POLITECNICO DI TORINO

Collegio di Ingegneria Informatica, Del Cinema e Meccatronica

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Implementazione, sviluppo ed evoluzione di Applicazioni Web basate su Firebase

Relatore

Prof. Malnati Giovanni

Candidato

Davide Ruscica

Sommario

Abstract

Lo scopo del progetto si estende su due fronti: automatizzazione dei processi ed evoluzione di un'Applicazione Web basata su *Firebase* chiamata *Blooming Harp*.

I processi automatizzati sono relativi a tutte le operazioni burocratiche di una piccola associazione no profit. Più nel dettaglio è stato studiato come automatizzare il più possibile tali processi sviluppando da zero un'Applicazione Web basata su Firebase (*Realtime Database*). Il prodotto ha dovuto inoltre seguire la business logic del sistema già esistente in modo da garantire tempi di training bassi e, al tempo stesso, testare l'elasticità dei sistemi basati su Firebase.

Il prodotto finale è stato successivamente soggetto ad una migrazione da un'implementazione tramite *Realtime Database* ad una tramite *Cloud Firestore*. Questo ha permesso sia di testare la complessità di tale operazione, misurata in cambiamenti lato Client, che di confrontare le due tipologie di database.

Introduzione

Il progetto nasce dalle esigenze di una piccola realtà associativa italiana, la cui crescita nel corso degli anni ha portato alla luce necessità precedentemente inesistenti e/o trascurabili. Il progetto è basato dunque sulla vita e le necessità dell'associazione *Bardi di Sadirvan – Loto Fiorito*.

L'associazione

Per una maggiore comprensione del progetto si riportano di seguito le principali caratteristiche dell'associazione *Bardi di Sadirvan – Loto Fiorito*.

In generale l'associazione porta avanti progetti creati dai soci, per lo più di stampo sociale, di varia natura ed entità. Nello specifico esistono dei gruppi interni addetti allo sviluppo di eventi più categorizzati.

La gerarchia è molto semplice: il *Consiglio direttivo* è un organo i cui membri vengono eletti dai *soci* stessi, tale organo è il responsabile dell'accettazione e dell'espulsione dei *soci* e della validazione di progetti associativi; altre cariche variano in base allo statuto e alle esigenze, queste possono nascere e decadere nel corso del tempo.

Al momento della stesura di questo progetto gli eventi nella quale l'associazione investe più risorse sono eventi *LARP* (o *GRV* in italiano), quindi si porrà particolare attenzione alle necessità derivanti da questa tipologia di eventi.

Le necessità

Nel corso degli anni la gestione dei soci è diventata via via più complessa, questo è dovuto dal fatto che il numero è cresciuto sensibilmente, passando da una decina a poche centinaia e continuando a crescere stabilmente, e alle necessità legali che sono state introdotte. Inizialmente infatti il rallentamento e la difficoltà di gestione delle operazioni di segreteria erano notevoli.

L'associazione necessita di un metodo automatico per: gestire i processi d'*iscrizione* all'associazione, gestire le richieste di *rinnovo* da parte dei soci, registrare i dati dei singoli soci senza *duplicazioni* e *disallineamenti*, gestire le richieste di partecipazioni ai singoli *eventi*.

L'evoluzione dell'idea

Le necessità descritte sono state inizialmente soddisfatte manualmente attraverso l'utilizzo di e-mail, fogli di calcolo, messaggi istantanei e accordi verbali, questi metodi si sono rivelati quasi immediatamente inadatti allo scopo e si è cercata un'alternativa economica e facilmente utilizzabile.

Per ammortizzare i problemi emersi è stato sviluppato un sistema informatico utilizzando le funzionalità offerte da *Google Spreadsheet*, *Google Form* e *Google Docs* (più in generale *Google Drive*), questo sistema ha facilitato enormemente il lavoro di gestione ma presenta problemi di sicurezza, affidabilità e scalabilità.

Obiettivi del progetto

La prima parte di questa Tesi consiste nel creare un sistema stabile, scalabile ed affidabile in grado di gestire tutte le esigenze associative.

La seconda parte di questa Tesi è di studiare tutti i meccanismi di migrazione tra sistemi differenti e sistemi affini: primo tra tutti migrare dal sistema attualmente in uso ad un'Applicazione Web, portando alla luce le difficoltà implementative derivanti dal replicare funzionalità già esistenti tramite una tecnologia differente; in secondo luogo, non per importanza, effettuare una migrazione parziale all'interno della stessa tecnologia, cambiando il dataModel di riferimento.

Nello specifico l'Applicazione Web è stata sviluppata tramite AngularJS in congiunzione con Firebase per semplificare la gestione back-end e creare un'applicazione scalabile.

Inizialmente ci si è appoggiati al *Realtime Database* per poi migrare al *Cloud Firestore*, queste sono le due tipologie di database, entrambe NoSQL, messe a disposizione da Firebase per la gestione dei dati.

1 Analisi dei requisiti

Le basi del progetto nascono grazie all'esperienza accumulata dall'intero gruppo organizzativo, questa ha permesso di individuare velocemente le necessità più grandi in modo da ottenere un punto di partenza e degli obiettivi ben definiti. Successivamente sono stati selezionati alcuni elementi del gruppo organizzativo con cui portare avanti la fase di progettazione, in quanto gran parte del sistema dovrà essere utilizzato da questi stessi (insieme ad altri) sempre ponendo particolare attenzione al lato utente.

Tutti i requisiti sono stati redatti sulla base del sistema attualmente in uso con l'aggiunta di requisiti di contorno per garantire un sistema stabile e facilmente utilizzabile.

I Requisiti sono stati raggruppati in ampie categorie (definite nella Tabella 2.1) in modo da migliorare la leggibilità di questo documento. I requisiti che appartengono ad una determinata categoria hanno caratteristiche comuni anche se possono avere dipendenze esterne.

ID	Descrizione
1	Gestione account
2	Gestione soci
3	Gestione eventi
4	Gestione transazioni
5	Altro

Tabella 1.1 - Categorie

I ruoli del sistema sono rappresentati tramite in valore numerico, tale valore influenza ciò che l'utente può fare e a quali dati ha accesso. Essendo rappresentati da un intero, i ruoli sono consequenziali: un utente avente ruolo N possiede tutti i diritti dei ruoli precedenti.

Titolo	Descrizione	Ruolo
Chiunque	Funzionalità pubblica.	0
Utente	Chiunque sia in possesso di un account. Diventa socio nel momento in cui completa tutti i passi per il tesseramento.	10
Organizzatore	Un utente a cui è stato affidato il compito di organizzare uno o più eventi.	15
Gestore	Un utente a cui è stato affidato il compito di gestire il sistema.	20
Direttivo	Un membro generico del *Consiglio direttivo* dell'associazione o un addetto alla segreteria.	30
Amministratore	Amministratore di sistema.	40

Tabella 1.2 – Ruoli

1.1 Gestione account

Tutto ciò che concerne la gestione di un account all'interno del sistema, un individuo qualunque deve poter effettuare l'accesso *prima* di procedere con l'iscrizione all'associazione, questo implica che il sistema è accessibile da chiunque abbia un indirizzo e-mail valido.

ID	Descrizione	Ruolo	Dipendenze
F1.1	Registrazione account	0	-
F1.2	Accesso al servizio	10	F1.1
F1.3	Uscita dal servizio	10	F1.2
F1.4	Conferma e-mail	10	F1.2
F1.5	Inserimento dei dati anagrafici	10	F1.2
F1.6	Modifica dei dati anagrafici	10	F1.5
F1.7	Effettuare una richiesta d'iscrizione	10	F1.2
F1.8	Recupero password	10	F1.1
NF1.1	Privacy dei dati	-	F1.5

Tabella 1.3 – Requisiti di gestione degli account

1.2 Gestione soci

La maggior parte dei servizi sono concettualmente disponibili solo ai *soci*, diventare socio però è un processo lungo e burocratico, pertanto gli utenti hanno dei limiti imposti solo sulla richiesta di tesseramento e sull'inserimento dei dati anagrafici.

ID	Descrizione	Diritti	Dipendenze
F2.1	Accettare una richiesta d'iscrizione	30	F1.7
F2.2	Generazione del modulo d'iscrizione	10	F1.5
F2.3	Espulsione di un socio	30	-
F2.4	Modifica della propria tessera[1]	10	F1.7
F2.5	Modifica ruolo di un utente	40	-
F2.6	Censimento di un'assicurazione	30	-
F2.7	Modifica di un'assicurazione	30	F2.7
F2.8	Rimozione di un'assicurazione	30	F2.6
F2.9	Visualizzazione dei dettagli di un utente	15	F1.1
F2.10	Visualizzazione dell'elenco degli utenti	15	-

Tabella 1.4 – Requisiti di gestione dei soci

1.3 Gestione eventi

Gli eventi costituiscono il centro dell'attività del sistema, gli utenti devono principalmente poter visualizzare ed iscriversi agli eventi pubblicati.

ID	Descrizione	Diritti	Dipendenze
F3.1	Creazione di un evento	20	-
F3.2	Modifica un evento	20	F3.1
F3.3	Acquistare un biglietto	20	F3.1
F3.4	Visualizzazione della lista degli iscritti ad un evento	20	F3.1

Tabella 1.5 – Requisiti di gestione degli eventi

[1] Questa operazione viene fatta dal sistema in automatico, tuttavia un membro del *Consiglio direttivo* deve poter forzare lo stato di una tessera ad un valore specifico.

1.4 Gestione transazioni

I pagamenti rappresentano le transazioni di denaro, pertanto questi devono essere il più possibile tracciabili ma al tempo stesso deve essere possibile per il *Consiglio direttivo* spostare somme di denaro da un *contenitore* ad un altro.

Gli utenti d'altro canto devono interagire con questo sistema in modo trasparente attraverso un carrello.

ID	Descrizione	Diritti	Dipendenze
F4.1	Aggiunta di una tessera al carrello	U	F1.4
F4.2	Aggiunta di un biglietto al carrello	U	F1.4
F4.3	Svuotare il carrello	U	F4.1, F4.2
F4.4	Effettuare il Checkout di un carrello	C	F4.3
F4.5	Censimento di un'entità autorizzata al raccoglimento dei pagamenti	U	-
F4.6	Visualizzazione dell'elenco delle transazioni	G	-
F4.7	Assegnazione di una transazione	O	F4.6
F4.1	Creazione di uno sconto	D	-
F4.2	Utilizzare uno sconto	U	F3.3

Tabella 1.6 – Requisiti di gestione delle transazioni

1.5 Altri requisiti

I seguenti requisiti sono nuove funzionalità che non sono esistenti nel precedente sistema in uso.

ID	Descrizione	Diritti	Dipendenze
F5.1	Inviare una richiesta di amicizia	U	-
F5.2	Accettare una richiesta di amicizia	U	F5.1
F5.3	Rifiutare una richiesta di amicizia	D	F5.1
F5.4	Creare una discussione	S	-
F5.5	Inviare un messaggio		F5.4
NF5.1	Variazione minima rispetto ai processi attuali	-	-

Tabella 1.8 – Altri requisiti

2 Casi d'uso

In questo capitolo vengono descritte nel dettaglio le interazioni delle varie tipologie di *utente* con il *sistema*. Verranno tralasciate le semplici operazioni di inserimento dei dati e, più in generale, quelle operazioni che non richiedono una logica complessa ma solo di un'interfaccia intuitiva e dei controlli che minimizzino gli errori umani.

Gran parte delle operazioni fanno riferimento al concetto di *Anno associativo*, con esso si intende il periodo di tempo in cui hanno validità le iscrizioni degli utenti.

Un *utente* deve poter accedere al sistema prima di ogni altra cosa dato che quasi la totalità delle funzionalità sono riservate ad utenti regolarmente iscritti. Si presuppone dunque che nei casi d'uso elencati sia sottinteso che l'*utente* abbia effettuato l'accesso al sistema utilizzando le proprie credenziali d'accesso.

2.1 Iscrizione

L'iscrizione all'associazione consiste nell'operazione di registrazione dell'utente al libro dei soci. Ogni utente deve essere in grado di inviare una richiesta d'iscrizione all'associazione, questa dovrà essere accompagnata dal pagamento della quota associativa e dalla consegna del modulo stampato e firmato contenente i dati anagrafici e le condizioni legali relative all'iscrizione stessa.

2.1.1 Dati anagrafici

Fondamentale per un corretto instradamento della richiesta è l'inserimento dei dati anagrafici, una volta passati i controlli di validazione base questi dovranno essere registrati e diventeranno in attesa di essere confermati da un gestore.

Nel sistema precedente si sono spesso verificati errori di inserimento da parte degli utenti, questo causava errori nel sistema e, nel caso peggiore, errori nei moduli; una buona validazione permette di filtrare molti di questi casi ma non tutti, pertanto è stata mantenuta la convalida lato gestore anche se questa non è

bloccante, l'utente infatti può scaricare il modulo compilato con i dati da lui/lei inseriti e pronto per essere firmato e consegnato anche prima della convalida.

2.1.1.1 Effort

Minimizzare l'effort impiegato lato associativo è estremamente importante, questo infatti non deve essere maggiore rispetto a quello attuale.

