Alma Mater Studiorum · Università di Bologna

**SCUOLA DI SCIENZE**

**Corso di Laurea in Informatica**

# EVENTUALLY: PROGETTAZIONE E SVILUPPO DI UNA APPLICAZIONE PER GESTIRE EVENTI

**Relatore:**
**Chiar.mo Prof.**
**LUCIANO BONONI**

**Presentata da:**
**SAMUELE EVANGELISTI**

**Sessione II - Primo Appello**
**Anno Accademico 2018/2019**

# Introduzione

Al giorno d'oggi la diffusione delle informazioni è pressochè immediata. Innumerevoli sono le piattaforme che operano nella gestione di queste informazioni. Il risultato è che in ogni momento possiamo venire a conoscenza di avvenimenti storici, fatti di cronaca ma anche informazioni personali di altre persone in maniera estremamente semplice.
Quando spostiamo l'attenzione sugli eventi (di seguito il termine "evento" sarà utilizzato per indicare raggruppamenti di persone) la diffusione delle informazioni relative a questi diventa meno incalzante, soprattutto se si parla di eventi di piccole dimensioni.
Le attuali piattaforme che forniscono una gestione delle informazioni relative agli eventi presentano dei difetti che riguardano principalmente la visibilità di questi e la difficoltà nella ricerca delle informazioni correlate. Quindi non si ha a disposizione un servizio dedicato in grado di valorizzare anche eventi locali e di piccole dimensioni.
Eventually nasce con l'intento di fornire una piattaforma semplice in grado di gestire e diffondere informazioni relative a eventi. È quindi possibile creare e cercare eventi, gestire liste di amici e di invitati ma soprattutto conoscere gli eventi nelle vicinanze.
Questo progetto nasce e si sviluppa insieme al mio collega, Dott. Gianluca Iacchini, pertanto questa tesi riporterà il mio contributo personale, vale a dire lo sviluppo dell'applicazione in ambiente Android e lo sviluppo del database per la gestione dei dati.

# Indice

# Elenco delle figure

# Capitolo 1

# Stato dell'arte

## 1.1 Facebook

Facebook è al momento il social network con il maggior numero di utenti attivi [1] ed è anche quello che offre la gestione più dettagliata riguardo le informazioni degli eventi. Infatti è possibile creare eventi, sia pubblici che privati, gestirne le informazioni come titolo, luogo e immagine di copertina, gestire la lista di invitati e gestire la lista degli eventuali organizzatori aggiuntivi.
Tuttavia quando si parla di ricercare un evento le opzioni sono:

1. ricerca degli eventi per nome

2. ricerca degli eventi tra gli amici che vi partecipano

3. ricerca degli eventi per posizione geografica

È facile rendersi conto dell'inefficacia di tali ricerche. Infatti se per i punti 1 e 2 l'utente deve già essere a conoscenza dell'evento per il punto 3 la difficoltà diventa la politica di Facebook che fornisce molta più visibilità eventi di grandi dimensioni anche se questi sono molto lontani dall'utente che effettua la ricerca. È comunque possibile applicare dei filtri alla ricerca ma con l'effetto di renderla più tediosa.

## 1.2 Snapchat

Snapchat fornisce un supporto minimo in ambito di eventi. Infatti è possibile, da parte dei creatori di un evento, generare un filtro personalizzato che i partecipanti possono applicare come *overlay* alle proprie foto o ai propri video. I filtri sono disponibili solo per determinati periodi di tempo e in determinate aree geografiche quindi la principale limitazione deriva dal fatto che usufruiscono del contenuto solo gli utenti effettivamente presenti e che sono già a conoscenza dell'evento.
Per quanto riguarda gli organizzatori che decidono di creare un filtro ci sono due aspetti considerevoli da mettere in conto:

- Il prezzo del filtro, che non è gratuito, dipende dall'estensione dell'area geografica e dalla durata del periodo di tempo per i quali sarà disponibile
- Il filtro diventa utilizzabile dagli utenti in seguito all'approvazione da parte di Snapchat quindi è necessario organizzare l'evento con un discreto anticipo

## 1.3 WhatsApp, Telegram

Queste due piattaforme non forniscono alcun supporto per la gestione di eventi. Tuttavia non è raro trovare gruppi creati ad hoc per agevolare la comunicazione tra i partecipanti o gli organizzatori di un evento nonchè per trasmettervi file multimediali correlati.

## 1.4 Eventbrite

Eventbrite è un'applicazione specializzata nella gestione di eventi. Si presenta con un'interfaccia semplice. Vengono fornite delle categorie, che l'utente può indicare tra i preferiti, per dare maggiore visibilità ad alcuni eventi che potrebbero interessare maggiormente. Gli eventi vengono ricercati utilizzando la posizione dell'utente.
Tuttavia l'applicazione non permette la creazione di eventi se non utilizzando un'altra

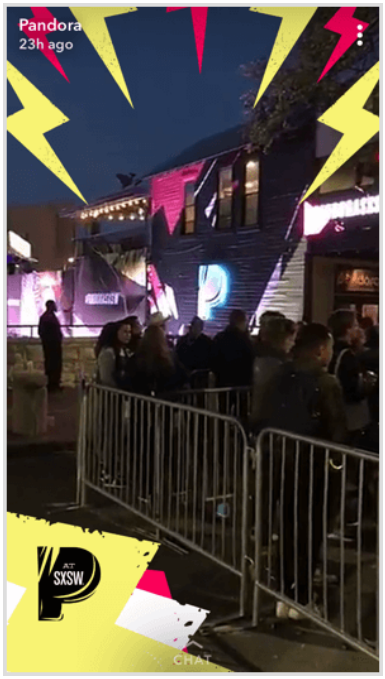



Figura 1.1: Filtro di Snapchat [2]

applicazione, *Eventbrite Organizer*. Non è comunque possibile l'interazione tra utenti (l'applicazione è stata installata e testata in data 18/09/2019).

## 1.5 Allevents.in

Allevents.in è un'altra applicazione specializzata nella gestione di eventi. Come nel caso precedente è possibile, da parte dell'utente, selezionare delle categorie per far sì che l'applicazione dia più visibilità a eventi di maggiore interesse. Un aspetto aggiuntivo presente in questa applicazione è quello di poter "seguire" le pagine che creano gli eventi, rimanendo così aggiornati sulle loro attività (l'applicazione è stata installata e testata in data 18/09/2019).

# Capitolo 2

# Progettazione

Seguendo le linee guida dello stile *agile* sono state definite poche specifiche ma precise, dando importanza alla realizzazione nel minor tempo possibile di una prima versione, già funzionante, del prodotto sulla quale apportare i giusti miglioramenti in base alle richieste degli utenti.
Ipotizzando di dover gestire grandi quantità di dati e di richieste in contemporanea il database è stato sviluppato con particolare attenzione verso la struttura dell'API sviluppata dal mio collega. Infatti le relazioni sono state costruite in modo da trovare i dati delle principali richieste già presenti nello stesso *record*.

## 2.1 Specifiche

Le specifiche che l'applicazione deve sicuramente soddisfare sono le seguenti:

- Gli utenti devono potersi registrare
- Gli utenti devono poter eliminare il proprio profilo
- Gli utenti devono poter creare eventi
- Gli utenti devono poter modificare i propri eventi
- Gli utenti devono poter eliminare i propri eventi
- Gli utenti devono poter invitare amici ai propri eventi
- Gli utenti devono poter aggiungere organizzatori ai propri eventi
- Gli utenti devono poter esprimere il loro stato di partecipazione agli eventi
- Gli utenti devono poter visualizzare i propri eventi
- Gli utenti devono poter visualizzare gli eventi nelle vicinanze
- Gli utenti devono poter cercare i propri amici
- Gli utenti devono poter aggiungere i propri amici
- Gli utenti devono poter rimuovere gli amici
- Gli eventi devono poter conoscere le persone presenti in un dato istante

### 2.1.1 Diagramma dei casi d'uso

La figura 2.1 mostra il diagramma dei casi dei casi d'uso.

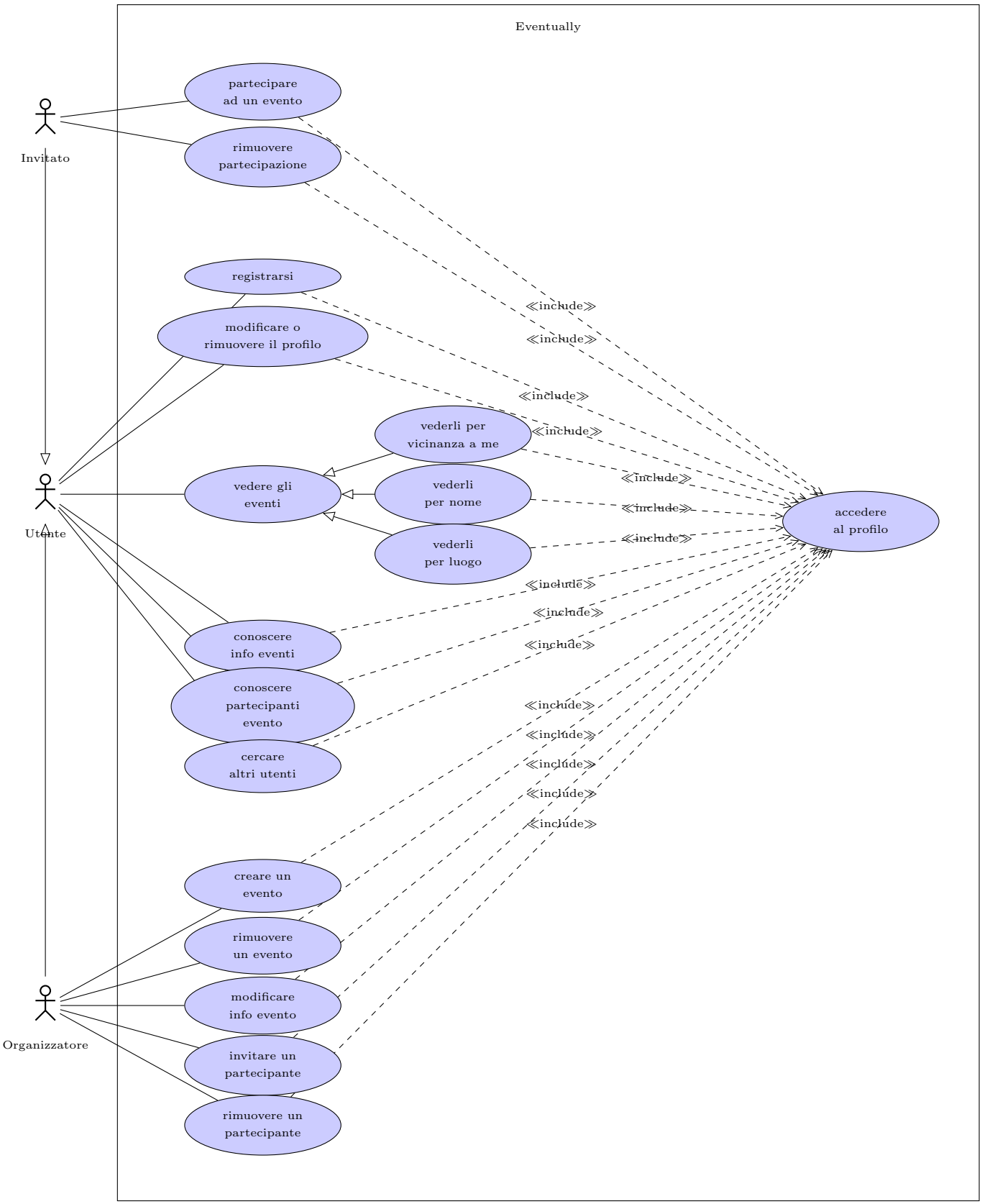



Figura 2.1: Diagramma dei casi d'uso

## 2.2 Applicazione Mobile

Le entità base, che rappresentano i componenti principali dell'applicazione, sono le seguenti:

