UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Informatica

Elaborato finale

Porting di un applicativo multiplatform

Supervisore
Maurizio Marchese

Laureando
Jacopo Martinelli

Anno accademico 2018/2019

Ringraziamenti

...thanks to...

Indice

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

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 obbiettivo 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[7], azienda informatica di Cesena. Consiste di un'applicazione mobile per dispositivi iOS[12] , di una seconda applicazione per dispositivi Android[2] e di un software in Filemaker[8] per il caricamento dei contenuti. In particolare io mi sono occupato dello sviluppo dell'applicazione mobile per iOS[12] in Swift 4[27] e dell'applicativo lato server in NodeJs[17].

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.

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[14] 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.

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[26] sviluppata in React[22]. Le applicazioni mobile sono state unificate utilizzando React-Native[23] 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.

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.

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 Javacript. 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.

La nuova versione non è ancora sviluppato nella sua totalità, quindi non è ancora disponibile all'accesso. Tutte le immagini presenti in questa tesi sono state create utilizzando dati di test e hanno solo scopo espositivo.

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[11], 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

La modalità *Esplora* citata nella sezione precedente è stata implementata sfruttando la tecnologia iBeacon basata su Bluetooth 4.0[4]. Essa 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.

1.2.1 UUID, major e minor

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 permettono di identificare i Beacon singolarmente. Per questo motivo, un gruppo di Beacon possiede il medesimo UUID ma differisce per major e minor. Questo tipo di stratagemma è utilizzato per identificare una Regione (Region) che viene utilizzata come area di ricerca.

1.2.2 Region

Una *Regione* può essere definita mediante i parametri identificativi dei Beacon e permette di eseguire la ricerca o *Ranging* solo sui Beacon che rispettano i parametri utilizzati durante la definizione della stessa. In particolare, creando una regione con uno specifico UUID, la procedura di ricerca rileverà soltanto i dispositivi con la stessa stringa alfanumerica. Questo è vero anche per major e minor. L'utilizzo di questi tre parametri permette quindi di definire molteplici regioni diverse.

1.2.3 Ranging e Proximity

Esistono due diversi sistemi di approccio alla ricerca in una specifica regione. Il *Ranging* fornisce un interfaccia per ottenere una lista ordinata di Beacon che si trovano nelle vicinanze del dispositivo. La lista è ordinata per distanza dallo smartphone: il Beacon più lontano si troverà nell'ultima posizione e quello più vicino nella prima. La ricerca tramite Proximity, invece, non fornisce informazioni sui Beacon specifici nelle vicinanze ma è utilizzata soltanto per identificare o meno l'ingresso in una regione.

Open Air museum utilizza solamente il primo parametro per identificare la regione di ricerca, quindi tutti i dispositivi Beacon sono impostati per utilizzare quello specifico UUID. Mentre per il protocollo di ricerca utilizza il Ranging perché permette un controllo più preciso sui dispositivi nelle vicinanze.

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[15]. 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[19] o Java[13]. L'implementazione di Nodejs di un I/O non bloccante[18] 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[10]. 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, perchè non vi è una correlazione diretta fra i due. Le API sono state disegnate per simulare questa correlazione, e ciò, 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[15], 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, cosi 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].

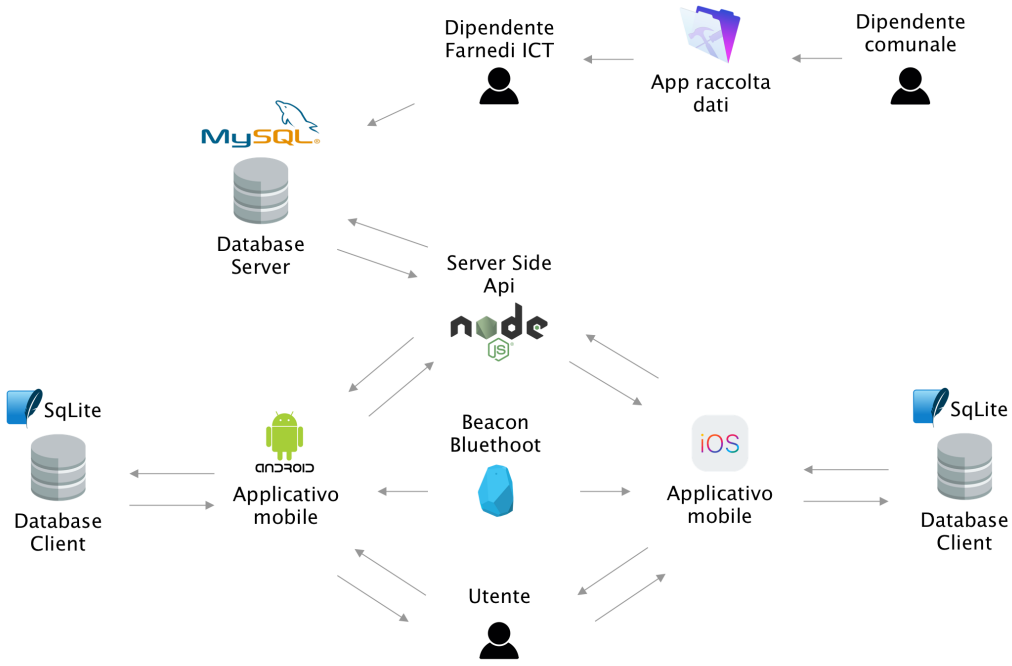

Figura 1.1: Schema logico dello stack di Open Air Museum

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.

