
# GathorApp

Documentazione Tecnica del Progetto

Alessandro Alfano

Matricola: 962485

Codice Persona: 10788970

Novembre 2025

# Contents

# 1. DESCRIZIONE DELL'AMBITO

## 1.1 Obiettivi del Progetto

GathorApp è un'applicazione web per la gestione di uscite e eventi con caratteristiche avanzate:

- **Organizzazione di uscite:** Gli utenti possono creare uscite (outings) indipendenti oppure associate a eventi
- **Gestione di eventi:** Gli utenti business possono creare eventi
- **Partecipazione:** Gestisce il sistema di approvazione delle partecipazioni, controlla e gestisce la concorrenza
- **Chat in tempo reale:** Implementa la comunicazione tra i partecipanti ad un'uscita tramite WebSocket. La chat si auto-disattiva dopo 7 giorni
- **Notifiche:** Salva le notifiche su database e le invia agli utenti (Pattern Observer)
- **Sistema di voucher:** Genera automaticamente i voucher per utenti premium che organizzano uscite che soddisfano i requisiti
- **Recensioni e valutazioni:** Permette agli utenti di valutare uscite ed eventi

## 1.2 Contesto di Utilizzo

L'applicazione è rivolta a:

- **Utenti standard:** Possono creare fino a 5 uscite al mese con massimo 10 partecipanti
- **Utenti premium:** Possono creare un numero illimitate di uscite associate a eventi e guadagnare voucher
- **Utenti business:** Possono creare eventi e premi
- **Amministratori:** Gestiscono utenti e configurazioni

## 1.3 Requisiti Funzionali Divisi per Ruolo

**Utente Standard (USER)**

- **RF1:** Può visualizzare le uscite e gli eventi nelle vicinanze (ricerca per raggio)
- **RF2:** Può creare uscite indipendenti (max 5 al mese, max 10 partecipanti)
- **RF3:** Può iscriversi a un'uscita e ricevere la notifica di approvazione/rifiuto
- **RF4:** Può inviare messaggi in chat durante l'uscita
- **RF5:** Può lasciare una recensione su un'uscita o un evento (1-5 stelle)
- **RF6:** Può visualizzare notifiche

**Utente Premium (PREMIUM)**

- **RF7:** Può creare uscite illimitate associate a eventi
- **RF8:** Può visualizzare i voucher che ha guadagnato dall'organizzazione di uscite
- **RF9:** Può utilizzare il voucher

**Utente Business (BUSINESS)**

- **RF10:** Può creare un evento e associare dei premi
- **RF11:** Può riscattare i voucher
- **RF12:** Può visualizzare i premi disponibili

**Amministratore (ADMIN)**

- **RF13:** Può gestire i ruoli degli utenti (promozione/retrocessione)
- **RF14:** Può bannare/sbannare gli utenti
- **RF15:** Può visualizzare le statistiche della piattaforma

### 1.4 Requisiti Non Funzionali

- **RNF1:** Interfaccia grafica responsiva (Angular)
- **RNF2:** Multithreading con lock per il controllo della concorrenza nel sistema di partecipazioni
- **RNF3:** Copertura di test >= 80% (JUnit 5 + JaCoCo)
- **RNF4:** Autenticazione JWT con refresh token
- **RNF5:** Documentazione UML completa con diagrammi
- **RNF6:** API documentata con OpenAPI/Swagger

---

## 2. DESCRIZIONE DELLA PROGETTAZIONE

### 2.1 Architettura Generale

L'applicazione segue il **pattern MVC (Model-View-Controller):**

- **Model:** Entità JPA
  - *backend/src/main/java/com/alfano/gathorapp/*/*.java*
- **View:** Componenti Angular standalone
  - *frontend/src/app/**
- **Controller:** Spring REST controllers
  - *backend/src/main/java/com/alfano/gathorapp/*/*Controller.java*
- **Service:** Logica di business
  - *backend/src/main/java/com/alfano/gathorapp/*/*Service.java*
- **Repository:** Accesso ai dati JPA
  - *backend/src/main/java/com/alfano/gathorapp/*/*Repository.java*

Ogni package funzionale contiene tutti i componenti correlati (Controller, Service, Repository, Entity, DTO, Mapper) nello stesso package, senza sottocartelle. Ad esempio, il package outing contiene:

- Outing.java (Entity)
- OutingController.java (REST Controller)
- OutingService.java (Business Logic)
- OutingRepository.java (Data Access)
- OutingMapper.java (DTO Mapping)
- dto/ (DTO classes)

## 2.2 Stack Tecnologico

**Backend:**

- Java 17
- Spring Boot 3.5.6
- Spring Security (JWT)
- Spring Data JPA
- Spring WebSocket (STOMP)
- H2 Database (in-memory/file-based)
- Gradle 8.5

**Frontend:**

- Angular 20
- TypeScript 5.7
- Componenti standalone
- OpenAPI Generator (client auto-generato)
- Leaflet (mappe)
- STOMP.js (WebSocket)

**Testing & Coverage:**

- JUnit 5
- Mockito
- JaCoCo

## 2.3 Package Backend

```
com.alfano.gathorapp
├── admin/           → Gestione amministratori
├── auth/            → Autenticazione JWT
├── chat/            → Sistema di chat
├── config/          → Configurazioni Spring
├── event/           → Gestione eventi
├── map/             → Servizi di geolocalizzazione
├── notification/    → Notifiche
├── outing/          → Gestione uscite
├── participation/   → Approvazione partecipazioni
├── pattern/         → Pattern di design
│   ├── observer/    → Observer Pattern (notifiche)
│   └── strategy/    → Strategy Pattern (limiti utente)
├── report/          → Segnalazioni
├── review/          → Recensioni
├── reward/          → Premi
├── security/        → Configurazioni sicurezza
├── user/            → Gestione utenti
├── voucher/         → Sistema voucher
└── websocket/       → Configurazione WebSocket
```

---

# 3. DESIGN PATTERN IMPLEMENTATI

## 3.1 Strategy Pattern (Limiti Utente)

**Ubicazione:** *backend/src/main/java/com/alfano/gathorapp/pattern/strategy/*

**Problema risolto:**

GathorApp ha 4 tipi di utenti (USER, PREMIUM, BUSINESS, ADMIN) ciascuno ha regole di business differenti per la creazione di uscite. Implementare queste regole con *if-else* basati su *user.getRole()* violerebbe l'Open/Closed Principle e renderebbe il codice fragile e difficile da testare.