**User** : Entità che rappresenta un utente dell'applicazione

**Event** : Entità che rappresenta un evento nell'applicazione

**ListManager** : Entità delegata alla gestione e alla persistenza delle liste di eventi e utenti

**CommunicationManager** : Entità delegata alla gestione delle comunicazioni con il server

### 2.2.1 User

Contiene le informazioni personali dell'utente. Oltre a queste sono presenti anche due dati chiamati *trust*:

**Organization Trust** : Affidabilità dell'utente come organizzatore di eventi, un valore elevato indica che gli eventi creati sono realmente presenti, viceversa un valore basso indica che gli eventi creati sono falsi e l'utente potrebbe essere rimosso dal servizio. Questo valore serve sia agli organizzatori, che sono spinti ad utilizzare il servizio di creazione eventi in maniera consapevole, sia agli utenti, che possono verificare l'affidabilità di certi organizzatori

**Partecipation Trust** : Affidabilità dell'utente come partecipante ad un evento, un valore elevato indica che l'utente è affidabile e prende parte agli eventi per i quali indica una partecipazione, viceversa un valore basso indica che lo stato di partecipazione dell'utente non è affidabile

Un utente può agire, in base alla relazione con un dato evento, in tre modi:

**Owner** : È il creatore dell'evento, può modificare l'evento, aggiungere invitati e organizzatori e può anche eliminare l'evento

**Organizer** : È un organizzatore aggiuntivo dell'evento, può modificare l'evento, aggiungere invitati ma non può eliminare l'evento

**Partecipant** : È un utente che prende parte ad un evento, non ha nessun potere di gestione dell'evento ma a differenza dei primi due, che necessariamente partecipano all'evento, questo può indicare il suo stato di partecipazione all'evento

### 2.2.2 Event

Contiene tutte le informazioni necessarie per identificare un evento. In particolare, utilizzando l'API di *Google Maps* al momento della creazione, viene identificato anche il posto nel quale si svolge l'evento per poter poi calcolare in tempo reale il numero di persone presenti. Infine è presente anche la relazione tra l'evento e l'utente che effettua la richiesta per conoscerne le informazioni in modo da stabilire le azioni che questo utente sia in grado di effettuare sull'evento.

### 2.2.3 ListManager

Si occupa di gestire e rendere persistenti le liste di eventi e di utenti. All'interno dell'applicazione sono presenti due liste eventi, una per gli eventi nelle vicinanze e una per gli eventi relativi all'utente, e una lista amici. Queste liste possono essere accedute in tempi e in modi differenti durante l'utilizzo dell'applicazione. ListManager si occupa di salvare le liste quando vengono richieste in modo da non dover di nuovo richiederle al server in caso di richieste successive.

### 2.2.4 CommunicarionManager

Si occupa della comunicazione con il server e della gestione delle risposte. In particolare viene gestito il cookie, i JSON e le immagini che vengono inviati dal server in risposta a determinate richieste.

### 2.2.5 Diagramma delle classi

La figura 2.2 rappresenta il diagramma delle classi.

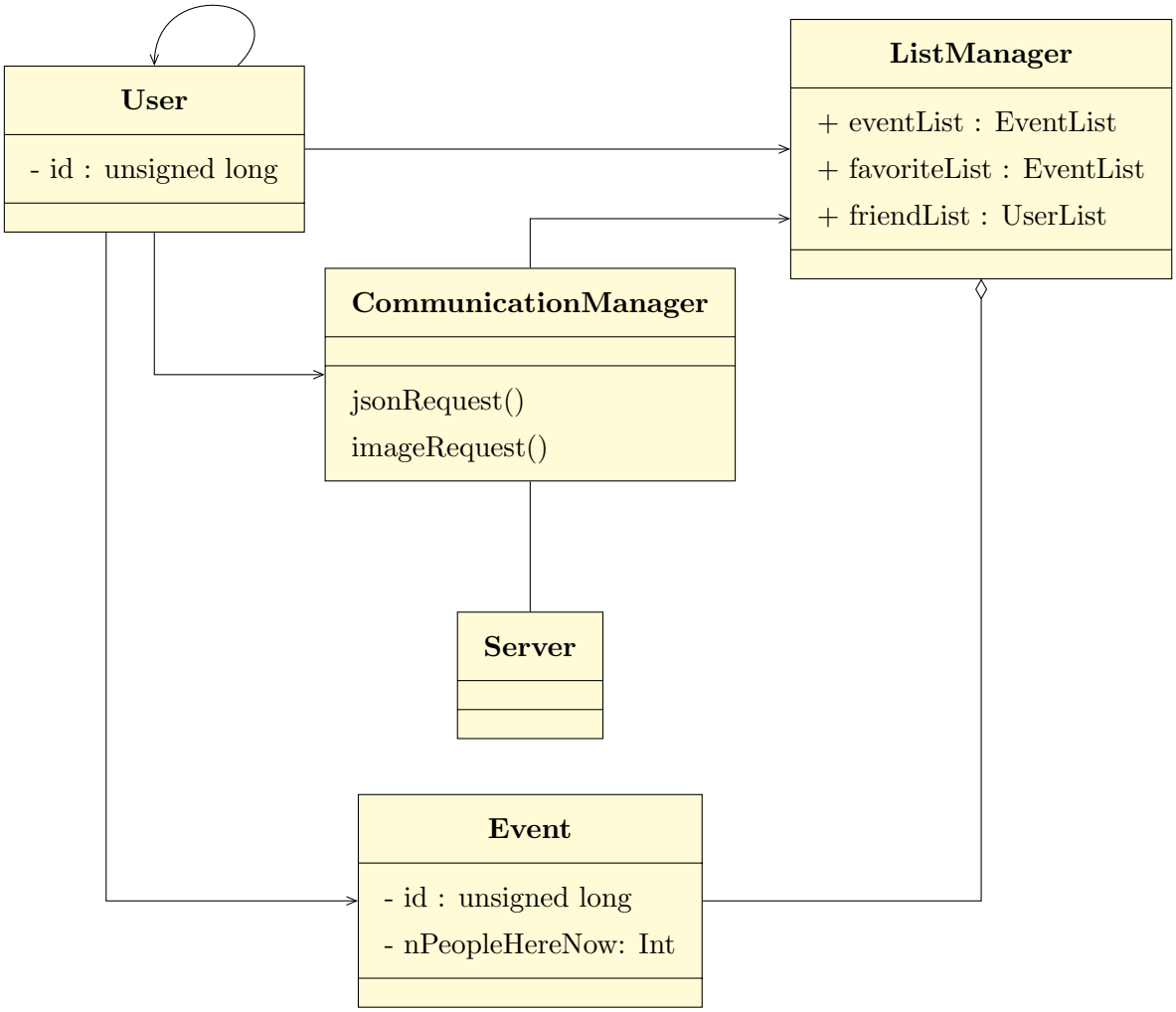



Figura 2.2: Diagramma delle classi

## 2.3 Database

Il database fornisce un insieme di *stored procedure* attraverso le quali svolgere le principali operazioni sai dati. Le *stored procedure* sono state progettate per interfac-

ciarsi nella maniera più comoda possibile con l'API REST e con le principali richieste che deve soddisfare.
Le principali interazioni con il database, dalle quali derivano le *stored procedure* sono:

- Creare, modificare e rimuovere un utente
- Creare, modificare e rimuovere un evento
- Cercare altri utenti
- Aggiungere e rimuovere utenti dalla lista amici
- Modificare lo stato di partecipazione di un utente rispetto ad un evento
- Ottenere gli eventi nelle vicinanze
- Ottenere gli eventi relativi ad un utente
- Notificare che un utente è fisicamente presente all'evento
- Notificare che un utente sta lasciando un evento

## 2.4 Tecnologie Utilizzate

### 2.4.1 Android Studio

Sviluppando l'applicazione per dispositivi Android la scelta dell'ambiente di sviluppo è ricaduta necessariamente su Android Studio. Infatti questo IDE fornisce tutti i tool necessari allo sviluppo e al test delle applicazioni.
In particolare l'applicazione è stata sviluppata usando i linguaggi Java e XML, il primo per lo sviluppo delle classi mentre il secondo per lo sviluppo delle interfacce.

### 2.4.2 Google Maps API

Google Maps è attualmente l'app più popolare per smartphone [3]. È un servizio che fornisce mappe globali molto dettagliate, correlate da informazioni in tempo reale

sulla viabilità, un navigatore satellitare e molto altro ancora. Queste due funzionalià citate sono alla base dello sviluppo del servizio di ricerca degli eventi nelle vicinanze e del calcolo del percorso per raggiungerli.
Infatti utilizzando Android Studio viene fornito un servizio di creazione automatica di activity basate su Google Maps API, rendendone l'integrazione con il progetto estremamente semplice. Da qui è poi possibile delegare a Google Maps il calcolo dei percorsi per raggiungere determinati eventi semplicemente creando dei *marker* sulla mappa e selezionandoli.
Infine è stata utilizzata la classe *Geocoder* nativa di Android per operazioni di *geocoding* e *reverse geocoding*. In questo modo è possibile, in fase di creazione o modifica di un evento, ottenere le coordinate geografiche dall'indirizzo inserito dall'utente oppure ottenere l'indirizzo utilizzando la posizione dell'utente [4].

### 2.4.3 Volley

Volley è una libreria HTTP che permette di semplificare l'invio e la ricezione di dati [7]. Nello specifico è in grado di gestire la coda delle richieste in maniera asincrona creando ed utilizzando nuovi *thread*, è in grado di risolvere le richieste gestendo la cache e infine dispone di richieste built-in per la trasmissione di JSON e di immagini, che sono i due tipi di richieste che l'applicazione fa al server.

### 2.4.4 MySQL

MySQL è un *relational database management system (RDBMS)*, formato da una architettura client-server, disponibile per sistemi Unix, Unix-like e Windows [5].
La scelta di questo RDBMS deriva dalla sua buona aderenza agli standard *ANSI SQL* e *ODBC SQL* [6].

### 2.4.5 Room

Room fornisce un livello di astrazione sopra SQLite e il suo utilizzo è fortemente consigliato da parte di Google rispetto al secondo [8].

All'interno dell'applicazione Room è stato utilizzato per creare un piccolo database in modo poter salvare alcuni dati localmente senza doverli richiedere ogni volta all'utente o al server.

