



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Informatica

ELABORATO FINALE

TITOLO

Sottotitolo (alcune volte lungo - opzionale)

Supervisore
Maurizio Marchese

Laureando
Jacopo Martinelli

Anno accademico 2018/2019

Ringraziamenti

...thanks to...

Indice

Sommario	4
0.1 Contesto	4
0.2 Motivazioni	4
0.3 Problema e Tecniche Utilizzate	4
0.4 Obiettivi	5
0.5 Conclusioni	5
0.6 Struttura della tesi	5
1 Open Air Museum	6
1.1 App native	6
1.2 iBeacon	6
1.3 Software gestionale	7
1.4 Server	7
1.5 Modelli ER	8
2 Strumenti e tecnologie	10
2.1 Javascript	10
2.1.1 Frameworks Javascript	10
2.2 Sviluppo mobile	11
2.2.1 Sviluppo mobile e Javascript	11
2.3 Audioguide	12
2.4 Tecnologie utilizzate in Open Air Museum 1.0	12
2.4.1 Filemaker	12
2.4.2 Node Js	12
2.4.3 MySql e SqLite	13
2.4.4 Java per Android e Swift	13
2.5 Tecnologie utilizzate in Open Air Museum 2.0	13
2.5.1 React	13
2.5.2 Redux	14
2.5.3 React-Native	14
2.5.4 Express	14
2.5.5 SqLite	14
3 Open Air Museum versione 2.0	15
3.1 Server	15
3.2 Applicativo Admin	15
3.3 Modello ER	15
3.4 Porting applicativi Nativi	16
3.4.1 Caratteristiche principali	16
3.4.2 Implementazione di Redux	20
3.4.3 Punti critici e problematiche	20
4 Implementazioni future	23
4.1 Esecuzione di processi in Nodejs	24

5 Analisi e Conclusioni	26
5.1 Confronto tra le due versioni	27
5.2 Punti critici	28
Bibliografia	28

Abstract

Ad oggi lo sviluppo software di applicativi client-server dispone di una ampia scelta di linguaggi e tecnologie. Solitamente per sviluppare con successo un applicativo è necessario integrare assieme molteplici di queste. Tale operazione non sempre è banale ed è fondamentale conoscere in modo approfondito gli ecosistemi che si desiderano integrare. Questa frammentazione rende necessario l'impiego di una serie di figure distribuite su tutto lo spettro tecnologico. Tutto ciò risulta in un incremento dei costi aziendali per medio-piccole realtà che desiderano affacciarsi a questo genere di mercato.

La rapida diffusione dei dispositivi mobili ha portato enti territoriali e pubbliche amministrazioni a utilizzare questo canale come nuovo vettore per catalizzare il turismo: Open Air Museum è nato in quest'ottica, mettendo a disposizione di un comune della Romagna un tool interattivo in grado di accompagnare i turisti nella visita della città. Gli applicativi di questo progetto sono stati sviluppati dal reparto software di Farnedi ICT, integrando assieme quattro tecnologie diverse: Filemaker, Nodejs, iOS e Android. Tale frammentazione ha costretto l'azienda ad avvalersi di un numero molto elevato di figure, aumentando i costi di sviluppo e diminuendo, di conseguenza, il profitto. Inoltre per mantenere lo stack tecnologico di Open Air Museum sarebbero necessarie le medesime figure, rendendo i costi del progetto insostenibili.

La mia tesi si propone di risolvere questa problematica riscrivendo la totalità degli applicativi utilizzando un solo linguaggio di programmazione. Tale operazione è possibile grazie ad una serie di tecnologie basate su Javascript che andranno a sostituire le quattro in uso in Open Air Museum.

Sommario

0.1 Contesto

Un ente pubblico come può essere un piccolo comune storico italiano necessita di una comunicazione efficiente e diretta con i propri visitatori. Open Air Museum, ha come obiettivo quello di guidare e indirizzare i turisti all'interno della città, storicamente e culturalmente ricca, con una guida virtuale che avrebbe potuto sostituire una persona fisica come guida turistica della zona. Data la grande varietà linguistica dei possibili utenti il prodotto è stato sviluppato multilingue.

Il progetto Open Air Museum è stato sviluppato da me e i miei colleghi del reparto di Sviluppo Software di Farnedi ICT[6], azienda informatica di Cesena. Consiste di un'applicazione mobile per dispositivi iOS[8] , di una seconda applicazione per dispositivi Android[2] e di un software in Filemaker[7] per il caricamento dei contenuti. In particolare io mi sono occupato dello sviluppo dell'applicazione mobile per iOS[8] in Swift 4[22] e dell'applicativo lato server in NodeJs[12].

Lo sviluppo del progetto ha richiesto circa sei mesi di lavoro e ha visto un solo upgrade. Durante il periodo di mantenimento del prodotto sono venute alla luce alcune criticità derivate dalle scelte tecniche iniziali: da queste considerazioni è nata la necessità di eseguire un porting dell'intero prodotto.

0.2 Motivazioni

Le motivazioni che mi hanno portato a scegliere questo tipo di progetto per la mia tesi triennale sono due: la prima è la necessità di invalidare la credenza che Javascript[9] sia un linguaggio di secondo ordine e che non possa essere utilizzato come linguaggio principale per grossi progetti. La seconda deriva dal desiderio di costruire un prodotto più performante e scalabile di quello precedentemente sviluppato per fornire all'azienda un'applicativo facilmente rivendibile e mantenibile.

0.3 Problema e Tecniche Utilizzate

La problematica principale affrontata era legata alla molteplicità di tecnologie utilizzate nel solution stack della prima versione dell'applicativo, che rendeva il mantenimento molto costoso. La soluzione che si è scelta per risolvere il problema è stata quella di costruire un nuovo solution stack utilizzando un unico linguaggio. Questo tipo di soluzione porta a una maggiore correlazione logica tra le varie parti del prodotto, alla possibilità di riutilizzare più codice in più zone del progetto e di utilizzare lo stesso pattern di sviluppo lungo tutta la codebase di Open Air Museum.

Per raggiungere questo obiettivo è stato necessario eseguire il porting di tutti gli applicativi che comprendevano Open Air Museum. Si è sostituito l'applicativo in Filemaker con una SPA[21] sviluppata in React[17]. Le applicazioni mobile sono state unificate utilizzando React-Native[18] in modo da poter sviluppare per due sistemi operativi differenti, sfruttando un'unica codebase. Anche il lato server ha subito un refactor: era infatti necessario adattarlo al meglio per le nuove richieste applicative, come la condivisione della configurazione attraverso l'intero solution stack.

Nel dettaglio mi sono occupato della ricerca sulle tecnologie da utilizzare, ho progettato l'infrastruttura applicativa ed ho attivamente sviluppato la parte server in Nodejs e gli applicativi mobile

utilizzando React-Native.

0.4 Obiettivi

L'obiettivo di questa tesi è dimostrare che è possibile, per un prodotto completo di questo tipo, riscrivere la totalità degli applicativi che lo compongono in Javascript, assicurandosi un vantaggio sia dal punto di vista dei costi di gestione che dell'effettiva manutenibilità del progetto. Tutto ciò mantenendo le medesime funzionalità e senza alterare la qualità del prodotto. Inoltre sarà valutata l'efficacia di adottare uno stack Javascript soppesando i costi di sviluppo e il peso tecnologico.

0.5 Conclusioni

Attraverso le tecnologie scelte, che verranno descritte approfonditamente nel secondo capitolo, è stato possibile riprodurre tutte le funzionalità della versione precedente, in quella full Javascript. Tale traguardo è stato possibile grazie al vastissimo ecosistema del linguaggio, che mette a disposizione tool e framework per creare applicativi ad ampio spettro: a partire dal lato server con Nodejs, alla parte mobile con React-Native. Questo tipo di approccio ha migliorato notevolmente l'esperienza di sviluppo, abbassando le barriere tecnologiche tra i vari applicativi che impediscono a sviluppatori frontend di approcciarsi al lato server e viceversa. Inoltre con questa soluzione è stato possibile rivalutare il concetto di riutilizzo del codice: ora è possibile condividere librerie in tutto lo spettro applicativo velocizzando ulteriormente la produzione software.

0.6 Struttura della tesi

Il capitolo 1 descrive nel dettaglio il progetto di Open Air Museum. Il capitolo 2 analizza le tecnologie impiegate nella prima versione dell'applicativo, e quelle disponibili attualmente sul mercato candidate a sostituirle. Il capitolo 3 presenta la nuova versione di Open Air Museum. Il capitolo 4 affronta la possibilità di espandere e migliorare ulteriormente il prodotto. Infine nel capitolo 5 viene eseguita un'analisi e una valutazione del lavoro svolto, con relative conclusioni.

1 Open Air Museum

1.1 App native

L'applicazione consiste in una guida turistica e multimediale in grado di mostrare ai suoi utenti una serie di punti di interesse collegati da percorsi preimpostati. Ogni punto dispone di un pacchetto multimediale di audio e immagini oltre ad una descrizione, un nome e delle informazioni tutte gestibili e configurabili in varie lingue. Tali punti possono essere navigati attraverso applicativi esterni di navigazione per smartphone - sfruttando la geolocalizzazione del dispositivo - oppure attraverso la modalità Esplora dell'applicazione. Quest'ultima permette, una volta attivata, di essere notificati automaticamente sulla presenza di punti di interesse nella zona circostante, dando la possibilità o meno, all'utente, di richiedere informazioni aggiuntive.

Questo è possibile grazie a due componenti: la tecnologia iBeacon, che verrà descritta nel dettaglio successivamente, e il *background processing*. Quest'ultimo è stato implementato mediante la registrazione di una *background task* per la localizzazione. Tale procedura permette all'applicativo di registrare all'interno del sistema operativo una finestra di esecuzione dedita a quell'operazione specifica. Questo genere di implementazione permette di risparmiare batteria, essendo controllabile quasi interamente dal sistema operativo, che nel caso in cui necessitasse di nuove risorse, potrebbe postporre l'esecuzione della task in background.

1.2 iBeacon

[APPROFONDISCI!]

La modalità esplora citata nella sezione precedente è stata implementata sfruttando la tecnologia iBeacon e Bluetooth 4.0, che permette al dispositivo di percepire l'ingresso in una specifica area, marcata da un piccolo radiofaro bluetooth, il quale è collegato a uno determinato punto di interesse dell'applicazione.

La metodologia con cui viene identificata una zona rispetto ad un'altra si basa sulla firma del Beacon, ovvero attraverso i suoi tre parametri identificativi: UUID, major e minor. Il primo è una stringa alfanumerica a 128 bit che ha lo scopo di identificare un gruppo di Beacon, mentre gli ultimi due identificano i singoli Beacon. Utilizzando lo UUID per definire una regione (Region) di ricerca è possibile escludere da essa tutti gli altri Beacon non appartenenti allo stesso gruppo. Infatti tutti i Beacon da noi posizionati sono contraddistinti da uno specifico UUID e differiscono per major e minor che permettono di identificarli singolarmente.