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.

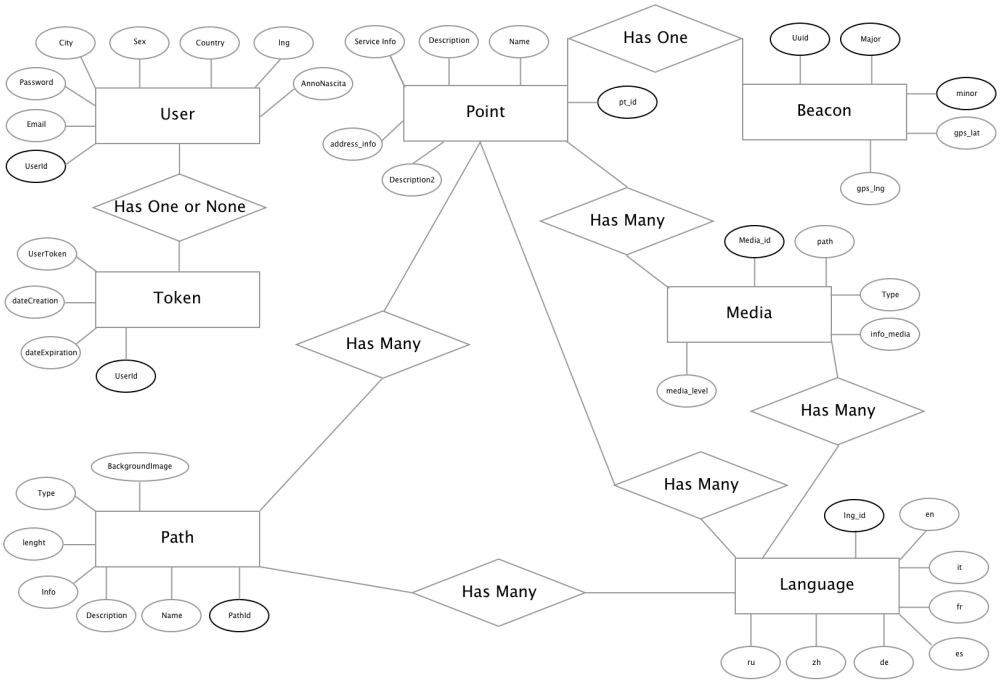

Figura 1.2: Schema ER vecchia versione server

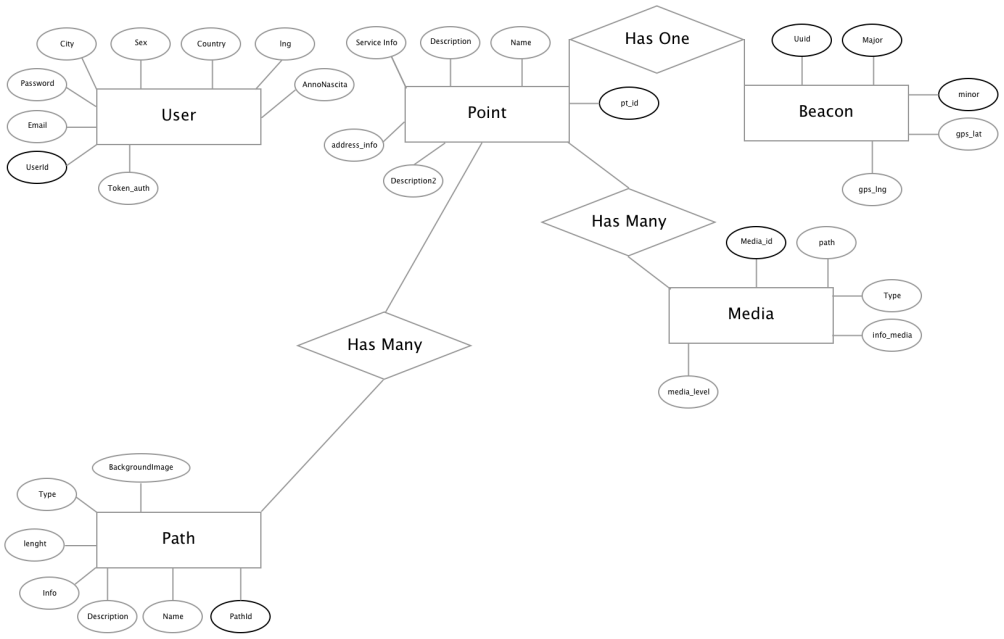

Figura 1.3: Schema ER vecchia versione smartphone

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 per arricchire l'interazione con le pagine web. Si tratta di un linguaggio interpretato e tipizzato dinamicamente, possiede un Garbage Collector[9] e si appoggia, per essere analizzato ed eseguito, ad un motore che solitamente risiede nel browser.

Uno dei motori più performanti è quello utilizzato nel web browser Chrome ed è conosciuto con il nome di V8[6]. La particolarità di tale engine è la sua natura Open Source che ha permesso negli anni di essere utilizzato in altri contesti al di fuori del browser. Questo tipo di diversificazione nei possibili ambienti di impiego di Javascript ha portato alla nascita di numerosi framework e tool anche lato server.