**Soluzione tramite Strategy Pattern:**

```
UserLimitationStrategy (interfaccia)
├── BaseUserStrategy (max 5 uscite/mese, 10 partecipanti)
├── PremiumUserStrategy (illimitate collegate ad evento, partecipanti illimitati)
└── BusinessUserStrategy (illimitati)

UserStrategyFactory → seleziona strategia per ruolo
```

**Implementazione dettagliata:**

1. **Interfaccia *UserLimitationStrategy***: definisce il contratto con 3 metodi
   - *canCreateOuting(User user, int monthlyCount)*: verifica se l'utente può creare l'uscita
   - *getMaxParticipants(User user)*: ritorna il limite massimo di partecipanti (10 per USER, illimitato per PREMIUM/BUSINESS)
   - *getMonthlyOutingLimit(User user)*: ritorna il limite massimo mensile di uscite create (5 per USER, illimitato per altri)
2. **Strategie concrete**:
   - *BaseUserStrategy*: implementa le regole per il ruolo USER (5/mese, max 10 partecipanti, solo uscite indipendenti)
   - *PremiumUserStrategy*: uscite illimitate se associate a eventi, partecipanti illimitati
   - *BusinessUserStrategy*: nessun limite (usato anche per ADMIN)
3. **Factory *UserStrategyFactory***: usa un *Map<Role, UserLimitationStrategy>* per selezionare la strategia

   ```
   public UserLimitationStrategy getStrategy(Role role) {
       return strategies.getOrDefault(role, baseUserStrategy);
   }
   ```
4. **Integrazione in *OutingService***:

   ```
   UserLimitationStrategy strategy = strategyFactory.getStrategy(user.getRole());
   if (!strategy.canCreateOuting(user, monthlyCount)) {
       throw new LimitExceededException("Monthly limit reached");
   }
   ```

**Benefici architetturali:**

- **Open/Closed Principle**: aggiungere un nuovo ruolo (es. MODERATOR) richiede solo di creare una nuova strategia, senza modificare *OutingService*
- **Single Responsibility**: ogni strategia gestisce solo le regole del proprio ruolo
- **Testabilità**: ogni strategia è testabile indipendentemente con unit test dedicati (vedi *UserLimitationStrategyTest.java*)
- **Polimorfismo**: il client (*OutingService*) lavora con l'interfaccia, non con le implementazioni concrete
- **Eliminazione condizionali**: nessun *if (role == USER)* sparso nel codice

**Visibilità nel Class Diagram:**

Nel class diagram, il pattern Strategy è rappresentato come sottografo separato con:

- Relazioni di implementazione (*implements*) tra strategie concrete e interfaccia
- Relazione di dipendenza (*creates*) tra factory e interfaccia
- Integrazione con *User* tramite *Role* enum

**File chiave:**

- *UserLimitationStrategy.java* (interfaccia)
- *BaseUserStrategy.java*, *PremiumUserStrategy.java*, *BusinessUserStrategy.java*
- *UserStrategyFactory.java*
- Test: *UserLimitationStrategyTest.java*

## 3.2 Observer Pattern (Notifiche Multi-Canale)

**Ubicazione:** *backend/src/main/java/com/alfano/gathorapp/pattern/observer/*

**Problema risolto:**

Il sistema deve inviare notifiche attraverso **2 canali** quando si verificano eventi importanti (partecipazione approvata, voucher generato, nuovo messaggio):

1. **Database**: persistenza per storico e consultazione offline
2. **WebSocket**: notifica real-time agli utenti connessi

Implementare questa logica direttamente nei service (es. *ParticipationService.approve()*) creerebbe:

- **Tight coupling**: i service dipenderebbero da *NotificationRepository* e *WebSocketMessagingTemplate*
- **Difficoltà di estensione**: aggiungere un 3° canale (email, SMS) richiederebbe di modificare tutti i service
- **Violazione SRP**: i service gestirebbero sia la business logic sia la notifica

**Soluzione tramite Observer Pattern:**

```
NotificationSubject (interface)
└── NotificationManager (Subject concreto thread-safe)


NotificationObserver (interface)
```

```
├── PersistenceNotificationObserver (salva in DB)
└── WebSocketNotificationObserver (invia via WebSocket STOMP)
```

**Implementazione dettagliata:**

1. **Interfaccia *NotificationSubject***: definisce il contratto del Subject
   - *attach(NotificationObserver observer)*: registra un observer
   - *detach(NotificationObserver observer)*: rimuove un observer
   - *notifyObservers(NotificationEvent event)*: propaga evento a tutti gli observer
2. **Classe *NotificationManager*** (Subject concreto):
   - Mantiene lista observer in *CopyOnWriteArrayList* (thread-safe)
   - Metodo *notifyObservers()* usa *parallelStream()* per eseguire observer in parallelo:

```
observers.parallelStream().forEach(observer -> {
    try {
        observer.update(event);
    } catch (Exception e) {
        log.error("Error notifying observer", e);
    }
});
```