Un ulteriore metodo per gestire la “scoperta” di Beacon in una regione è quello di verificarne la distanza dal telefono, per scartare i Beacon più lontani e notificare all'utente solamente quelli nel raggio scelto. Questo è stato fondamentale nello sviluppo di Open Air Museum, data la maggiore densità di beacon in alcune aree della città, dove una semplice ricerca non basterebbe a definire la prossimità di un utente a un punto rispetto ad un altro. Questa features ha aiutato a realizzare una localizzazione più precisa, risultando in un'esperienza d'uso migliore.

1.3 Software gestionale

Attraverso l'applicativo in Filemaker, esposto tramite un'interfaccia web, i dipendenti del Comune potevano gestire l'inserimento dei punti e dei percorsi, compresi di media e testi relativi.

La visualizzazione dei punti avviene attraverso due schermate: la prima è una lista che mostra tutti i punti di interesse con la possibilità di entrare nel dettaglio e modificare le informazioni del punto selezionato e i contenuti multimediali associati ad esso; la seconda mostra l'elenco di tutti i percorsi disponibili e il dettaglio di ognuno di essi, contenente l'elenco dei punti da cui è composto e l'ordine di visita relativo. Per ogni testo vi è la possibilità di fornire una traduzione in tutte le lingue supportate dall'applicativo: italiano, tedesco, inglese, spagnolo e cinese semplificato.

Filemaker mette a disposizione un interfaccia drag and drop per la creazione della parte visuale dell'applicativo, permettendo di collegare i vari elementi trascinati nell'area di lavoro ad una specifica entry del database. Tale interfaccia viene quindi tenuta sempre in sincronia dal motore di Filemaker per riflettere i dati contenuti nel database: questo permette di avere un interfaccia facilmente modificabile e una raccolta dati consistente.

Il vantaggio principale dell'utilizzare questo genere di software si è trovato nella possibilità di convertirlo in una web application, con cui siamo stati in grado di fornire al Comune una piattaforma, senza particolari requisiti di utilizzo, per il caricamento dati. Tale scelta si è rivelata vincente vista la grande frammentazione dei dispositivi in uso negli uffici della suddetta P.A.

1.4 Server

La parte server, invece, espone delle Rest API che forniscono sia i dati dell'applicazione che un servizio di autenticazione mediante Json Web Token[10]. Questo applicativo è stato sviluppato senza l'ausilio di framework esterni sia per il routing e la gestione dell'autenticazione che per la generazione dei token di sessione.

Vista la grande quantità di file e la tipologia di dati da servire, è stato scelto NodeJs per la sua efficienza nelle operazioni di I/O rispetto ad una tecnologia più canonica come PHP[14] o Java[?]. L'implementazione di Nodejs di un I/O non bloccante[13] risulta in un invio più efficiente di file di medie dimensioni come audio o immagini, e un minor utilizzo di risorse. Tali media sono serviti utilizzando il file system come base, e mediante l'interfaccia messa a disposizione da Nodejs, vengono serviti attraverso una specifica API HTTP. I media sono salvati all'interno di un database MySql attraverso il percorso che hanno nel file system rispetto alla radice del progetto.

Questo approccio permette di svincolare il database da una task onerosa come quella di dover convertire i file ricevuti in base64 e di doverli salvare come entry tabellari. Tale procedura inserisce una criticità: i dati presenti nella memoria possono non riflettere quelli salvati a database e vice versa, perché non vi è una correlazione diretta fra i due. Le API sono state disegnate per simulare questa correlazione, e cioè, coadiuvato da un blocco di operazioni di lettura e scrittura nella specifica directory, mantiene i media e i dati a database in sincronia.

L'autenticazione, come anticipato in precedenza, avviene attraverso una tecnica chiamata Json Web Token[10], che consiste in uno standard open source basato su JSON per la creazione di token di accesso. Questi token sono firmati per essere piccoli e criptati mediante una chiave privata, così da essere decodificati solamente dalle parti autorizzate. Sono generati per essere URL-safe e per eseguire piccole transazioni di dati o per approvare un autorizzazione.

Nel caso di Open Air Museum vengono utilizzati per validare l'autenticazione: fin quando un token

resta valido, cioè finchè la sua data di scadenza non è maggiore della data della richiesta, il client che lo conserva può utilizzarlo per autenticare tutte le richieste http. Per fare questo, l'header di ogni richiesta http dovrà contenere la chiave 'Authorization' con valore 'Bearer' seguito da uno spazio e dal token. Tale autenticazione implementa lo standard RFC 6750: OAuth 2.0 Bearer Token[1].

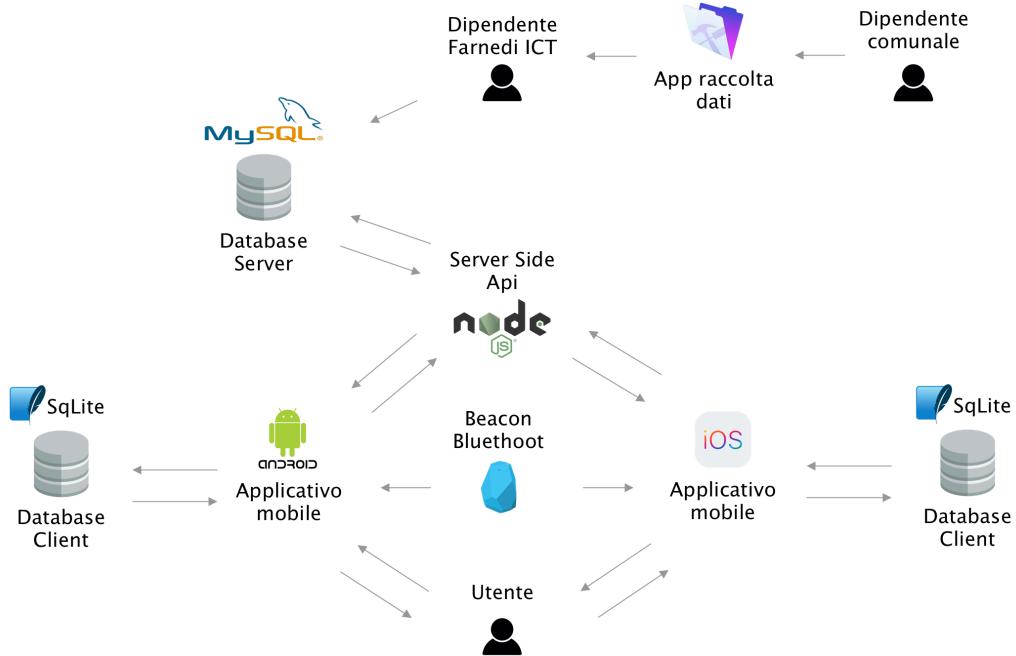


Figura 1.1: Schema logico dello stack di Open Air Museum

1.5 Modelli ER

Il lato server utilizza come database relazionale MySql, che viene sfruttato attraverso un interfaccia Javascript che permette di connettersi al database creando query e inviandole mediante questa connessione. Tutto questo in modo asincrono. Lo schema relazionale per il lato server è presentato in figura 1.2.

Per quanto riguarda le due applicazioni mobile, viene utilizzato sempre un database relazionale, ma sfruttando SQLite come tecnologia, che risulta essere più leggera e adatta a essere eseguita su dispositivi dalle prestazioni ridotte. La figura 1.3 mostra lo schema relazionale per il lato mobile.

[espandi descrivendo i vari quadrati/entità]

Come si può vedere, la gestione delle funzionalità multilingue dei punti e dei percorsi avviene attraverso una relazione da uno a molti con la tabella 'language'. Questa tabella contiene tutti i testi per lingua supportata. Tale accorgimento è assente nello schema relazionale dei dispositivi mobili dato che, al cambiamento della lingua, le nuove traduzioni verranno scaricate dalle Rest API fornite a lato server.

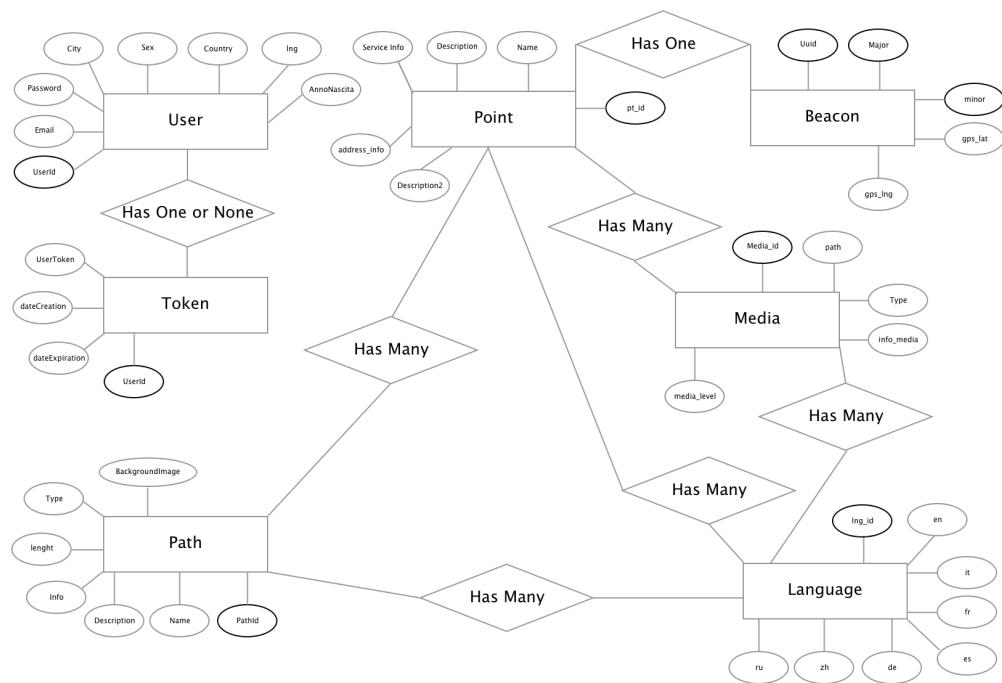


Figura 1.2: Schema ER vecchia versione server

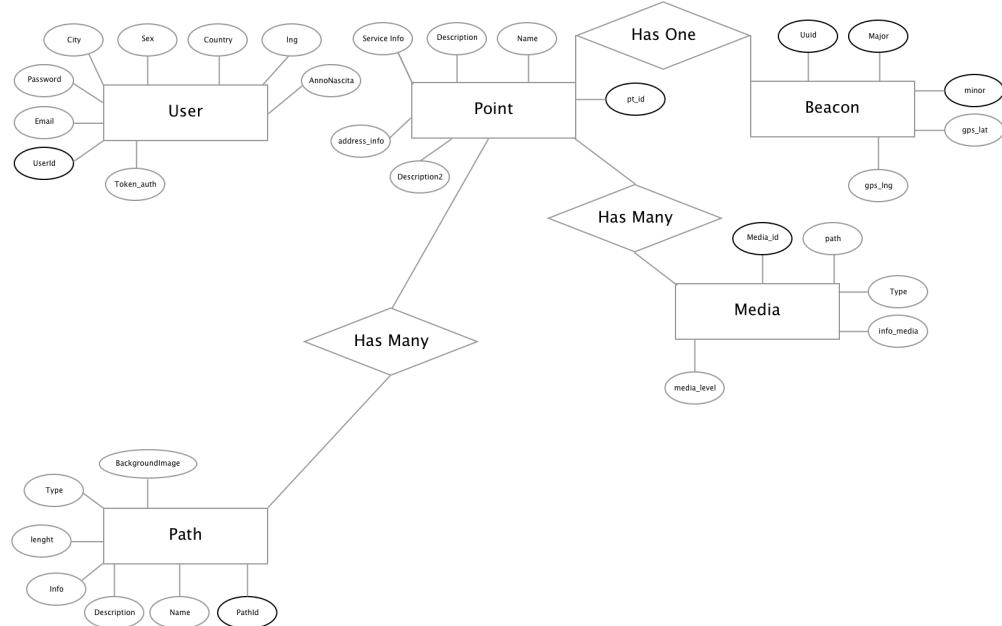


Figura 1.3: Schema ER vecchia versione smartphone

2 Strumenti e tecnologie

Questo capitolo descrive le varie tecnologie Javascript disponibili sul mercato e in che modo possono essere combinate per costruire il nuovo stack applicativo.