# Capitolo 3

# Implementazione

## 3.1 Applicazione Android

Durante lo sviluppo si è cercato di seguire il più possibile le linee guida che Google fornisce per lo sviluppo delle interfacce utente [9], ne consegue una gestione della navigazione per mezzo di un *Navigation Drawer*, l'utilizzo del *Floating Action Button* per le azioni principali delle activity e l'impiego di una libreria di colori built-in appositamente creata per il *Material Design* (*material 900*).
Altro punto focale è stata la gestione dei permessi. Infatti l'applicazione necessita dell'accesso alla posizione dell'utente tramite GPS, anche se non tutte le funzionalità fornite dipendono da essa. Quindi l'accesso alla posizione utente viene controllato e richiesto a *runtime*, in questo modo l'utente può decidere di non fornire i dati relativi alla sua posizione geografica pur continuando ad utilizzare l'applicazione, seppur in modo molto limitato.
Prima di proseguire è necessario puntualizzare le classi fondamentali intorno alle quali è stata costruita l'applicazione:

**Activity** : è il principale componente che interagisce con l'utente, si occupa di creare la schermata nella quale inserire il layout [10]

**Fragment** : rappresenta il comportamento di una porzione di schermo, è una sorta di sub-activity riutilizzabile da activity diverse e combinabile con altri fragment

nella stessa activity [11]

**Service** : è un componente dell'applicazione in grado di eseguire operazioni in background senza fornire un'interfaccia utente [12]. In particolare nell'applicazione è stato utilizzato un foreground service che, a differenza dei service base, è in grado di svolgere operazioni in background anche quando l'applicazione non è attiva

### 3.1.1 Design

Il primo passo è stato scegliere il colore caratteristico dell'applicazione. Per fare questo è stato effettuato un sondaggio su 31 persone di età compresa tra i 20 e i 23 anni chiedendo di associare un colore alla parola "festa". Al termine il colore più popolare è risultato l'arancione seguito dal verde, dal viola e dal rosso.
La navigazione è gestita dal *Navigation Drawer*, qui troviamo l'accesso a due activity, quattro fragment e il logout. Nello specifico:

**New Event** : avvia l'activity responsabile della creazione di un nuovo evento

**Near Me** : avvia l'activity in grado di mostrare la mappa e la posizione geografica degli eventi nelle vicinanze

**Home** : fragment che mostra la lista di eventi nelle vicinanze

**Favorites** : fragment che mostra la lista di eventi correlati all'utente

**Profile** : fragment che mostra i dati dell'utente

**Friends** : fragment che mostra la lista amici

**Logout** : effettua il logout

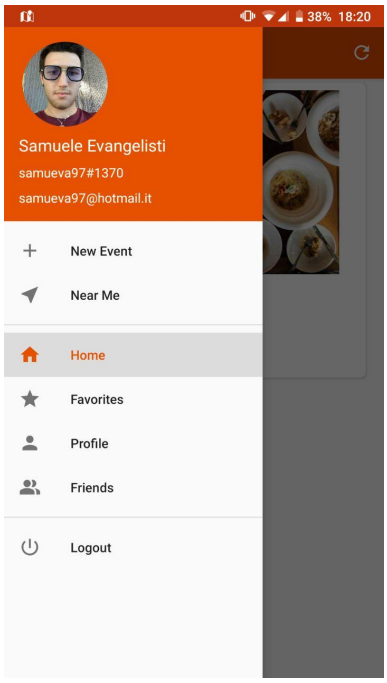
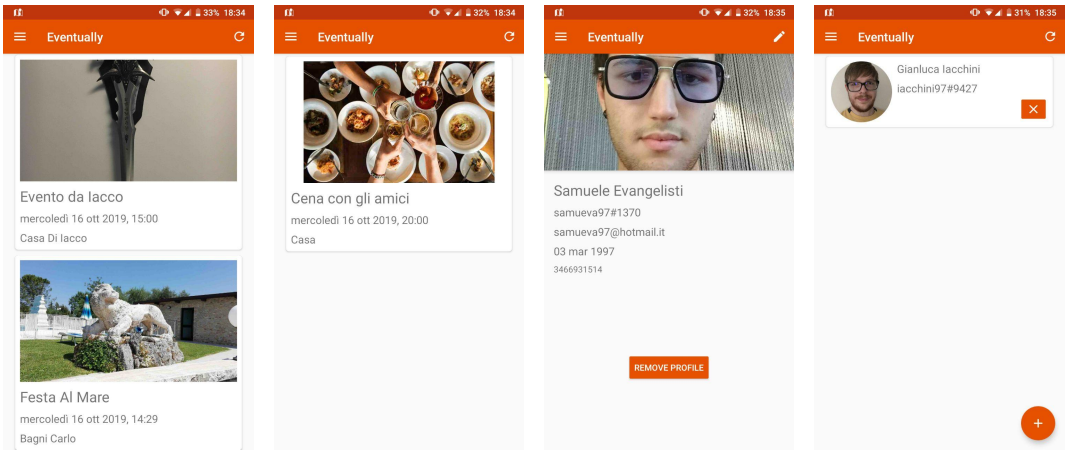



Figura 3.1: Il Navigation Drawer




(1) (2) (3) (4)

Figura 3.2: I quattro Fragment dell'applicazione, (1) HomeFragment, (2) FavoriteFragment, (3) ProfileFragment, (4) FriendFragment

### 3.1.2 User

La classe *User* rappresenta un utente dell'applicazione.
I suoi campi sono Id, Nome, Cognome, Immagine, Data di nascita, Numero di telefono, OrgTrust, PartTrust. I metodi presenti consentono la creazione di una nuova istanza partendo dal JSON che si ottiene dal server in seguito al login e tutti i metodi setter e getter dei campi.
Il ruolo che l'utente ricopre in un evento e il suo stato di partecipazione non vengono definiti in questa classe, che di fatto si occupa solo di gestire i dati dell'utente, ma nella classe Event.
Per automatizzare il login senza chiedere tutte le volte le credenziali dell'utente è presente una *@Entity* del database locale dell'applicazione (*Credentials.class*) che contiene esattamente questi dati. Nel database è sempre presente al massimo un record di questo tipo che viene memorizzato nel momento in cui il login va a buon fine e viene cancellato nel momento in cui l'utente effettua il logout.

### 3.1.3 Event

La classe *Event* rappresenta un evento nell'applicazione.
I suoi campi sono Id, Titolo, Descrizione, Immagine, Coordinate, Indirizzo, Nome del posto, Distanza, Relazione con l'utente, Data di inizio, Data di fine, Richiesta dell'invito. I metodi presenti consentono la creazione di una nuova istanza partendo dal JSON che si ottiene dal server in seguito al login e tutti i metodi setter e getter dei campi.
Per la gestione dell'immagine sono presenti due campi, il primo contiene l'url dell'immagine mentre il secondo contiene l'immagine fornita dal server (questa gestione è stata implementata in questo modo dal mio collega, Dott. Gianluca Iacchini, per i ridotti tempi di risposta che presenta).
L'applicazione ottiene le informazioni sugli eventi in tre momenti diversi:

1. quando viene mostrato un fragment che visualizza una lista eventi, il fragment verifica l'esistenza della lista in memoria e se questa non è presente invia l'ap-

posita richiesta, asincrona, al server. Quando ottiene la risposta popola la lista istanziando gli eventi con le informazioni base

2. nel momento in cui viene popolata una lista viene verificata la presenza dell'immagine per i vari eventi, in caso positivo vengono inoltrate nuove richieste asincrone al server, una per ogni immagine evento

3. quando l'utente seleziona un evento vengono inviate al server due richieste asincrone, la prima per ottenere le informazioni aggiuntive non presenti nella lista di partenza, come la descrizione e il numero di persone invitate o partecipanti, mentre la seconda per popolare la lista delle persone invitate con indicato il relativo stato di partecipazione

### 3.1.4 ListManager

Potendo accedere alle liste di eventi e utenti da activity differenti e in tempi differenti viene utilizzata una classe con il solo scopo di centralizzare la gestione dei dati che non riguardano l'utente. La classe *ListManager* garantisce la persistenza delle liste e diminuisce il numero di comunicazioni con il server. Infatti prima di ogni richiesta per ottenere dal server una lista di eventi o di utenti (come la lista amici) viene controllata l'esistenza della stessa e in caso positivo viene vengono mostrati all'utente i dati già presenti. È comunque possibile, da parte dell'utente, inviare una nuova richiesta per aggiornare i dati presenti nel dispositivo con i dati presenti sul server.
Nello specifico questa classe si compone di quattro liste *public* (accessibile da altre classi) e *static* (le liste sono sempre le stesse per ogni istanza della classe):

**List<Event> eventListHome** : contiene la lista di eventi da mostrare nell'*HomeFragment*

**List<Event> eventListFavorites** : contiene la lista di eventi da mostrare nell'*FavoriteFragmen*

**List<User> friendList** : contiene la lista amici

**List<User> userList** : contiene la lista di utenti correlati all'evento che si sta visualizzando

In queste quattro liste può saltare all'occhio la quarta (*userList*). Questa lista viene popolata nel momento in cui si visualizzano i dati relativi ad un evento. Il principale scopo di questa lista è quello di interagire con la lista amici (*friendList*) nel momento in cui si vogliono invitare amici ad un evento. In questo modo non vengono visualizzati gli utenti, presenti nella lista amici, che sono già stati invitati all'evento.

### 3.1.5 CommunicationManager

La classe *CommunicationManager* è costruita intorno a Volley [7]. Questa classe si compone di alcuni componenti:

**String url** : contiene il nome del server in modo da non dover inserire l'URL completo per ogni richiesta

**String cookie** : il server gestisce la comunicazione attraverso un cookie che viene mandato in risposta alla richiesta di login

**RequestQueue requestQueue** : questo campo è il principale motivo della creazione di questa classe. Volley gestisce le comunicazioni inserendole in una *RequestQueue* [13] appositamente configurata all'avvio dell'applicazione. La lista in questione è *static*. Il motivo di questa decisione è la semplificazione nella gestione che ne consegue. Infatti il *CommunicationManager* è stato implementato in modo da accodare una nuova richiesta semplicemente istanziando un nuovo *CommunicationManager* configurato ad hoc

Oltre ai dati necessari per la risoluzione delle richieste, il *CommunicationManager* necessita anche dell'implementazione di due funzioni, la prima da richiamare in caso di successo nella comunicazione, la seconda da richiamare in caso di fallimento.
Il codice viene riportato in *Appendice A - CommunicationManager*.

### 3.1.6 GeofenceService

Questo service ha il principale scopo di rendere il sistema di *geofence* utilizzabile sulle nuove versioni di Android.

Eventually oltre a fornire un servizio incentrato sulla gestione degli eventi cerca anche di fornire un servizio di sicurezza. Anche se questo dato, completamente anonimo, non viene mostrato agli utenti, il database tiene traccia del numero di utenti effettivamente presenti ad un evento. In questo modo è possibile verificare se gli utenti presenti ad un evento siano effettivamente in situazioni sicure o si trovino in situazioni di sovraffollamento.
Più in dettaglio quando l'applicazione ottiene dal server la lista di eventi nelle vicinanze per ognuno di questi crea una *geofence* [14] di raggio 100 metri. Tenendo attivo il GeofenceService, quando l'utente entra o esce da una di queste aree di 100 metri di raggio, viene inviata una comunicazione al server che provvede ad aggiornare il numero di partecipanti effettivi all'evento interessato.
Questo sistema presenta alcune limitazioni per le quali si è cercato di proporre delle soluzioni:

1. Per poter effettuare questo tipo di operazioni il service deve necessariamente essere un *Foreground Service* [12]. Per poter essere definito tale il service deve mostrare una notifica permanente all'utente. Questa soluzione può sembrare fastidiosa quindi dalla notifica è possibile interrompere l'esecuzione del service, anche se è fortemente sconsigliato, pur continuando ad utilizzare l'applicazione normalmente. Logicamente se il service viene interrotto non è più utilizzabile il sistema di geofence in maniera attiva e non è possibile ovviare a questo problema

2. L'utente può decidere in ogni momento di non comunicare la propria posizione spegnendo il GPS. Infatti l'applicazione necessita della posizione dell'utente solo all'avvio, quando vengono richiesti al server gli eventi nelle vicinanze. Quindi in una situazione di questo tipo l'utente è comunque in grado di utilizzare l'applicazione. Per risolvere questo scenario è stato implementato nel database un sistema di permanenza dei dati dove vengono salvati gli ID degli eventi nei quali l'utente risulta presente fisicamente. In questo modo nel momento in cui l'utente si avvicina o si allontana da un evento è possibile, lavorando sui dati memorizzati, inviare le giuste comunicazioni al server per garantire la consistenza del dato relativo alle persone fisicamente presenti agli eventi