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

In alternativa alle modalità di sviluppo descritte in precedenza, vi sono molteplici soluzioni che offrono la possibilità di sviluppare per entrambi i dispositivi, iOS ed Android, utilizzando Javascript. Si differenziano per due aspetti fondamentali:

- Soluzioni Ibride
- Soluzioni Compilate

Con un approccio ibrido, l'applicativo consiste in 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 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 la 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*: chi utilizza l'applicativo deve prima indicare a quale zona è interessato per poi essere reindirizzato alla sezione relativa di 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.

Il primo approccio descritto permette di dover mantenere un solo prodotto e di servire molteplici enti, questo però limita molto la customizzazione che tali applicativi possono possedere. Al contrario, 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 richiederà probabilmente funzionalità e caratteristiche differenti.

Un ulteriore punto di riflessione riguarda 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 il cui scopo è guidare l'utente all'interno della zona di interesse. Gli applicativi che presentano questa features utilizzano come tecnologia di localizzazione il gps che per zone all'aperto, come lo possono essere città o parchi naturali, si rivela essere un'ottima soluzione. Di contro, per quanto riguarda la localizzazione indoor, come in un museo, 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 su triangolazioni satellitari per calcolare la posizione dell'utente.

2.4 Tecnologie utilizzate in Open Air Museum 1.0

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 l'inserimento di punti e percorsi che viene utilizzata da alcuni addetti comunali. Essi forniranno i media, le descrizioni e i testi tradotti per le varie lingue supportate. Tale applicativo è standalone e quindi slegato dal lato server che espone le API. Quindi 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, Nodejs è un runtime di Javascript al di fuori del browser, utilizza Chrome V8[6] come motore per eseguire Javascript e fornisce una suite di interfacce per la comunicazione

con l'hardware. L'implementazione dell'interazione con l'hardware avviene attraverso un I/O non bloccante[3]. Queste API mettono a disposizione la possibilità di utilizzare la scheda di rete, e quindi di avviare server TCP e HTTP, e di dare accesso al file system. Il tutto attraverso un interfaccia lato Javascript asincrona.

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 da questa piattaforma che inoltre gestisce l'autenticazione e la gestione degli utenti.

2.4.3 MySql e SqLite

Per la gestione dei database è stato utilizzato MySql per la parte server e SqLite per la parte mobile. Sono entrambe soluzioni relazionali ma con una sostanziale differenza: la prima utilizza un applicativo che gestisce le interazioni con il database, accettando richieste e inviando dati attraverso un interfaccia tcp; la seconda invece utilizza una gestione a file in cui 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 scrivere gli applicativi mobile è stato utilizzato Java su Android Studio e Swift su XCode, rispettivamente per l'applicativo Android e l'applicativo IOs. L'utilizzo dei linguaggi nativi ha permesso un maggior accesso a quelle che sono le funzionalità dell'hardware ed in particolare al bluetooth. 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 bluetooth 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

2.5.1 React

React[22] è una Libreria Javascript per la costruzione di interfacce grafiche a componenti e 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, e che viene poi renderizzato a seconda delle modifiche effettuate. Il secondo, invece, indica la metodologia utilizzata da React per trasferire i dati attraverso la virtualizzazione appena descritta. L'efficenza 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 semplice codice Javascript che utilizza i metodi forniti da React per interagire con il DOM Virtuale. Ciò permette di costruire la pagina html nella struttura virtuale che poi React renderizzerà in modo efficiente nel DOM del browser.

2.5.2 Redux

Redux è una libreria Javascript che permette di gestire lo stato applicativo come un unico oggetto statico, modificabile soltanto attraverso della azioni ben definite. Ogni modifica viene fatta attraverso una funzione pura[21] chiamata reducer che trasforma a seconda dell'implementazione una parte specifica dello store. La firma della funzione prende due parametri: il primo è l'oggetto che rappresenta lo stato attuale dell'applicativo, il secondo indica l'azione che si sta compiendo. L'azione dovrà sempre avere una tipologia, mentre il payload di dati è facoltativo. Ciò che ritorna il reducer corrisponde a un oggetto che diventa il nuovo stato applicativo.

Una delle specifiche di Redux è che i Reducer siano sincroni, e che quindi all'interno di essi non sia possibile eseguire richieste tramite la rete o utilizzando risorse hardware come l'accesso alla memoria. Per sopperire a questa mancanza, si utilizza un costrutto chiamato *middleware*. Il compito del middleware è quello di eseguire operazioni quando una specifica azione viene inviata al reducer. Le operazioni possono essere asincrone e l'unica specifica è che alla terminazione esse dovranno lanciare un altra azione sullo store. In questo modo si può integrare l'implementazione di Redux con delle Rest API: eseguendo un azione per avviare la richiesta ed una per inserire i dati nello store a richiesta terminata.

2.5.3 React-Native

React-Native è l'implementazione di React per 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 esso viene pilotato attraverso 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 del codice nativo, mediante una specifica dichiarazione. 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 thread 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 dà 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 un certo ordine. 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 routing delle Rest API rendendo molto veloce e predittivo lo sviluppo.