2.1 Javascript

Javascript è nato come linguaggio di scripting lato client

[Descrivo come funziona Javascript e V8]

2.1.1 Frameworks Javascript

L'ecosistema Javascript per il web è molto ricco di soluzioni per tutti gli ambiti di impiego. Per quanto riguarda il lato applicativo front-end vi sono numerose scelte, sia che si intenda operare nel browser che fuori da esso. I principali framework per lo sviluppo web front-end sono React, AngularJs, Vue e Ember. I primi tre implementano il pattern MVC “Model-View-Controller” che implica la separazione di ciò che detiene i dati da ciò che li renderizza all’utente attraverso una logica che fa da moderatore del flusso. Sia Angular che Vue che Ember offrono nativamente “*two-way data binding*” ovvero la capacità di mantenere la sincronia tra View e Model. React, d’altro canto, è stato pensato per essere una libreria e non come un vero e proprio framework, infatti il suo approccio è puramente legato alla costruzione dell’interfaccia grafica, slegata da quelle che sono tutte le implementazioni possibili nella gestione del flusso dei dati.

Nel caso di applicativi enterprise che devono gestire molti dati, questa parte può risultare complessa e particolarmente sensibile a bug. Per questo, soluzioni come Redux possono risultare funzionali per risolvere questo genere di problemi. Lo scopo di tale libreria è quello di contenere lo stato applicativo e fare in modo che sia sempre in uno stato certo e consistente, permettendo la modifica di esso solo attraverso delle “Action”. Ad ogni azione corrisponde una tipologia che il reducer interpreta per modificare lo stato. Il reducer è una funzione pura che prende in input lo stato attuale e l’azione eseguita e ritorna il nuovo stato applicativo. Essendo i reducer funzioni pure, sono prevedibili, prive di “Side effects” e facilmente testabili.

Per quanto riguarda la parte applicativa per le due piattaforme mobili, le scelte possibili sono racchiuse in Cordova/Phonegap, Ionic e React-Native. I primi due sono dei “wrap” attorno ad una webView nativa con una serie di API rese disponibili a Javascript che gira all’interno del browser che a sua volta risiede nella webView. Tale soluzione, seppur funzionale, non è ottimale per il caso d’uso specifico di Open Air Museum. Infatti l’applicativo mobile necessita di avere accesso al bluetooth e ad altre API non disponibili negli ambienti di Cordova/PhoneGap. Da notare che Ionic, seppur basato sulla medesima tecnologia, mette a disposizione delle API per interagire in modo completo con il bluetooth. Questo e la sua maturità lo rendono un’ottima scelta per sviluppare la parte mobile dell’applicativo.

D’altro canto, React-Native è un framework basato sull’utilizzo di un motore di compilazione che non fa altro che prendere i costrutti Javascript, dichiarati utilizzando librerie, e convertirli in codice nativo a seconda della piattaforma. Inoltre permette, se necessario, di interfacciarsi con l’ambiente nativo a piacimento, dando la possibilità di utilizzare un costrutto chiamato “bridge” che offre un’interfaccia a Javascript verso il nativo e viceversa. La differenza principale tra i sistemi che usano

Cordova e React-native è che il primo ha un ideale legato al concetto di “*Write once, run anywhere*” mentre il secondo a quello di “*Learn once, use everywhere*”.

Anche per la parte server è disponibile un’ampia scelta di tecnologie, che seppur orientate alla stessa soluzione, risultano molto diverse tra di loro. Vi sono librerie come ExpressJs che sono pensate per essere molto leggere e di lieve impatto nella strutturazione dei dati. D’altra parte, ve ne sono altre, come MeteorJs e SailsJs, che offrono un profondo controllo su tutti gli aspetti dell’applicativo e su una serie di features: la generazione automatica di API seguendo la specifica delle “CROUD operation”, la generazione automatica di modelli per descrivere nuove entità e la gestione automatica dell’autenticazione e degli utenti.

2.2 Sviluppo mobile

Al giorno d’oggi sono disponibili sul mercato una vasta gamma di smartphone che variano per dimensione e potenza. Tutti condividono la possibilità di utilizzare App: software provenienti da terze parti, installabili mediante uno store digitale messo a disposizione dal produttore del sistema operativo.

Solitamente per lo sviluppo di App si utilizzano i tool messi a disposizione dall’azienda produttrice del sistema operativo e cioè, nel caso di iOS e Android: rispettivamente XCode e Android Studio. Ogni OS ha a disposizione un sdk che permette di sfruttare l’hardware del dispositivo: le SDK di sistemi operativi differenti, come si può immaginare, presentano molte differenze, sia tecniche che logiche. La differenza principale sta nel linguaggio utilizzato: iOS utilizza ObjectiveC o Swift mentre Android si avvale di Java o Kotlin. Nel caso si desideri sviluppare per entrambi i sistemi operativi, l’enorme distanza tra i due comporta la necessità di avere una conoscenza approfondita di entrambi i linguaggi e dei rispettivi sdk.

2.2.1 Sviluppo mobile e Javascript

Oltre alle tecnologie descritte in precedenza, però, vi sono molteplici soluzioni che offrono la possibilità di sviluppare per entrambi i dispositivi iOS ed Android utilizzando come linguaggio Javascript. Queste soluzioni si differenziano per due aspetti fondamentali:

- Soluzione Ibrida
- Soluzione Compilata

La prima utilizza un approccio *ibrido*. L’applicativo, in questo caso, è una *Web Application* che viene eseguita all’interno di un browser che ha accesso all’hardware dello smartphone mediante un’interfaccia software. Questa interfaccia può essere vista come un layer al di sopra degli sdk nativi che permettono alla web app di accedere a funzionalità come il bluethoot e la fotocamera.

La soluzione compilata invece utilizza uno stratagemma diverso. Tutti i componenti descritti attraverso una particolare struttura vengono compilati dal framework e convertiti in componenti nativi, mentre la logica applicativa viene lanciata su di un thread separato che comunica con la controparte nativa attraverso un costrutto software chiamato *bridge*. Questo thread utilizza un motore per eseguire il codice Javascript, che viene fornito, in modalità sviluppo, da un server locale che ha il compito di compilare il sorgente; mentre viene incluso nei binari post compilazione una volta che l’applicativo deve essere rilasciato sullo store. Tale approccio permette di avere prestazioni più elevate e maggior flessibilità rispetto all’ibrido ma risulta più complesso. L’assenza di un’layer che astrae i rispettivi sdk obbliga quindi lo sviluppatore ad utilizzare direttamente il reame nativo sfruttando al comunicazione data dal *bridge*.

La scelta sulla tipologia da utilizzare va fatta a seconda delle specifiche progettuali. Se non sono necessari particolari accessi all'hardware del dispositivo o non sono necessarie prestazioni molto elevate, l'approccio ibrido risulta più efficiente e rapido. La soluzione compilata è da preferire in contesti dove sono necessari particolari accessi all'hardware o è necessario l'utilizzo di una libreria disponibile solamente nel lato nativo.

2.3 Audioguide

Vi sono disponibili diversi prodotti che offrono la possibilità di creare audioguide per determinate zone o città. Alcuni di questi utilizzano un approccio di *selezione luogo*; utilizza l'applicativo deve prima indicare a quale zona è interessato per poi essere reindirizzato alla sezione relativa a quell'area geografica. Altri prodotti come Open Air Museum utilizzano un approccio *mono scopo*. L'audioguida è relativa ad una sola area specifica e sviluppata ad Hoc per quel caso d'uso. L'approccio descritto per primo permette di dover mantenere un solo prodotto e di servire molteplici enti, questo però limita molto la customizzazione che tali applicativi possono possedere. Di contro il secondo approccio permette una customizzazione elevata, essendo il prodotto sviluppato appositamente per l'ente. Questo risulta dispendioso se si vogliono servire un gran numero di realtà, infatti ognuno degli applicativi avrà peculiarità e caratteristiche differenti.

Un ulteriore punto di riflessione sono le funzionalità. La totalità dei competitor presi in considerazione permette di visualizzare contenuti multimediali collegati a punti di interesse o percorsi, ma in pochi offrono un servizio di localizzazione attivo a guidare all'interno della zona di interesse.

La totalità degli applicativi presi in considerazione che hanno questa features utilizzano come tecnologia di localizzazione il gps che per zone all'aperto, come lo possono essere città o parchi naturali. Questa è un ottima soluzione, però per quanto riguarda la localizzazione indoor, come in un museo per esempio, questa tecnologia risulta imprecisa e poco utilizzabile. iBeacon in questo caso risulta la tecnologia migliore per localizzazione degli utenti dato che non deve fare affidamento a triangolazioni satellitari.

2.4 Tecnologie utilizzate in Open Air Museum 1.0

Indicherò le varie tecnologie utilizzate per l'attuale versione dell'applicativo.

2.4.1 Filemaker

Filemaker è un database con cui è possibile creare applicativi personalizzati. Open Air Museum utilizza questa tecnologia per creare l'interfaccia per la creazione di punti e percorsi che verrà utilizzata da alcuni addetti comunali che, oltre a questo, arricchiranno la piattaforma con i testi e media tradotti per le varie lingue supportate. Tale applicativo è standalone e quindi slegato dal lato server che espone le API e l'importazione dei dati provenienti da questa soluzione deve essere fatta manualmente. Ciò comporta uno svantaggio non indifferente e ne preclude una scalabilità rapida del prodotto. La motivazione dietro alla scelta di questa tecnologia era quella di avere il più rapidamente possibile un prodotto che mettesse il comune nelle condizioni di fornire dati consistenti.