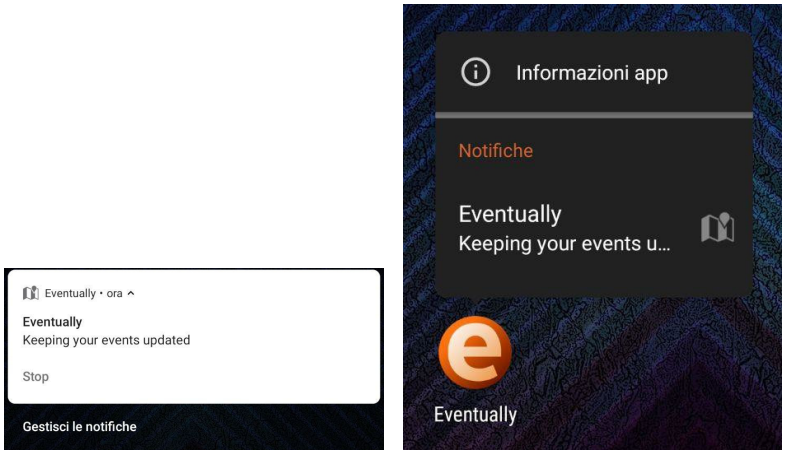



Figura 3.3: Notifica del service

### 3.1.7 LauncherActivity

La *LauncherActivity* è la prima activity che viene mostrata all'utente quando l'applicazione viene avviata. È un'activity molto semplice con una barra di caricamento circolare nella metà inferiore. Quando questa activity viene lanciata viene interrogato il database locale per ottenere le credenziali dell'utente. Successivamente possono essere avviate due activity:

- se le credenziali presenti nel database non sono utilizzabili per fare login nell'applicazione o se non sono presenti credenziali allora viene avviata la *LoginActivity*
- se le credenziali presenti nel database sono corrette allora viene effettuato il login e viene avviata la *MainActivity*

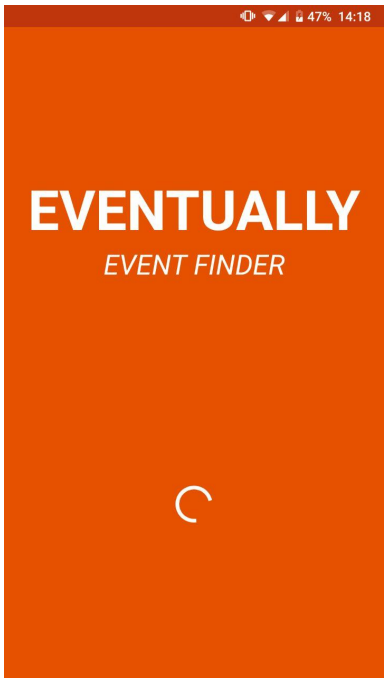



Figura 3.4: LauncherActivity

### 3.1.8 LoginActivity

La *LoginActivity* si occupa dell'autenticazione dell'utente. Sono presenti i due campi per l'autenticazione, Email e Password, e due bottoni, New User e Login. Con il primo viene avviata l'activity per effettuare la registrazione dell'utente mentre con il secondo vengono inviati al server i dati per l'autenticazione. In caso di risposta positiva il server comunica all'applicazione i dati dell'utente che vengono utilizzati per istanziare la classe *User*.

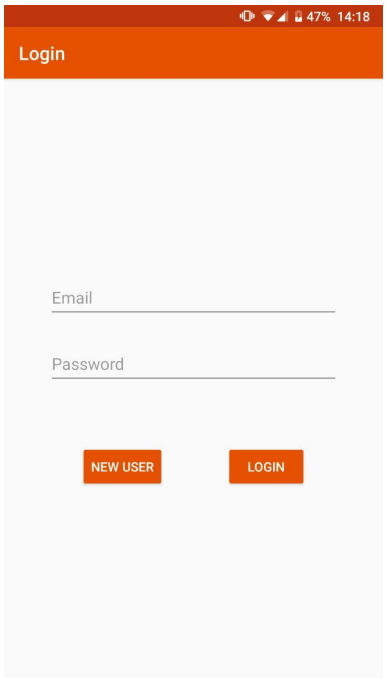



Figura 3.5: LoginActivity

### 3.1.9 NewUserActivity

La *NewUserActivity* raccoglie i dati necessari alla registrazione di un nuovo utente. I dati necessari all'applicazione sono:

- Email
- Password
- Nome
- Cognome
- Username
- Data di nascita

Inoltre, a discrezione dell'utente, è possibile fornire altri due dati facoltativi:

- Numero di telefono
- Immagine di profilo

Tutti i campi obbligatori sono necessari al servizio per identificare un utente in fase di autenticazione o in caso questo venga cercato da altri utenti all'interno dell'applicazione. Viceversa i due campi facoltativi non sono necessari al servizio per identificare l'utente ma sono comunque molto utili per l'identificazione da parte di altri utenti.

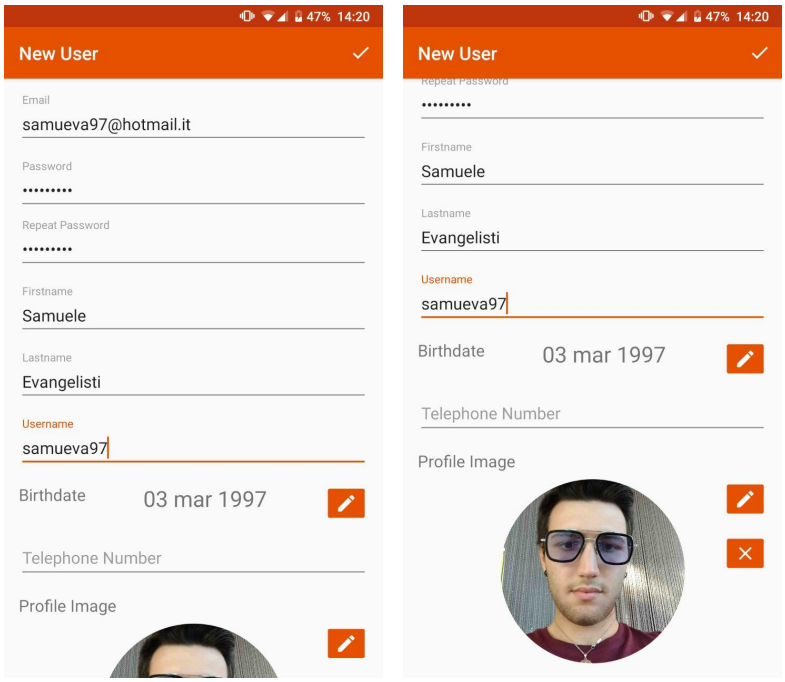



Figura 3.6: NewUserActivity

### 3.1.10 MainActivity

La *MainActivity* è il nucleo dell'applicazione. Si compone di quattro fragment e di un navigation drawer che si occupa di gestire la navigazione tra i fragment. Le funzionalità dei vari fragment sono gestite dagli stessi quindi la *MainActivity* una volta

avviata si occupa solo di istanziare il primo Fragment, vale a dire l'*HomeFragment*. Di seguito verranno trattati nel dettaglio i vari componenti di questa activity.

### Navigation Drawer

Come detto in precedenza si è cercato di seguire le linee guida di Google in termini di interfaccia utente, quindi la navigazione viene gestita attraverso un navigation drawer.
Il navigation drawer è composto da due parti:

- Nella parte superiore troviamo un header con sfondo arancione nel quale vengono mostrate l'immagine utente e le principali informazioni che lo identificano nell'applicazione, vale a dire nome e cognome, username e email
- Nella parte inferiore troviamo gli elementi che gestiscono la navigazione. Quando uno di questi elementi viene selezionato dall'utente il navigation drawer comunica la scelta alla *MainActivity* che provvede a visualizzare il fragment corretto, a lanciare una nuova activity o a effettuare il logout. Per ogni elemento è presente un'icona che lo caratterizza in modo da rendere la navigazione più intuitiva

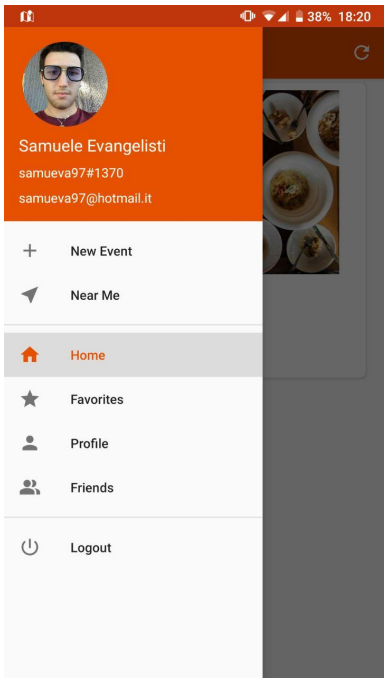



Figura 3.7: Navigation Drawer

### HomeFragment

In questo fragment viene mostrata la lista degli eventi nelle vicinanze.
La prima cosa che questo fragment fa quando viene istanziato è verificare, e in caso negativo richiedere all'utente, i permessi per accedere alla posizione dell'utente. Successivamente viene verificato che il GPS sia acceso e in caso contrario viene notificato all'utente di accenderlo. Anche se l'utente decide di non fornire la propria posizione è possibile utilizzare l'applicazione normalmente, ad eccezione dei servizi per i quali è necessaria.
Quando il fragment ottiene la posizione dell'utente la comunica al server che in risposta fornisce tutti gli eventi nelle vicinanze. La lista viene fornita al *ListManager* e subito dopo mostrata all'utente. Da qui è poi possibile visualizzare nel dettaglio le informazioni dei vari eventi selezionando l'evento d'interesse.

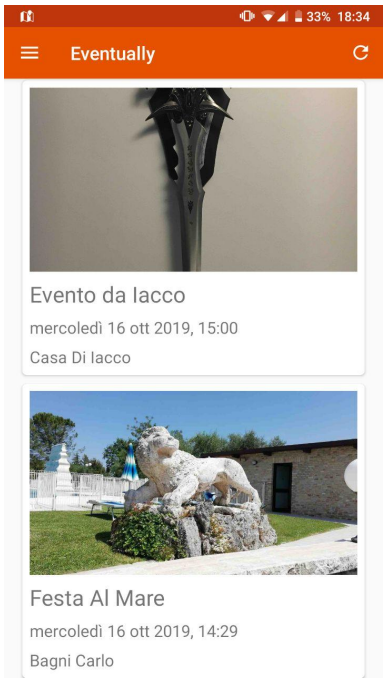



Figura 3.8: HomeFragment

### FavoriteFragment

In questo fragment viene mostrata la lista degli eventi con i quali l'utente ha una correlazione.
La prima cosa che questo fragment fa quando viene istanziato è comunicare al server l'id dell'utente che in risposta fornisce la lista descritta in precedenza. La lista viene poi fornita al *ListManager* e subito dopo mostrata all'utente. Come nel caso precedente è possibile visualizzare le informazioni dettagliate dei vari eventi selezionando l'evento d'interesse.