Il sistema in uso richiede un intervento manuale pesante da parte dei gestori nel caso di errori di inserimento, in tal caso la richiesta deve essere manualmente annullata e l'utente deve essere contattato tramite mail, telefono o simili.

Nel nuovo sistema l'utente potrà visualizzare direttamente dal proprio profilo la situazione in cui si trova, in tal modo potrà autonomamente reinserire i dati anagrafici e generare il modulo corretto.

2.1.2 Moduli cartacei

La presenza dei moduli cartacei è di natura legale, l'associazione deve conservare tali moduli firmati per diversi anni, è importante dunque che un gestore possa registrare la corretta ricezione del modulo e che l'utente sia in grado di visualizzare lo stato della consegna dei moduli.

2.1.3 Tessere e assicurazioni

Ad ogni iscrizione è associata una tessera, molte tessere concedono benefici differenti tra i quali ad esempio un'assicurazione contro gli infortuni. Al momento dell'iscrizione ogni utente deve scegliere la tessera che preferisce.

2.1.3.1 Effort

Nel sistema attuale le assicurazioni vengono emesse da terzi, queste richiedono spesso parte dei dati anagrafici. I gestori devono recuperare i dati e manualmente inserirli nel sistema assicurativo, successivamente devono registrare l'assicurazione emessa nel sistema.

Non potendo intervenire nel sistema assicurativo si è scelto di permettere ai gestori di creare *assicurazioni* all'interno del sistema, una volta censita un'assicurazione l'utente dovrà selezionare un elenco di utenti da aggiungere a

tale assicurazione, in questo modo l'effort è leggermente ridotto in quanto non è necessario aggiungere su ogni singolo record i dati assicurativi ma questo viene fatto tramite semplici checkbox.

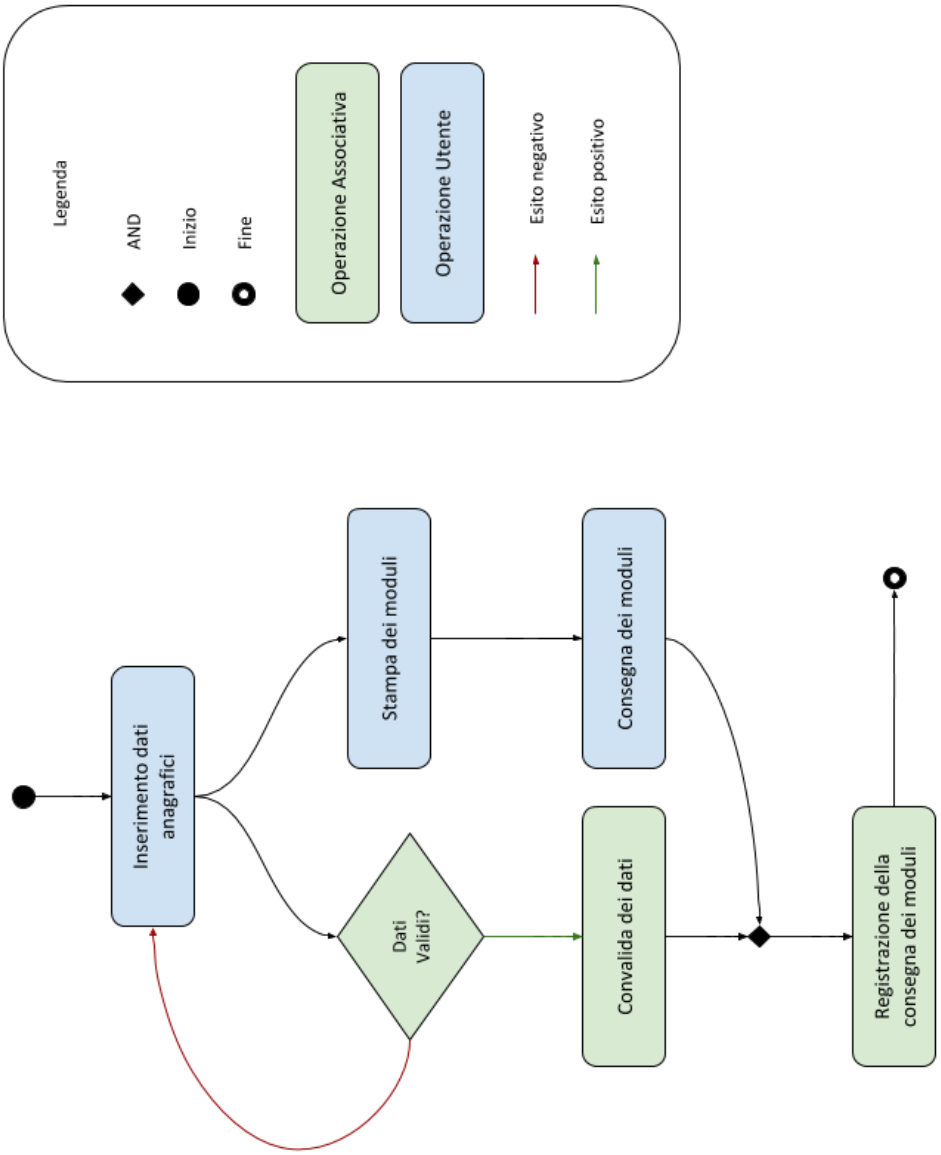

Figura 1 - Flusso di un'iscrizione

2.2 Eventi

Di norma solo un *socio* può iscriversi ad un *evento associativo*, tuttavia effettuare l'*iscrizione all'associazione* in contemporanea alla richiesta di partecipazione ad un *evento* è un caso molto frequente.

2.2.1 Utenti non tesserati

L'elenco degli eventi deve essere visibile, gli utenti devono poter scegliere eventuali preferenze di partecipazione e richiedere un biglietto. Se l'utente non è in possesso di una tessera deve poter semplicemente selezionarne una dalla schermata e questa deve essere inclusa nel carrello.

2.2.2 Tipologie di tessere

Alcune tipologie di tessere associative non includono alcuna *assicurazione contro gli infortuni* e, la maggior parte degli eventi (in particolare gli eventi *LARP*), richiedono necessariamente un'assicurazione valida per poter partecipare, in questo caso il tipo di *tessera* dell'utente deve includere un'assicurazione o l'utente deve essere in possesso di un'assicurazione accettata dal *Consiglio direttivo*.

2.2.3 Assicurazioni

Le assicurazioni vengono fornite da terzi su richiesta dell'associazione, i *gestori* devono poter registrare la data di scadenza dell'assicurazione relativa ad ogni utente. Questo processo avviene spesso in massa, ciò significa che molti soci avranno un'assicurazione con la stessa data di scadenza.

Un *gestore* può accedere alla pagina apposita in cui gestire le assicurazioni: creando, eliminando o modificando le assicurazioni stesse. Le assicurazioni hanno una *data di scadenza*, che indica il termine oltre il quale l'assicurazione non sarà più valida, delle *note*, che possono essere inserite da vari gestori per annotare informazioni significative (ad esempio quante sono state pagate e da chi), e un elenco di assicurati con quella specifica assicurazione.

Solo poche tipologie di assicurazione esterne vengono accettate, ed è un caso che richiede la diretta attenzione del *Consiglio direttivo*, in questo infatti il rallentamento del processo dal punto di vista dell'*utente* è accettabile.

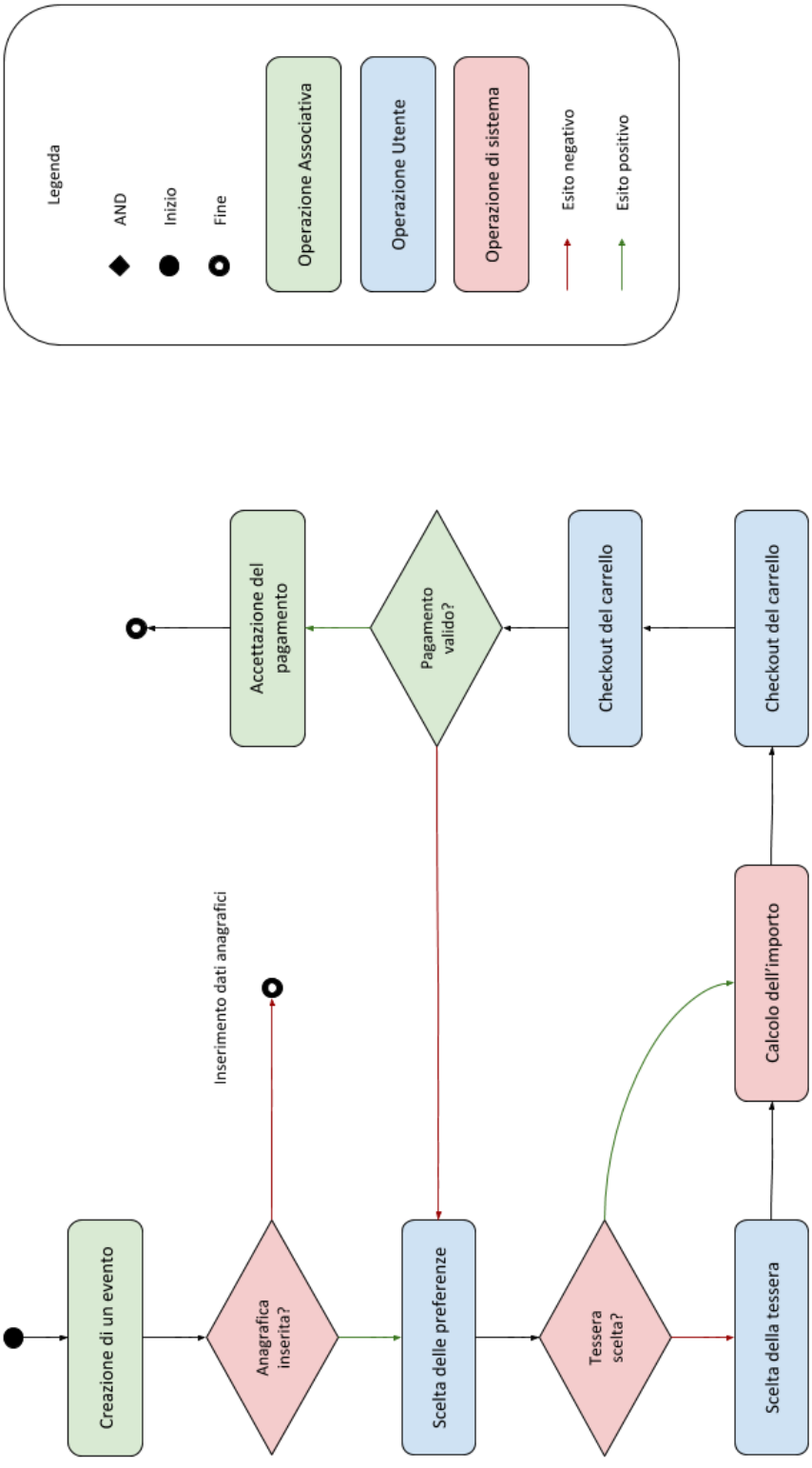

Figura 2 - Iscrizione ad un evento

2.2.4 Preferenze di partecipazione

Gli *eventi* possono avere *preferenze di partecipazione*, queste rappresentano delle scelte specifiche relative all'evento (es. posto a sedere, giorni in cui si intende partecipare). Le *preferenze* vengono impostate dagli *Organizzatori*, devono avere delle *opzioni* selezionabili, per questo motivo quando un *utente* effettua una richiesta di *partecipazione* tutte le coppie *Preferenza-Opzione* devono essere salvate al suo interno anche in modo statico[2]. Le *opzioni* inoltre possono avere un *limite di selezioni* oltre il quale non è più possibile selezionarla, questo può variare da *0* a *infinito*, cioè nessun limite.

2.2.5 Età minima

Gli *eventi* possono avere un requisito minimo di età per parteciparvi, ciò significa che potranno effettuare la richiesta di partecipazione solo i soci che entro l'inizio dell'evento stesso avranno compiuto l'età richiesta, questo requisito può essere evitato *accompagnando* il minore all'evento stesso.

2.3 Anni associativi

Ogni iscrizione ha validità circoscritta in un anno associativo, ognuno di questi anni anno ha un inizio ed una fine che spesso discostano da quello solare. Ad un anno associativo inoltre deve essere possibile iscriversi prima del suo inizio.

Nel sistema in uso questa operazione è fatta manualmente, le tessere devono essere manualmente invalidate e gli utenti contattati direttamente.

Nel nuovo sistema le iscrizioni sono automatiche, gli eventi e le iscrizioni verranno legate agli anni associativi in modo da automatizzare completamente tale operazione.

[2] Con "statico" si intende senza alcun riferimento, in questo modo se il dato originale dovesse essere modificato il campo "statico" non rileverebbe tali variazioni.

2.4 Pagamenti

Il sistema dei pagamenti è stato trasposto dal sistema attuale, implementando migliorie ove possibile.

2.4.1 La logica attuale

Nel sistema in uso i pagamenti vengono generati in automatico, questi sono identificati da un importo e da un metodo di pagamento. Gli utenti non hanno modo di interagire con esso, pertanto devono contattare l'associazione tramite e-mail e notificare l'avvenuto pagamento. I gestori devono registrare tali pagamenti nel sistema inserendo il metodo di pagamento scelto dall'utente.