2.4.2 Node Js

Come descritto in precedenza, Node js è un runtime di Javascript al di fuori del browser. Tale programma, come dicuterò più avanti, utilizza Chrome V8[5] come motore per eseguire Javascript ed grazie ciò NodeJs è stato fornito di una suite di interfacce per la comunicazione con l'hardware. In

particolare questa particolare implementazione viene indicata come I/O non bloccante[3]. La particolarità di questa soluzione è che mette a disposizione una serie di API a basso livello che permettono di utilizzare la scheda di rete e quindi di avviare server TCP e HTTP oltre che accesso al file system quindi lettura e scrittura su disco. In Open air museum questo framework è utilizzato per costruire il lato server ed esporre una serie di API Http Rest che permettono la fruizione dei dati. Tutti i media, quindi audio e immagini vengono serviti anch'essi da questa piattaforma che oltre a questo gestisce anche l'autenticazione degli utenti e la gestione degli stessi.

2.4.3 MySql e SqLite

Per la gestione dei database è stato utilizzato MySql per la parte server, mentre per la parte mobile SqLite. Sono entrambe soluzioni relazionali, la differenza tra le due è che la prima utilizza un applicativo che gestisce le interazioni con il database, accettando richieste e inviando dati attraverso un interfaccia tcp. Il secondo invece utilizza una gestione a file e cioè non sono necessari applicativi intermedi ma l'estrazione dei dati è fatta attraverso il file system. E' chiaro come la seconda soluzione sia più adatta su dispositivi che dispongono di poca potenza e una batteria limitata dato che non vi è necessità di avviare un ulteriore processo che gestisca l'interazione con i dati.

2.4.4 Java per Android e Swift

Per sviluppare gli applicativi mobili si sono utilizzati Java con Android Studio e Swift con XCode, rispettivamente per l'applicativo Android e IOs. L'utilizzo dei linguaggi nativi ha permesso un maggior accesso a quello che sono le funzionalità dell'hardware ed in particolare il bluethoot. Grazie alla tecnologia IBeacon è possibile localizzare gli utenti e mostrare contenuti coerenti con la loro posizione. Tale tecnologia è supportata da un gran numero di dispositivi che montano bluethoot LE, e posseggono una versione del sistema operativo superiore al 4.3 per android e 7.0 per IOs.

2.5 Tecnologie utilizzate in Open Air Museum 2.0

Indicherò le varie tecnologie utilizzate per la nuova versione dell'applicativo ponendo particolare attenzione a che vantaggi portano.

2.5.1 React

React[17] è una Libreria Javascript per lo sviluppo di interfacce grafiche a componenti, permette lo sviluppo di single page application (SPA). Tale libreria è basata sul concetto di Dom virtuale e One-way data flow. Il primo è un astrazione del Dom HTML che permette di eseguire manipolazioni dello stesso in maniera molto più efficiente, per poi essere renderizzato a seconda delle modifiche effettuate su di esso. Il secondo invece è il modo che utilizza React per trasferire i dati attraverso la virtualizzazione appena descritta; l'efficienza di React deriva dalla scelta di gestire il passaggio dei dati in un solo verso, e cioè da nodo padre a nodo figlio.

React utilizza una sintassi chiamata JSX per descrivere i componenti che andranno poi renderizzati mediante l'interazione tra Dom virtuale e Dom HTML. Tale sintassi ricorda l'html ma si tratta di zucchero sintattico. Infatti è necessario configurare un Transpiler come Babel per poter utilizzare questo tipo di sintassi. Una volta che il codice in JSX viene trasposto può essere utilizzato da qualunque browser infatti si tratta di una serie di chiamate a dei metodi che react usa per creare elementi nel Dom virtuale.

2.5.2 Redux

Redux è una libreria Javascript che permette di gestire lo stato applicativo come un unico oggetto statico, modificabile soltanto attraverso delle azioni ben definite applicate ad esso. Ogni modifica viene fatta attraverso una funzione pura[16] chiamata reducer che modifica a seconda dell'implementazione una parte specifica dello store. La firma di tale funzione prende due parametri, il primo è l'oggetto che rappresenta lo stato attuale dell'applicativo, il secondo è ciò che rappresenta l'azione che si sta compiendo. Tale azione dovrà sempre possedere una tipologia mentre può o non può avere un payload di dati che si riferiscono a quell'azione specifica. Ciò che ritornerà il reducer sarà un oggetto che diventerà il nuovo stato applicativo.

Una delle specifiche di Redux è quella che i Reducer siano sincroni e quindi all'interno di essi non è possibile eseguire richieste tramite la rete o utilizzando risorse hardware come l'accesso alla memoria. Per sopporire a questa mancanza si utilizza un costrutto chiamato middleware. Il compito del middleware è quello di eseguire un'operazione quando una specifica azione viene eseguita. Tale operazione può essere asincrona e l'unica specifica è che alla sua terminazione essa dovrà lanciare un'altra azione sullo store. In questo modo si possono integrare con delle Rest API, eseguendo un'azione per avviare la richiesta ed una per inserire i dati nello store.

2.5.3 React-Native

React-Native è l'implementazione di React ma per l'ambiente nativo. Si basa sul concetto di visualizzazione del Dom descritto in precedenza con l'unica differenza che la renderizzazione viene fatta tramite un processo che converte i vari nodi del Dom in componenti grafici nativi. In questo modo si mantiene la resa del codice nativo ma pilotata tramite il motore di React. Tale approccio permette di scrivere applicativi nativi utilizzando le conoscenze che si posseggono per l'ambiente web senza per forza rinunciare alle prestazioni dovute alle costrizioni imposte da un web browser. Le API di React-native permettono inoltre di controllare mediante una specifica dichiarazione del codice nativo. La comunicazione tra il reame Javascript e il reame nativo avviene attraverso un costrutto software chiamato bridge. Tale interfaccia permette di avere una comunicazione full duplex tra il thread Javascript ed altri Nativi. Tale comunicazione avviene in modo completamente asincrono e non bloccante.

2.5.4 Express

Express è l'implementazione del pattern middleware per Node Js. Tale framework permette di gestire con granularità il routing delle API e da la possibilità di centralizzare la gestione degli errori. L'implementazione di tale pattern si riflette in un applicativo che esegue le computazioni attraverso una cascata di funzioni chiamate una dopo l'altra in ordine, ed ognuna di esse ha la facoltà di decidere se continuare la catena o interrompere la computazione. Questo permette di avere una struttura affidabile per gestire il routeing delle Rest API rendendo molto veloce e predittivo lo sviluppo.

2.5.5 SqLite

SqLite è “File-based” database utilizzato lato server per il salvataggio delle informazioni fornite dagli utenti admin attraverso l'applicativo web per la raccolta dei dati. Tale scelta è stata fatta per la flessibilità di questa tecnologia che permette di gestire internamente all'applicativo la creazione dello stesso e non attraverso una dipendenza esterna. Questo è stato fatto per semplificare ulteriormente il deploy eliminando le necessità di aggiungere una configurazione ulteriore all'ambiente.

3 Open Air Museum versione 2.0

3.1 Server

L'applicativo lato server è stato riscritto riscritto ma le sue funzionalità principali sono rimaste invariate. Ciò che è stato modificato sono le tecnologie utilizzate. Ora la gestione del database viene fatta attraverso un ORM chiamato Sequelize che offre una serie di funzionalità fondamentali per rendere l'applicativo. In primis da la possibilità, nel caso dell'utilizzo di SQLite come tecnologia, di generare un database all'avvio dell'applicativo se al percorso indicato non ne esiste alcuno. Genera, la struttura del database utilizzando la dichiarazione di modelli che avvengono mediante le sue api. Infine offre la possibilità di astrarre le query tramite un interfaccia più famigliare a Javascript senza obbligare il programmatore a generarle tramite concatenazione di stringhe o altri metodi. Un altro importante framework utilizzato per il routing delle api è Express.

Una features che differenzia la vecchia implementazione rispetto alla nuova sono una serie di endpoint che permettono di gestire e inviare la configurazione degli applicativi lato client così da centralizzare il più possibile la personalizzazione degli applicativi sia mobile che admin.

3.2 Applicativo Admin

Il compito dell'applicativo admin è quello di sostituire quello in Filemaker ed espandere alcune sue funzionalità. Come descritto in precedenza questo applicativo è stato sviluppato come una SPA utilizzando React come framework di sviluppo. Le features che possiede sono: la possibilità di modificare testi e media per i vari punti e percorsi di interesse, la possibilità di creare percorsi selezionando e ordinando punti precedentemente creati e la capacità di poter monitorare l'utilizzo della piattaforma attraverso un pannello che mostra i punti più visitati e i percorsi più seguiti. L'accesso a tale applicativo è possibile solo attraverso autenticazione contro il server mediante le Rest Api da esso fornite.

3.3 Modello ER

Ora descriverò brevemente il nuovo schema logico dell'applicativo e come le varie tecnologie, tutte basate su Javascript interagiscono fra loro.

Come si può vedere in figura 2.4 lo stack risulta meno complesso e vanta l'utilizzo di meno tecnologie per raggiungere le medesime funzionalità. Come si può notare i punti di interazione con la piattaforma sono gestiti da due applicativi basati entrambi su React, il primo è l'applicativo mobile che utilizza React-native mentre il secondo è la webApp admin. Entrambi questi applicativi ricevono input dall'utente ma il primo utilizza hardware esterno per arricchire la propria raccolta informazioni con la posizione dell'utente, ricevuta attraverso il gps o Beacon bluethooth. Tali applicativi interagiscono con il server mediante delle Rest Api che, come nella versione precedente, forniscono i dati sui punti e i percorsi; ma, in contrasto con l'applicativo prima del refactor, portano anche la configurazione impostata dall'applicativo admin in modo da adattarlo alle richieste di chi gestisce i punti e i percorsi. Tali dati vengono salvati in un database SQLite il cui unico punto di accesso è attraverso le api fornite dal server.