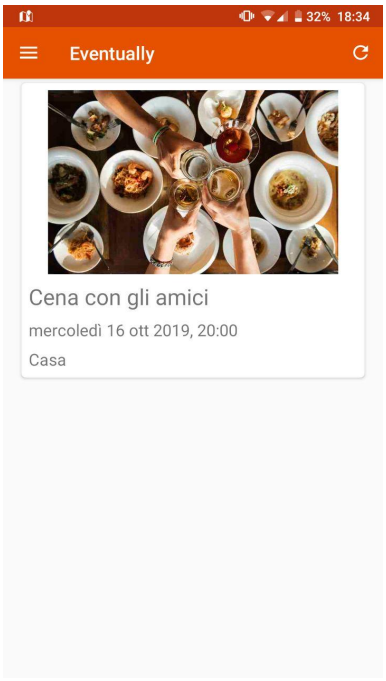
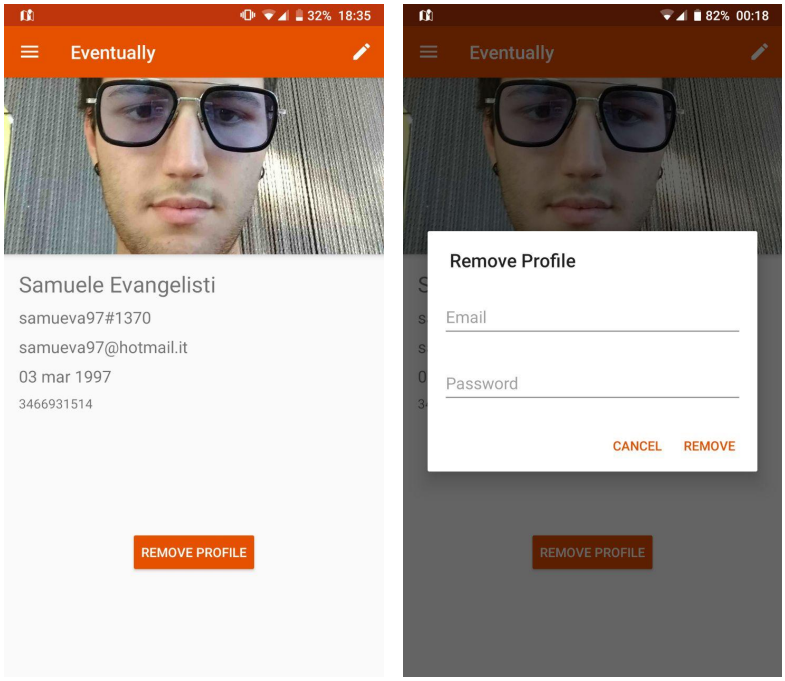



Figura 3.9: FavoriteFragment

### ProfileFragment

In questo fragment vengono mostrati i dati dell'utente e il pulsante per rimuovere il profilo.
I dati dell'utente sono forniti dal server nel momento dell'autenticazione quindi questo fragment non effettua richieste al server se non per la rimozione del profilo. I dati mostrati sono nome e cognome, username, email, data di nascita e, in caso l'utente li abbia forniti, il numero di telefono e l'immagine profilo. Nel momento in cui l'utente decida di rimuovere il profilo dall'applicazione vengono nuovamente chieste le credenziali per confermare tale decisione.

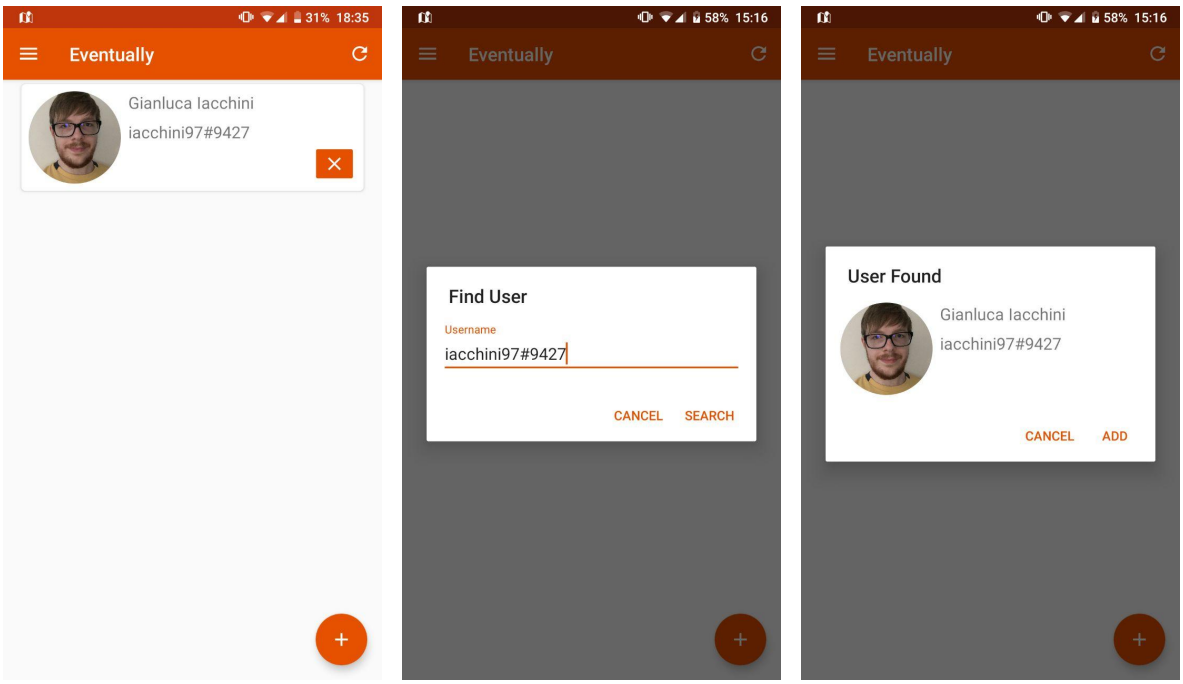



(1) (2)

Figura 3.10: (1) ProfileFragment, (2) autenticazione per la rimozione del profilo

### FriendFragment

In questo fragment viene mostrata la lista amici dell'utente.
La prima cosa che questo fragment fa quando viene istanziato è comunicare al server l'id dell'utente che in risposta fornisce la lista descritta in precedenza. La lista viene poi fornita al *ListManager* e subito dopo mostrata all'utente. Da questo fragment e possibile rimuovere gli utenti dalla lista amici o aggiungerne di nuovi. L'aggiunta di nuovi amici può essere considerata l'azione principale di questa activity quindi, nel rispetto delle linee guida di Google, viene delegata a un floating action button [15].



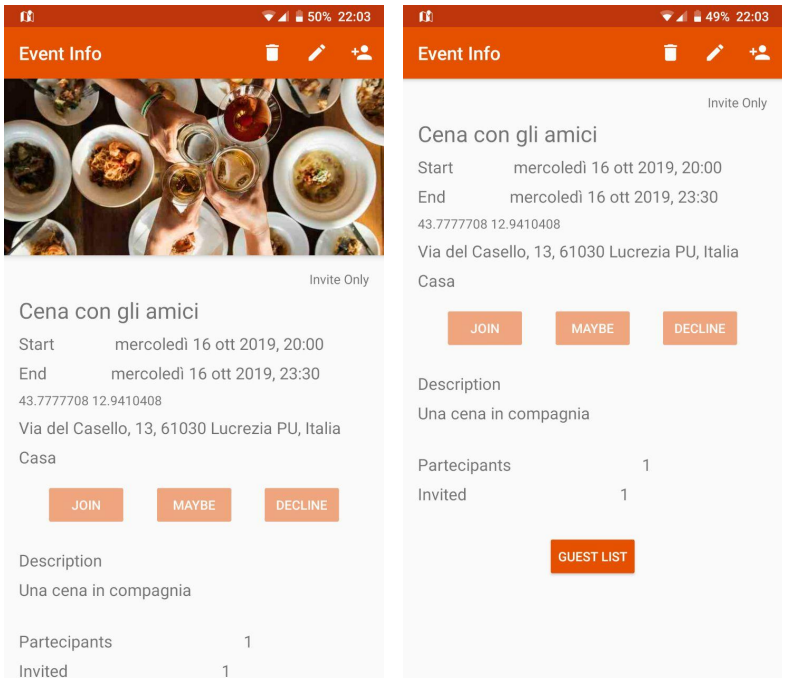

(1) (2) (3)

Figura 3.11: (1) FriendFragment, (2) ricerca di un utente, (3) aggiunta di un amico

### 3.1.11 EventInfoActivity

Nell'*HomeFragment* e nel *FavoriteFragment* gli eventi vengono mostrati in liste nelle quali sono mostrati i dati piú significativi come Titolo, Data, Luogo e, in caso l'utente l'abbia inserita, l'immagine. É possibile visualizzare i dati completi degli eventi selezionando l'evento di interesse.
La *EventInfoActivity* si occupa della visualizzazione dei dati completi oltre che fornire la possibilità agli organizzatori di aggiungere invitati, modificare o eliminare l'evento, mentre agli utenti la possibilità di esprimere il proprio stato di partecipazione.
Quando un evento viene selezionato dall'utente il fragment che in quel momento è istanziato avvia la *EventInfoActivity* fornendole le indicazioni sulla lista e sull'indice dell'evento. Infatti avendo già ottenuto i dati principali dell'evento nel momento

dell'istanziazione del fragment e avendoli salvati nel *ListManager* è necessario solamente richiedere al server solamente i dati aggiuntivi come Descrizione, numero degli invitati, numero dei partecipanti e lista degli invitati. Questi non sono, volutamente, presenti nel *ListManager* in quanto, soprattutto il numero di invitati e di partecipanti, possono cambiare frequentemente.
Come detto in precedenza questa activity fornisce anche la possibilità di indicare il proprio stato di partecipazione all'evento selezionato comportando la necessità di aggiornare il *ListManager*. A tale scopo la *EventInfoActivity* nel momento in cui viene chiusa notifica al fragment che l'ha avviata la necessità di aggiornare i dati.

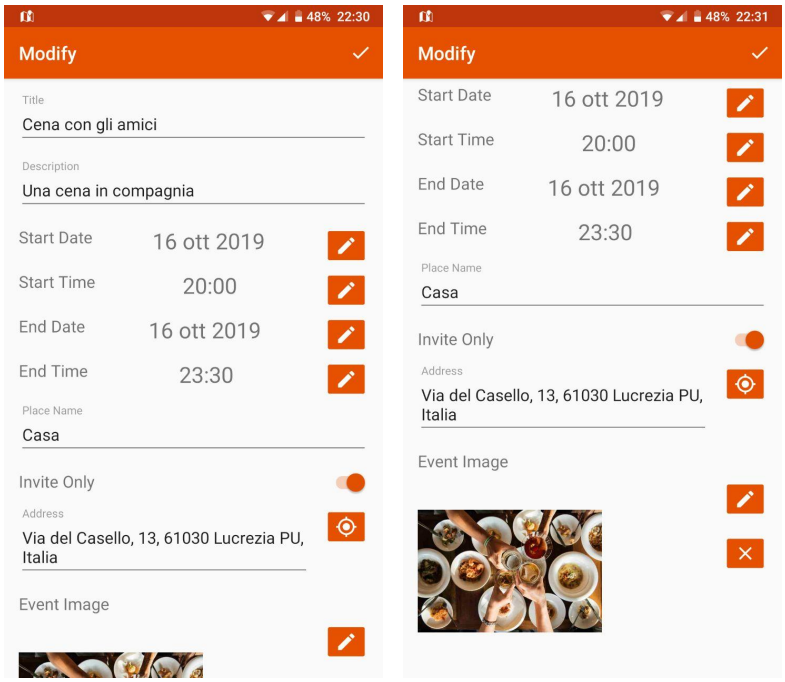



(1) (2)

Figura 3.12: EventInfoActivity

### 3.1.12 ModifyEventActivity

Nel caso in cui l'utente sia il creatore o un organizzatore di un evento la *EventInfoActivity* mostra la possibilità di modificarlo.
La *ModifyEventActivity* Presenta la stessa interfaccia della *NewEventActivity* con l'unica differenza che quando viene avviata i campi vengono precompilati con le informazioni sull'evento da modificare.
Questa activity può essere avviata solo dalla *EventInfoActivity* la quale fornisce le indicazioni su come trovare l'evento nel *ListManager*, vale a dire la lista nel quale è inserito l'evento e il suo indice nella lista. Quando l'utente modifica l'evento per prima cosa i nuovi dati vengono mandati al server, successivamente l'activity termina e notifica alla *EventInfoActivity* la necessità di aggiornare i dati. Come nel caso precedente verranno aggiornati i dati presenti nel *ListManager* per poi essere visualizzati.