Questo processo è uno dei processi più inefficienti sia lato utente che associativo.

2.4.2 Implementazione

Il modo più semplice per ottimizzare questo processo è quello di includere la parte comunicativa all'interno del sistema.

Lato utente devono essere unificati la scelta del metodo e la segnalazione all'associazione, la cosa viene fatta tramite il sistema di un carrello. Ogni utente deve avere un carrello al quale deve poter aggiungere tessere e biglietti, una volta completati gli acquisti dovrà scegliere il metodo di pagamento e inserire i dettagli per facilitarne il tracciamento.

Lato associativo il sistema deve permettere di visualizzare i pagamenti inviati dall'utente e accettarli.

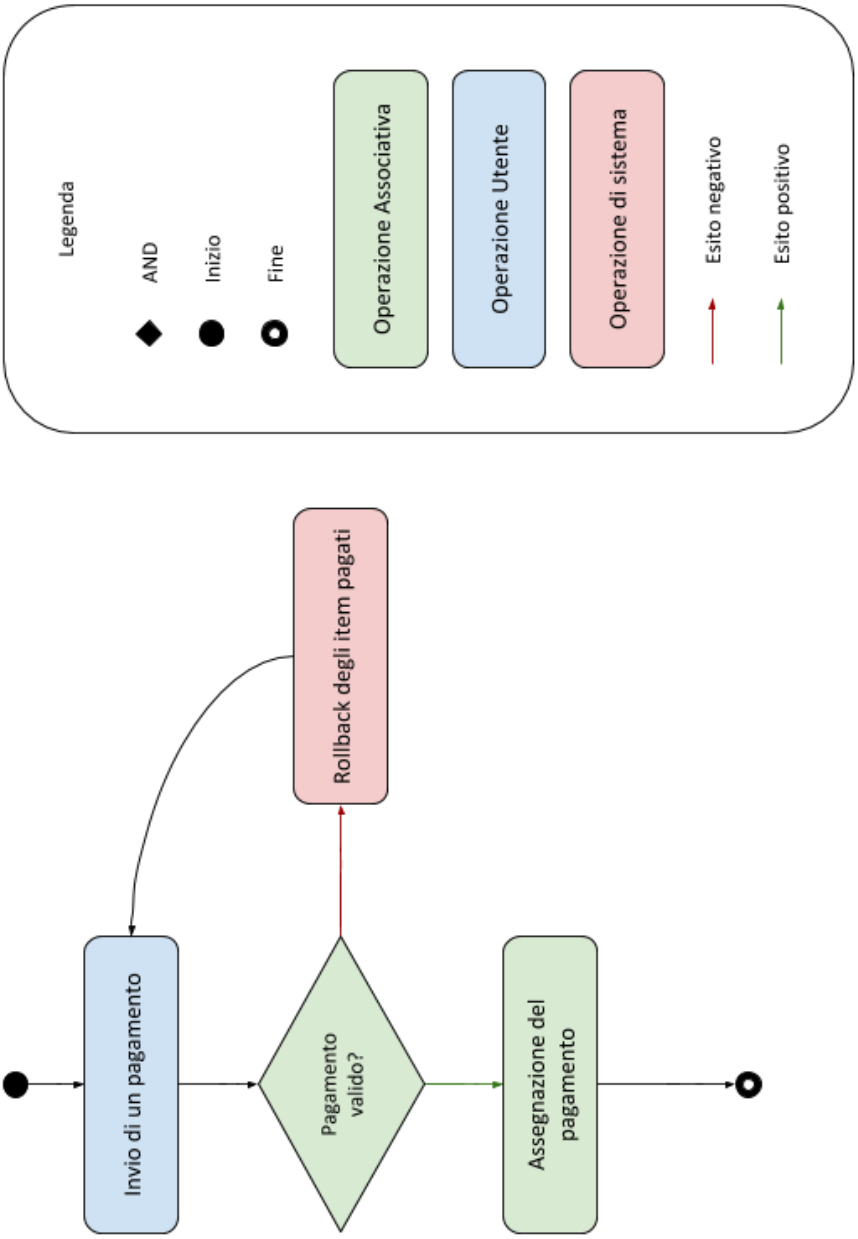

Figura 3 - Registrazione di un pagamento

2.4.3 Metodi di pagamento e movimenti

Non sono stati implementati nuovi metodi di pagamento, questi rimangono dunque: *Bonifico bancario*, *PayPal* e *Brevi manu*. Tutti i pagamenti devono essere accompagnati da alcuni dati aggiuntivi per facilitarne la tracciabilità.

È da porre particolare attenzione al pagamento *Brevi Manu*, questa è una tipologia di pagamento che consiste nel pagamento fisico ad un socio autorizzato. Al contrario degli altri metodi di pagamento questo non è diretto, il socio autorizzato che ha raccolto la quota dovrà successivamente depositare l'ammontare in conti intestati all'associazione o consegnarli al *tesoriere* in carica.

I movimenti di denaro interni al gruppo organizzativo avvengono per lo più da una persona al conto bancario, al conto PayPal o alla cassa fisica dell'associazione, non è però da escludere il trasferimento di denaro da una persona fisica ad un'altra (es. al *tesoriere*).

2.4.4 Sconti

Gli sconti sono resi nel sistema attualmente in uso tramite codici sconto, questi devono essere creati da un gestore per poi essere comunicati all'utente. Quando l'utente richiede l'iscrizione ad un evento può inserire il codice, se questo è valido allora viene applicato lo sconto e l'importo aggiornato.

Nel nuovo sistema tale processo risulta semplificato: gli sconti sono generati da un gestore e l'utente può utilizzarlo immediatamente dalla schermata d'iscrizione all'evento; in questo modo non è necessario comunicare codici che devono essere memorizzati dall'utente in modi non convenzionali.

2.5 Social

L'intero sistema social è del tutto nuovo, non esiste alcuna funzionalità simile all'interno del sistema attualmente in uso.

Gli utenti hanno la possibilità di interagire tra loro attraverso discussioni pubbliche e chat private, tutto questo deve essere supportato da un sistema di amicizie.

Le discussioni pubbliche devono poter essere visualizzate da chiunque, le discussioni private invece sono chat tra due persone.

Gli utenti devono poter inviare richieste di amicizia a tutti gli utenti che vedono nelle chat pubbliche e inviare una richiesta diretta utilizzando Nome utente e Tag, un codice numerico univoco per ogni utente.

Tutto l'ambito social ha permesso di enfatizzare l'aspetto realtime di Firebase ed è stato progettato in modo da poter accogliere nuove funzionalità che lo possano rendere più unico e funzionale per la categoria di utenti che dovranno utilizzarla.

3 Firebase

Firebase è la piattaforma a cui si appoggia *Blooming Harp*, permettendo di limitare ampiamente le implementazioni lato back end, offrendo strumenti in grado di supportare lo sviluppo di ogni genere di applicazioni Web.

Uno dei punti focali di *Firebase* è il database, di tipo NoSQL, del quale ne esistono due varianti: *Realtime Database* e *Cloud Firestore*, il primo è la versione originaria offerta da *Firebase* mentre il secondo è la versione più recente ancora in Beta al momento dello sviluppo di *Blooming Harp*.

Uno dei punti focali di questo progetto è quello di analizzare quanto possa essere complessa (e quali vantaggi potrebbe portare) una migrazione dal *Realtime Database* al *Cloud Firestore*.

3.1 Strumenti lato Client

Firebase si interfaccia bene con diverse tipologie di Client, in questo progetto è stata utilizzata la libreria angularfire per accedere a diverse funzionalità tramite AngularJS.

L'inizializzazione è immediata e le guide messe a disposizione sono chiare e dirette.

3.1.1 Accesso tramite AngularJS

Comunicare con Firebase tramite AngularJS richiede, dopo il setup generale, l'inizializzazione di una variabile globale identificato come "firebase", attraverso il quale è possibile accedere a tutte le funzionalità necessarie.

L'oggetto firebase può essere creato come oggetto con diritti di amministratore o con gli stessi diritti dell'utente che sta utilizzando l'applicazione. Nel progetto Blooming Harp l'applicazione accede solamente come semplice utente.

3.1.2 Autenticazione

Esistono diversi metodi di autenticazione, nel progetto viene utilizzato un semplice sistema di Email + Password. L'autenticazione lato Server è gestita interamente dalla piattaforma, lato Client è necessario utilizzare diversi strumenti per le varie operazioni di autenticazione, tali strumenti ritornano una *Promise*, questo permette un'implementazione uniforme e pulita.

3.1.3 Sessioni

Il comportamento di default nelle applicazioni Web prevede una sessione persistente, ovvero che rimane attiva finché l'utente mantiene aperta la scheda o la finestra. Lato Client è stata implementata una funzione per catturare i cambiamenti di stato della sessione, in questo modo è possibile creare, modificare e pulire le variabili di sessione necessarie alla corretta navigazione.

3.2 Cloud Functions

Le *Cloud Functions* sono funzioni *JavaScript* o *TypeScript* che possono essere caricate ed essere eseguite lato server.

Questo strumento può supportare ogni tipologia di esigenza, tuttavia il suo comportamento spezza, in alcuni casi, l'aspetto realtime di Firebase, queste infatti vengono eseguite in modo asincrono ed il client non ha modo di attendere il completamento di tali funzioni.

3.2.1 Funzionamento

Le funzioni vengono chiamate tramite dei trigger sulle operazioni effettuate sui dati stessi (nel caso di Realtime Database trigger), gli eventi catturabili descritti nella tabella 6.1 insieme alla possibilità di specificare il percorso del sottoalbero da ascoltare permettono di gestire liberamente tutti i casi necessari.

Operazione	**Descrizione**
Create	Il dato è stato creato.
Update	Il dato è stato aggiornato.

Delete	Il dato è stato cancellato.
Write	Il dato è stato modificato in qualunque modo.

Tabella 3.1 - Realtime Database Trigger

I percorsi possono essere anche espressi tramite “variabili” (in formato ${<nome chiave>}) che permettono di gestire chiavi dinamiche ed acquisirne i valori.

Esistono diversi tipi di trigger che offrono la possibilità di gestire anche molti altri casi, ad esempio i Firebase Authentication Trigger permettono di gestire eventi come la registrazione di un utente, gli HTTPTrigger consentono l’intercettazione di richieste (GET, POST, PUT, DELETE e OPTIONS), in questo caso il Client è in grado di attendere il completamento dell’operazione.

3.2.2 Vantaggi

Le Cloud Functions permettono di creare del codice indipendente dal client in grado di mantenere l’architettura classica Client Server consentendo la creazione di ambienti separati e garantire una maggiore sicurezza delle operazioni.

Possono essere utili nella cattura di eventi difficilmente rilevabili lato Client e permettono di implementare funzionalità interessanti come la pulizia dei dati sistematica e analisi dei dati.

3.2.3 Svantaggi

Gli strumenti forniti lato Client spingono verso una grande responsività ed una grossa indipendenza dell’applicazione, questo va in netto contrasto con le operazioni scatenate sulla modifica dei dati, in quanto queste vengono eseguite in modo asincrono ed il Client non ha un modo diretto per attendere il completamento dell’esecuzione delle funzioni, pertanto il Client riceverà una risposta immediatamente dopo aver effettuato un’operazione sui dati.

Un’altra problematica si verifica quando si vuole elaborare i dati lato Server, tali operazioni infatti richiedono tempo in quanto vengono eseguite dal server non appena possibile creando una discrepanza temporale tra la richiesta e

l'elaborazione effettiva dei dati, questo richiede l'implementazione di lock sui dati (i quali non sono supportati nativamente dalla piattaforma) per non incorrere in violazioni delle proprietà ACID.

4 Realtime Database

Inizialmente *Blooming Harp* è stato modellato per assecondare i bisogni del *Realtime Database*, utilizzando le strutture dati più consone mirate ad ottenere una buona efficienza e scalabilità.

4.1 Security Rules

Le regole di sicurezza vengono definite tramite una struttura che ricalca l'albero dei dati del Realtime *Database*, tale struttura è a tutti gli effetti un albero JSON che ammette solo due tipi di chiavi: statica e dinamica; e quattro tipi di regole: *.read, .write, .validate, .indexOn*.

Tutte le regole di sicurezza possono accedere ai dati relativi all'account dell'utente che effettua la richiesta (es. email, id), è possibile muoversi sull'albero dei dati tramite *.parent()*, *.child()* e *.root()*, permettendo allo sviluppatore di accedere a qualunque altro dato contenuto nel *Realtime Database*.

Tutte le regole inoltre possono accedere ai dati correnti e ai dati potenzialmente modificati dalla richiesta in corso, nello specifico nelle operazioni di scrittura le regole sono in grado di differenziare i dati memorizzati prima della richiesta e i dati modificati in seguito a tale richiesta.

Utilizzate correttamente è possibile delegare al Client il compito di elaborare la totalità della richiesta ed inviare al server dei dati che necessitano solo di una validazione, in questo modo non è necessario implementare funzioni lato server.

4.1.1 Chiavi statiche e dinamiche

Il mapping tra le regole e la struttura dati può essere effettuato attraverso chiavi statiche (es. "utenti") o chiavi dinamiche (es. "$uid"), le prime permettono di allinearsi direttamente con la chiave specifica mentre le seconde sono da usare in caso di chiavi il cui valore non è predeterminato, come ad esempio ID o chiavi auto-generate.