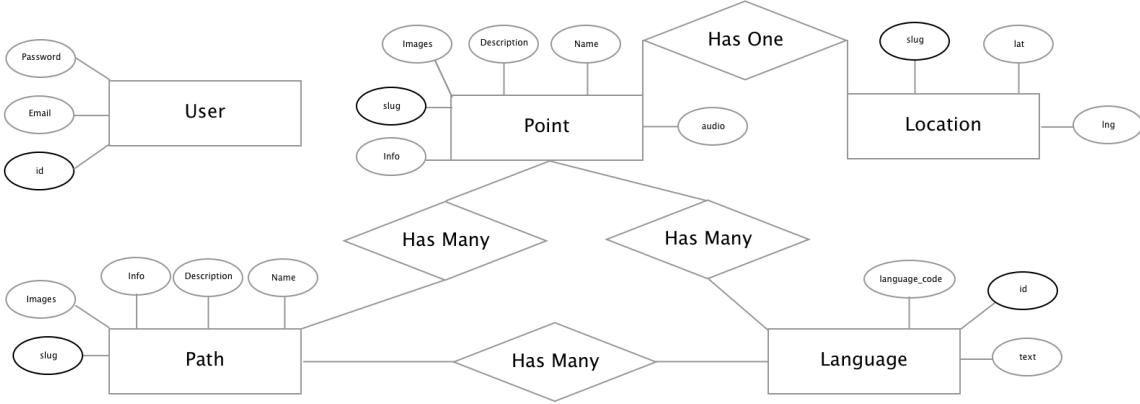


Figura 3.1: Schema ER nuova versione

Tale approccio garantisce che i dati siano sempre consistenti con quello che è l'applicativo server in modo da mantenere la struttura consistente con il versionamento del prodotto. Un ulteriore struttura per mantenere i dati è presente a lato mobile mediante Redux, questo è utile per poter salvare le configurazioni e mantenerle tali in modo consistente attraverso l'app in modo da creare una coesione tra tutti i componenti di React che necessitano di usufruire dei dati. Tale stato applicativo viene idratato attraverso le api Rest e quando questo non è possibile per mancanza di rete permette di utilizzare tramite React-native lo Storage locale del dispositivo come luogo di backup. Ogni mutazione dello stato applicativo viene infatti seguita da un'operazione asincrona che ha lo scopo di salvare uno snapshot dello store nella memoria del dispositivo in modo da poter essere sempre consistente nel mostrare gli ultimi dati disponibili anche se il dispositivo risulta offline.

3.4 Porting applicativi Nativi

Qui descriverò nel dettaglio il porting degli applicativi nativi a React-native, descrivendo il percorso implementativo e riassumendo i punti critici.

3.4.1 Caratteristiche principali

La caratterizzazione fondamentale di questo tipo di porting è quello di non compromettere la funzionalità iniziale dell'applicativo, mantenendo un “lookAndfeel” il più possibile vicino all'applicativo iniziale, ma apportando allo stesso le possibili migliorie date dal nuovo ambiente. La prima problematica affrontata è stata quella di poter gestire la modalità Esplora da Javascript e cioè essere in grado di controllare il bluetooth dello smartphone e mediante una task in background monitorare l'avvicinarsi o meno a specifiche aree marcate da un dongle. La tecnologia di comunicazione bluetooth supportata dalla versione nativa era IBeacon mediante un SDK sviluppato da Estimote. La decisione di mantenerla anche nel porting è stata data dall'hardware già configurato e presente sul territorio. Come detto in precedenza la comunicazione tra il reame nativo e quello Javascript avviene attraverso una particolare struttura software chiamata bridge. Tramite le Api messe a disposizione da react-native è possibile utilizzare questa struttura per passare dati tra i due reami. Ora descriverò nel dettaglio in che modo utilizzare tali api per garantire nel reame Javascript le stesse Api dell'SDK Estimote proximity, sia su iOS che su Android.

Per quanto riguarda iOS è necessario aggiungere alle dipendenze del progetto Estimote Proximity SDK, mediante CocoaPods, un dependency manager per XCode. Inoltre è necessario per abilitare la localizzazione in background aggiungere tra le “Capabilities” dell'applicativo nella voce “Background Mode” la voce “Uses Bluetooth LE accessories”.

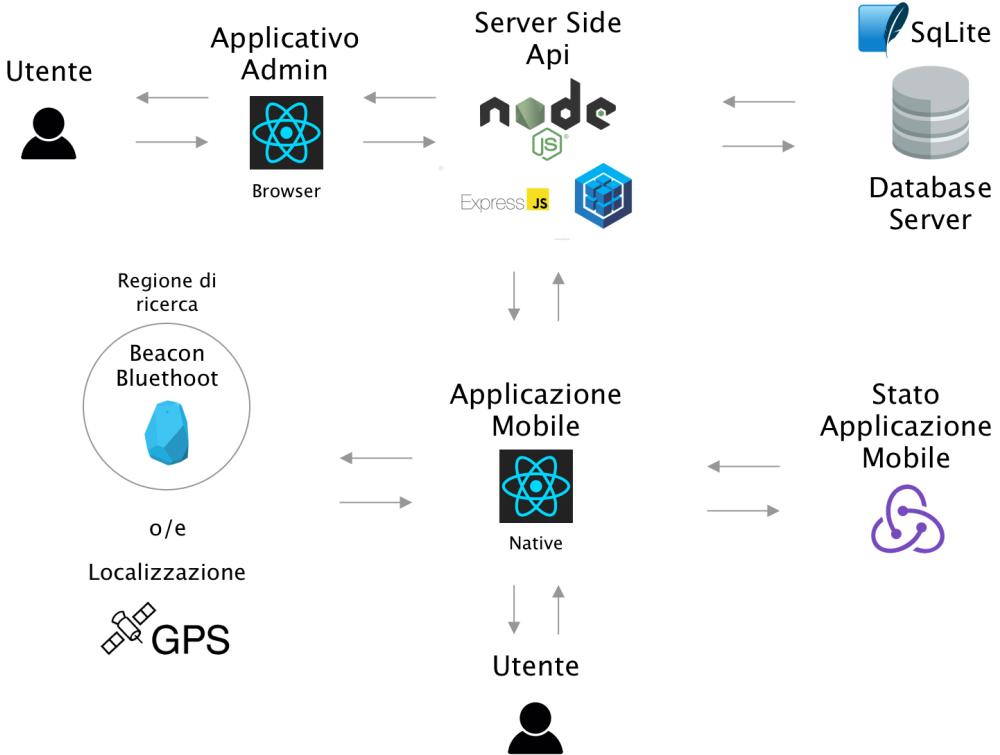


Figura 3.2: Schema logico della struttura 2.0

A questo punto è necessario creare un file che permetta di dichiarare ed esportare i metodi di configurazione e gestione della libreria dichiarandoli utilizzando l'interfaccia fornita da RCTBridgeModule di react-native per renderli visibili lato Javascript al momento della compilazione e dell'esecuzione dell'applicativo. Un ulteriore punto importante è che la comunicazione attraverso il “bridge” è asincrona e per questo è possibile passare valori da nativo a javascript mediante callback o eventi.

Nel caso specifico la scelta è ricaduta su di un sistema ad eventi, per la maggior flessibilità di utilizzo. In particolare i metodi messi a disposizione attraverso il “bridge” sono:

- initialize:(NSDictionary *)config
- startObservingZones:(NSArray *)zonesJSON
- stopObservingZones()

Il primo ha lo scopo di inizializzare l'sdk Estimote attraverso un oggetto che contiene i vari parametri di configurazione. Il secondo permette di mettere il device in ascolto su una lista di zone identificate mediante dei tag, quando il telefono entrerà in una zona così configurata verrà lanciato l'evento “Enter”. Per l'uscita da una regione verrà lanciato l'evento “Exit” mentre per un cambio di contesto mentre si è in una zona verrà lanciato “Change”. Il contesto è un parametro passato nella funzione chiamata dalla sottoscrizione a questi eventi e contiene le informazioni del beacon che rappresenta quella regione.

Per quanto riguarda Android lato Javascript le api sono le medesime per dare una sensazione di uniformità tra le due piattaforme. Per quanto riguarda l'implementazione nativa l'implementazione dell'sdk avviene in modo paritetico che con ios ed anche l'esportazione dei metodi attraverso il “bridge”; ciò che cambia effettivamente sono le configurazioni necessarie date da un ambiente diverso (Java).

A questo punto mediante un interfaccia messa a disposizione da React-native è possibile quindi utilizzare i metodi esportati dalle due piattaforme. Tali metodi sono esportati attraverso l'oggetto "NativeModules" che a sua volta contiene un oggetto per ogni classe che implementa l'interfaccia RCT-BridgeModule. A questo punto è stato scelto utilizzare un file Javascript che normalizza l'esportazione dei vari metodi e nel caso in cui vi siano differenze tra una piattaforma e l'altra è bene esportare solamente quelli disponibili per quel Sistema operativo. Nel caso di questa particolare implementazione le api sono paritetiche per cui non sono stati necessari accorgimenti specifici.

Voglio far notare che, seppur i due sistemi operativi sono molto differenti, le api lato Javascript risultano paritetiche. Questo offre un astrazione che elimina la complessità di dover gestire singolarmente due linguaggi diversi con ambienti diversi. React-native offre mediante le "Bridge Api" la possibilità di costruire librerie non "cross platform" ma "multi platform". Tale valore non è indifferente e può risultare cruciale per quanto riguarda tempi di sviluppo e features del prodotto dando la possibilità di scrivere solo il codice nativo necessario, sviluppando poi il resto della logica in un'ottica "write once run anywhere".

Dal punto di vista grafico inoltre sono state apportate molte modifiche dati i feedback di molti utenti che trovavano l'applicativo precedente poco intuitivo. Si è passati da una visualizzazione dei punti di interesse da lista a mappa. Infatti molti utenti si lamentavano del fatto che non era immediato capire dove fossero i punti rispetto alla loro posizione.

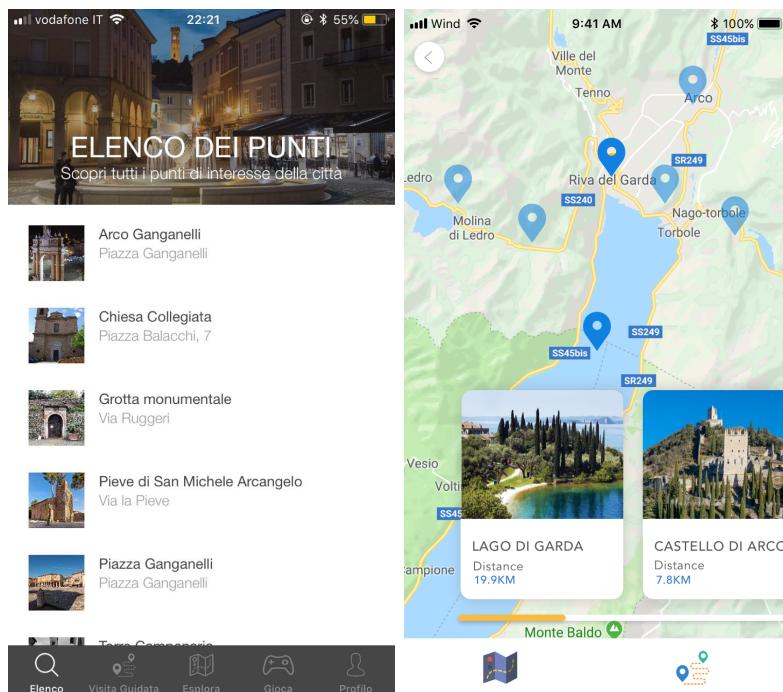


Figura 3.3: Confronto HomeScreen Open Air Museum vecchia e nuova versione

Sfruttando la potente libreria di animazioni fornite da React-native è stato possibile creare un'interfaccia ricca e moderna mantenendo le prestazioni elevate anche su dispositivi non recenti e di alta fascia. Il "driver" di tale libreria è infatti nativo ed implementato nel modo più efficiente per i due sistemi operativi. Questo offre solide prestazioni anche nel caso di animazioni molto complesse ma aggiunge anche alcune limitazioni. La più evidente è l'obbligo di implementare tali animazioni con un pattern dichiarativo indicando quindi in precedenza il modo in cui l'oggetto deve comportarsi per poi avere la possibilità di eseguirlo mediante un'interfaccia di start e stop. Tale problematica non