2.5.5 SqLite

SqLite è un database *"file-based"* utilizzato lato server per il salvataggio delle informazioni fornite dagli admin attraverso l'applicativo web per la raccolta dei dati. Tale scelta è legata alla flessibilità di questa tecnologia, che permette di gestire internamente all'applicativo la creazione del database, 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 interamente ma le sue funzionalità sono rimaste invariate. Le modifiche apportate infatti riguardano principalmente le tecnologie. Ora la gestione del database viene fatta attraverso un ORM chiamato Sequelize che offre una serie di funzionalità fondamentali: in primis dà la possibilità, nel caso dell'utilizzo di SqLite come tecnologia, di generare un database all'avvio dell'applicativo se al percorso indicato non esiste alcun file di SqLite. Inoltre 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 familiare a Javascript senza obbligare il programmatore a generarle tramite concatenazione di stringhe o altri metodi. Il framework utilizzato per il routing delle api è Express.

Quello che differenzia la vecchia implementazione rispetto alla nuova, è una serie di endpoint che permette di gestire e inviare la configurazione degli applicativi lato client in modo da centralizzare il più possibile la personalizzazione degli applicativi mobile e dell'applicativo admin.

3.2 Applicativo Admin

Il compito dell'applicativo admin è quello di sostituire quello in Filemaker ed espandere alcune delle sue funzionalità. Come descritto in precedenza, questo applicativo è stato sviluppato come una SPA utilizzando React come framework di sviluppo. Le feature 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 un numero minore di 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 ricevono input dall'utente, ma il primo utilizza hardware esterno per arricchire la propria raccolta informazioni. Infatti la posizione dell'utente è individuata attraverso il Gps o il Bluetooth. Tali applicativi interagiscono con il server mediante delle Rest Api che, come nella versione precedente, forniscono i dati sui punti e i percorsi. A differenza del primo l'applicativo, però, portano anche la configurazione impostata dal lato 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.

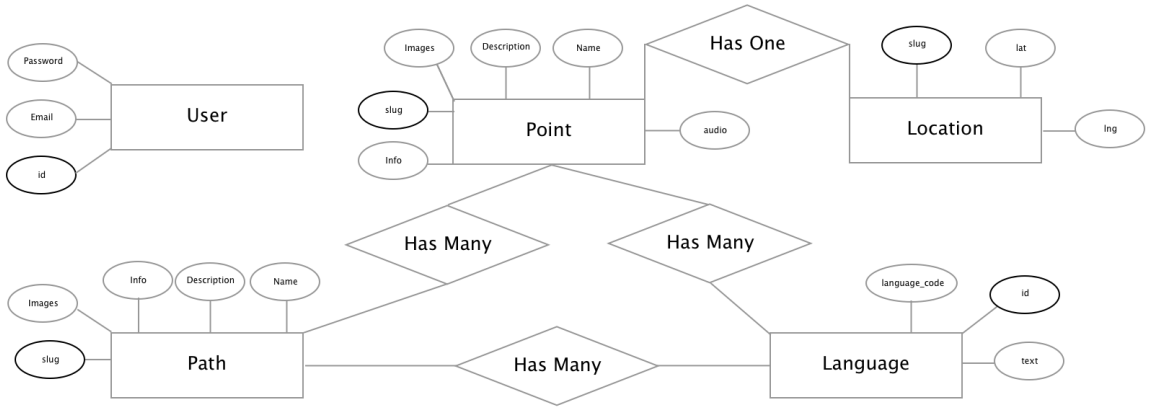

Figura 3.1: Schema ER nuova versione

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 al fine di 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 questo modo è garantita consistenza nel mostrare gli ultimi dati disponibili anche se il dispositivo risulta offline.

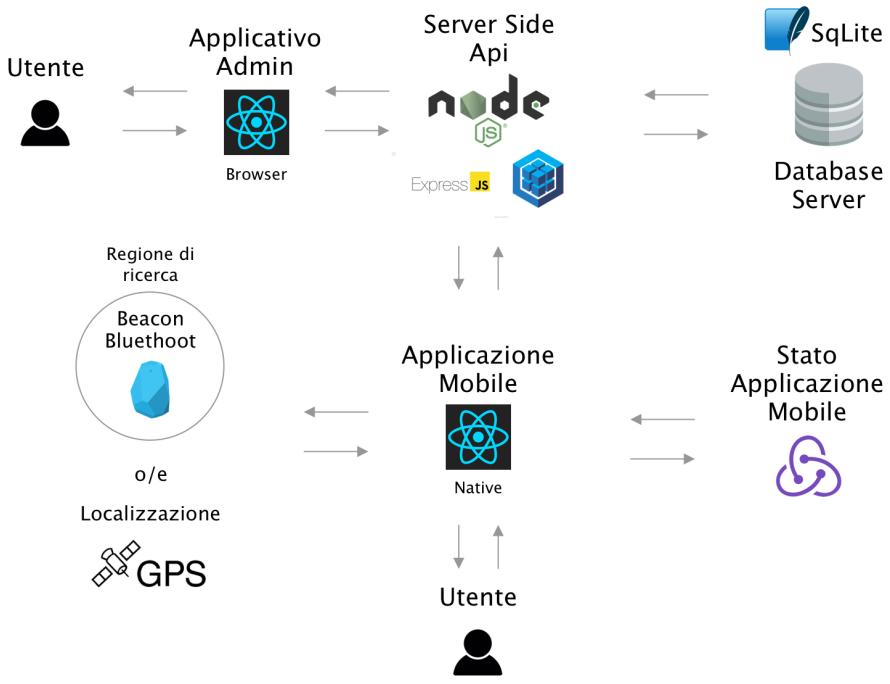

Figura 3.2: Schema logico della struttura 2.0

3.4 Porting applicativi Nativi

Ora verrà descritto nel dettaglio il porting degli applicativi nativi a React-native, il percorso implementativo e 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 *look-and-feel* il più possibile vicino a quello iniziale e apportando le possibili migliorie date dal nuovo ambiente. La prima problematica affrontata è stata quella di poter gestire la modalità Esplora da Javascript: ovvero essere in grado di controllare il bluetooth dello smartphone e monitorare l'avvicinarsi a specifiche aree marcate dai Beacon, mediante l'utilizzo di task in background. La versione nativa si avvaleva della tecnologia iBeacon sfruttata dall'SDK di Estimote. E' stato deciso di mantenerla anche nella seconda versione, vista la presenza di hardware già configurato 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 trasferire i dati.