4.1.1.1 .read e .write

Le regole di lettura e scrittura vengono espresse attraverso questi campi, il loro effetto è ereditario e non viene sovrascritto dai figli, ciò significa che il sistema scende di livello sull'albero delle regole solo se l'utente non è autorizzato ad effettuare la richiesta.

4.1.1.2 .validate

A differenza delle regole di sicurezza .read e .write, tutte le regole .validate devono essere soddisfatte prima di poter completare l'operazione, questa proprietà permette di gestire gran parte dei casi grazie ai controlli effettuati sull'intero sottoalbero.

4.1.1.3 .indexOn

Questa particolare regola non controlla gli accessi ma permette di rendere efficaci le ricerche dei dati creando degli index su dati differenti dalle chiavi stesse (le chiavi sono sempre indicizzate).

La corretta impostazione degli indici è di fondamentale importanza per garantire l'efficienza di gran parte delle query, configurarle risulta dunque di vitale importanza per il sistema.

4.2 Operazioni sui dati

L'accesso ai dati lato Client è basato su diverse funzioni di libreria che ritornano Promises, in questo modo è possibile utilizzare tutti gli strumenti in grado di interfacciarsi con esse.

Al ritorno di una chiamata si ha come risposta una struttura dati chiamata Snapshot, ovvero un'immagine istantanea dei dati, questa espone il metodo `.val()` per recuperare i dati effettivi ed altri metodi utili ad eventuali analisi o controlli sui dati (es. data di ultima modifica, id, path).

4.2.1 Accedere ai dati

Lo strumento principale per accedere ai dati contenuti nel Realtime Database è firebase.database().ref(), tramite il quale è possibile effettuare tutte le operazioni di Read e Write.

Le operazioni disponibili sono: set, on, once, update, transaction.

4.2.2 In scrittura

I metodi per la scrittura sono set, update, push e transaction:

`set()` permette effettuare una richiesta di write sovrascrivendo i dati presenti nel percorso specificato.

`update()` consente l'aggiornamento dei dati applicando modifiche non distruttive, è veloce e permette di aggiornare diversi percorsi con la stessa richiesta.

`push()` di base ritorna un ID univoco sulla lista dei dati richiesta, tuttavia è possibile passare un oggetto tra i parametri per effettuare anche un set() con l'ID univoco creato.

`transaction()`, operazione onerosa fondamentale nella gestione degli accessi concorrenti, è lenta e pesante ma garantisce l'assenza dei conflitti.

4.2.3 In lettura

I metodi legati alla lettura sono once, on e off:

`once()` legge i dati una volta, creando uno snapshot degli stessi.

`on()` legge i dati e mantiene aggiornato il client sulle modifiche e richiede la specifica del m.

`off()` interrompe l'ascolto creato tramite on() sulla risorsa specificata.

Evento	Descrizione	Snapshot
Child_added	Viene scatenato una volta per ogni figlio e ogni volta che ne viene aggiunto uno.	Nuovo figlio
Child_changed	Viene scatenato ogni volta che un nodo figlio viene modificato.	Figlio modificato
Child_removed	Viene scatenato ogni volta che un nodo figlio viene rimosso.	Nodo rimosso
Child_moved	Rileva cambiamenti in una lista ordinata.	
Value	Viene scatenato ad ogni modifica della risorsa.	La risorsa.

Tabella 4.1 - Eventi dei listener

4.2.4 Timestamp

Firebase offre una costante dell'oggetto *firebase.database.ServerValue* (contenuto della struttura dati "firebase.database" inizializzata lato Client) chiamata `TIMESTAMP`, questa permette di gestire appunto i timestamp in modo estremamente efficiente. Quando si crea una richiesta di scrittura dei dati (create o update) è possibile specificare un valore speciale del dato che Firebase è in grado di rilevare e tradurre sotto forma di timestamp (come millisecondi trascorsi dalla Unix epoch) al momento della ricezione della richiesta. Questa stessa costante permette in modo similare di sincronizzare il Client con il Server per effettuare controlli sui timestamp, come ad esempio la scadenza di un valore.

5 Cloud Firestore

Il nuovo database NoSQL offerto da Firebase, in fase Beta al momento dello sviluppo di questo progetto.

I servizi offerti da questa tecnologia sono molto simili a quelli offerti da *Realtime Database* (aggiornamenti in realtime, scalabilità, servizi offline, eccetera) ma la logica richiesta nella modellazione della struttura dati è molto diversa e si basa sul concetto di *Documenti* e *Collezioni*.

Firestore, esattamente come Realtime Database, non ha uno schema, pertanto non esistono vere e proprie limitazioni intrinseche sulla tipologia di dati che possono essere inserite dall'utente, nonostante ciò queste possono essere simulate parzialmente attraverso regole di sicurezza.

5.1.1 Documenti

Un documento è identificato da un nome univoco e rappresenta un record contenente una quantità variabile di chiavi associate a valori in una struttura JSON. Tutti i documenti possono contenere una quantità indefinita di *Collezioni*, permettendo in questo modo la creazione di creare un vero e proprio albero dei dati.

Recuperare un documento ne carica solamente i dati diretti, lasciando dunque ad una *query* puntuale il compito di recuperare i dati contenuti in eventuali sotto collezioni del documento.

5.1.2 Collezioni

Le collezioni sono liste di Documenti ottimizzate per l'ordinamento, il recupero e la gestione di lunghe liste di documenti, ciò significa che per garantire una buona scalabilità dell'applicazione è necessario tenere bene a mente che è sempre preferibile avere collezioni ampie invece di grandi documenti.

5.1.3 Reference

Per accedere a documenti e collezioni è necessario creare Reference al quale poi vengono applicate le funzioni di lettura/scrittura che variano in base alla tipologia di oggetto referenziato, ad esempio è possibile ordinare e filtrare la lista dei documenti contenuti all'interno di una collezione da una *CollectionReference* mentre è possibile aggiornare il contenuto di uno specifico documento solo da una *DocumentReference*.

Ogni reference può essere concatenata seguendo un'unica semplice regola: Una *CollectionReference* può essere susseguita unicamente da una *DocumentReference* e viceversa, ciò significa che le due tipologie di reference devono necessariamente susseguirsi.

5.1.3.1 Operazioni di lettura

I dati referenziati da una reference possono essere letti staticamente, creando un singolo snapshot, oppure ascoltati specificando una callback da chiamare ogni volta che viene rilevato un cambiamento dei dati.

Il metodo `get()` permette di leggere direttamente creando uno snapshot del dato, `onSnapshot()` è utilizzato invece per creare un listener che chiamerà la callback passata come argomento.

Le CollectionReference espongono metodi che permettono di filtrare la lista dei documenti, tali filtri vengono applicati tramite i diversi metodi descritti in tabella e possono essere ordinati tramite il metodo `orderBy()`.

5.1.3.2 Operazioni di scrittura

Tutti i metodi di scrittura permettono di aggiornare, modificare, creare e cancellare i dati referenziati, tutte le modifiche vengono riflettute in realtime sul database innescando l'aggiornamento di tutte le risorse aggiornate da tutti i client.

Metodo	Descrizione
`endAt()`	Tutti i documenti fino a quello specificato come argomento (incluso).
`endBefore()`	Tutti i documenti fino a quello specificato come argomento (escluso).
`limit()`	Numero massimo di documenti.
`startAt()`	Tutti i documenti a partire da quello specificato come argomento (incluso).
`startAfter()`	Tutti i documenti a partire da quello specificato come argomento (escluso).
`where()`	Condizione che deve essere soddisfatta affinché il singolo documento della collezione venga caricato, tale condizione può essere specificata su più campi e vengono utilizzati gli operatori ("<", "<=", "==", ">", ">=") per determinare la condizione.

Tabella 5.1 - Filtri

5.1.4 Snapshot

I dati ritornati da un'operazione di lettura vengono identificati come snapshot, ovvero un'immagine dei dati recuperati in quell'istante. Esistono snapshot relativi sia a collezioni che documenti, la sua gestione in entrambi i casi richiede un'attenzione particolare.

5.1.4.1 DocumentSnapshot

Questo tipo di snapshot contiene il riferimento ad uno specifico documento, da questo è possibile recuperare tutti i dati contenuti all'interno del documento (`.data()`) o recuperare uno specifico *fieldPath*.

5.1.4.2 QuerySnapshot

Uno snapshot di questo tipo permette di gestire un riferimento ad una collezione, da questa è possibile effettuare tutte le operazioni di lettura sui dati, come l'array dei documenti, i metadati, le modifiche dall'ultimo snapshot, un iteratore sui documenti e tutto ciò che è descritto nella tabella.

Metodo/Campo	Descrizione
docs	Array dei documenti contenuti nello snapshot
empty	Booleano che indica se non ci sono documenti nello snapshot.
metadata	Metadati sullo snapshot inclusi fonte e se contiene modifiche locali.
query	La query effettiva chiamata sul get() o sul onSnapshot().
size	Numero dei documenti nello snapshot.
docChanges()	Ritorna l'array delle differenze dall'ultimo snapshot.
forEach()	Enumera i documenti dello snapshot, permettendo di chiamare una funzione di callback su ogni documento.
isEqual	Compara this con un altro QuerySnapshot.

Tabella 5.2 - Metodi e Campi di QuerySnapshot

6 Migrazione

Uno dei punti focali del progetto è stato lo studio sulle migrazioni, questo infatti ha visto due modifiche sostanziali: la prima è stata un cambiamento dell'intero sistema, la seconda invece un cambiamento interno della struttura dati.

Le logiche del sistema precedentemente in uso (basato su *GoogleSpreadsheet* e *GoogleForm*) sono state replicate il più fedelmente possibile nell'applicazione *Blooming Harp*. Questo ha permesso di analizzare le difficoltà pratiche di un utente nell'interfacciarsi ad un sistema differente, richiedendo dunque uno studio approfondito sulla *user experience* in modo da creare dei processi che non discostino molto da un semplice foglio di calcolo o da un form.

Il cambiamento della struttura dati è stata una scelta più tecnica, questa non ha influenzato l'utente finale in alcun modo ed è stata presa sia per motivi di studio che per migliorare le prestazioni del sistema. La migrazione da *Realtime Database* a *Cloud Firestore* sarà l'oggetto principale del seguente capitolo, scendendo nei particolari e analizzando vantaggi, svantaggi e difficoltà tecniche.

Si identificano come *differenze critiche* tutte quelle differenze che sono contrastanti tra loro e richiedono un'attenzione particolare in un contesto di migrazione.

6.1 Data model

Entrambe le tipologie di database sono NoSQL ma hanno una struttura molto diversa, i dati memorizzati nel *Realtime Database* sono memorizzati in un albero JSON mentre *Cloud Firestore* memorizza i dati in Collezioni e Documenti, come descritto più nel dettaglio rispettivamente nei capitoli 4 - Realtime Database e 5 - Cloud Firestore.

6.1.1 Data flattening e Denormalization

Da un lato l'albero JSON concede estrema flessibilità ma dall'altro richiede un uso intensivo di data flattening e denormalization, ciò significa che per garantire una

buona efficienza delle operazioni di lettura è necessario suddividere i dati in alberi differenti per assecondare le query più comuni ed evitare di recuperare l'intero albero ad ogni query.

Prendiamo come esempio la lista di utenti: si supponga per semplicità che ogni utente deve contenere alcuni dati di base (nickname, ruolo, eccetera), l'elenco degli anni di iscrizione e l'elenco degli amici; implementare una tale struttura senza denormalizzazione porterebbe ad un eccessivo scambio di dati del tutto non necessario, pertanto nel caso in analisi il requisito minimo è di separare una lista "snella" di utenti dal quale è possibile recuperare dati di base comunemente utilizzati e suddividere gli anni di iscrizione e la lista degli amici in alberi del tutto separati, in questo modo quando è necessario elencare gli utenti attivi non sarà necessario scaricare ogni singolo dato riguardante tutti gli utenti ma solo una piccola porzione di esso.

L'utilizzo delle Collection permette di ridurre drasticamente l'uso di queste tecniche in quanto leggere un Document di una Collection non recupera le Subcollection, ciò significa che, riprendendo l'esempio precedentemente esposto, un utente conterrebbe una piccola struttura dati JSON per i dati di base, una Collection per l'elenco degli amici e un'altra Collection per l'elenco degli anni d'iscrizione; in questo modo leggere l'intera Collection degli utenti ne scaricherebbe solo i dati di base e non i contenuti delle Subcollection.

6.1.2 Scalabilità dei dati

Entrambe le soluzioni presentano "simili" capacità di scalare, tuttavia creare un singolo albero JSON adatto ad una grande mole di dati potrebbe risultare complesso da progettare e da implementare, le Collection invece sono intuitivamente più adatte ad accogliere grandi quantità di documenti al loro interno permettendo una più semplice implementazione anche con grandi mole di dati.