   - **Gestione errori**: se un observer fallisce, gli altri continuano l'esecuzione
3. **Interfaccia *NotificationObserver***:
   - Singolo metodo: *update(NotificationEvent event)*
4. **Observer concreti**:
   - *PersistenceNotificationObserver*: crea un'entità *Notification* e salva nel DB tramite *NotificationRepository*
   - *WebSocketNotificationObserver*: invia un messaggio STOMP a */topic/notifications/{user* tramite *SimpMessagingTemplate*
5. **Inizializzazione** (*NotificationObserverInitializer*):
   - Bean **@PostConstruct** che registra i 2 observer al manager all'avvio dell'applicazione
6. **Integrazione nei Service**:

```
// In ParticipationService.approve()
NotificationEvent event = new NotificationEvent(
    userId,
    NotificationType.PARTICIPATION_APPROVED,
    "La tua richiesta è stata approvata"
);
notificationManager.notifyObservers(event);
```

**Multithreading e prestazioni:**

- **Thread-safety**: *CopyOnWriteArrayList* permette letture concorrenti senza lock. Scritture (attach/detach) copiano l'array intero (rare, solo all'avvio)
- **Parallelizzazione**: *parallelStream()* esegue observer su thread del ForkJoinPool comune
  - Osservatore DB: ~10ms (INSERT query)
  - Osservatore WebSocket: ~50ms (network latency)
  - **Tempo totale sequenziale**: 60ms
  - **Tempo totale parallelo**: ~50ms (il più lento dei due)
  - **Speedup**: 1.2x con 2 observer, scala linearmente con più observer
- **No blocking**: se il DB è lento, il WebSocket non viene bloccato (e viceversa)
- **Fault tolerance**: eccezione in un observer non propaga agli altri (try-catch in *forEach*)

**Benefici architetturali:**

- **Decoupling**: *ParticipationService* non conosce *NotificationRepository* né *SimpMessagingTemplate*, dipende solo da *NotificationManager*
- **Estensibilità (Open/Closed)**: aggiungere email/SMS richiede solo:
  1. Creare *EmailNotificationObserver implements NotificationObserver*
  2. Registrare nel *NotificationObserverInitializer*
  3. **Zero modifiche** ai service esistenti
- **Single Responsibility**: ogni observer gestisce 1 solo canale
- **Testabilità**: gli observer sono testabili individualmente, i service sono testabili con mock del manager
- **Concorrenza**: le notifiche inviate in parallelo migliorano performance

**Visibilità nel Class Diagram:**

Nel class diagram, l'Observer Pattern è visualizzato con:

- Relazioni di implementazione tra observer concreti e interfaccia *NotificationObserver*
- Relazione di composizione (*o–*) tra *NotificationManager* e lista observer
- Annotazione <> su *NotificationManager*
- Annotazioni <> sui 2 observer concreti
- Nota dedicata che spiega il meccanismo di *parallelStream()*

**Integrazione con altri componenti:**

Il pattern Observer è usato da:

- *ParticipationService*: notifica approvazione/rifiuto partecipazioni
- *VoucherService*: notifica generazione/riscatto voucher
- *ChatService*: notifica nuovi messaggi
- *OutingService*: notifica creazione uscite
- *EventService*: notifica creazione eventi

**File chiave:**

- *NotificationSubject.java* (interfaccia Subject)
- *NotificationManager.java* (Subject concreto thread-safe)
- *NotificationObserver.java* (interfaccia Observer)
- *PersistenceNotificationObserver.java* (observer DB)

- *WebSocketNotificationObserver.java* (observer WebSocket)
- *NotificationObserverInitializer.java* (registrazione observer)
- *NotificationEvent.java* (evento propagato)
- Test: *NotificationObserverTest.java* (verifica parallelizzazione e fault tolerance)

---

# 4. MULTITHREADING E SINCRONIZZAZIONE

## 4.1 Meccanismi di Concorrenza

### A. Pessimistic Locking (ParticipationService)

**Ubicazione:** *participation/ParticipationService.java*

```
@Transactional(isolation = Isolation.SERIALIZABLE)
public synchronized ParticipationResponse joinOuting(UUID outingId, UUID userId) {
    // ...
    long approvedCount = participationRepository.countApprovedByOuting(outing);
    if (approvedCount >= outing.getMaxParticipants()) {
        throw new RuntimeException("Outing is full");
    }
    // ...
}
```

**Meccanismo:**

- *@Transactional(isolation = Isolation.SERIALIZABLE)*: Isolamento massimo
- *synchronized*: Sincronizzazione a livello di metodo
- *@Lock(LockModeType.PESSIMISTIC_WRITE)* su repository

**Scopo:** Prevenire race condition quando più utenti cercano di unirsi simultaneamente.

**Resource condivisa:** *Participation* (conteggio partecipanti approvati)

### B. Observer Pattern con Parallelizzazione

**Ubicazione:** *NotificationManager.notifyObservers()*

```
observers.parallelStream().forEach(observer -> {
    try {
        observer.onNotification(notification);
    } catch (Exception e) {
        log.error("Error notifying observer", e);
    }
});
```

**Meccanismo:**

- *CopyOnWriteArrayList*: Thread-safe per accesso concorrente
- *parallelStream()*: Esecuzione parallela di observer

- Gestione errori: un osservatore non blocca gli altri

**Scopo:** Notificare utenti in tempo reale senza blocking.

#### C. Scheduled Tasks

**Ubicazione 1:** *chat/ChatDeactivationScheduler.java*

```
@Scheduled(cron = "0 0 2 * * *")  // 2:00 ogni giorno
public void deactivateExpiredChats() {
    // Disattiva chat più vecchie di 7 giorni
}
```

**Ubicazione 2:** *voucher/VoucherExpirationScheduler.java*

```
@Scheduled(cron = "0 0 3 * * *")  // 3:00 ogni giorno
public void expireVouchers() {
    // Scade voucher più vecchi di 60 giorni
}
```

---

## 5. DIAGRAMMI UML

Tutti i diagrammi sono consultabili come file allegati. Nei seguenti paragrafi si specificano i nomi dei file di riferimento.

