://react-redux.js.org/introduction/getting-started>, 2022.
- [7] Giuneco S.r.l. Applicazioni native android e ios. <https://tech.giuneco.it/applicazioni-native-android-e-ios-parte-1/>, 2021.
- [8] Jet Brains. Kotlin documentation. <https://kotlinlang.org/docs/android-overview.html>, 2022.
- [9] Martin Heller. What is kotlin? the java alternative explained, 2020. <https://www.infoworld.com/article/3224868/what-is-kotlin-the-java-alternative-explained.html#:~:text=Kotlin%20is%20a%20general%20purpose,%2C%20clarity%2C%20and%20tooling%20support>.
- [10] Dawn griffiths. *Head First Kotlin*, chapter A quik Dip. O'Reilly Media, 2019.
- [11] Anshul Bansal. Java vs. kotlin. <https://www.baeldung.com/kotlin/java-vs-kotlin#:~:text=Java%20is%20a%20strictly%20typed, type%20of%20the%20assignment%20value>, 2021.
- [12] Apple. Swift overview. <https://developer.apple.com/swift/#open-source>, 2022.
- [13] Apple. Swift about. <https://www.swift.org/about/>, 2022.
- [14] Massimo Carli. Il garbage collector di java. <http://www.di-srv.unisa.it/~ads/corso-security/www/CORSO-9900/java/mirror/mokabyte/garbagecollector.htm>, 1998.
- [15] Code Academy. Kotlin vs swift. <https://www.codecademy.com/resources/blog/kotlin-vs-swift/>, 2021.
- [16] Lerma Gray. Kotlin vs swift. <https://devcount.com/kotlin-vs-swift/>, 2022.

- [17] Satinder Singh. Native app e multiplatform app. <https://www.netguru.com/blog/cross-platform-vs-native-app-development>, 2021.
- [18] Expo, Software Mansion, and Callstack. React navigation documentation. <https://reactnavigation.org/docs/navigation-container/>, 2022.
- [19] Expo, Software Mansion, and Callstack. React navigation documentation. <https://reactnavigation.org/docs/stack-navigator#api-definition>, 2022.
- [20] Expo, Software Mansion, and Callstack. Getting started. <https://www.knowledgehut.com/blog/web-development/redux-in-react-native>, 2022.
- [21] John Jakob. Getting started. <https://axios-http.com/docs/intro>, 2022.
- [22] Mozilla. promise. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise, 2022.
- [23] React Native. Flatlist. <https://reactnative.dev/docs/flatlist#listemptycomponent>, 2022.
- [24] React Native Component. Component rendering. <https://reactnative.dev/docs/intro-react-native-components>, 2022.