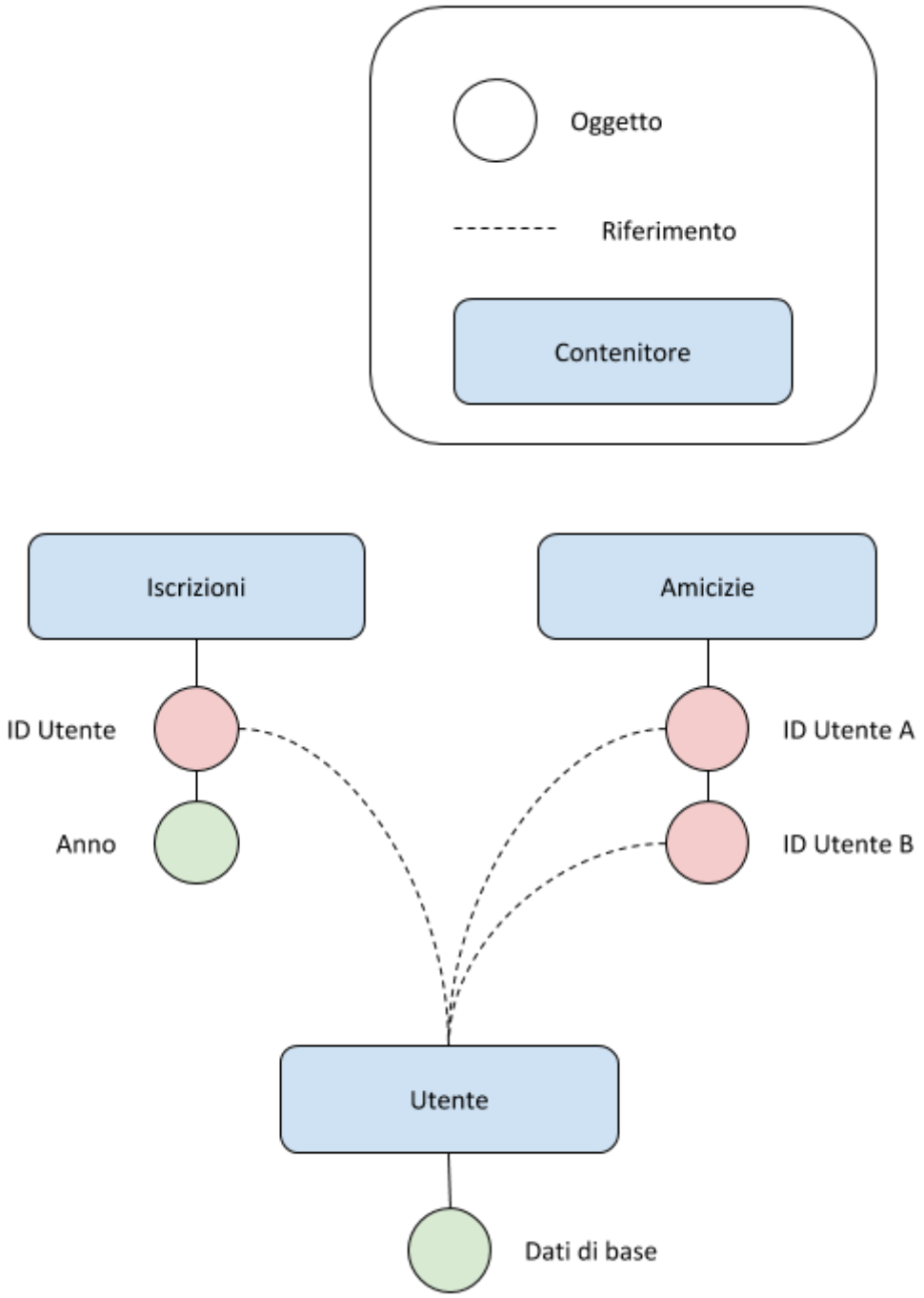

Figura 4 - Esempio di data flattening

6.1.3 Timestamp (differenza Critica)

Nel Realtime Database i timestamp vengono memorizzati come millisecondi trascorsi dalla Unix Epoch, Cloud Firestore invece li rappresenta in oggetti contenenti secondi e microsecondi, tale differenza può creare delle grosse discrepanze in una migrazione in quanto un sistema basato su Realtime Database

potrebbe recuperare il timestamp e direttamente convertirlo in formati più adatti (come ad esempio una javascript Date), la sua controparte invece richiede l'elaborazione dell'oggetto prima di poter essere convertito, pur non essendo complesso (anche per via dei metodi a disposizione da Firebase) la transizione non è trasparente, richiedendo un intervento potenzialmente su diversi punti.

6.1.3.1 Soluzione adottata

Nel progetto è stato necessario applicare un metodo nel servizio che gestisce l'accesso al database in carico di tradurre il timestamp in un formato unico, per farlo è stato necessario andare a cambiare tutte le implementazioni di lettura dei timestamp, tale modifica ha influenzato diversi file.

In una migrazione nella direzione opposta questa differenza potrebbe risultare meno critica in quanto è comunque necessario elaborare il timestamp ricevuto da Cloud Firestoe, pertanto è possibile centralizzare la modifica nel metodo incaricato di elaborare tale timestamp per rendere trasparente la migrazione al Realtime Database.

6.1.4 Snapshot (differenza critica)

Recuperare il contenuto da uno Snapshot ritornato dal Realtime Database viene effettuato utilizzando il metodo `.val()`, nel Cloud Firestore invece ci sono da distinguere i casi in cui si recupera una Collection, che bisogna iterare tramite forEach() o recuperare l'array dei documenti, o un singolo Document, nel quale si accede ai dati tramite il metodo `.data()`.

6.1.4.1 Soluzione adottata

È stato implementato il metodo `.val()` nei prototipi di QuerySnapshot e di DocumentSnapshot, tale metodo ha lo scopo di convertire in un Object Javascript i dati dei documenti contenuti nello snapshot. Nel caso di un DocumentSnapshot è possibile invocare semplicemente il metodo `.data()`, nel QuerySnapshot invece è necessario mappare gli id con i dati dei singoli documenti iterando attraverso un `forEach()`.

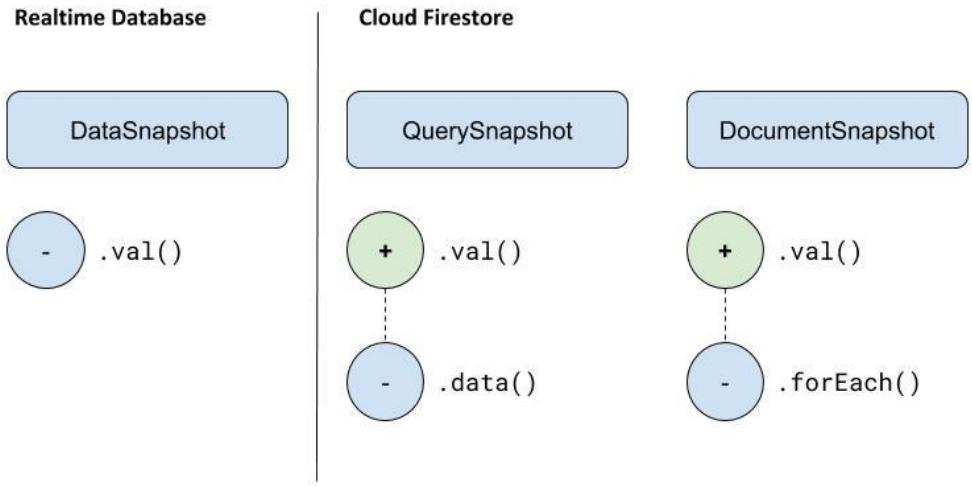

Figura 5 - Adattamento degli Snapshot

6.1.5 Import & Export (differenza critica)

Realtime Database permette di importare ed esportare un qualunque albero JSON a partire dal root fino ai nodi più bassi, questa funzionalità permette di facilitare una migrazione sia cross sistema, convertendo i dati in JSON e ribaltandoli direttamente nella struttura dati, sia da un sistema basato su Firestore ad un altro. Cloud Firestore d'altra parte non permette di gestire manualmente import ed export, per farlo è necessario creare degli script o dei metodi in grado di gestire tali operazioni.

Importare i dati è un'operazione potenzialmente critica, questo deriva dal fatto che le strutture dati possono essere disallineate, pertanto effettuare un'importazione senza i dovuti controlli presenta alte probabilità di generare inconsistenze e criticità, il che significa che in entrambi i casi sarebbe necessario implementare script o metodi ad hoc in grado di purgare, normalizzare ed allineare la struttura dati in ingresso seguendo le regole della struttura dati in uscita.

6.2 Querying

Interrogare il Data Model per recuperare specifici dati è di fondamentale importanza per un'applicazione di qualunque tipo, entrambe le soluzioni infatti

permettono di creare query poco complesse in grado principalmente di filtrare e ordinare i dati.

6.2.1 Ordinamenti e Filtri

Entrambe le tipologie di database offrono diversi metodi per l'ordinamento dei dati e per l'impostazione dei filtri, tuttavia utilizzando il Realtime Database non è possibile applicare più di una regola di ordinamento ad una singola query anche se è possibile concatenare un qualunque numero di filtri; utilizzando Cloud Firestore invece è possibile creare query concatenando un qualunque numero di clausole where (filtri) e ordinamenti.

6.2.2 Index

La gestione degli indici è un argomento molto critico perché influenza drasticamente le performance dell'intero sistema, avere una corretta implementazione potrebbe tuttavia essere complesso e inefficiente. Molti database relazionali infatti spesso indicizzano i dati automaticamente, Cloud Firestore ricalca questo comportamento generando indici ma consentendo comunque allo sviluppatore di creare tutti gli indici che ritiene necessari. Realtime Database invece crea indici automatici solo sulle chiavi, i quali spesso sono id autogenerati, ma devono essere creati manualmente sui vari campi, questo appunto può facilmente ridurre le performance.

6.3 Scrittura

Essendo i database offerti da Firebase di tipo NoSQL è fondamentale l'uso di Transazioni e Batched writes, tali operazioni permettono di sopperire ai difetti di un sistema non relazionale garantendo le proprietà ACID.

6.3.1 Transazioni

Una transazione rappresenta un insieme di operazioni read e write che permettano di aggiornare dati senza il rischio di collisioni, ad esempio aggiornare il numero di reazioni ad un post non sarebbe possibile in un ambiente dinamico

nel quale diversi utenti potrebbero modificare contemporaneamente lo stesso dato.

È possibile creare transazioni in entrambi i sistemi, queste presentano delle differenze implementative ma non sussistono differenze critiche.

6.3.2 Batch Operation

Una Batch Operation consiste nella scrittura simultanea di diverse parti del database contemporaneamente, tali operazioni sono connesse e devono fallire se almeno una fallisce. Cloud Firestore implementa questa funzionalità nativamente mentre Realtime Database no.

Nello sviluppo del sistema si è incorso spesso nella necessità di aggiornare diversi campi contemporaneamente senza dover passare dalla onerosa Transaction, in queste occasioni è stato sfruttato il comportamento dell'operazione di Update: utilizzando come riferimento un nodo root e andando a creare un array di aggiornamenti (anche su diversi punti dell'albero) è possibile simulare il comportamento di un batch. La differenza sostanziale risiede nell'inserimento dei nuovi campi, un'operazione di update infatti non è in grado di effettuare un push automatico di un nuovo dato, è possibile però richiedere una nuova chiave tramite `push().key` e utilizzare tale chiave per aggiungere nuovi nodi all'albero.

6.4 Security Rules

Le regole di sicurezza costituiscono una parte molto importante per rendere il sistema più stabile e sicuro. Tramite queste è possibile limitare l'accesso a dati sensibili e/o privati in modo del tutto indipendente dalla piattaforma.

6.4.1 Liste di dati (differenza critica)

Una regola di sicurezza sul Realtime Database applicata ad una lista si riflette su tutto il sottoalbero, ciò significa che se l'utente può leggere la lista può accedere anche ai singoli dati senza limiti, su Cloud Firestore invece è possibile specificare una regola sui singoli documenti, se durante una query l'utente cerca di recuperare dati a cui non dovrebbe accedere l'intera query fallisce.

La libertà offerta dal Realtime Database permette di creare alberi del tutto personalizzati in grado di riflettere tutte le regole di sicurezza necessarie, tuttavia questo richiede uno sforzo non indifferente da parte di uno sviluppatore, aumentando le possibilità di bloccare alcune query legittime o di esporre dati privati. Una volta combinate la struttura dei dati e le regole di sicurezza tuttavia si ottiene una protezione solida, affidabile e trasparente per il client.

Il vantaggio principale delle regole di sicurezza in Cloud Firestore è la semplicità di implementazione, queste infatti possono essere applicate indipendentemente dalla struttura dati utilizzata e dalle regole definite nei nodi più alti. Lo svantaggio tuttavia si presenta lato client, è necessario infatti limitare le query in modo adatto per non incorrere in richieste bloccate.