(1) (2)

Figura 3.13: ModifyEventActivity

### 3.1.13 UserInfoActivity

Come per gli eventi è possibile visualizzare le informazioni dettagliate relative ad un utente.
Selezionando un utente dalla lista amici mostrata nel *FriendFragment* quest'ultimo provvederà ad avviare la *UserInfoActivity* fornendo le informazioni su come ottenere i dati dell'utente nel *ListManager*, quindi la lista in cui trovare l'utente e l'indice nella lista.
La *UserInfoActivity* è una activity molto semplice che si occupa solo di mostrare i dati dell'utente selezionato. Ricordiamo che la lista di amici viene chiesta al server nel momento in cui si istanzia il *FriendFragment* quindi non vengono effettuate altre richieste al server quando viene avviata la *UserInfoActivity*.

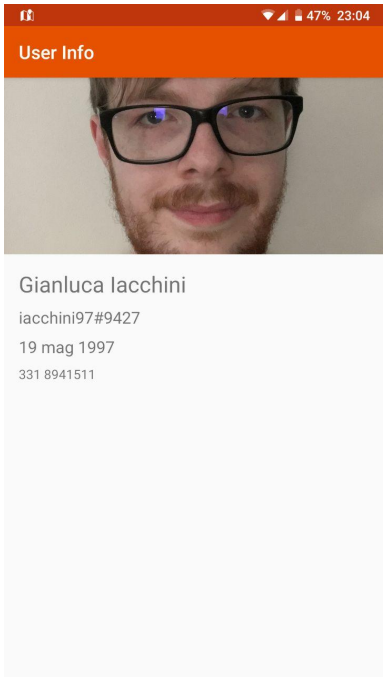


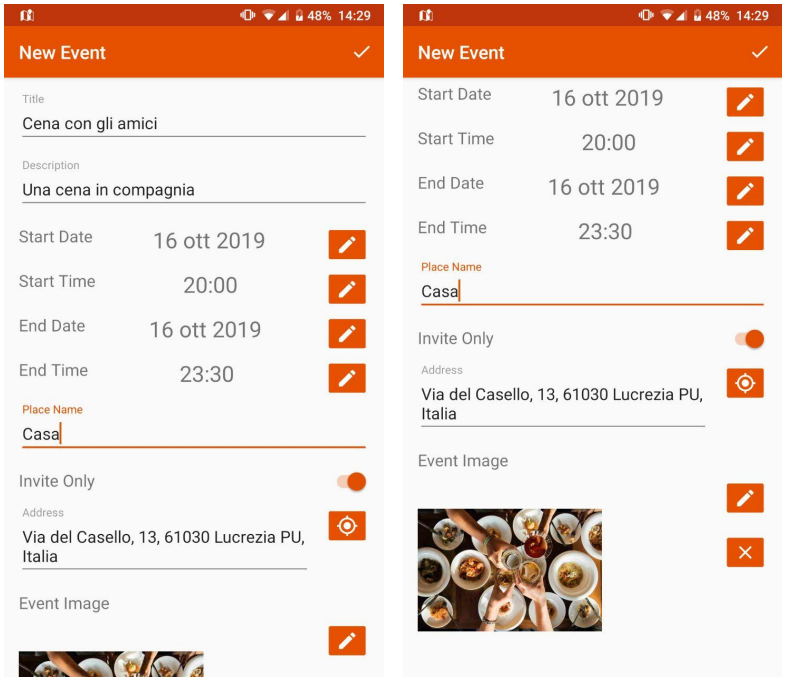

Figura 3.14: UserInfoActivity

### 3.1.14 NewEventActivity

La *NewEventActivity* si occupa di creare un nuovo evento e comunicarlo al server. L'activity presenta vari campi dove inserire le informazioni relative all'evento. Parte di queste informazioni sono obbligatorie come Titolo, Data e ora di inizio, Data e ora di fine, Indirizzo. Altre sono facoltative, ma comunque consigliate per caratterizzare l'evento, come Descrizione, Nome del luogo, Immagine evento.
Analizzando le informazioni che vengono inserite è possibile notare che l'indicazione del luogo è presente sia come indirizzo sia come nome del posto. In realtà queste due informazioni si occupano di aspetti distinti dell'evento. Infatti mentre il nome del luogo identifica il posto per come è conosciuto dagli utenti (per esempio il nome di un locale) l'indirizzo si occupa di identificare il luogo geograficamente per mostrare l'evento nella mappa e fornire il servizio di indicazioni stradali.
L'indirizzo dell'evento può essere inserito in due modi:

1. L'utente conosce l'indirizzo del luogo. In questo caso l'activity, attraverso un'operazione di *geocoding*, ottiene le coordinate geografiche derivanti da quell'indirizzo

2. L'utente non conosce l'indirizzo ma si trova nel luogo dove avverrà l'evento. In questo caso, premendo il bottone a fianco del campo correlato, l'activity otterrà, attraverso il GPS, la posizione dell'utente. Attraverso un'operazione di *reverse geocoding* sarà poi possibile trasformare le coordinate geografiche in indirizzo autocompilando il campo presente nell'activity



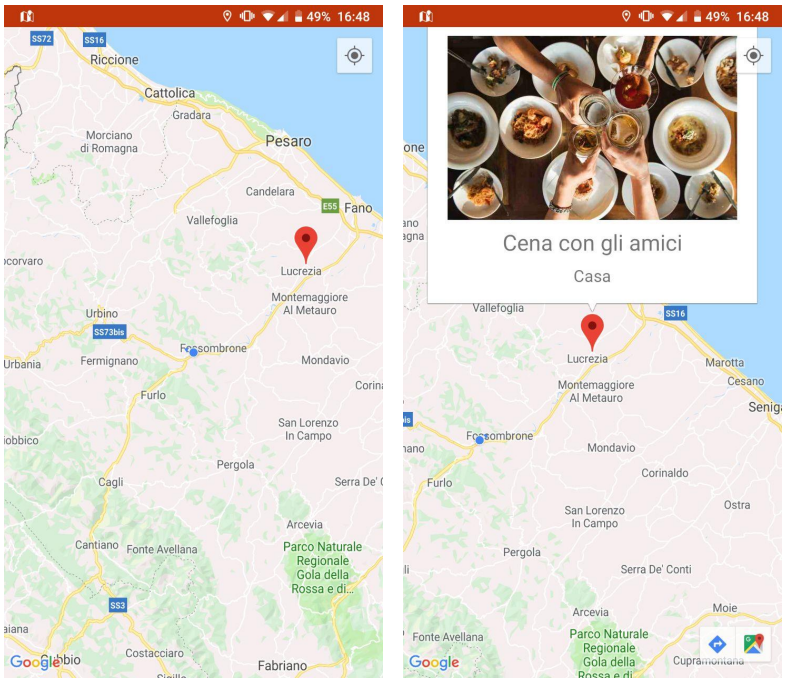

(1) (2)

Figura 3.15: NewEventActivity

### 3.1.15 MapsActivity

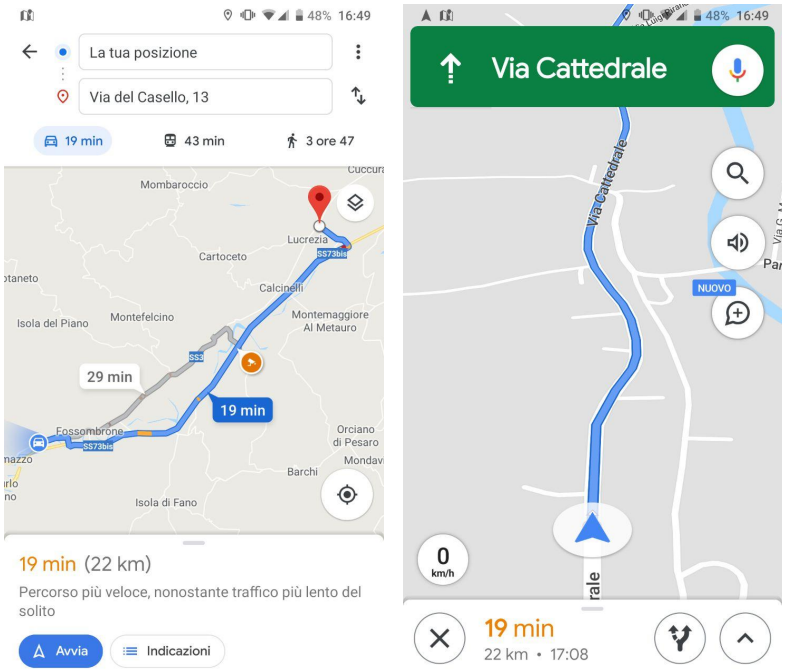
la *MapsActivity* si occupa di mostrare la posizione geografica degli eventi nelle vicinanze dell'utente.
Quando questa activity viene avviata ottiene le informazioni sugli eventi da visualizzare prendendole dalla lista correlata presente nel *ListManager*. Ricordiamo che quando viene creato un evento vengono sempre registrate le coordinate geografiche del luogo dove avverrà, grazie a questo dato è possibile posizionare i vari marker nei luoghi degli eventi. Selezionando un marker è poi possibile visualizzare le informazioni base dell'evento identificato dal marker oltre che poter avviare l'applicazione Google Maps per ottenere le indicazioni stradali. In questo modo non è necessario implementare da zero un nuovo navigatore ed è possibile mantenere questo servizio

sempre aggiornato indipendentemente dall'applicazione.




(1) (2)

Figura 3.16: (1) MapsActivity, (2) informazioni sull'evento




(1) (2)

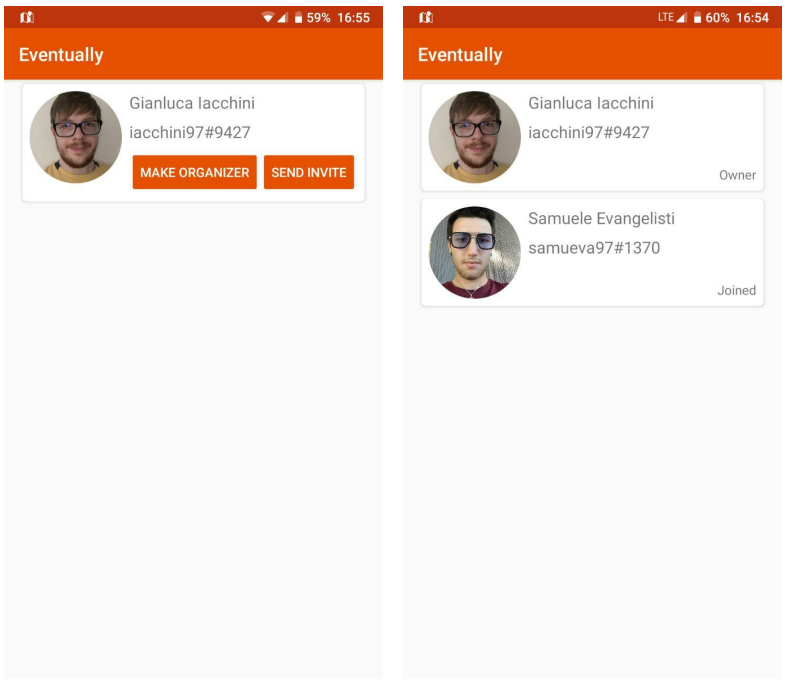
Figura 3.17: (1) indicazioni stradali, (2) navigatore

### 3.1.16 FriendListActivity e UserListActivity

Queste due activity sono molto simili in termini di layout e comportamento. Le principali differenze che presentano sono i dati che mostrano e le principali operazioni che effettuano su di essi.