I paragrafi seguenti descrivono 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, l'Estimote Proximity SDK utilizzando CocoaPods, un dependency manager per XCode. Inoltre, per abilitare la localizzazione in background, è necessario aggiungere tra le *capabilities* dell'applicativo nella sezione *Background Mode* la voce *Uses Bluetooth LE accessories*.

Il passo successivo consiste nel creare un file che permetta di dichiarare ed esportare i metodi di configurazione e gestione della libreria di Estimote. Per fare questo si utilizza l'interfaccia fornita da RCTBridgeModule di React-Native, in modo da rendere visibili i metodi dichiarati nel codice nativo al lato Javascript quando l'applicativo viene compilato ed eseguito. Un ulteriore punto fondamentale riguarda la comunicazione attraverso il "bridge": tale procedura è asincrona e per questo è possibile passare valori da nativo a Javascript solamente mediante callback o eventi.

Nel caso specifico, la scelta è ricaduta su di un sistema ad eventi, vista 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 regioni identificate mediante dei tag. Nel momento in cui un telefono entrerà in una zona così configurata verrà lanciato l'evento "Enter" dal reame nativo a Javascript attraverso il bridge. 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 l'evento "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. Questo per dare una sensazione di uniformità tra le due piattaforme. Per quanto riguarda l'implementazione nativa: l'uso dell'SDK avviene nello stesso modo di iOS; così come l'esportazione dei metodi attraverso il "bridge". Ciò che cambia effettivamente sono solo le configurazioni necessarie date dal diverso tipo di ambiente (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. In seguito è stato scelto di utilizzare un file Javascript che normalizzasse l'esportazione dei vari metodi.

Il risultato ottenuto da questa implementazione ha permesso di ottenere delle api paritetiche lato Javascript, nonostante le grandi differenze tra i due sistemi operativi. Questo offre un astrazione che elimina la complessità di dover gestire singolarmente due linguaggi diversi con ambienti diversi.

Quindi mediante le "Bridge Api", React-Native offre 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 feature del prodotto, dando la possibilità di scrivere solo il codice nativo necessario e sviluppando poi il resto della logica in un'ottica *"write once run anywhere"*.

Dal punto di vista grafico, inoltre, sono state apportate molte modifiche date dai 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.