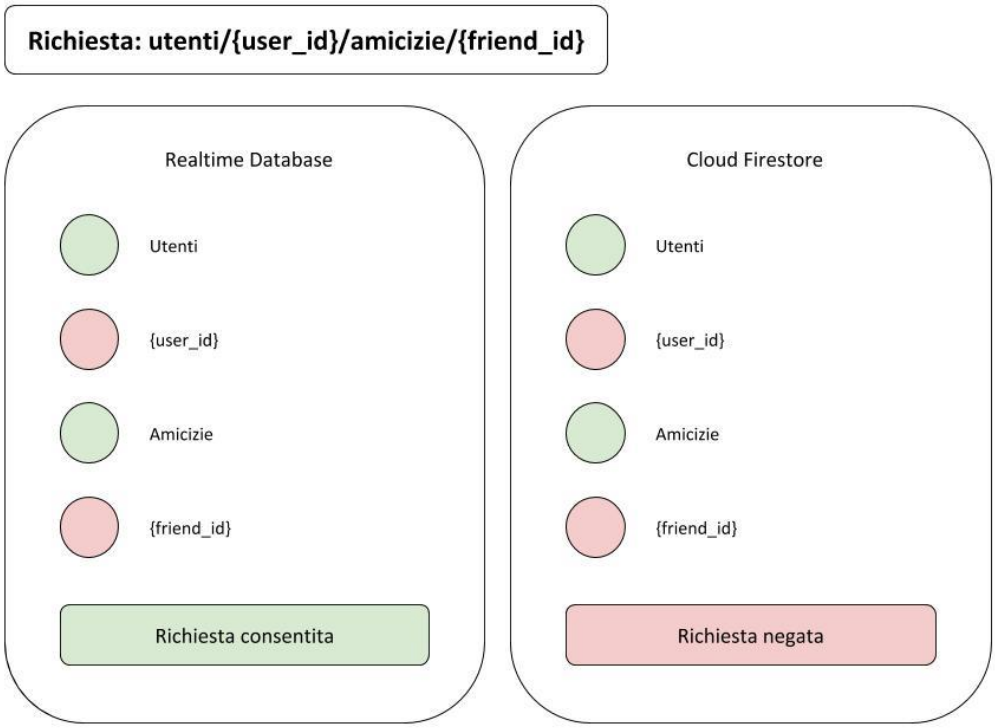

Figura 6 - Regole in cascata

6.5 Scalabilità, stabilità e performance

Sia Realtime Database che Cloud Firestore permettono di creare sistemi scalabili e performanti, il punto focale da tenere a mente è che il primo è in uso da diversi anni e il suo funzionamento è assodato e il comportamento affidabile, il secondo invece è in versione beta al momento della redazione di questo documento, questo rende impossibile avere dei dati comparabili in modo equo fino al rilascio completo del prodotto finale.

7 Struttura dati

In questo capitolo verranno esplorati i punti critici e degni di nota della struttura dati, comparando inoltre le implementazioni tramite *Realtime Database* e *Cloud Firestore*, evidenziandone le differenze.

Durante la migrazione si è cercato di trasporre la struttura dati in funzione del rispetto dei requisiti, efficienza e scalabilità.

7.1 Utenti

Ogni utente è legato univocamente ad un record contenente tutti i dati che lo riguardano tramite un identificativo univoco definito come `UID`.

Quando un utente effettua la registrazione viene inizializzato un record Utente utilizzando l'`UID` generato da Firebase, il record viene popolato con un ruolo di default pari a 10, il `nickname` uguale al `displayName` (recuperato dai dati di autenticazione[3]) e il `tag` uguale al primo numero progressivo disponibile.

L'insieme delle informazioni contenuti direttamente dentro il record dell'utente vengono definite come *dati di base*.

7.1.1 Tag

Gli utenti vengono identificati da un tag, questo è un numero progressivo che viene aggiornato ogni volta che viene richiesto da un utente attraverso il metodo `pushTag()`, il quale sfrutta una *transaction* per recuperare ed aggiornare l'indice `lastTag`.

Semplici regole di sicurezza possono impedire agli utenti di cancellare o diminuire il valore pubblico `lastTag`, in modo da garantire un indice crescente ed univoco.

[3] Quando un utente effettua l'accesso Firebase invia al client una struttura dati contenente diverse informazioni relative all'account come ad esempio l'indirizzo e-mail, il nome utente, la data relativa all'ultimo accesso e diverse altre informazioni.

7.1.2 Iscrizioni

Le iscrizioni sono strettamente legate ad un anno associativo, ogni utente può essere iscritto ad un singolo anno associativo, pertanto ogni elemento della lista di iscrizioni utilizza come ID quello dell'anno associativo corrispondente.

Ogni iscrizione deve contenere principalmente riferimenti alle transazioni relative al tesseramento, la tessera attuale e un'eventuale tessera in attesa di convalida.

L'elenco di iscrizioni è potenzialmente grande quanto il numero di anni associativi, la sua crescita dunque è lenta e le operazioni di ricerca all'interno di essa risulterà sempre poco onerosa.

7.1.3 Profilo

Il profilo rappresenta l'insieme dei dati anagrafici di un utente, questi devono essere convalidati lato gestore, pertanto è necessario sempre differenziare ciò che ha inserito l'utente da ciò che è stato convalidato. La struttura dati per memorizzare i profili risulta dunque molto statica, contenendo al massimo il doppio dei campi anagrafici.

7.1.4 Amici

La lista degli amici lega gli utenti tra loro, i dettagli di un'amicizia sono molto semplici e riguardano lo stato dell'amicizia (richiesta inviata, richiesta ricevuta, amicizia confermata, blocco). Questo sottoalbero cresce linearmente al numero di utenti iscritti, è prioritario dunque implementare correttamente la struttura dati in modo da garantire una buona scalabilità.

7.1.5 Carrello

Gli utenti possono aggiungere Tessere o Biglietti al carrello, è possibile aggiungere una singola tessera alla volta mentre è possibile aggiungere un qualunque numero di biglietti (max. uno per evento). La dimensione del carrello risulta dunque estremamente controllabile, questo perché un utente può registrarsi solo agli eventi aperti e perché i biglietti hanno un tempo di vita limitato, al termine del quale invalidano l'intero carrello svuotandolo.

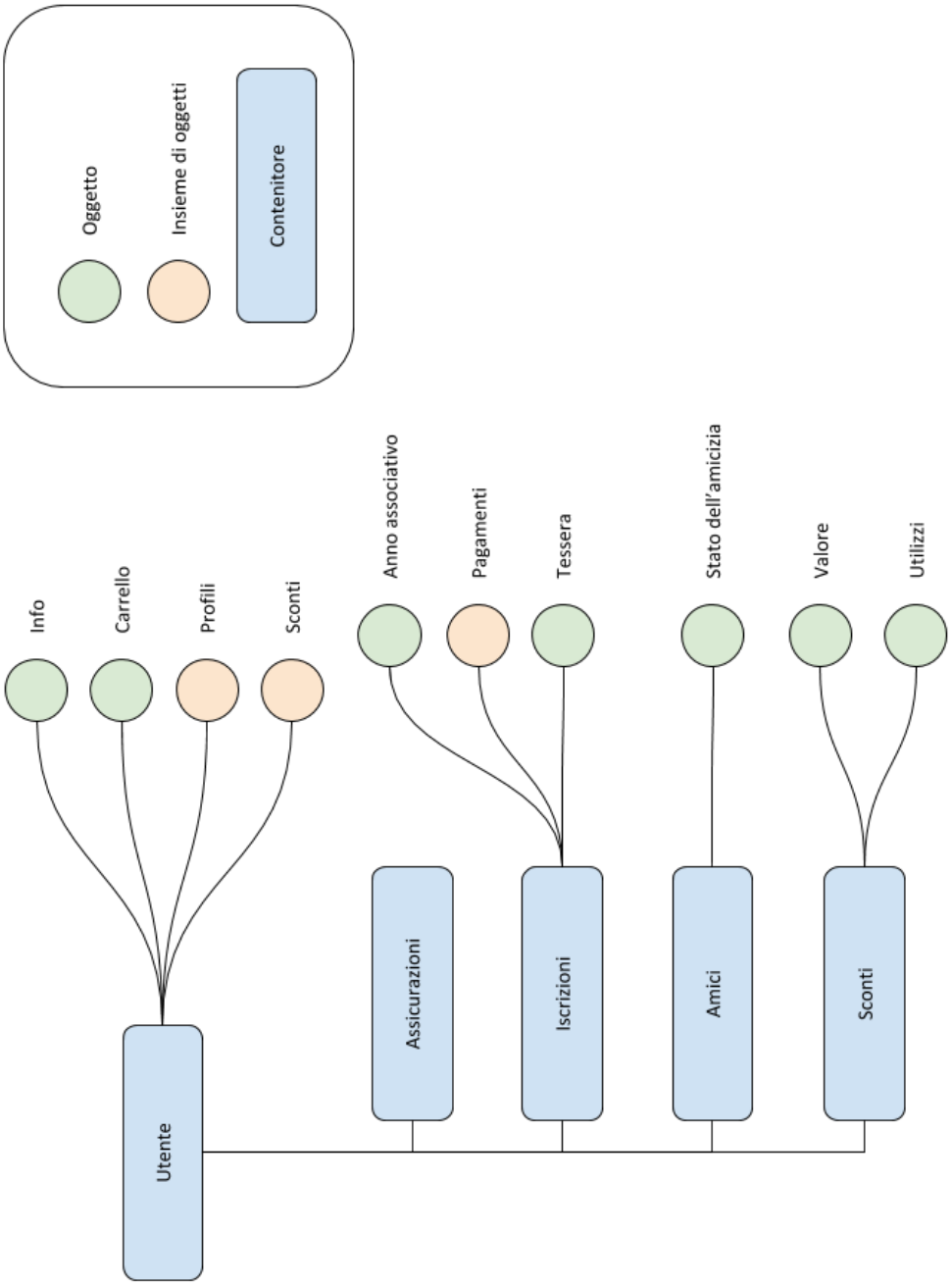

Figura 7 - Struttura di un utente

7.1.6 Realtime Database vs Cloud Firestore

La struttura dati dell'utente è quella su cui vengono effettuate il maggior numero di query durante la navigazione, quindi è di fondamentale importanza progettarla correttamente in modo da garantire una buona scalabilità del sistema.

Utilizzando *Realtime Database* è stato necessario separare le informazioni di base (es. nickname, ruolo) dal resto delle informazioni, considerando che l'unico sottoalbero che potrebbe beneficiare di ulteriore data flattening è quello degli amici, non si è proseguito ulteriormente.

Tramite Cloud Firestore non è necessario denormalizzare la struttura dati in alcun modo: le informazioni di base sono contenute all'interno dello stesso record mentre amici, iscrizioni e altri dati (come il carrello e i profili) sono contenuti in subcollection, in questo modo recuperare l'elenco degli utenti ne recupererà unicamente i dati di base senza i contenuti delle diverse subcollection.

Performance e Scalabilità sono migliori nella struttura di Cloud Firestore ma la differenza è sensibile unicamente nel recupero degli amici di un utente e nelle query con filtri sui dati iniziali degli utenti. Utilizzando Cloud Firestore la denormalizzazione non è necessaria, ciò permette di creare del codice più snello e leggibile, permettendo inoltre di creare regole di sicurezza più comprensibili e meno sensibili ad errori.

7.2 Eventi

Gli eventi rappresentano un altro punto focale dell'applicazione, uno dei principali scopi del sistema è infatti la gestione delle iscrizioni agli eventi da parte degli utenti.

7.2.1 La struttura

La struttura dati degli eventi è relativamente articolata, questa è divisa in cinque sezioni: info, location, preferenze, prezzo, fasce. Una volta creata tuttavia la sua dimensione risulta limitata e poco dinamica.

7.2.1.1 Info

Queste sono le informazioni di base come il titolo dell'evento, la descrizione, la data di inizio, di fine e l'anno associativo a cui è legato; tutte queste informazioni sono poche ma necessarie per creare un elenco di eventi.

7.2.1.2 Location

Vengono inserite qui tutte le informazioni riguardo al luogo in cui si svolge l'evento, in questa sezione vi è un campo di dimensione variabile riguardante l'elenco di province italiane che si trovano in prossimità della location, questo dato è necessario alle logiche per il calcolo del prezzo.

7.2.1.3 Preferenze

Questa è una sezione interamente dinamica, la sua dimensione non è fissa e rappresenta tutte le opzioni che l'utente può o deve selezionare al momento dell'iscrizione. Le preferenze possono essere obbligatorie, con posti limitati e possono influenzare il prezzo del biglietto.

Di norma sono presenti dalle due alle cinque preferenze massime, questo tuttavia non è un limite.

Le preferenze possono essere a posti limitati, questo richiede un'attenta gestione dei lock delle preferenze inserite in quanto queste vengono bloccate unicamente quando l'utente registra il pagamento.

7.2.1.4 Prezzo

Il prezzo di un evento è dinamico, dipende da diversi fattori che possono essere definiti a priori (come l'età di un utente) o in corso di acquisto (come le preferenze selezionate).

I fattori che influenzano il prezzo sono: categoria dell'utente (età, provenienza, disabilità), preferenze, fascia e sconti applicabili.

7.2.1.5 Fasce

Le fasce di pagamento definiscono i periodi di pagamento, questo è il punto principale che aiuta a definire il prezzo.

7.2.2 Biglietti

I biglietti vengono creati al momento del pagamento di un evento, questo contiene semplicemente il prezzo, la fascia, il tipo di biglietto, il riferimento alla transazione e alcune informazioni di controllo. Un utente può avere un solo biglietto per evento.