### 5.1 Class Diagram

**File:** *class-diagram*

Mostra:

- **12 entità JPA:** User, Event, Outing, Participation, Reward, Voucher, Chat, ChatMessage, Notification, Review, RefreshToken, Report
- **10 enumerazioni:** Role, ParticipationStatus, VoucherStatus, NotificationType, ReportType, ReportStatus
- **2 design pattern completi:** Strategy Pattern (limitazioni utente) + Observer Pattern (notifiche multi-canale)
- **Tutti i vincoli e relazioni:** associazioni, molteplicità, dipendenze tra package

**Approccio progettuale:**

Si è scelto un **approccio unificato** con un singolo class diagram comprensivo di tutti i package, anziché creare diagrammi separati per ogni package. Questa scelta è motivata da:

1. **Leggibilità:** visualizzazione immediata delle relazioni cross-package (es. User → Outing → Voucher)
2. **Completezza:** ogni entità è contestualizzata rispetto all'intero sistema
3. **Pattern integration:** i design pattern sono mostrati come sottografi integrati con le entità di dominio, evidenziando come Strategy e Observer interagiscono con il resto del sistema

I package sono comunque identificabili tramite:

- Commenti di sezione (*%% Core Entities*, *%% Design Pattern: Strategy Pattern*)
- Raggruppamento logico delle classi (entità di dominio, enumerazioni, pattern)
- Note esplicative per meccanismi di multithreading

**Discussione:**

Il diagramma delle classi illustra la struttura completa del dominio applicativo. Le entità core rappresentano i concetti principali del sistema: utenti con ruoli differenziati (User + Role), eventi organizzati da business (Event + Reward), uscite sociali (Outing + Participation), comunicazione real-time (Chat + ChatMessage), sistema di incentivi (Voucher), feedback (Review) e moderazione (Report).

Le 12 entità sono interconnesse da relazioni che rappresentano le dipendenze funzionali:

- **User è il fulcro** del sistema: crea eventi, organizza uscite, partecipa, scrive recensioni, guadagna voucher, invia messaggi, riceve notifiche, sottomette e riceve segnalazioni
- **Event e Outing** sono separati ma collegabili: gli utenti PREMIUM possono creare uscite associate a eventi business
- **Report** permette la moderazione della piattaforma: utenti possono segnalare contenuti/utenti inappropriati, gli admin revisionano le segnalazioni

**Design Pattern integrati:**

Il diagramma include due design pattern come sottografi visibili:

1. **Strategy Pattern** (*pattern.strategy*): implementa il polimorfismo delle limitazioni utente
    - Interfaccia *UserLimitationStrategy* definisce il contratto
    - 3 strategie concrete (*BaseUserStrategy*, *PremiumUserStrategy*, *BusinessUserStrategy*) implementano regole specifiche per ogni ruolo
    - *UserStrategyFactory* seleziona la strategia appropriata basandosi su *User.role*
    - **Integrazione**: *OutingService* usa la factory per validare la creazione di uscite in base al ruolo dell'organizzatore
2. **Observer Pattern** (*pattern.observer*): gestisce le notifiche multi-canale in modo disaccoppiato
    - *NotificationSubject* (interfaccia) definisce attach/detach/notify
    - *NotificationManager* (Subject concreto) mantiene lista thread-safe di observer tramite *CopyOnWriteArrayList*
    - 2 Observer concreti: *PersistenceNotificationObserver* (salva nel DB), *WebSocketNotificationObserver* (invia via STOMP)
    - **Multithreading**: *parallelStream()* esegue observer in parallelo (no blocking)
    - **Integrazione**: *ParticipationService*, *VoucherService*, *ChatService* notificano eventi al manager, che propaga a tutti gli observer registrati

**Meccanismi di sincronizzazione:**

Il diagramma evidenzia 3 meccanismi di controllo della concorrenza tramite note:

- **Participation:** *synchronized* + *@Transactional(SERIALIZABLE)* + pessimistic locking per prevenire race condition nell'approvazione (scenario critico: 100 utenti per 10 posti)
- **Voucher:** *@Lock(PESSIMISTIC_WRITE)* per evitare double redemption
- **NotificationManager:** *CopyOnWriteArrayList* + *parallelStream()* per notifiche concurrent senza blocking

Questo approccio integrato permette di comprendere sia la struttura statica del sistema sia i meccanismi dinamici di estensibilità (pattern) e concorrenza (multithreading).

## 5.2 Sequence Diagram - Partecipazione

**File:** *sequence-diagram*

Mostra:

- Uno user richiede di unirsi a un'uscita
- Il sistema valida la disponibilità di posti
- Si attiva Observer Pattern per le notifiche
- Le notifiche sono salvate nel DB e inviate via WebSocket in parallelo
- L'organizzatore riceve la notifica in tempo reale

**Discussione:** Il diagramma di sequenza per la partecipazione illustra il flusso completo dall'iscrizione alla ricezione della notifica real-time. L'elemento *par* (parallelo) mostra come i due osservatori (*PersistenceObserver* e *WebSocketObserver*) vengono eseguiti in parallelo tramite *parallelStream()*.

## 5.3 Sequence Diagram - Chat WebSocket

**File:** *sequence-diagram-chat-websocket*

Mostra:

- La connessione WebSocket con autenticazione JWT
- L'invio dei messaggi
- La presenza dell'indicatore di digitazione
- L'auto-disattivazione tramite lo scheduler

**Discussione:** Il diagramma della chat WebSocket mostra come i messaggi vengono trasmessi in tempo reale tramite STOMP broker. L'autenticazione è garantita dal JWT token, che viene verificato dall'interceptor del WebSocket. L'auto-disattivazione della chat dopo 7 giorni è gestita da un task scheduler che viene eseguito quotidianamente alle 2:00.

## 5.4 Sequence Diagram - Voucher

**File:** *sequence-diagram-voucher-redemption*

Mostra:

- La creazione automatica voucher dopo approvazione
- Il riscatto tramite QR code
- La gestione della scadenza tramite lo scheduler

**Discussione:** Il flusso del voucher è composto da tre fasi principali: emissione (automatica dopo l'approvazione di 5+ partecipanti), visualizzazione e riscatto. L'emissione automatica è verificata nel metodo *checkAndIssueVoucher()* del *VoucherService*, che viene chiamato durante l'approvazione di ogni partecipazione. La scadenza è gestita da uno scheduler che viene eseguito quotidianamente alle 3:00.