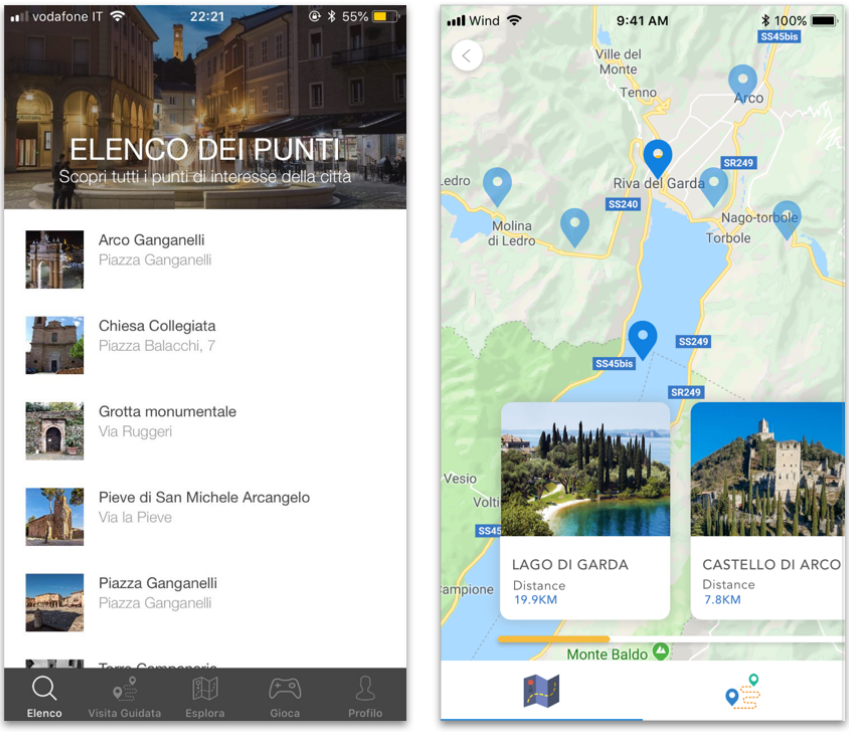

Figura 3.3: Confronto HomeScreen Open Air Museum vecchia e nuova versione

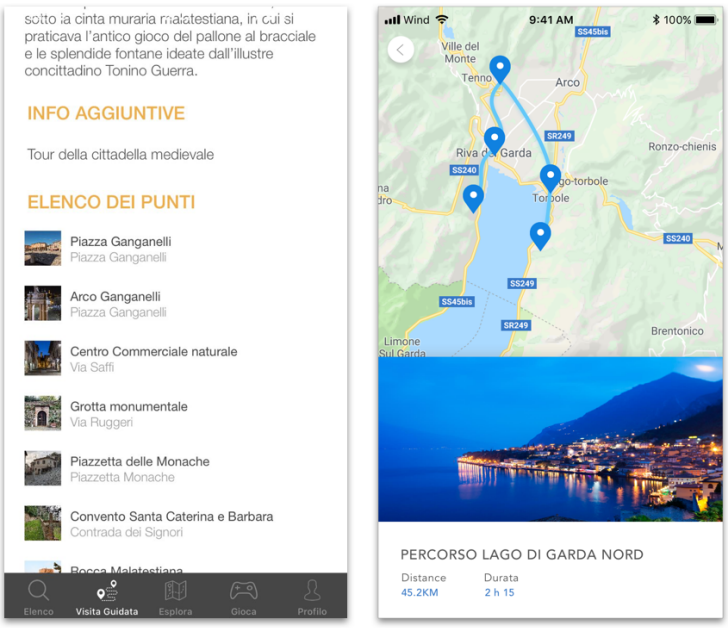

Figura 3.4: Confronto HomeScreen Open Air Museum vecchia e nuova versione

Sfruttando la potente libreria di animazioni fornita 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.

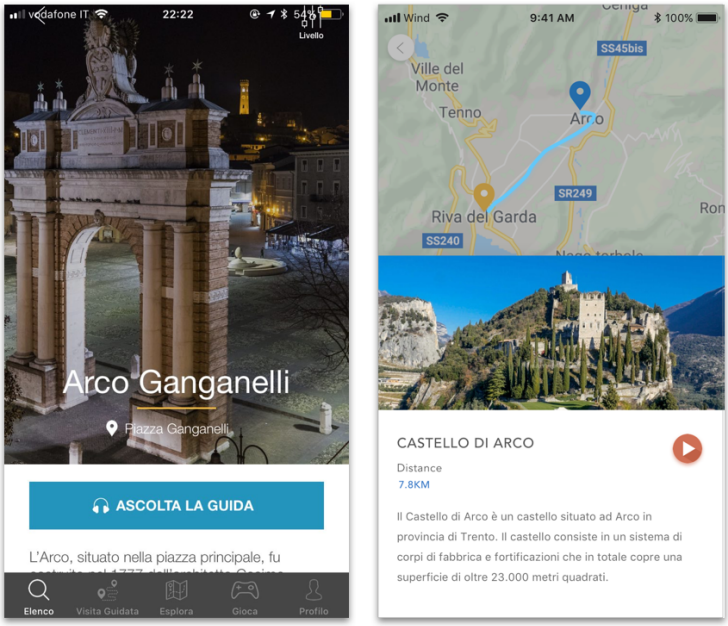

Figura 3.5: Confronto HomeScreen Open Air Museum vecchia e nuova versione

Un'ulteriore funzionalità richiesta dal porting è stata quella di poter riprodurre dei contenuti audio all'interno dell'applicativo. In particolare questi contenuti 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. L'utente era 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 vista 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*. Il package permette di configurare uno stream audio mediante una url e di controllarne la riproduzione. Le API lato javascript sono le medesime seppur a lato nativo il package utilizzi due librerie diverse: per Ios *DuoAudioStreamer* e per Android *ExoPlayer*. L'utilizzo di tale package ha accelerato lo sviluppo, mostrando il vantaggio di React-native sopra a soluzioni simili: la maturità dell'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 feature 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 feature 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.