7.2.3 Realtime Database vs Cloud Firestore

Nella versione originale implementata tramite *Realtime Database* si è scelto di dividere il campo delle info dal resto, in modo tale da ottenere una struttura snella nella creazione dell'elenco degli eventi e chiedendo i dettagli solo in caso di selezione dell'evento.

Cloud Firestore aiuta in questo caso permettendo di creare una subcollection di dettagli in cui inserire tutti gli altri campi. Sarebbe possibile dividere ulteriormente i dati, creando subcollection apposite per preferenze, prezzi e fasce, la cosa porterebbe solo parzialmente dei vantaggi in quanto queste sezioni, per quanto dinamiche, non raggiungono mai una dimensione elevata.

Un aspetto importante della gestione degli eventi riguarda invece i biglietti, questi infatti, nel data model del *Realtime Database,* sono memorizzati in alberi separati, referenziati solo grazie all'id dell'evento stesso, tramite *Cloud Firestore* invece è stato possibile creare una semplice subcollection all'interno dello stesso evento contenente tutti i dati relativi ai biglietti emessi.

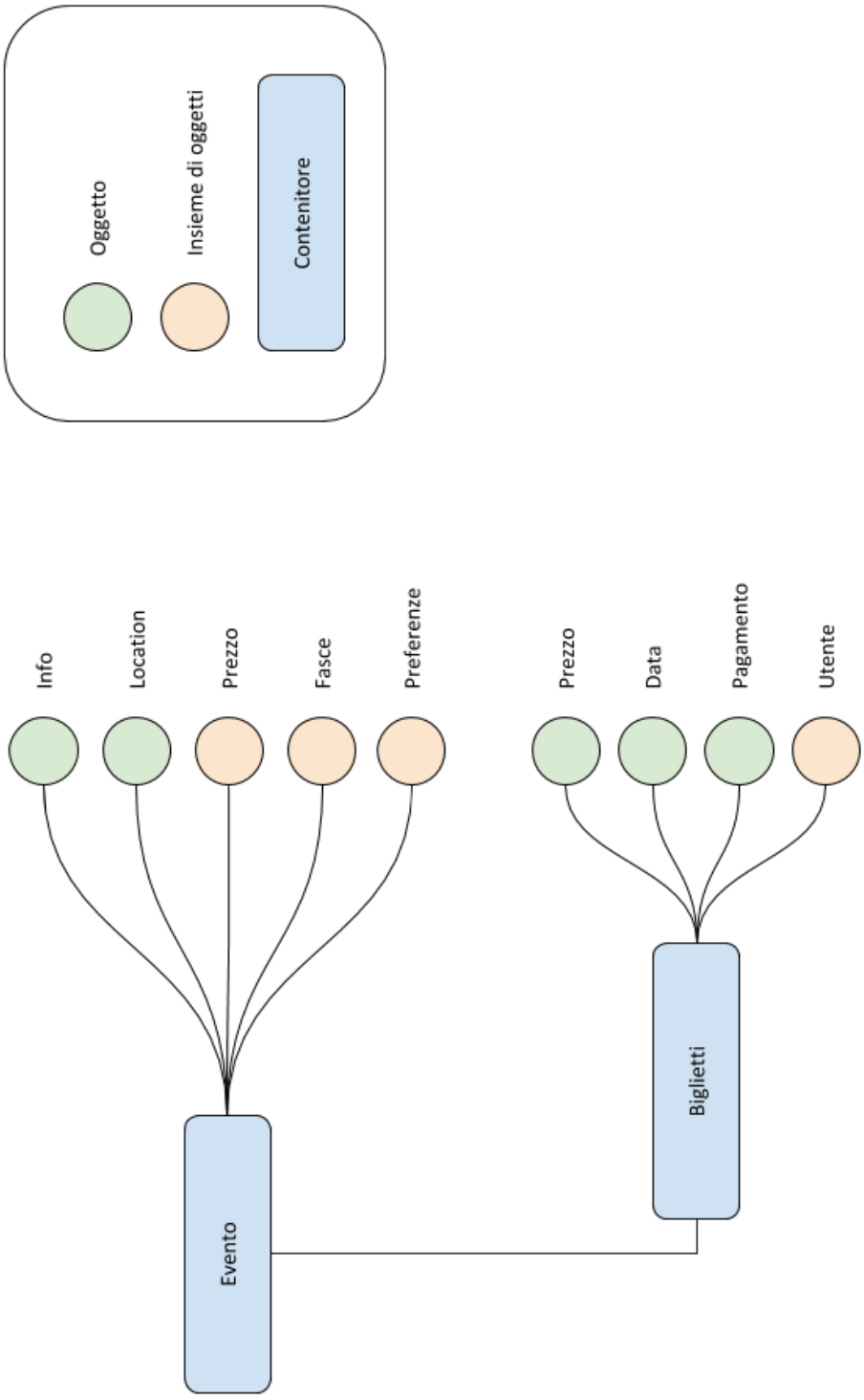

Figura 8 - Struttura di un evento

7.3 Pagamenti

Ogni pagamento è identificato da una transazione, questa struttura ricalca quella del carrello includendo in più dati riguardanti allo stato della transazione e ai dettagli del pagamento.

I pagamenti vengono gestiti anche utilizzando due strutture dati per la gestione dei movimenti di denaro interni al *Consiglio Direttivo* e/o ad eventuali gruppi organizzativi.

7.3.1 Autorizzati

Le entità in grado di accettare pagamenti sono identificate come autorizzati, questi possono essere creati dall'applicazione e rappresentano principalmente membri dell'associazione in carico di ricevere denaro o contenitori generici di denaro (Banca, PayPal). Ogni autorizzato deve contenere un nome, data di creazione e flag di staticità, quest'ultimo in particolare indica che l'autorizzato è in grado di "generare" denaro, ovvero può accogliere *movimenti* senza origine.

7.3.2 Movimenti

Gli spostamenti di denaro interni sono identificati dai movimenti, questi servono principalmente a gestire il saldo di un autorizzato, che appunto potrebbe ad esempio effettuare un bonifico delle quote da lui raccolte, ma sono utili anche a tracciare il cash flow. I movimenti devono avere un autorizzato di destinazione, un importo e un autorizzato sorgente, quest'ultimo è opzionale nel caso in cui l'autorizzato di destinazione sia statico.

7.3.3 Transazioni

Le transazioni sono generate in automatico al momento di un checkout, queste nascono aperte (stato *open*) e deve essere un gestore ad assegnarla ad un autorizzato accettandola (stato *closed*), o a rifiutarla (stato *rejected*) invitando l'utente a provvedere al nuovo pagamento (o alla sua corretta registrazione).

7.3.4 Realtime Database vs Cloud Firestore

Il data model in entrambi i casi è simile: non sono richieste denormalizzazioni in quanto ogni transazione è di dimensioni ridotte, tuttavia la dimensione della lista delle transazioni ha una velocità di crescita proporzionale sia al numero di utenti che al numero di eventi che al numero di anni associativi.

Considerando il rateo di crescita delle transazioni Cloud Firestore è dunque il database che permette un'efficacia migliore per quanto riguarda performance e scalabilità, soprattutto una volta conclusa la Beta.

7.4 Social

Il sistema della chat è suddiviso tra discussioni pubbliche e discussioni private, le prime sono accessibili a chiunque mentre le seconde sono chat tra due o più persone. Ogni discussione deve contenere la lista dei messaggi e poche altre informazioni (data di creazione, titolo e descrizione della discussione), il che le rende delle strutture dati molto semplici ma al tempo stesso incredibilmente ampie.

7.4.1 Realtime Database vs Cloud Firestore

La struttura dati implementata sul *Realtime Database* è stata suddivisa tra la lista (contenente solo titolo, descrizione e creatore) e i dettagli, all'interno del quale vengono memorizzati messaggi e, nel caso di discussioni pubbliche, l'elenco degli utenti autorizzati a visualizzare la discussione.

In fase di migrazione la struttura è stata trasposta creando dunque una Collection per le discussioni pubbliche ed una per le discussioni private

7.4.1.1 Discussioni Private

La differenza più grande tra le due implementazioni risiede proprio nelle discussioni private. L'implementazione su *Realtime Database* è simile a quella utilizzata nelle discussioni pubbliche, ovvero separando lista e dettagli, tuttavia la lista delle discussioni è articolata diversamente: nella lista è presente un record per ogni utente, indicizzato per ID utente; quando due utenti diventano amici

viene inizializzata la loro discussione privata, questo viene fatto creando un record con ID autogenerato nel sottoalbero dei dettagli, al tempo stesso vengono creati altri due record per connettere l'ID della discussione ad entrambi gli utenti. Ad esempio: supponiamo che l'utente A (idA) invii una richiesta di amicizia all'utente B (idB), B accetta e i due vengono registrati come amici; viene inizializzata allora la loro discussione privata, questo viene fatto creando una nuova discussione al path `social/private/details/<idD>` (dove idD è autogenerato) e questa viene referenziata ai path `social/private/list/idA/idB` e `social/private/list/idB/idA` in modo tale che entrambi gli utenti possono recuperare velocemente la discussione relativa ad uno specifico utente.

Il motivo di questa struttura risiede nel fatto che le regole di sicurezza del Realtime Database vanno in cascata e limitando la visibilità di un singolo record non permetterebbe ad un utente di effettuare una query sui dati, seppur filtrata lato client per vedere il sotto-elenco a cui avrebbe accesso.

Per proteggere i dati in modo adeguato esistono due possibili opzioni:

- Applicare le regole di sicurezza unicamente ai dettagli, in questo modo la lista del mapping delle discussioni risulterebbe di libero accesso e tutti potrebbero visualizzarne i contenuti, lato client sarebbe necessario implementare filtri corretti per ridurre la visibilità dall'applicazione.
- Strutturare i dati per singolo utente, questa è la soluzione che è stata adottata, le regole di sicurezza in questo modo possono essere applicate al sottoalbero dell'utente e lato client non sono necessari particolari filtri.

L'implementazione tramite Cloud Firestore ha seguito l'approccio opposto, questo perché la combinazione di filtri e regole di sicurezza permette di ridurre la visibilità dei singoli record permettendo comunque di elencare il sottoinsieme di oggetti su cui l'utente ha visibilità. In aggiunta è stato possibile non denormalizzare la struttura dati, questa infatti contiene la lista dei messaggi in una subcollection mentre i dati di base sono inseriti all'interno del documento stesso.

L’approccio utilizzato su Cloud Firestore permette inoltre di implementare facilmente chat di gruppo private, la struttura dati infatti è stata progettata proprio per accogliere una modifica del genere senza dover modificare le regole di sicurezza.

8 AngularJS

L'applicazione lato front end è stata sviluppata tramite AngularJS realizzando una Single Page Application. Seguendo il paradigma MVC è possibile identificare:

Il Model è formato dall'insieme dei dati ricevuti e dei dati generati.

La View è formata dall'insieme di pagine HTML i cui componenti comunicano con il Controller tramite i tag e gli attributi di Angular, accede al Model solo tramite il Controller.

Il Controller è organizzato tramite un controllore principale e l'implementazione di Servizi, ogni servizio ha un ambito assegnato in modo tale da rendere il più possibile separati gli ambiti d'azione anche se non è raro che i Servizi comunichino tra di loro.

8.1 Scope

Il punto focale del funzionamento di AngularJS è lo $scope: punto di connessione tra le pagine HTML (view) e il codice JavaScript (controller). Lo $scope può essere suddiviso in sotto parti per limitare l'accesso delle risorse, tutti gli *$scope* derivano dal *$rootScope* attraverso il quale è possibile accedere a tutti gli *$scope* esistenti.

8.1.1 Aggiornamento dinamico

I dati presenti nello $scope possono essere modificati dagli input inseriti dall'utente o dal controller stesso, ogni modifica viene rilevata automaticamente e la view risulta sempre aggiornata.

8.1.2 Eventi

È possibile inviare e ricevere segnali attraverso i vari *$scope*, questo risulta utile in quei casi in cui gli eventi devono essere rilevati da diversi punti del sistema o da punti semplicemente non accessibili. Ad esempio, nell'applicazione finale viene sollevato un evento nel momento in cui l'autenticazione cambia, questo deve poter essere rilevato da tutti i punti dell'applicazione in modo diretto e immediato.

Le due metodologie per propagare gli eventi sono $emit e $broadcast, per catturare un evento si utilizza $on.

8.1.2.1 $emit

Invia l'evento a tutti i genitori dello $scope, questo è utile quando bisogna segnalare un cambiamento di stato da parte di una foglia del $rootScope.

8.1.2.2 $broadcast

Invia l'evento a tutti i figli dello $scope, risulta utile per aggiornare i figli su di una modifica effettuata ad alto livello, si può utilizzare $broadcast in congiunzione con il $rootScope per inviare l'evento a tutti gli $scope.

8.1.2.3 $on

Imposta l'ascolto dell'evento specificato sullo $scope, è possibile catturare solo gli eventi visibili a questo determinato livello.