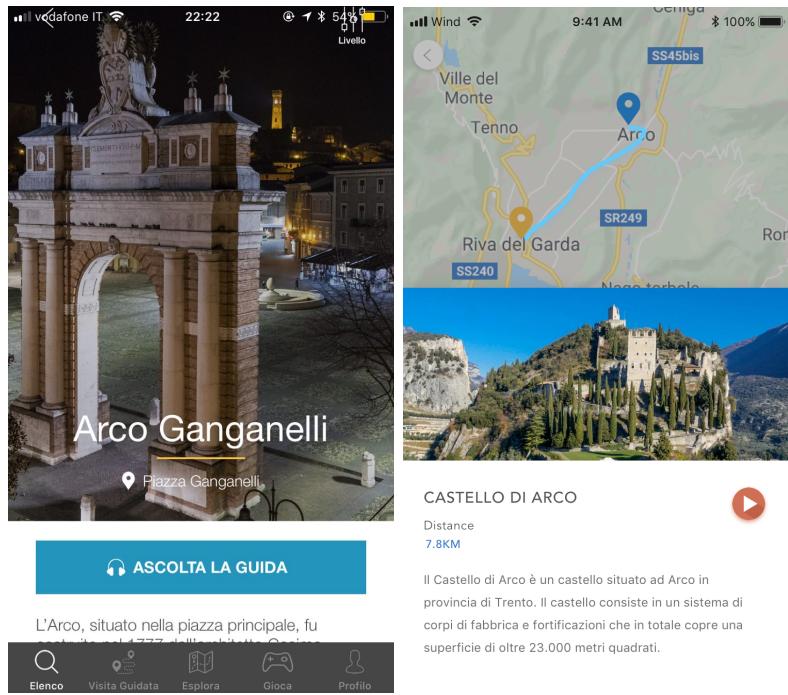


Figura 3.4: Confronto HomeScreen Open Air Museum vecchia e nuova versione

ha composto un grave problema implementativo. DI fatto si è trattato solamente di implementare il tutto seguendo il pattern descritto sopra.

Un’ulteriore funzionalità richiesta al porting è quella di poter, riprodurre dei contenuti audio all’interno dell’applicativo. In particolare questi contenuti audio sono presenti sul server e sono serviti attraverso un’api che permette il loro streaming. In precedenza era stato sviluppato un player che utilizzava i rispettivi sdk nativi per controllare l’audio, erano in grado di avviare e stoppare l’audio e proseguire avanti veloce mediante una barra di trascinamento. Le funzionalità per i due sistemi operativi erano le stesse ma viste le diversità tra le due piattaforme è stato necessario re implementare la stessa logica due volte. Utilizzando React-native è stato possibile scrivere la logica una sola volta e condividerla tra le due piattaforme utilizzando un package open source chiamato react-native-audio-streamer. Tale package permette di configurare uno stream audio mediante un url e di controllare la riproduzione, tale libreria utilizza la medesima implementazione attraverso il “bridge” sfruttando a livello nativo due librerie distinte. Per Ios DuoAudioStreamer mentre per Android ExoPlayer. L’utilizzo di tale package accelera lo sviluppo di questa parte mostrando il vantaggio di React-native sopra a soluzioni simili e cioè l’ecosistema. Creare una dipendenza esterna in un progetto enterprise come questo può essere una scelta rischiosa. Legarsi ad un software di terze parti per una features cruciale dell’applicativo può sfociare in problematiche che spaziano dal mantenimento futuro a bug non facilmente tracciabili ma data la semplicità di questa particolare libreria tale rischio è ragionevolmente contenuto.

Come si è visto l’impiego di React-native non ha soppiantato la totalità della parte nativa, anzi questo approccio “misto” ha permesso di non trovarsi limitati da una tecnologia rispetto ad un’altra offrendo sempre il giusto tool per la features richiesta. Durante le ricerche legate allo sviluppo di questo prodotto non sono stati trovati linguaggi o framework più flessibili e adatti alle molteplici configurazioni di questo progetto, questo specifico caso è un ottimo esempio di come l’impiego di Javascript al di fuori dal browser sia la scelta vincente per creare un applicativo moderno e flessibile senza sacrificare manutenibilità e consistenza nel tempo.

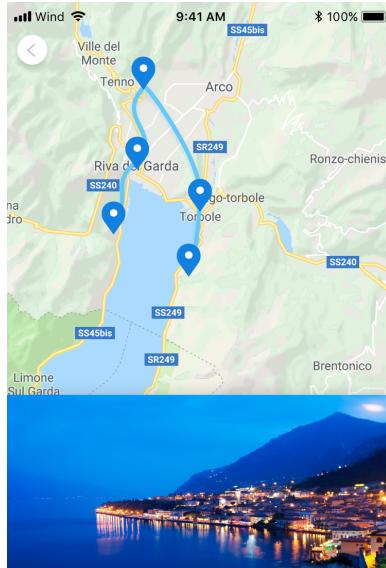
sotto la cinta muraria malatestiana, in cui si praticava l'antico gioco del pallone al bracciale e le splendide fontane ideate dall'illustre concittadino Tonino Guerra.

INFO AGGIUNTIVE

Tour della cittadella medievale

ELENCO DEI PUNTI

	Piazza Ganganelli Piazza Ganganelli
	Arco Ganganelli Piazza Ganganelli
	Centro Commerciale naturale Via Saffi
	Grotta monumentale Via Ruggeri
	Piazzetta delle Monache Piazzetta Monache
	Convento Santa Caterina e Barbara Contrada dei Signori



PERCORSO LAGO DI GARDA NORD



Figura 3.5: Confronto HomeScreen Open Air Museum vecchia e nuova versione

3.4.2 Implementazione di Redux

Per l'implementazione di Redux si è scelto di sviluppare una piccola libreria da condividere tra l'applicativo mobile e la web-app admin. Questa libreria comprende l'insieme dei Reducer e delle Action disponibili sia nell'applicativo mobile che in quello web e un costrutto software per usufruire delle Rest Api lato server. Questo approccio ha permesso di velocizzare notevolmente lo sviluppo permettendo di riutilizzare la parte che gestiva il flusso logico applicativo concentrandosi solamente sui componenti dediti alla visualizzazione grafica dei dati.

Come descritto in precedenza lo stato applicativo consiste in un oggetto immutabile che detiene tutti i dati necessari all'applicativo. Quindi oltre ai vari punti e percorsi nello store sono contenute anche delle impostazioni relative all'applicativo specifico, come la lingua selezionata dall'utente o l'utilizzo o meno di IBeacon come sistema di localizzazione.

3.4.3 Punti critici e problematiche

Durante questo porting sono stati evidenziati una serie di punti critici implementativi, ne discuterò indicando la soluzione da me intrapresa e se presente una soluzione alternativa.

L'utilizzo di una libreria esterna come quella per la gestione degli stream audio oppure quella per gestire la mappa, creano delle dipendenze che aggiungono una serie di incognite sulla longevità del prodotto. E' importante legare il progetto il meno possibile a risorse esterne non del tutto affidabili. Per tale ragione entrambe le risorse esterne che l'applicativo utilizza e cioè la libreria per lo streaming audio e l'SDK Estimote sono gestite attraverso un layer software. L'applicativo utilizza la libreria attraverso le Api fornite da questa interfaccia, in caso di necessità si può sostituire più agevolmente la vecchia libreria con la nuova mappando le nuove api sull'interfaccia. Ovviamente questo è possibile nei limiti di una coesione logica tra la il vecchio e nuovo SDK. Da notare che un'implementazione di questo tipo permette inoltre di abbandonare del tutto il software esterno implementando la propria libreria integrandola nel progetto facilmente attraverso tale interfaccia. Un ulteriore possibile soluzione a questo problema sarebbe quello di gestire internamente la maggior parte del software ma in alcuni

```

interface StatusApp {
  settings: {
    useBeacon: boolean,
  };
  userData: {
    lang: string;
  };
  points: [
    {
      index: number;
      images?: number[];
      slug: string;
      description?: string;
      info?: string;
      metadata?: {
        ebTikeiID: string
      };
      location: {
        slug: string;
        lat: string;
        lng: string
      }
    }
  ];
  paths: [
    {
      slug: string;
      desctiption: string;
      duration: number;
      points:[
        {
          order: number;
          slug: string;
        }
      ]
    }
  ]
}

```

Figura 3.6: Interfaccia Store Redux

casi questa soluzione deve essere scartata per l'elevato costo.

Altra problematica è data dalla coesistenza di tre linguaggi differenti nella stessa codebase. In alcuni casi React-native può aggiungere complessità invece che rimuoverla, infatti se la quantità di codice nativo a cui dover attingere è molto grande si rischia di dover mantenere tre ecosistemi a differenza di due. Maggiori sono le tecnologie da conoscere per operare su di un prodotto maggiore è il bagaglio necessario per mantenerlo, questo rende il software più complesso da gestire. In questo caso un'attenta analisi del progetto prima dello sviluppo può guidare verso una soluzione differente ed evitare il problema. Nel caso specifico la quantità di codice nativo necessaria non è stata molta ma tale problema potrà presentarsi in futuro con il proseguire del progetto.

Un alto punto critico da mantenere in considerazione con questo tipo di porting è la completa scomparsa di database Sql in favore di uno state manager ad oggetti. Data l'impossibilità di gestire i dati in modo paritetico rispetto alla versione nativa è stato necessario riadattare la logica applicativa a questa nuova tecnologia. Non esistendo più tempi di latenza per accedere ai dati questo ha migliorato molto la pulizia e l'asciuttezza del codice. La modifica dei dati avviene attraverso mutazioni allo stato e questo comporta in caso di applicativi molto grandi una crescente complessità se non si dividono

in modo accorto le varie aree dello stato dividendo in modo netto una area dall'altra usando nodi il più possibile vicini alla radice. Tale approccio infatti permette di ottenere una buona "separation of concerns" [19] delle varie aree logiche dell'applicativo ponendo entrypoint diversi per l'accesso ai dati a seconda della zona logica prestabilita. Ciò permette di risolvere il problema descritto sopra, offrendo un guadagno in termini di semplicità rispetto ad una soluzione basata su SqLite.