3.4.2 Implementazione di Redux

```
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

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 e concentrandosi solamente sui componenti dediti alla visualizzazione grafica dei dati.

Come si può vedere dalla figura 3.6, lo stato applicativo consiste in un oggetto immutabile che detiene tutti i dati necessari. Quindi oltre ai vari punti e percorsi, nello store, sono contenute anche le 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 Mappa interattiva

Come si pùò vedere dalle figure 3.3, 3.4 e 3.5, il componente principale dell'interfaccia applicativa è composto da una mappa interattiva. Questo componente è messo a disposizione da una libreria chiamata *react-native-maps* che offre una serie di feature come la generazione di *marker* customizzati e la gestione delle interazioni con la mappa stessa attraverso eventi di *pan* e *drag*. E' stato scelto di utilizzare questo package, rispetto a quello sviluppato per la precedente versione, per via delle numerose funzionalità che la nuova interfaccia grafica richiede. Infatti l'implementazione proprietaria precedente è basata su Open Street Maps e non è in grado di supportare la generazione di percorsi. Mentre React-native-maps offre nativamente un canvas con cui è possibile disegnare sulla mappa tutti i poligoni necessari per la creazione di un percorso interattivo.

3.4.4 Punti critici e problematiche

Durante questo porting sono stati evidenziati una serie di punti critici implementativi: nei seguenti paragrafi sarà descritta la soluzione adottata per ogni problema specifico ed un'eventuale soluzione alternativa.

L'utilizzo di una libreria esterna, come quella per la gestione degli stream audio o quella per 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 e 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 casi questa soluzione deve essere scartata per l'elevato costo.

Un'altra problematica è data dalla coesistenza di tre linguaggi differenti nella stessa codebase. In alcuni casi React-native può aggiungere complessità invece che rimuoverla: 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 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 di Open Air Museum 2.0, 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 tenere in considerazione quando si effettua questo genere 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. L'assenza dei tempi di latenza per accedere ai dati ha migliorato molto la forma del codice, che risulta molto più conciso e pulito.

La modifica dei dati avviene attraverso mutazioni allo stato. Questo comporta, in caso di applicativi molto grandi, una crescente complessità se non si dividono in modo accorto le varie aree dello stato: è possibile dividere in modo netto un'area dall'altra usando nodi il più possibile vicini alla radice. Tale approccio, infatti, permette di ottenere una buona *separation of concerns*[24] 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 Sviluppi futuri

Uno degli aspetti principali che hanno portato al refactor di questo progetto è stato quello di renderlo più flessibile ed adattabile ad altri contesti e non solo legato al comune che lo ha commissionato. Un ulteriore passo in questa direzione consisterebbe nello sviluppo di un altro applicativo che abbia le funzionalità di orchestratore. Tale applicativo, anch'esso sviluppato mediante Nodejs sarà in grado di sfruttare a pieno l'alta configurabilità di ogni applicativo compreso nello stack di Open Air Museum 2.0, 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 verranno analizzati nei paragrafi successivi descrivono le linee guida generali per lo sviluppo.

Il primo tra questi è possedere un portale dove richiedere all'utente le configurazioni necessarie: il nome dell'applicativo; se si desidera utilizzare l'Estimote SDK o si preferisce utilizzare il GPS; le Api Key di software esterni che si vogliono prevedere all'interno dell'applicazione; le chiavi degli store per la firma degli applicativi per IOS e per Android etc.

Ottenute queste informazioni, è necessario costruire un manager che, una volta importata la configurazione, svolga le seguenti operazioni: esegua un deploy della parte server in Nodejs eil *transpiling* dell'applicativo admin in React; 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. Per questo risultano anche veloci da avviare e da arrestare. Tutto ciò 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. Il sistema operativo della macchina deve essere obbligatoriamente una versione di macOs visto che Apple non fornisce ad ora una versione di XCode per sistemi Linux o Windows. Per quanto riguarda Android, invece, basterà 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 li 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 4.1.

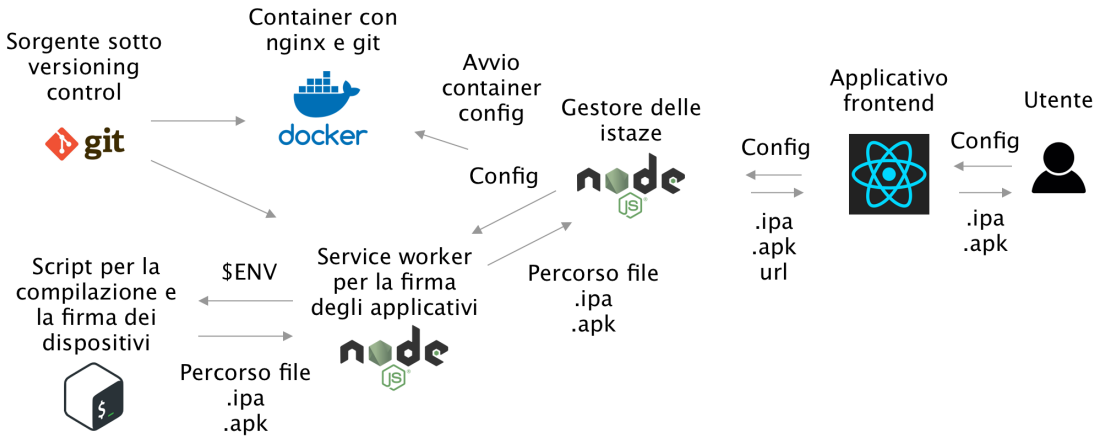

Figura 4.1: Schema del funzionamento della possibile futura del Control Manager di istanze di Open Air Museum

Come si può notare, è previsto anche un service worker sviluppato in NodeJs che controllerà l'esecuzione dello script bash iniettando tutte le variabili di environment come i percorsi ai certificati, precedentemente caricati, che sono 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[5] 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 il processo utilizza come standard output.

Si può vedere un esempio di funzionamento nelle immagini della figura 4.2, nel caso di un semplice programma in c++ che stampa a console la somma tra due numeri passati come parametri. Lo standard output è descritto dalla libreria iostream il cui 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 sinistra mostra il codice in c++ che verifica l'esistenza di soli due argomenti, e nel caso esistano, li 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, attraverso il modulo child-process e il metodo spawn, si crei 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 terminazione. 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++ siano diversi da due, il return del main ritorni 1. Ciò sta ad indicare che vi è stato un errore.

```cpp
#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;
  }
}
```

```javascript
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

Come si può intuire questa soluzione non è molto scalabile dato che utilizzare lo standard output 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 [20] per poter comunicare attivamente in modo asincrono da entrambi i lati. Non descriverò nel dettaglio l'implementazione di questa soluzione perchè 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 al progetto, sono state divise 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 potrebbe essere soggetto a variazioni future.