**FriendListActivity** : si avvia quando l'utente vuole aggiungere invitati ad un evento per il quale è il creatore o un organizzatore. Gli utenti visualizzati in questa activity sono tutti gli utenti presenti nella lista amici esclusi quelli già invitati all'evento in questione. Da questa activity è possibile sia invitare un utente sia aggiungerlo come organizzatore dell'evento utilizzando i bottoni che vengono visualizzati per ogni utente nell'activity

**UserListActivity** : si avvia quando l'utente vuole consultare la lista di invitati di un evento utilizzando l'apposito bottone presente nella *EventInfoActivity*. In questo caso vengono visualizzati tutti gli utenti che presentano una correlazione con l'evento in questione e il loro ruolo o il loro stato di partecipazione.




(1) (2)

Figura 3.18: (1) FriendListActivity, (2) UserListActivity

## 3.2 Database

Il database di Eventually è stato sviluppato di pari passo con le necessità dovute allo sviluppo dell'API REST, implementata dal Dott. Gianluca Iacchini. Infatti, dove possibile, si è cercato di unire nelle stesse relazioni i dati forniti in contemporanea dalle varie richieste.

### 3.2.1 Relazioni

| Credentials | | | |
|---|---|---|---|
| Id | Email | PasswdHash | Salt |

*Credentials* contiene tutte le informazioni per l'autenticazione dell'utente:

- Id BIGINT(255) UNSIGNED AUTO_INCREMENT UNIQUE NOT NULL: id utente, chiave primaria della relazione
- Email VARCHAR(255) UNIQUE NOT NULL: email dell'utente, necessaria in fase di autenticazione
- PasswdHash VARCHAR(512) NOT NULL: hashing della password utente, fornita dal server, necessaria per l'autenticazione
- Salt VARCHAR(64) NOT NULL: sale [16], fornito dal server, necessario per l'autenticazione

| Users | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Id | Firstname | Lastname | Username | Birthdate | Image | TelNo | OrgTrust | PartTrust |

*Users* contiene tutte le informazioni relative ad un utente dell'applicazione:

- Id BIGINT(255) UNSIGNED UNIQUE NOT NULL: id utente, chiave primaria della relazione e chiave esterna collegata a *Id* della relazione *Credentials*
- Firstname VARCHAR(255) NOT NULL: nome dell'utente
- Lastname VARCHAR(255) NOT NULL: cognome dell'utente
- Username VARCHAR(255) UNIQUE NOT NULL: username dell'utente
- Birthdate DATE NOT NULL: data di nascita dell'utente
- Image VARCHAR(4096): link al file contenente l'immagine dell'utente

- TelNo VARCHAR(255): numero di telefono dell'utente
- OrgTrust DECIMAL(7, 6) NOT NULL DEFAULT 0.000000: affidabilità dell'utente relativamente all'organizzazione di eventi
- PartTrust DECIMAL(7, 6) NOT NULL DEFAULT 0.000000: affidabilità dell'utente relativamente al rispetto del proprio stato di partecipazione rispetto agli eventi

Quando un utente viene rimosso dalla relazione *Credentials* i vincoli fanno in modo di eliminare i dati corrispondenti nella relazione *Users*.

| **Events** | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Id | Title | Image | Start Date | Start Time | End Date | End Time | Latitude | Longitude | Address | Place Name | Invite Only |

*Events* contiene le informazioni che caratterizzano l'evento:

- Id BIGINT(255) UNSIGNED AUTO_INCREMENT UNIQUE NOT NULL: id evento, chiave primaria della relazione
- Title VARCHAR(255) NOT NULL: titolo dell'evento
- Image VARCHAR(4096): link al file contenente l'immagine dell'evento
- StartDate DATE NOT NULL: data di inizio dell'evento
- StartTime TIME NOT NULL: orario di inizio dell'evento
- EndDate DATE NOT NULL: data di fine dell'evento
- EndTime TIME NOT NULL: orario di fine dell'evento
- Latitude DECIMAL(10, 8) NOT NULL: latitudine del posto dove si svolgerà l'evento

- Longitude DECIMAL(11, 8) NOT NULL: longitudine del posto dove si svolgerà l'evento
- Address VARCHAR(255): indirizzo del posto dove si svolgerà l'evento
- PlaceName VARCHAR(255): nome del posto dove si svolgerà l'evento
- InviteOnly BOOLEAN NOT NULL DEFAULT true: questo dato indica se l'evento sia su invito o pubblico

| **EventInfo** | | | | |
|---|---|---|---|---|
| Id | Description | nInvited | nPartecipants | nPeopleHereNow |

*EventInfo* contiene le informazioni aggiuntive riguardanti l'evento. Queste informazioni non sono caratterizzanti per l'evento ma l'utente può comunque volerle visualizzare. Queste informazioni sono:

- Id BIGINT(255) UNSIGNED UNIQUE NOT NULL: id evento, chiave primaria della relazione e chiave esterna collegata a *Id* della relazione *Events*
- Description MEDIUMTEXT: descrizione dell'evento
- nInvited INTEGER(255) UNSIGNED NOT NULL DEFAULT 1: numero di invitati all'evento, il valore di default considera il creatore come primo invitato
- nPartecipants INTEGER(255) UNSIGNED NOT NULL DEFAULT 1: numero di partecipanti all'evento, il valore di default considera il creatore come primo partecipante
- nPeopleHereNow INTEGER(255) UNSIGNED NOT NULL DEFAULT 0: numero di persone fisicamente presenti all'evento. Questo valore viene aggiornato dall'applicazione attraverso il service di gestione delle geofence. Tuttavia questo valore non può essere consultato dagli utenti dell'applicazione

Quando un evento viene rimosso dalla relazione *Events* i vincoli fanno i modo di rimuoverlo dalla relazione *EventInfo*.

| **UserUserRelations** | |
|---|---|
| UserId | FriendId |

*UserUserRelations* indica quali altri utenti si trovano nella lista amici di un utente. Nello specifico:

- UserId BIGINT(255) UNSIGNED NOT NULL: id dell'utente per il quale si sta cercando la lista amici, chiave esterna della relazione collegata a *Id* della relazione *Credentials*
- FriendId BIGINT(255) UNSIGNED NOT NULL: id dell'utente che si trova nella lista amici, chiave esterna della relazione collegata a *Id* della relazione *Credentials*

La chiave primaria di questa relazione è composta *UserId* e da *FriendId*. Quando un utente viene rimosso dalla relazione *Credentials* i vincoli fanno in modo che vengano rimossi tutti i record dove è presente, sia come *UserId* sia come *FriendId*.

| **UserUserRelations** | | | | |
|---|---|---|---|---|
| UserId | EventId | Status | Arrival | Departure |

*UserEventRelations* indica lo stato di partecipazione di un utente rispetto ad un evento. I campi presenti sono:

- UserId BIGINT(255) UNSIGNED NOT NULL: id utente, chiave esterna della relazione collegata a *Id* della relazione *Credentials*
- EventId BIGINT(255) UNSIGNED NOT NULL: id evento, chiave esterna della relazione collegata a *Id* della relazione *Events*
- Status TINYINT(255) UNSIGNED NOT NULL: stato di partecipazione dell'utente rispetto all'evento

- Arrival DATETIME DEFAULT NULL: istante di arrivo dell'utente all'evento

- Departure DATETIME DEFAULT NULL: istante di uscita dell'utente dall'evento

La chiave primaria di questa relazione è composta da *UserId* e da *EventId*. Quando un utente viene rimosso dalla relazione *Credentials* i vincoli fanno in modo che vengano rimossi tutti i record dove l'utente è presente come *UserId*. La stessa cosa accade per *EventId*. Gli ultimi due campi presenti registrano le informazioni relative all'arrivo dell'utente e al momento quando l'utente lascia l'evento. Il valore di *Arrival* viene impostato con l'istante temporale in cui l'utente entra per la prima volta nell'evento mentre il valore di *Departure* viene impostato con l'ultimo istante temporale in cui l'utente lascia l'evento. Infatti utilizzando il sistema di geofence è possibile che l'utente si avvicini e si allontani dall'evento molte volte comportando ingressi e uscite multiple. Volendo considerare la presenza complessiva dell'utente vengono presi il primo ingresso e l'ultima uscita. Questo dato non è visibile agli utenti attualmente ma in futuro verrà utilizzato per analizzare la durata della permanenza degli utenti agli eventi e i picchi massimi di presenze. Tutto questo dopo aver reso anonimi i dati in questione.

### 3.2.2 Procedure

Per interfacciare il DBMS con il server sono state implementate varie procedure. Richiamando le procedure il server è in grado di comunicare al DBMS quali operazioni effettuare sui dati. Le procedure danno anche la possibilità di indicare parametri di ingresso e parametri di uscita, i primi vengono forniti per la computazione, i secondi vengono letti a computazione terminata.
Di seguito verranno descritte le principali procedure utilizzate presentandone i tratti principali dell'implementazione. Verranno tralasciate le procedure che presentano un'implementazione ovvia.

**create_user**

La procedura *create_user* si occupa di inserire un nuovo utente nel database con i dati forniti in input. In caso l'utente sia già presente nel database il parametro *alreadyin* viene impostato come **true**, altrimenti come **false**.

```
proc_label:BEGIN
START TRANSACTION;
IF EXISTS (SELECT * FROM Credentials WHERE Credentials.Email
    = email) THEN
  SET alreadyin = -1;
  COMMIT;
  LEAVE proc_label;
ELSE
  IF NOT EXISTS (SELECT * FROM Users WHERE Users.Username =
      username) THEN
    INSERT INTO Credentials (Email, PasswdHash, Salt)
    VALUES (email, passwdhash, salt);
    SET @id = LAST_INSERT_ID();
    INSERT INTO Users (Id, Firstname, Lastname, Username,
        Birthdate, Image, TelNo)
    VALUES (@id, firstname, lastname, username, birthdate,
        image, telno);
    set alreadyin = 0;
  ELSE
    SET alreadyin = -1;
  END IF;
END IF;
COMMIT;
END
```

Listing 3.1: procedura create_user

**delete_user**

La procedura *delete_user* si occupa per prima cosa di aggiornare i dati relativi alla partecipazione dell'utente agli eventi. Successivamente rimuove l'utente e con esso tutti i dati e gli eventi ad esso correlati grazie ai vincoli presenti nelle relazioni.

```
BEGIN
START TRANSACTION;
UPDATE EventInfo
SET EventInfo.nInvited = EventInfo.nInvited - 1
WHERE EventInfo.Id IN (
  SELECT UserEventRelations.EventId
  FROM UserEventRelations JOIN Events ON UserEventRelations.
      EventId = Events.id
  WHERE UserEventRelations.UserId = id AND Events.InviteOnly
      = true);
UPDATE EventInfo
SET EventInfo.nPartecipants = EventInfo.nPartecipants - 1
WHERE EventInfo.Id IN (
  SELECT UserEventRelations.EventId
  FROM UserEventRelations
  WHERE UserEventRelations.UserId = id AND (
      UserEventRelations.status = 0 OR UserEventRelations.
      status = 1 OR UserEventRelations.status = 2));
DELETE FROM Credentials
WHERE Credentials.Id = id;
COMMIT;
END
```

Listing 3.2: procedura delete_user

**create_event**

La procedura *create_event* si occupa di inserire un nuovo evento nel database con i dati forniti in input. Una volta inserito l'evento il parametro *id* viene impostato con l'id di questo evento.

```
BEGIN
START TRANSACTION;
INSERT INTO Events (Title, Image, StartDate, StartTime,
    EndDate, EndTime, Latitude, Longitude, Address, PlaceName,
     InviteOnly)
VALUES (title, image, startdate, starttime, enddate, endtime,
     latitude, longitude, address, placename, inviteonly);
SET id = LAST_INSERT_ID();
INSERT INTO UserEventRelations (UserId, EventId, Status)
VALUES (ownerid, id, 0);
INSERT INTO EventInfo (Id, Description)
VALUES (id, description);
COMMIT;
END
```