## 5.5 Use Case Diagram

**File:** *use-case-diagram-1-events-outings*, *use-case-diagram-2-communication-rewards*, *use-case-diagram-3-administration*

Mostra:

- 22 use case
- 4 attori (User, Premium, Business, Admin)
- Dipendenze tra use case

**Discussione:** Il diagramma dei casi d'uso rappresenta tutte le funzionalità del sistema organizzate per attore. I 22 use case coprono le operazioni principali: gestione uscite (UC1-UC5, UC13-UC14), comunicazione (UC6, UC12, UC15), premi e voucher (UC7, UC11, UC16-UC18), recensioni (UC8, UC19) e amministrazione (UC10, UC20-UC22). Le dipendenze mostrano come certi casi d'uso estendono o richiedono altri; ad esempio, UC4 (Join Outing) è necessario per UC6 (Send Chat Message) e UC12 (View Chat History).

---

# 6. INTERFACCIA GRAFICA - SCREENSHOT APPLICAZIONE

Questa sezione presenta 18 screenshot dell'applicazione frontend Angular che illustrano le principali funzionalità e il design dell'interfaccia utente. Gli screenshot sono organizzati per flusso funzionale e mostrano l'implementazione concreta dei requisiti descritti nelle sezioni precedenti.

## 6.1 Autenticazione e Navigazione

**Screenshot 6.1.1: Pagina di Login**

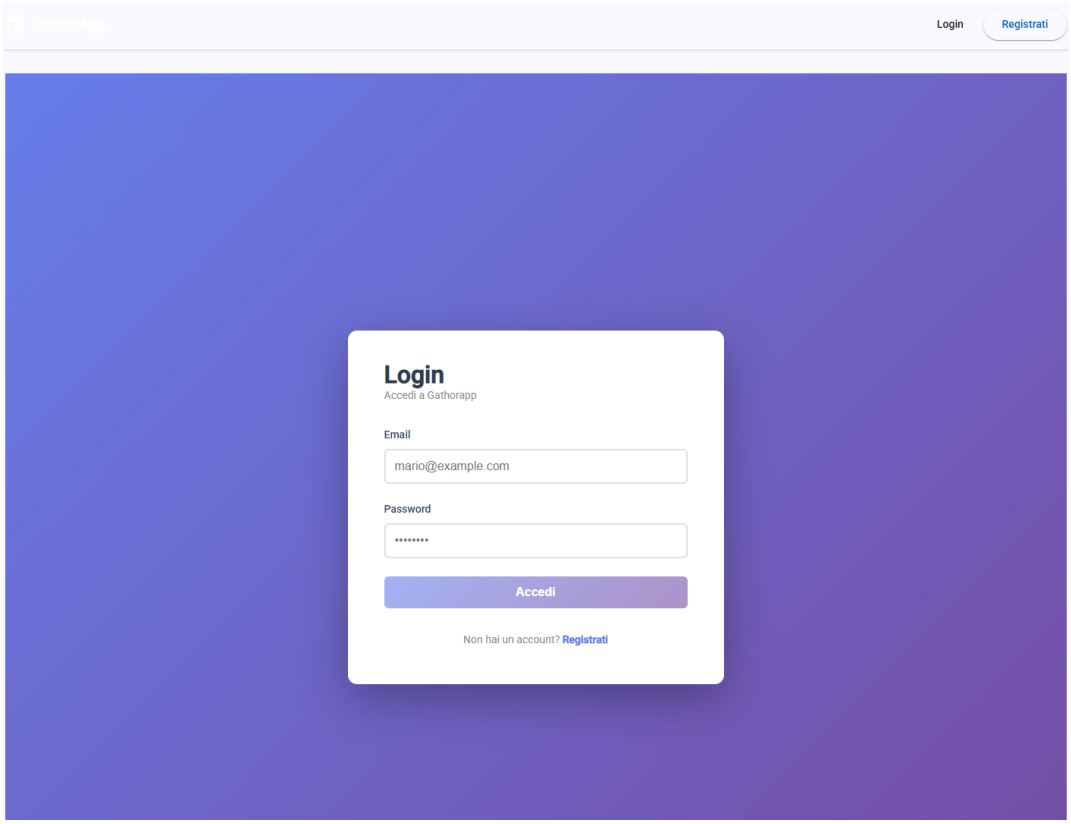


Figure 1: Login

**Descrizione:** Form di login con autenticazione JWT, validazione client-side.

**Screenshot 6.1.2: Pagina di Registrazione**

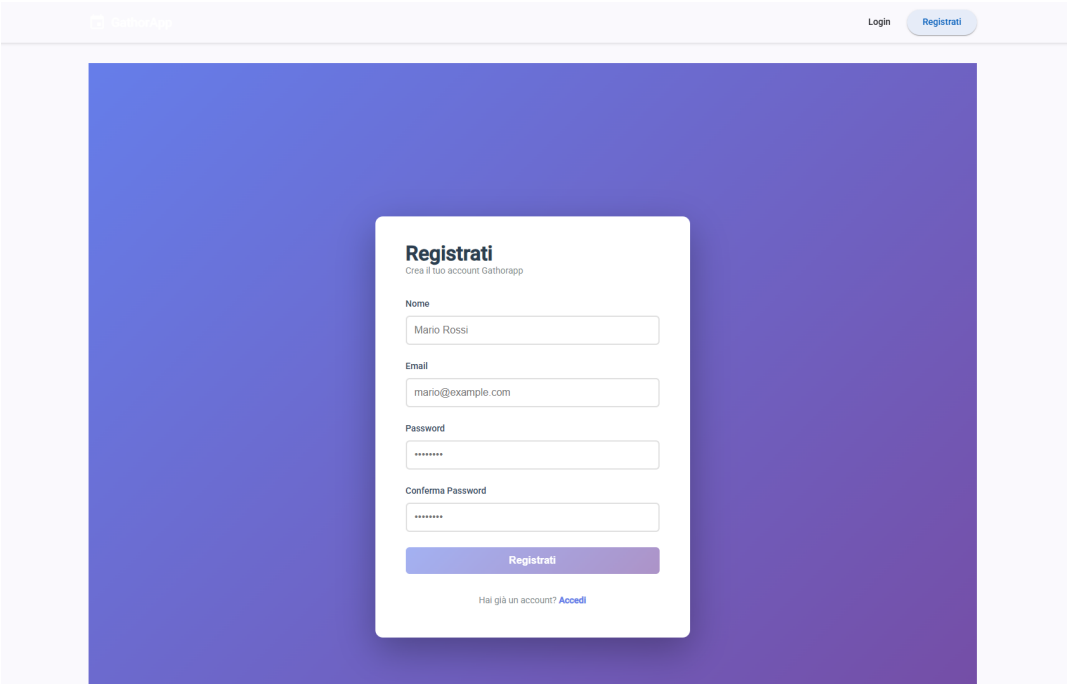


Figure 2: Registrazione

**Descrizione:** Form registrazione con validazione real-time dei campi (nome, email, password, conferma). Redirect a login dopo creazione account.

**Screenshot 6.1.3: Navbar con Menu Utente**

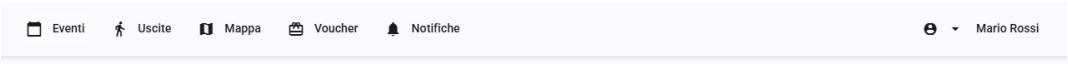


Figure 3: Navbar

**Descrizione:** Navbar con logo, menu principale (Eventi, Uscite, Mappa, Voucher), icona notifiche, menu utente. Voci differenziate per ruolo (Admin ha pannello).