Prendendo in considerazione il lato umano, è subito risultato lampante come, nonostante la divisione iniziale del lavoro, i membri del team siano stati in grado di interscambiarsi facilmente tra sviluppo lato server e lato client. La presenza di una linguaggio condiviso tra client e server ha velocizzato le operazioni di integrazione tra le varie piattaforme e le code review settimanali. Infatti il vantaggio principale di utilizzare un linguaggio comune è quello di rendere più veloce la comprensione delle feature scritte da altri sviluppatori e di trovare eventuali bug o problemi in modo più immediato.

Grazie a questo tipo di approccio si è potuta 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 chi aveva il compito di implementare una particolare funzionalità, si occupava sia del lato client che del lato server. Questo ha accelerato lo sviluppo delle Rest API, evitando di dover eseguire lo step intermedio di accordarsi sulla loro struttura. Ora chi sviluppa la nuova feature lato utente ha anche il compito di sviluppare le Api lato server, 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. Pattern come la programmazione funzionale possono essere distribuiti senza difficoltà nell'intera codebase del progetto. Con uno stack full Javascript, come quello proposto, si possono risolvere senza complessità aggiuntiva problemi di *Coding conventions* forzandone uno unico in tutta la code base e superando i limiti tecnologici imposti da un cambio di linguaggio tra due aree.

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 sopratutto in modi impraticabili precedentemente. In particolare tra l'applicativo admin sviluppato come Web Application e gli applicativi nativi si è condivisa 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, cioè i dipendenti comunali. Tale peculiarità però non influisce nella possibilità di riutilizzare parte della libreria perchè essa implementa sia la sezione admin che la sezione utente senza privilegi.

Inoltre, 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 utilizzando, di fatto, lo stesso framework. Ciò permette di avere un livello di complessità all'approccio di questo progetto più bassa della versione precedente. Ciò comporta tempi più corti per l'apprendimento dei pattern e delle logiche e di conseguenza risulta in una miglior produttività per un'eventuale nuova figura da inserire nel team.

L'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 canonico. La possibilità di configurare rapidamente la tecnologia da utilizzare è un plus importante. Ciò è stato possibile grazie all'impiego di un ORM come Sequelize[25]. Esso permette anche di implementare velocemente migrazioni e seeder restando sempre agnostico sulla tecnologia da utilizzare a lato 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.

Il lato "admin", cioè quello che va a sostituire Filemaker nella raccolta dati, è stato sviluppato con React. La scelta di sviluppare una SPA[26] 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. Questa tecnica inoltre astrae ulteriormente la struttura del database durante il processo di inserimento dei dati offrendo quindi flessibilità su modifiche future ad essa.

5.2 Punti critici

L'utilizzo di una sola tecnologia in più zone distinte dello stack porta con sè i numerosi vantaggi descritti precedentemente. Và detto però, che se nel caso trattato, le tecnologie Javascript andavano a completare in modo ottimale le necessità applicative, questo può non essere sempre applicabile. In linea generale è sempre meglio scegliere la tecnologia migliore per la funzionalità che si vuole creare. 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: potrebbe eseguire i processi e restare in attesa di risultati sfruttando l'asincronicità 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 manutenibilità 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[16] in grado di animare istanze a seconda del carico, ogni istanza non sarebbe in grado di condividere dati con le altre istanze, 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[25] che permette di mantenere la stessa struttura dei modelli ma di cambiare la tecnologia del database. Grazie a questa tipo di astrazione software sarà possibile adattare il nuovo applicativo ad altri contesti di utilizzo.

Bibliografia

[1] 0auth2.0 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.

[4] Bluetooth - wikipedia. https://en.wikipedia.org/wiki/Bluetooth.

[5] Child process - node.js. https://nodejs.org/api/child-process.html.

[6] Chrome v8 - wikipedia. https://en.wikipedia.org/wiki/Chrome-V8.

[7] Farnedi ict. https://www.farnedi.it/.

[8] Filemaker - wikipedia. https://en.wikipedia.org/wiki/FileMaker.

[9] Garbage collector - wikipedia.

[10] Hypertext transfer protocol - wikipedia. https://it.wikipedia.org/wiki/Hypertext-Transfer-Protocol.

[11] ibeacon - wikipedia.

[12] ios. https://en.wikipedia.org/wiki/IOS.

[13] Java - wikipedia. https://en.wikipedia.org/wiki/Javaprogramminglanguage.

[14] Javasctip - wikipedia. https://it.wikipedia.org/wiki/JavaScript.

[15] Json web token - wikipedia. https://en.wikipedia.org/wiki/JSON-Web-Token.

[16] Load balancing - wikipedia. https://it.wikipedia.org/wiki/Load-balancing.

[17] Node.js. https://nodejs.org/it/.

[18] Overview of blocking vs non-blocking - nodejs. https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/.

[19] Php. https://secure.php.net/.

[20] Pub/sub - redis. https://redis.io/topics/pubsub.

[21] Pure function - wikipedia. https://en.wikipedia.org/wiki/Pure-function.

[22] React. https://reactjs.org/.

[23] Reactnative. https://facebook.github.io/react-native/.

[24] Separation of concerns - wikipedia. https://en.wikipedia.org/wiki/Separation-of-concerns.

[25] Sequelize. https://sequelize.readthedocs.io/en/v3/.

[26] Single page application - wikipedia. https://it.wikipedia.org/wiki/Single-page-application.

[27] Swift.org. https://swift.org/about/swiftorg-and-open-source.