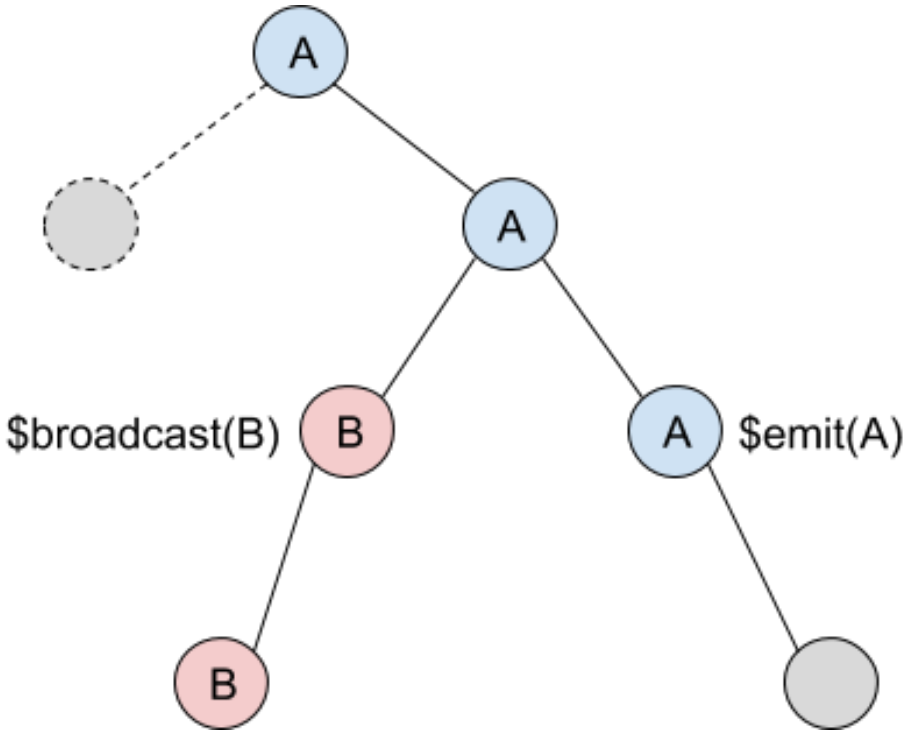

Figura 9 - Emissione e ricezione dei segnali

8.2 Servizi

I diversi ambiti dell'applicazione *Blooming Harp* sono gestiti dai Servizi, ognuno di essi è stato infatti progettato in modo da gestire un singolo ambito in modo da centralizzare il flusso e rendere modifiche successive molto più intuitive.

8.2.1 Authentication Service

Questo servizio gestisce tutte le interazioni con il sistema di autenticazione: espone dei metodi che permettono di operare nell'ambito dell'autenticazione, nello specifico permette di effettuare l'autenticazione, chiudere una sessione, registrare un account e recuperare i dati relativi alla sessione corrente.

8.2.1.1 Stato di una sessione

Quando lo stato dell'autenticazione cambia, il servizio fa in modo che i dati di sessione vengano puliti e valorizzati nel modo corretto, questo processo avviene automaticamente in modo isolato dal resto del sistema. Il cambiamento di stato viene catturato tramite i metodi disponibili nella libreria Firebase.

8.2.1.2 Operazioni sull'account

La creazione di un account è gestita quasi del tutto interamente da Firebase, attraverso il quale è possibile creare, sospendere e cancellare un account, insieme a tutte le funzionalità di supporto come il recupero delle credenziali di accesso e la convalida di un indirizzo e-mail.

Il servizio mette a disposizione i metodi utilizzati dalle interfacce che permettono all'utente di effettuare le operazioni sul proprio account senza l'intervento di un amministratore.

8.2.2 Cart Service

Il carrello è la parte più sensibile dell'applicazione, questo richiede una buona sincronizzazione con il database e dei controlli rilevanti, questo ha reso il codice talmente importante ed esteso da diventare un servizio a sé stante. Il Database Service (insieme al Validation Service) avrebbe potuto includere concettualmente tutto il codice di questo servizio.

8.2.2.1 Formattazione dei dati

Per poter aggiungere un oggetto nel carrello, questo deve essere formattato correttamente, necessita di informazioni particolari che richiedono una breve elaborazione, pertanto il servizio riceve in input i dati inseriti dall'utente per poi creare l'oggetto senza creare errori lato Server.

8.2.3 Database Service

Questo servizio ha il compito di comunicare con il Realtime Database di Firebase, ogni richiesta passa da questo servizio ed è l'unico punto di contatto con il database.

Forzando una convergenza su questo servizio è possibile gestire in modo più fluido e pulito il traffico verso il database, il resto del Client non è cosciente di come si debba comunicare con il database ma richiama le funzioni esposte dal servizio passando i dati necessari.

8.2.3.1 Watch

Quando l'applicazione deve richiedere dei dati tramite on(), ovvero ha bisogno di ascoltare tali dati nei loro cambiamenti, non lo fa direttamente ma utilizza un suo metodo watch(). Questo metodo chiama a sua volta on() ma prima memorizza in una mappa il dato ascoltato, in questo modo è possibile liberare la risorsa quando non è più necessaria. watch() permette anche di specificare che una risorsa è persistente, pertanto deve essere liberata manualmente. Gli ascolti istanziati tramite watch() possono essere interrotti tramite il metodo stopWatch(), è possibile inoltre liberare tutti gli ascolti con un'unica chiamata utilizzando stopWatches(), entrambi i metodo non interrompono di default gli ascolti persistenti, tale comportamento deve essere forzato.

8.2.3.2 Server Time

Quando è necessario inserire un timestamp sul database è possibile utilizzare la metodologia descritta in Strumenti lato Client, quando invece è necessario effettuare un controllo lato Client su di una data è possibile richiedere al Server il proprio Timestamp attraverso questo servizio.

Sono esposte due funzioni che permettono di recuperare il Server Time: la prima invia una richiesta al Server e sincronizza in una variabile locale l'offset temporale tra il Client e il Server, la seconda ritorna il Server time applicando l'offset memorizzato al timestamp corrente. Entrambe le funzioni non ritorneranno mai un tempo accurato ma ci sarà sempre uno scarto di alcuni millisecondi, inoltre ogni parte del Client può scegliere arbitrariamente di richiamare sia la prima che la seconda versione.

8.2.4 Navigation Service

I menu da visualizzare sono creati dinamicamente, il livello di accesso, nome e link di ogni item vengono definiti in questo servizio ed elaborati poi dalla view. È concesso creare un menu all'interno di un item, la view stessa è in grado di gestire il nesting dichiarato fornendo gli strumenti necessari alla navigazione.

Alcune variabili necessarie alla navigazione vengono inizializzate/istanziate in questo servizio, queste risultano accessibili a tutti gli altri servizi.

8.2.5 Validation Service

Tutti i controlli lato Client sono registrati in questo servizio, è possibile convalidare preventivamente i dati prima di inviarli al database in modo da rilevare alcuni errori ed evitare traffico inutile. La maggior parte dei controlli ricalcano quelli registrati lato Server attraverso le Security Rules, anche se per lo più risultano meno stretti e sono applicabili solo al dato da inviare in sé, in quanto risulterebbe superfluo effettuare una convalida dei dati sulle relazioni con l'intero albero (convalida che viene effettuata lato Server).

9 Conclusioni

Questo progetto ha permesso di valutare le prestazioni di *Firebase* in un ambiente reale, ne consegue che è stato possibile carpire vantaggi e svantaggi sull'utilizzo di tale piattaforma.

Il punto di forza più grande di *Firebase* consiste nell'aspetto realtime: creare un sistema responsivo e costantemente aggiornato è ciò per cui la piattaforma è stata progettata. I mezzi messi a disposizione sono semplici e numerosi, l'implementazione dei listener è immediata e la loro gestione è quasi del tutto a carico delle librerie disponibili lasciando comunque allo sviluppatore un'ottima libertà nella progettazione della struttura dati.

Creare un sistema realtime basato su Database relazionali non è immediato, richiede il supporto di software terzi che potrebbero introdurre, nella maggior parte dei casi, riduzioni di performance o perdita in scalabilità, senza considerare tutte le difficoltà implementative.

Esistono vantaggi dal punto di vista economico per realtà di piccole e medie dimensioni, permettendo loro di avere un sistema stabile, scalabile e responsivo investendo un capitale proporzionale all'utilizzo[4] del sistema.

La scalabilità è un altro fattore fondamentale: la stessa applicazione può scalare in modo automatico senza richiedere un intervento lato codice, semplificando drasticamente la gestione di crescite improvvise nell'utilizzo dell'applicativo.

Esistono diverse altre soluzioni NoSQL in grado di supportare sistemi realtime, queste tuttavia non sono state esplorate in profondità durante la stesura di questo documento. Sotto questo aspetto *Firebase* tuttavia mette a disposizione diverse funzionalità limitrofe come *Hosting*, *Atuhentication* e *Storage* che

[4] Tra le opzioni disponibili per le metodologie di pagamento vi è la possibilità di pagare *as you go*, il cui costo dipende strettamente dall'utilizzo del sistema.

permetterebbero, in numerosi casi, agli sviluppatori di interfacciarsi con un'unica piattaforma per gestire l'aspetto back end della propria applicazione.

I sistemi basati su *Firebase* presentano tuttavia un grosso limite nelle query, queste infatti mancano di operatori basilari come LIKE e OR impedendo un'implementazione pulita nel caso in cui siano richieste query più complesse. Esistono due soluzioni al problema: appoggiarsi ad alcuni framework terzi come *ElasticSearch*, il quale nello specifico è perfettamente integrato e supportato da *Firebase*, in grado di estendere le potenzialità delle query; oppure la creazione di *Cloud Function* custom adattabili ai casi più specifici, quest'ultima soluzione potrebbe essere rischiosa in quanto graverebbe sullo sviluppatore l'onere di creare algoritmi efficienti le cui prestazioni si rifletterebbero potenzialmente sull'intero sistema.

9.1 Realtime Database e Cloud Firestore

Realtime Database è in grado di fornire quasi tutti gli stessi servizi di Cloud Firestore, per farlo però è spesso necessario un effort maggiore sia nella progettazione che nello sviluppo di un applicativo.

Il secondo inoltre è il sistema più recente, pertanto è plausibile che riceva un supporto più a lungo rispetto al suo predecessore.

9.2 Quando scegliere Firebase

Quando si deve creare un'applicazione ex-novo nella quale le funzionalità realtime ricoprano una mansione fondamentale, questo si verifica in molte delle applicazioni più recenti, nelle quali il focus è più l'interazione tra gli utenti complesse logiche di business.

Quando si vuole creare una piccola o media applicazione con una business logic non troppo articolata, questo deriva dal fatto che non esiste un back end in grado di operare in maniera complessa, gran parte della logica deve essere implementata lato Client.

In particolar modo *Firebase* offre un supporto estensivo per il mondo mobile.

9.3 Quando non scegliere Firebase

Nella migrazione cross sistema a partire da una realtà già avviata che si appoggia ad un database relazionale:

Il ribaltamento di un database relazionale contenente una discreta quantità di record risulterebbe gravosa, tale operazione richiederebbe diversi script ad hoc in grado di popolare il nuovo database, la cosa è ovviamente realizzabile ma richiede sforzi di progettazione non indifferenti, dato che replicare la struttura di un database relazionale richiederebbe anche l'implementazione di pesanti regole di sicurezza o l'introduzione di logiche lato Client.

L'applicazione presenta business logic articolate, tutte le logiche dovrebbero essere implementate lato Client, appesantendo l'applicazione stessa rendendola poco utilizzabile. In particolar modo se l'applicazione non riesce a trarre beneficio dall'aspetto realtime di *Firebase*.

Sitografia

AngularFire, Google Inc., https://github.com/angular/angularfire2
Data di ultima consultazione: 12/07/2018

AngularJS, Google Inc., https://angularjs.org/
Data di ultima consultazione: 12/07/2018

Oretti, F., Andrei, D., Angular-tooltips,
https://github.com/720kb/angular-tooltips
Data di ultima consultazione: 12/07/2018

AngularUI bootstrap, AngularUI Team, https://angular-ui.github.io/bootstrap/
Data di ultima consultazione: 12/07/2018

Rowls A., Bootstrap-datepicker, https://bootstrap-datepicker.readthedocs.io
Data di ultima consultazione: 12/07/2018

Bootstrap, Bootstrap Development Team, https://getbootstrap.com/docs/3.3/
Data di ultima consultazione: 12/07/2018

Firebase, Google Inc., https://firebase.google.com/
Data di ultima consultazione: 12/07/2018

Font Awesome, Font Awesome Team, https://fontawesome.com/
Data di ultima consultazione: 12/07/2018

Geonames, http://www.geonames.org/
Data di ultima consultazione: 12/07/2018

Alman, B., Grunt, https://gruntjs.com/
Data di ultima consultazione: 12/07/2018

Von Hertzen, N., html2canvas, https://html2canvas.hertzen.com/
Data di ultima consultazione: 12/07/2018

JQuery, the JQuery Team, https://jquery.com/
Data di ultima consultazione: 12/07/2018

NodeJS, Joyent Inc., https://nodejs.org/
Data di ultima consultazione: 12/07/2018

NPM Inc., https://docs.npmjs.com/
Data di ultima consultazione: 12/07/2018

Bpampuch, pdfmake, http://pdfmake.org/#/
Data di ultima consultazione: 12/07/2018

Visual Studio Code, Microsoft, https://code.visualstudio.com/
Data di ultima consultazione: 12/07/2018