**Screenshot 6.1.4: Dropdown Notifiche**

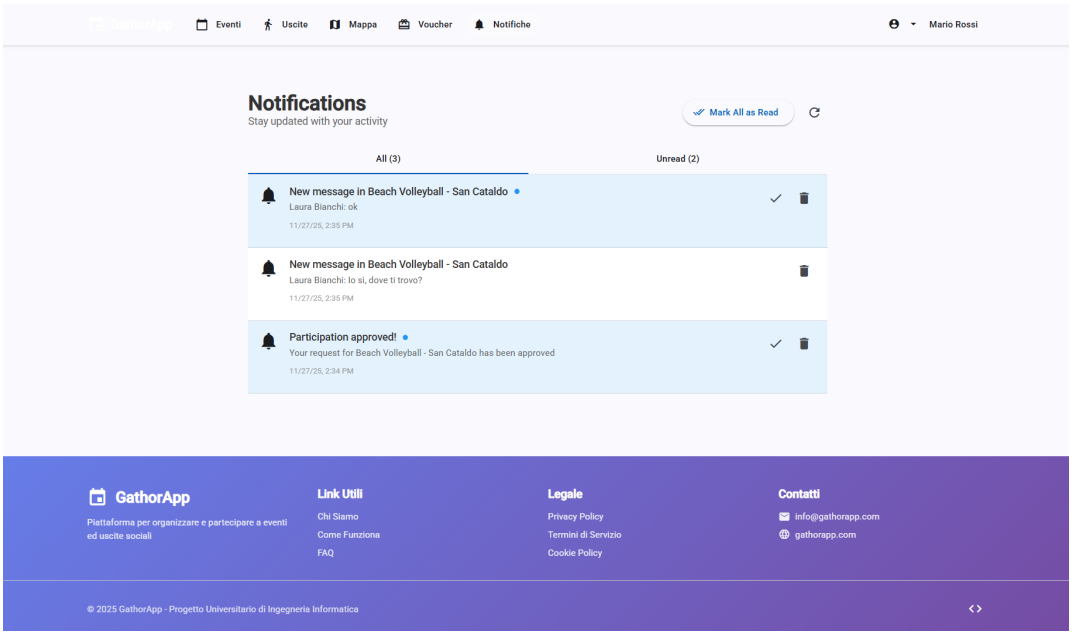


Figure 4: Notifiche

**Descrizione:** Lista notifiche, timestamp relativo. Non lette evidenziate.

## 6.2 Eventi

### Screenshot 6.2.1: Lista Eventi

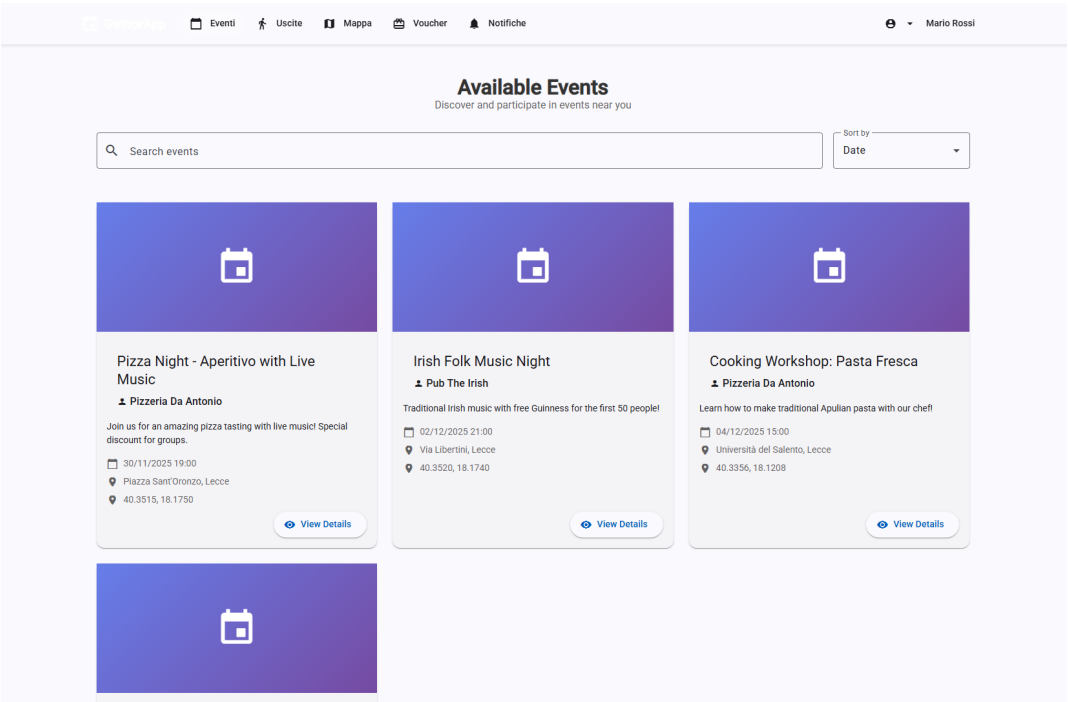


Figure 5: Lista Eventi

**Descrizione:** Griglia responsive di card eventi con immagine, titolo, location, data. Filtri e ordinamento.