Listing 3.3: procedura create_event

**delete_event**

La procedura *delete_event* si occupa di verificare che l'utente che ha richiesto l'eliminazione dell'evento sia il creatore, se questo succede l'eliminazione avviene, altrimenti no. Successivamente, come nel caso dell'eliminazione di un utente, vengono rimossi tutti i dati relativi all'evento grazie ai vincoli presenti nelle relazioni.

```
BEGIN
SET success = FALSE;
SET @status = -1;
START TRANSACTION;
```

```
SELECT @status:=UserEventRelations.Status
FROM UserEventRelations
WHERE UserEventRelations.UserId = userid AND
    UserEventRelations.EventId = eventid;
IF (@status = 0) THEN
  DELETE FROM Events
  WHERE Events.Id = eventid;
  SET success = TRUE;
END IF;
COMMIT;
END
```

Listing 3.4: procedura delete_event

**find_passwdhash_by_email**

Questa procedura è utilizzata dal server solo per ottenere la password dell'utente ed effettuare l'autenticazione.

```
BEGIN
SELECT Credentials.PasswdHash
FROM Credentials
WHERE Credentials.Email = email;
END
```

Listing 3.5: procedura find_passwdhash_by_email

**find_events_by_userid**

Questa procedura si occupa di fornire al server la lista di eventi che hanno una correlazione con un utente. Oltre le informazioni relative all'evento viene anche fornita lo stato di partecipazione dell'utente rispetto a ciascuno degli eventi forniti.

```
BEGIN
```

```
SELECT Events.*, UserEventRelations.Status
FROM Events JOIN UserEventRelations ON Events.Id =
    UserEventRelations.EventId
WHERE UserEventRelations.UserId = userid;
END
```

Listing 3.6: procedura find_events_by_userid

È presente anche una seconda procedura, la procedura *find_events_by_distance*, di cui non verrà riportato il codice. Questa procedura fornisce la lista di eventi presenti nelle vicinanze dell'utente operando un calcolo sulle coordinate geografiche dell'utente e sulle coordinate geografiche dell'evento:

```
6371 * ACOS(COS(RADIANS(latitude)) * COS(RADIANS(Events.
    Latitude)) * COS(RADIANS(Events.Longitude) - RADIANS(
    longitude)) + SIN(RADIANS(latitude)) * SIN(RADIANS(Events.
    Latitude)))
```

Listing 3.7: formula utilizzata per calcolare la distanza tra un utente e un evento

dove *latitude* e *longitude* sono le coordinate geografiche dell'utente, mentre *Events.Latitude* e *Events.Longitude* sono le coordinate geografiche dell'evento.

### make_joined

La procedura *make_joined* fa parte di un gruppo di procedure che si occupano di cambiare lo stato di partecipazione dell'utente rispetto ad un evento. Conseguentemente la procedura aggiorna i dati dell'evento relativi al numero di invitati e al numero di partecipanti in base al vecchio stato di partecipazione dell'utente e in base al nuovo stato di partecipazione.

```
BEGIN
SET success = false;
SET @status = -1;
START TRANSACTION;
```

```
SELECT @status:=UserEventRelations.Status
FROM UserEventRelations
WHERE UserEventRelations.UserId = userid AND
    UserEventRelations.EventId = eventid;
IF (@status = -1) THEN
  INSERT INTO UserEventRelations (UserId, EventId, Status)
  VALUES (userid, eventid, 2);
  UPDATE EventInfo
  SET EventInfo.nInvited = EventInfo.nInvited + 1, EventInfo.
      nPartecipants = EventInfo.nPartecipants + 1
  WHERE EventInfo.Id = eventid;
  SET success = true;
ELSEIF (@status = 1 OR @status = 3 OR @status = 4) THEN
  UPDATE UserEventRelations
  SET UserEventRelations.Status = 2
  WHERE UserEventRelations.UserId = userid AND
      UserEventRelations.EventId = eventid;
   IF (@status = 3 OR @status = 4) THEN
     UPDATE EventInfo
     SET EventInfo.nPartecipants = EventInfo.nPartecipants + 1
     WHERE EventInfo.Id = eventid;
  END IF;
  SET success = true;
END IF;
COMMIT;
END
```

Listing 3.8: procedura make_joined

Come detto in precedenza sono presenti altre procedure per gestire gli altri cambiamenti di stato, in particolare le procedure sono *make_maybe*, *make_declined*, *make_organized*, rispettivamente per impostare lo stato dell'utente rispetto all'evento

come "forse", "non partecipante" o "organizzatore".

### set_user_arrival e set_user_departure

La procedura *set_user_arival* imposta l'arrivo dell'utente con la data e ora in cui viene avviata questa procedura. Andando a impostare il primo arrivo dell'utente, la procedura verifica anche che questo valore non sia stato precedentemente impostato, caso in cui l'utente è entrato nell'evento in precedenza.

```
BEGIN
START TRANSACTION;
UPDATE UserEventRelations
SET UserEventRelations.Arrival = NOW()
WHERE UserEventRelations.UserId = userid AND
    UserEventRelations.EventId = eventid AND
    UserEventRelations.Arrival IS NULL;
UPDATE EventInfo
SET EventInfo.nPeopleHereNow = EventInfo.nPeopleHereNow + 1
WHERE EventInfo.Id = eventid;
COMMIT;
END
```

Listing 3.9: procedura set_user_arrival

Allo stesso modo la procedura *set_user_departure*, di cui non verrà presentato il codice, imposta l'uscita dell'utente dall'evento con la data e ora corrente. Andando a impostare l'ultima uscita dall'evento questa procedura non ha bisogno di fare controlli sul valore stesso ma semplicemente lo sovrascrive ogni volta che l'utente esce nuovamente dall'evento.

# Conclusioni

Sono molti i social network che forniscono un supporto per la gestione degli eventi. Tuttavia questi servizi presentano difetti. Alcuni, come nel caso di Snapchat, forniscono un servizio carente e comunque costoso, altri, come nel caso di Facebook, forniscono un servizio completo ma incentrato soprattutto sulla parte economica, derivante dalla promozione di eventi di grandi dimensioni, trascurando l'informatività del servizio oltre agli eventi di piccole dimensioni.
La nascita di Eventually è quindi dettata dalla necessità di un servizio in grado di valorizzare allo stesso tempo eventi di grandi dimensione ed eventi di nicchia oltre che fornire all'utente informazioni sugli eventi a cui può accedere. Principalmente lo scopo dell'applicazione è la gestione di eventi anche se presenta funzionalità aggiuntive come la possibilità di visualizzare gli eventi sulla mappa, di ottenere indicazioni stradali in modo semplice e veloce e di fornire un rating di organizzatori e invitati. Infine è presente anche una funzionalità per il controllo del sovraffollamento dei luoghi dove si tengono gli eventi.

## Sviluppi futuri

Gli sviluppi futuri possono essere molteplici e di certo dipenderanno soprattutto dal feedback con gli utenti. In generale alcuni punti focali da cui partire potranno essere:

- Miglioramento dell'interfaccia utente
- Miglioramento della profilazione utente e della gestione della lista amici

- Miglioramento della gestione dei valori di *trust* degli utenti
- Aggiunta dei filtri per la ricerca degli eventi
- Aggiunta di funzionalità come pubblicazione di post o invio di messaggi
- Consentire l'accesso al servizio utilizzando i social network già presenti
- Aggiunta di un servizio di gestione degli inviti e di compravendita online dei biglietti tramite Qr code

# Bibliografia


[1] J.Clement. *Most famous social network sites 2019, by active users*

https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users

27 Maggio 2019

[2] https://www.socialmediaexaminer.com/how-to-create-facebook-event-frame-instagram-event-geotag-and-snapchat-event-filter - consultato il 25 giugno 2019

[3] Cooper Smith. *Google+ Is The Foruth Most-Used Smartphone App*,

https://www.businessinsider.com/google-smartphone-app-popularity-2013-9?IR=T#infographic

5 Settembre 2013

[4] https://developer.android.com/training/location/display-address - consultato il 3 agosto 2019

[5] https://it.wikipedia.org/wiki/MySQL - consultato il 20 settembre 2019

[6] Oracle Corporation. *MySQL Differences from Standard SQL*

https://dev.mysql.com/doc/refman/5.7/en/differences-from-ansi.html

22 giugno 2014

[7] https://developer.android.com/training/volley - consultato il 5 agosto 2019

[8] https://developer.android.com/training/data-storage/room - consultato il 15 luglio 2019

[9] https://developer.android.com/guide/topics/ui - consultato il 15 luglio 2019

[10] https://developer.android.com/reference/android/app/Activity - consultato il 15 luglio 2019

[11] https://developer.android.com/guide/components/fragments - consultato il 15 luglio 2019

[12] https://developer.android.com/guide/components/services - consultato il 3 settembre 2019

[13] https://developer.android.com/training/volley/requestqueue - consultato il 5 agosto 2019

[14] https://developer.android.com/training/location/geofencing - consultato il 25 agosto 2019

[15] https://developer.android.com/guide/topics/ui/floating-action-button - consultato il 15 luglio 2019

[16] https://it.wikipedia.org/wiki/Sale_(crittografia) - consultato il 15 luglio 2019

# Ringraziamenti

Ringrazio il Professor Luciano Bononi per in consigli forniti, il tempo e la disponibilità dedicati. Un ringraziamento speciale va al Dott. Gianluca Iacchini, amico, compagno di studi e collega senza il quale Eventually non sarebbe stata realizzata, per il tempo dedicato e il supporto fornito in fase di progettazione e sviluppo. Infine ringrazio la mia famiglia per il supporto fornitomi durante gli studi universitari.

# Appendice A

# CommunicationManager

```
public class CommunicationManager {

  private static final String url = "https://www.
     eventuallyapp.com";
  private static String cookie;
  private static RequestQueue requestQueue;
  private CommunicationManagerListener
     communicationManagerListener;

  public interface CommunicationManagerListener {
    void onParseNetworkResponse(NetworkResponse response);
  }



  public CommunicationManager(@Nullable
     CommunicationManagerListener
     communicationManagerListener) {
```

```
    this.communicationManagerListener =
        communicationManagerListener;
}

public static void init(Context context) {
    requestQueue = Volley.newRequestQueue(context);
}



//////// GETTERS AND SETTERS ////////

public static String getCookie() {
    return cookie;
}

public static void setCookie(String newCookie) {
    cookie = newCookie;
}



//////// COMMUNICATION MANAGEMENT ////////

public void jsonRequest(int method, String path, @Nullable
    JSONObject jsonObject, Response.Listener<JSONObject>
    listener, Response.ErrorListener errorListener) {
    JsonObjectRequest jsonObjectRequest = new
        JsonObjectRequest(method, url + path, jsonObject,
        listener, errorListener) {
```

```
      @Override
      public Map<String, String> getHeaders() {
        Map<String, String> headers = new HashMap<String,
            String>();
        headers.put("Cookie", cookie);
        return headers;
      }

      @Override
      protected Response<JSONObject> parseNetworkResponse(
          NetworkResponse response) {
        if (communicationManagerListener != null) {
          communicationManagerListener.onParseNetworkResponse
              (response);
        }
        return super.parseNetworkResponse(response);
      }
    };
    requestQueue.add(jsonObjectRequest);
  }

  public void imageRequest(String path, Response.Listener<
      Bitmap> listener, Response.ErrorListener errorListener)
      {
    ImageRequest imageRequest = new ImageRequest(url + path,
        listener, 0, 0, null, null, errorListener) {
      @Override
      public Map<String, String> getHeaders() {
        Map<String, String> headers = new HashMap<String,
            String>();
```

```
            headers.put("Cookie", cookie);
            return headers;
        }
    };
    requestQueue.add(imageRequest);
  }
}
```