4 Implementazioni future

Uno degli aspetti principali che hanno portato al refactor di questo progetto è stata quella di renderlo più flessibile ed adattabile ad altri contesti e non solo legato al comune che ha commissionato l'applicativo. Un ulteriore passo in questa direzione è lo sviluppo di un altro applicativo che abbia le funzionalità di un orchestratore. Tale applicativo, anch'esso sviluppato mediante Nodejs sarà in grado di sfruttare a pieno l'alta configurabilità di ogni applicativo che comprende lo stack Alakai, eseguendo in modo automatizzato la configurazione e il deploy di tutti i servizi necessari. Per raggiungere tale scopo è necessario implementare alcuni accorgimenti. I punti che descriverò ora sono delle linee guida generali e non passaggi sequenziali verso un prodotto finito.

Il primo tra questi è possedere un portale dove richiedere all'utente le configurazioni necessarie, come il nome dell'applicativo, se si desidera utilizzare o meno l'Estimote SDK oppure utilizzare il GPS, le Api key di Eventbrite e le chiavi degli store per la firma degli applicativi per IOS e per Android. Una volta ottenute queste informazioni è necessario costruire un manager che, una volta importata la configurazione, esegua un deploy della parte server in Nodejs, trasponga l'applicativo admin in React e lo inserisca all'interno di una directory prestabilita sul server in modo che possa essere servito staticamente e infine compili e firmi i due applicativi per Android e IOS. Le problematiche da risolvere per raggiungere un prodotto di questo tipo sono molteplici, ne discuterò solo alcune.

E' necessario possedere un ambiente in grado di ospitare un applicativo in Nodejs e servire mediante un proxyPass sia l'applicativo in React sia le Api sullo stesso dominio. Per fare questo è necessario che la macchina abbia installata una versione di Node Js superiore al 6 e aver installato e configurato Nginx o Apache in modo da implementare il proxyPass descritto sopra. Tali configurazioni sono di fatto uguali ad ogni istanza che si desidera allocare per cui una soluzione per gestire ed automatizzare il tutto è utilizzare un'immagine Docker. Docker permette di creare delle istanze, chiamate container, separate dal sistema operativo che le ospita. Tali istanze sono meno pesanti rispetto ad una macchina virtuale, questo grazie alla condivisione delle risorse del kernel tra i vari container. Oltre a ciò sono, grazie alla condivisione delle risorse a basso livello, molto veloci da avviare e da arrestare e questo rende Docker la tecnologia perfetta per questo genere di problematiche. L'idea è quella di creare un immagine docker che contiene NodeJs e Nginx insieme alla sua configurazione ed iniettare all'interno di essa, al momento della creazione dell'istanza, il codice sorgente configurato mediante i dati inviati dall'utente.

Per quanto riguarda invece la compilazione e la firma degli applicativi mobili è necessario configurare in modo particolare la macchina che eseguirà questo tipo di task. Di fatto bisogna che sia una versione di macOs visto che Apple non fornisce ad ora una versione di XCode per sistemi linux o Windows. Per quanto riguarda Android è necessario installare la command line interface del Gradle. A questo punto sarà necessario preparare uno script in bash che eseguirà una volta chiamato, tutte le procedure di compilazione e di firma degli applicativi. Tale procedura dovrà renderli disponibili al download inserendoli in una apposita directory dove un applicativo gli servirà e potranno essere scaricati e distribuiti attraverso gli store.

Questi sono alcuni dei punti implementativi da svolgere per arrivare ad avere un prodotto con queste features. L'architettura progettuale risultante sarà simile a quella descritta in figura

Come si può notare vi sarà un service worker sviluppato in node che controllerà l'esecuzione dello

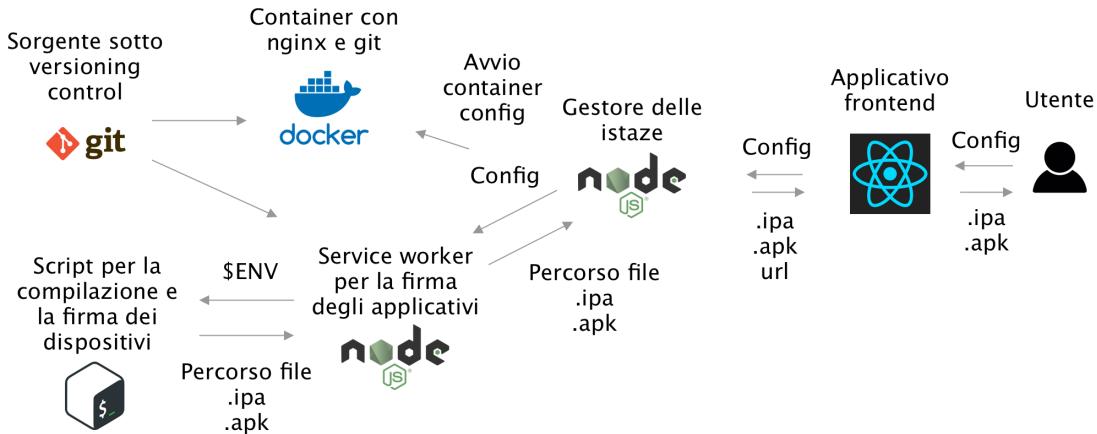


Figura 4.1: Schema del funzionamento della possibile futura del Control Manager di istanze di Open Air Museum

script bash iniettando tutte le variabili di environment come i percorsi ai certificati ,precedentemente caricati, indispensabili per la procedura di compilazione e di firma.

4.1 Esecuzione di processi in Nodejs

Per rendere possibile questo tipo di prodotto è necessario che l'applicativo in Javascript sia in grado di pilotare uno script bash. Per fare questo è possibile utilizzare il modulo ChildProcess[4] che espone una serie di metodi di esecuzione possibili. Tra quelle disponibili vi è l'esecuzione attraverso spawn che esegue lo script indicato all'interno di un processo generato come figlio di quello principale. Il controllo dello stato di questo processo avviene attraverso gli stream di output che questo processo utilizza come standard output. Per esempio nel caso di un programma in c++ che stampa a console la somma tra due numeri passati come parametri, lo standard output è descritto dalla libreria iostream il quale compito è fare da ponte tra l'esecuzione del processo e la shell che lo sta eseguendo. Node.js tramite l'interfaccia a stream è in grado di agganciarsi a questo output e ridirigerlo in modo asincrono a Javascript che sarà in grado di mettere in coda l'esecuzione della funzione associata all'evento 'data' e di usufruire in modo asincrono e non bloccante dell'output del programma in c++ sopracitato.

Come si può vedere dalla figura 4.2 l'immagine a destra è il codice in c++ che verifica l'esistenza di soli due argomenti e nel caso gli converte in interi ed esegue la somma stampandola mediante cout che non fa altro che convertire tale valore in stringa inviandolo poi allo standard output. Passando all'immagine di destra si può vedere come mediante il modulo child-process e il metodo spawn si crea un processo utilizzando il risultato della compilazione del precedente codice c++ passando come parametri i due numeri che si desiderano sommare. Successivamente Javascript, in modo asincrono, si mette in ascolto dello standard output del processo catturandone i valori. Una volta che il processo ha completato l'esecuzione viene lanciato l'evento 'close' passando come parametro nel callback il codice di uscita del processo. Nel caso in cui esso sia uguale a zero il processo è stato eseguito correttamente e si può mostrare i risultati raccolti dall'esecuzione, altrimenti si mostrano gli errori. Da notare come nel caso in cui i numeri passati al processo in c++ sono diversi da sue il return del main torna 1 e sta ad indicare che vi è stato un errore e quindi vengono mostrati invece tutti gli output raccolti da stderr che è lo standard output per gli errori.

Come si può intuire questa soluzione non è molto scalabile dato che utilizzare lo standard output

```

#include <iostream>

//usa lo spazio dei nomi standard
using namespace std;

int main(int argc, char*argv[]) {
    int numberArg = argc - 1;
    if (numberArg != 2) {
        cerr<<"Fornire due numeri"<<endl;
        return 1;
    } else {
        int first = stoi(argv[1]);
        int second = stoi(argv[2]);
        cout<<first + second<<endl;
        return 0;
    }
}

const { spawn } = require('child_process');
const ls = spawn('./a.out', ['2', '10']);

var sum = 0;
var errors = ''

ls.stdout.on('data', (data) => {
    sum = data.toString()
});

ls.stderr.on('data', (data) => {
    errors = `${errors}${data}`
});

ls.on('close', (code) => {
    if (code === 0){
        console.log(sum)
    } else {
        console.log(errors)
    }
});

```

Figura 4.2: Esempio di esecuzione codice c++ da Javascript

per ricevere il risultato di un'operazione potrebbe essere difficile nel caso di logiche più complesse. La soluzione potrebbe essere quella di utilizzare un ponte di comunicazione tra processi come Redis Pub/Sub [15] in modo da poter comunicare attivamente in modo asincrono da entrambi i lati. Non descriverò nel dettaglio l'implementazione di questa soluzione perché esula dallo scopo della tesi, ma offre un punto di partenza per lo sviluppo di una libreria per NodeJs per la compilazione automatica di applicativi nativi (ios e android).

5 Analisi e Conclusioni

Per discutere di ciò che il refactor ha portato dividerò le conclusioni in due parti prendendo in considerazione prima il lato tecnologico e poi il lato umano. Ponendo che il progetto è in fase di terminazione non vi è la possibilità di accedere a dati definitivi per cui quello che riporterò di seguito potrebbero essere soggetto a variazioni future.

Previa analisi è necessario fornire un contesto per qualificare al meglio i dati che andrò a descrivere, il primo fattore sono il numero di sviluppatori dedicati a questo progetto. La composizione dell'organico è il seguente, un grafico, due programmati ed un project manager il quale compito era gestire la qualità del prodotto e far rispettare le tempistiche promesse al cliente.

Prendendo in considerazione il lato umano è subito risultato lampante come, seppur la divisione dei compiti dello sviluppo era fatta per ambienti e cioè lato client e lato server, la facilità con cui il team ha potuto interscambiarsi è stata molto evidente. Si è mostrato come, seppur la codebase fosse nuova e solo la logica generale fosse conosciuta, possedere una 'lingua franca' in cui esprimersi è stato fondamentale per velocizzare tutte quelle operazioni che richiedevano di muoversi in un differente. Un altro esempio di questo è la velocità con cui si sono eseguite le code review settimanali. Possedere un dialetto comune permette di rendere più veloce la comprensione delle features aggiunte dal collega e trovare eventuali bug o problemi in modo più immediato. Ecco che grazie a questo tipo di approccio si è potuto delineare una tecnica di divisione dei compiti non più per zona di lavoro (server o client) ma per funzionalità. Si è notato che la velocità di sviluppo aumentava se, isolata una feature, chi aveva il compito di implementarla creava sia il lato client che il lato server. Questo ha evitato anche lo sviluppo di rest api ridondanti con dati mai utilizzati a front end, aumentando l'efficienza del software, evitando di dover eseguire lo step intermedio di accordarsi, per ogni nuova funzionalità che si andava ad inserire, sulla forma delle api. Ora di fatto chi sviluppa la funzionalità lato utente ha anche il compito di sviluppare le api che dovranno essere consumate per fornire tale funzionalità creando un prodotto meno frammentato e più integrato.