**Screenshot 6.2.2: Dettaglio Evento**

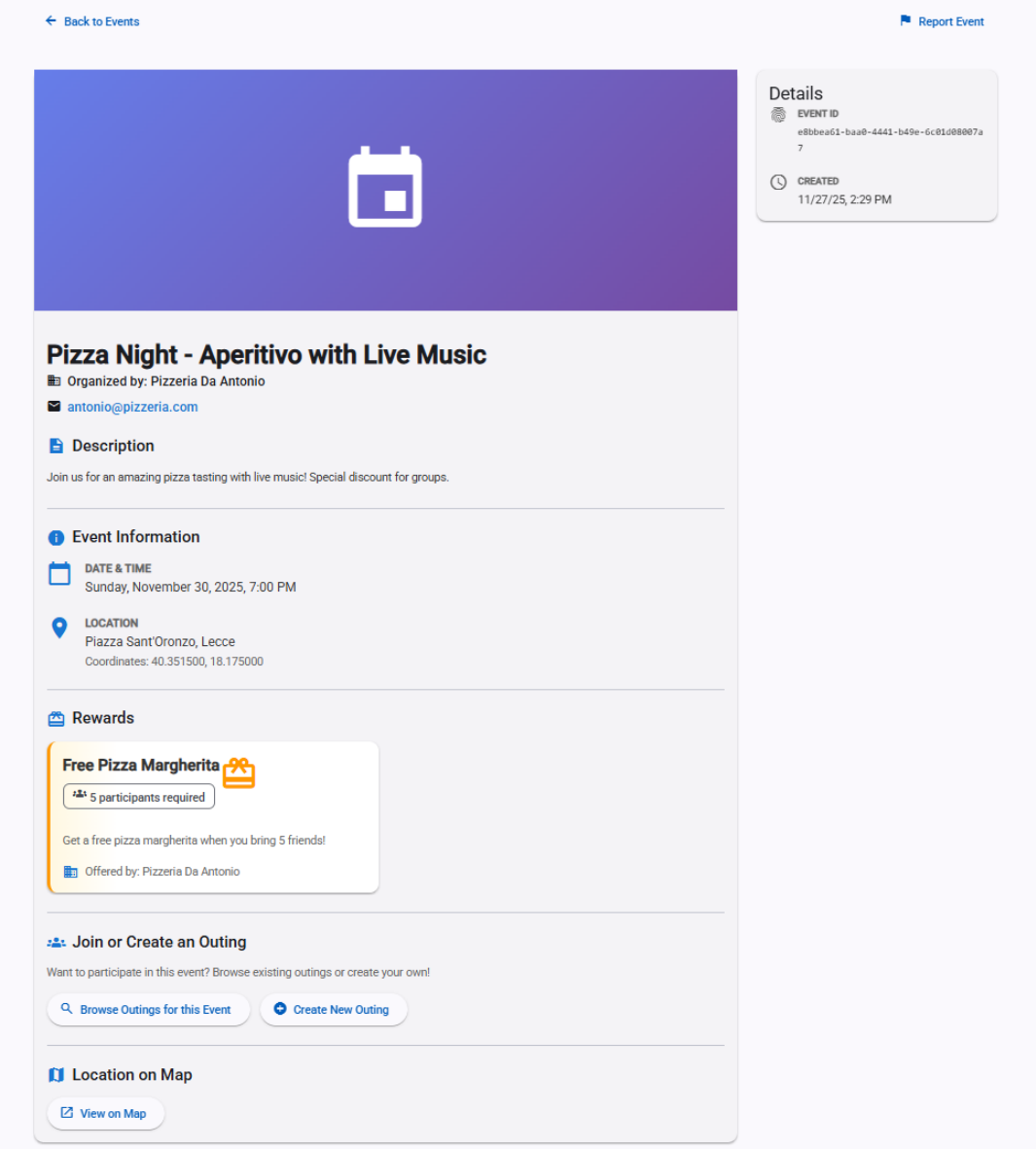


Figure 6: Dettaglio Evento

**Descrizione:** Dettaglio con titolo, organizzatore, descrizione, dettagli. Sezione "Ricompense" per utenti premium, sezione per cercare uscite collegate o creare un'uscita collegata. Link per pagina mappa. Possibilità di segnalazione.