Un altro punto focale è stata la possibilità di condividere conoscenze in modo più diretto. Molte delle tecnologie e dei pattern di programmazione utilizzati principalmente a lato client hanno contaminato il lato server e vice versa. Librerie come Redux sono agnostiche e possono adattarsi facilmente al contesto server e pattern come la programmazione funzionale possono essere distribuite senza difficoltà nell'intera codebase del progetto evitando "l'effetto palude". Tale effetto è quando durante lo sviluppo una parte di codice non mantenuta diventa ingestibile a causa dell'incapacità dei colleghi di fare refactor su quella parte perché scritta con un certo linguaggio non conosciuto da tutti i componenti del team, o con un pattern conosciuto solamente dal suo scrittore. Con uno stack full Javascript come quello proposto nella soluzione descritta in questa tesi si può risolvere senza overhead questo genere di problemi forzando un vero standard in tutta la code base superando i limiti tecnologici imposti da un cambio di linguaggio tra due aree.

Ulteriore appunto che è emerso durante lo sviluppo è stata la possibilità di condividere alcune logiche tra client e server, tagliando notevolmente i tempi di sviluppo. Tutte le logiche per il recupero dei dati sono state racchiuse in una piccola libreria che è stata condivisa tra gli applicativi mobili e il lato admin. La scrittura di un solo pezzo di codice destinato ad uno scopo preciso e portabile su più piattaforme ha inoltre evitato bug derivati da modifiche alle api dato che per più volte è bastato aggiornare internamente la libreria senza dover cambiare null'altro.

Un ultimo punto che voglio includere che non è legato ad una situazione generica ma solamente al mio ambiente lavorativo; essendo Farnedi ICT in primis un'azienda che fa supporto tecnico ad aziende e p.a. parte dei suoi dipendenti non hanno una conoscenza diretta di sviluppo software. Javascript è risultato essere molto chiaro anche a chi è al di fuori del campo, permettendo anche al lato manageriale di entrare, in minima parte, nello sviluppo. Tale considerazione potrebbe essere legata alla mia sola realtà aziendale ma è stata di grande importanza per la produzione di funzionalità qualitativamente più in linea con le richieste del direttivo.

Dal punto di vista tecnologico non vi sono stati cambiamenti importanti, le funzionalità applicative trattandosi di un refactor hanno seguito la linea del prodotto iniziale. Però la dismissione di un applicativo come filemaker per il recupero dei dati è stata la fonte di un incremento sostanziale nella mantenibilità e nella rapidità dell'applicativo in tutte le sue parti. La scelta di passare da un applicativo sviluppato con un framework proprietario ad un tool open source come React ha portato in generale ad ottenere un ambiente di sviluppo più moderno e flessibile. Il passaggio ad una SPA[21] ha migliorato non solo l'esperienza per chi ha sviluppato il software ma la qualità del prodotto in se.

Per quanto riguarda i costi di refactor degli applicativi è risultato in un consumo di ore uomo differente a seconda dell'ambito del refactor. Nel caso dell'applicativo admin lo sviluppo è stato più costoso in ore lavoro rispetto alla versione precedente in Filemaker ma, ciò ha tolto ulteriore lavoro necessario per adattare i dati provenienti da filemaker alla struttura del database. Per quanto riguarda invece lo sviluppo degli applicativi mobili React-native ha portato un enorme vantaggio riducendo alle versioni diverse, infatti lo sviluppo necessario per il completamento di entrambe le versioni è stato un terzo rispetto al precedente.

5.1 Confronto tra le due versioni

Confrontando gli stack delle due applicazioni in termini di tecnologie utilizzate possiamo vedere i motivi per cui le scelte fatte migliorano il software e sotto quali aspetti.

Grazie all'utilizzo del medesimo framework lato mobile e lato web è stato possibile condividere gran parte del codice e delle logiche di interazione con le api lato server. Un'altra zona di condivisione è l'implementazione di Redux. Data la sua agnosticità sulla piattaforma utilizzata questa libreria è un perfetto esempio di come l'utilizzo di uno stack basato interamente su Javascript porti ad un riutilizzo del codice in modo importante ma soprattutto in modi impraticabili precedentemente. In particolare tra l'applicativo admin sviluppato come webapp per il browser e gli applicativi nativi ho condiviso la totalità dell'implementazione gestendo lo store allo stesso modo. Le differenze principali sono sulla libreria che permette di interfacciarsi con le api lato server. Seppur molti metodi sono condivisi, quelli relativi alla mutazione dei dati sono riservati ai soli utenti autenticati e cioè Admin. Tale peculiarità però non influenza nella possibilità di riutilizzare parte della libreria che implementa sia i metodi lato amministratore che quelli lato utente senza privilegi.

Un ulteriore punto è che rispetto alla versione precedente vi è una somiglianza molto forte tra quello che è l'applicativo web e l'applicativo nativo, le due applicazioni non sono compatibili ma condividono gran parte delle logiche e dei pattern essendo di fatto lo stesso framework. Ciò permette di avere un livello di complessità all'approccio a questo progetto più bassa della versione precedente. Questo porta a tempi più corti per l'apprendimento dei pattern e delle logiche e di conseguenza risulta in una miglior produttività.

La scelta dell'utilizzo di un database a file con la possibilità di configurare e passare ad un database "classico" si è rivelata molto importante per la scalabilità dell'applicativo. SQLite risulta più comodo in fase di deploy di contro, non ha le prestazioni di un database "classico" e la possibilità di configurare rapidamente la tecnologia da utilizzare è un plus importante. Questo si ottiene grazie all'impiego di

un ORM in grado di fornire questa features come Sequelize. Oltre a questo permette di implementare in modo veloce migrazioni e seeder restando sempre agnostico sulla tecnologia a database. La versione precedente conteneva una pesante assunzione sull'utilizzo di un database MySql e ciò abbassava sostanzialmente la flessibilità e il riutilizzo del prodotto.

Un'altra differenza rispetto alla precedente è l'esistenza dell'applicativo admin in react che dovrà essere servito dall'applicativo server. Per fare questo è necessario che tutte le richieste non inviate direttamente a /api, vengano tutte reindirizzate all'applicativo in modo che la SPA utilizzi il router interno a React per mostrare i contenuti corrispondenti all'url della richiesta.

Riguardo al lato "admin", cioè quello che va a sostituire Filemaker nella raccolta dati, è stato sviluppato con React. La scelta di sviluppare una SPA[21] per questo compito è stata dettata da un bisogno di rendere questa operazione iterabile e ripetibile. Nel caso in cui il cliente volesse cambiare dei testi o correggere delle traduzioni può farlo in autonomia, senza dover passare da una figura che traduca le modifiche richieste aggiungendole a database manualmente. Questo processo inoltre astrae ulteriormente la struttura del database durante il processo di inserimento dei dati offrendo quindi flessibilità su modifiche future alla struttura. Un altro punto a favore della scelta di una SPA

5.2 Punti critici

L'utilizzo di una sola tecnologia in più zone distinte dello stack porta con sé numerosi vantaggi che ho descritto precedentemente. Va detto però che se nello specifico caso applicativo preso da me come esempio le tecnologie Javascript esistenti sul mercato andavano a completare in modo ottimale le necessità applicative, questo può non essere vero per tutti i casi. In linea generale è sempre meglio scegliere la tecnologia migliore per la funzionalità che si vuole svolgere. Un esempio è la capacità computazionale limitata di Javascript che a confronto con Python non permette di svolgere in modo efficiente operazioni su matrici limitandone l'uso per quanto riguarda l'ambito del machine learning. In questo caso Javascript potrebbe far eseguire le operazioni di calcolo sfruttando l'ambiente fornito da Node e in particolare eseguire dei processi stando in ascolto di essi, in attesa di risultati sfruttando l'asicronicità dell'I/O fornita dal framework. Tali processi, disegnati per eseguire micro task di computazione matematica, potrebbero essere sviluppati in c o c++ in modo da essere il più ottimizzati possibile per il compito. Tale approccio però offre il fianco ad una serie di problematiche legate alla mantenibilità della parte in c++ e alla dipendenza non diretta del software Javascript a tale libreria. Per avere questo tipo di dipendenza è necessario crearla attraverso una struttura software per interrompere l'esecuzione o mostrare degli errori all'avvio in caso tali dipendenze non siano esplicitate.

La scelta di utilizzare un database a file per conservare i dati sui vari punti rende problematico scalare il prodotto. Infatti, nel caso si voglia far pilotare questo applicativo da un loadbalancer[11] in grado di animare istanze a seconda del carico, ogni istanza non sarà in grado di condividere dati tra di essa creando un grave problema di consistenza. Per questa ragione la scelta di utilizzare questo tipo di tecnologia è fallimentare in quest'ottica. Una soluzione per dare la possibilità di scalare all'applicativo è quella di sfruttare la flessibilità di Sequelize[20] che permette di mantenere la stessa struttura dei modelli ma di cambiare la tecnologia del database. E' grazie a questa astrazione software che si è in grado di adattare il nuovo applicativo ad altri contesti di utilizzo.

Bibliografia

- [1] OAuth2.0 bearer token. <https://oauth.net/2/bearer-tokens/>.
- [2] Android. <https://it.wikipedia.org/wiki/Android>.
- [3] Asynchronous i/o - wikipedia. [https://en.wikipedia.org/wiki/Asynchronous-I/O](https://en.wikipedia.org/wiki/Asynchronous_I/O).
- [4] Child process - node.js. <https://nodejs.org/api/child-process.html>.
- [5] Chrome v8 - wikipedia. <https://en.wikipedia.org/wiki/Chrome-V8>.
- [6] Farnedi ict. <https://www.farnedi.it/>.
- [7] Filemaker - wikipedia. <https://en.wikipedia.org/wiki/FileMaker>.
- [8] ios. <https://en.wikipedia.org/wiki/IOS>.
- [9] Javasctip - wikipedia. <https://it.wikipedia.org/wiki/JavaScript>.
- [10] Json web token - wikipedia. <https://en.wikipedia.org/wiki/JSON-Web-Token>.
- [11] Load balancing - wikipedia. <https://it.wikipedia.org/wiki/Load-balancing>.
- [12] Node.js. <https://nodejs.org/it/>.
- [13] Overview of blocking vs non-blocking - node.js. <https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/>.
- [14] Php. <https://secure.php.net/>.
- [15] Pub/sub - redis. <https://redis.io/topics/pubsub>.
- [16] Pure function - wikipedia. <https://en.wikipedia.org/wiki/Pure-function>.
- [17] React. <https://reactjs.org/>.
- [18] Reactnative. <https://facebook.github.io/react-native/>.
- [19] Separation of concerns - wikipedia. <https://en.wikipedia.org/wiki/Separation-of-concerns>.
- [20] Sequelize. <https://sequelize.readthedocs.io/en/v3/>.
- [21] Single page application - wikipedia. <https://it.wikipedia.org/wiki/Single-page-application>.
- [22] Swift.org. <https://swift.org/about/swiftorg-and-open-source>.