**Screenshot 6.2.3: Form Creazione Evento**

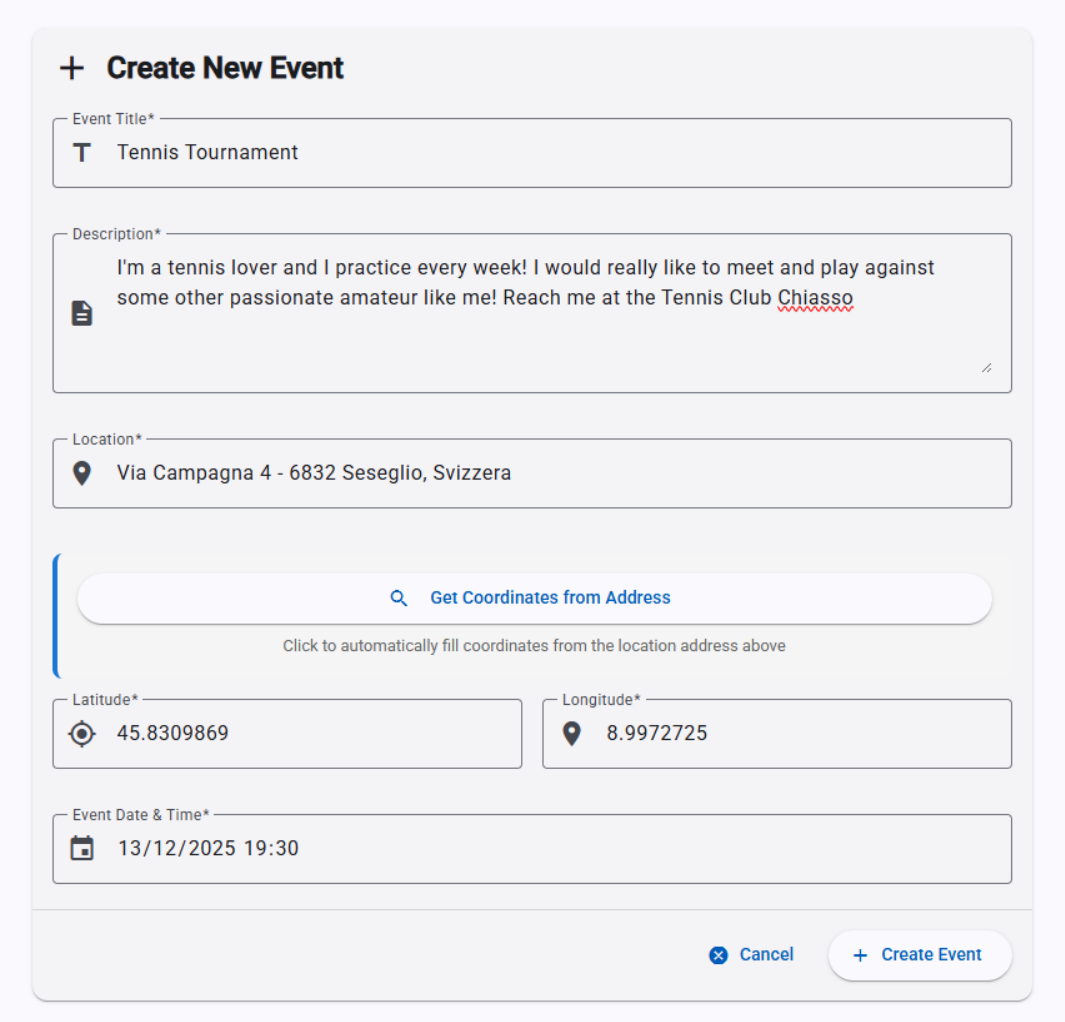


Figure 7: Creazione Evento

**Descrizione:** Form: titolo/descrizione, risoluzione automatica coordinate da indirizzo, datepicker data/ora. Validazione real-time.

## 6.3 Uscite

### Screenshot 6.3.1: Lista Uscite

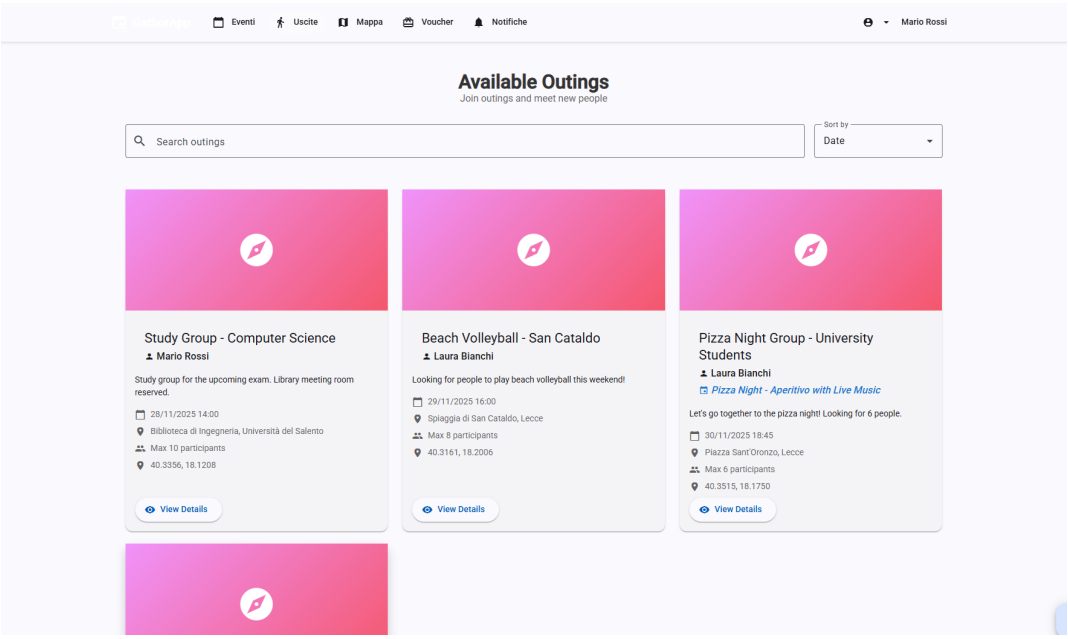


Figure 8: Lista Uscite

**Descrizione:** Griglia card con organizzatore, max posti disponibili, location. Indicazione per uscite collegate ad eventi. Filtri e ordinamento.

**Screenshot 6.3.2: Dettaglio Uscita**

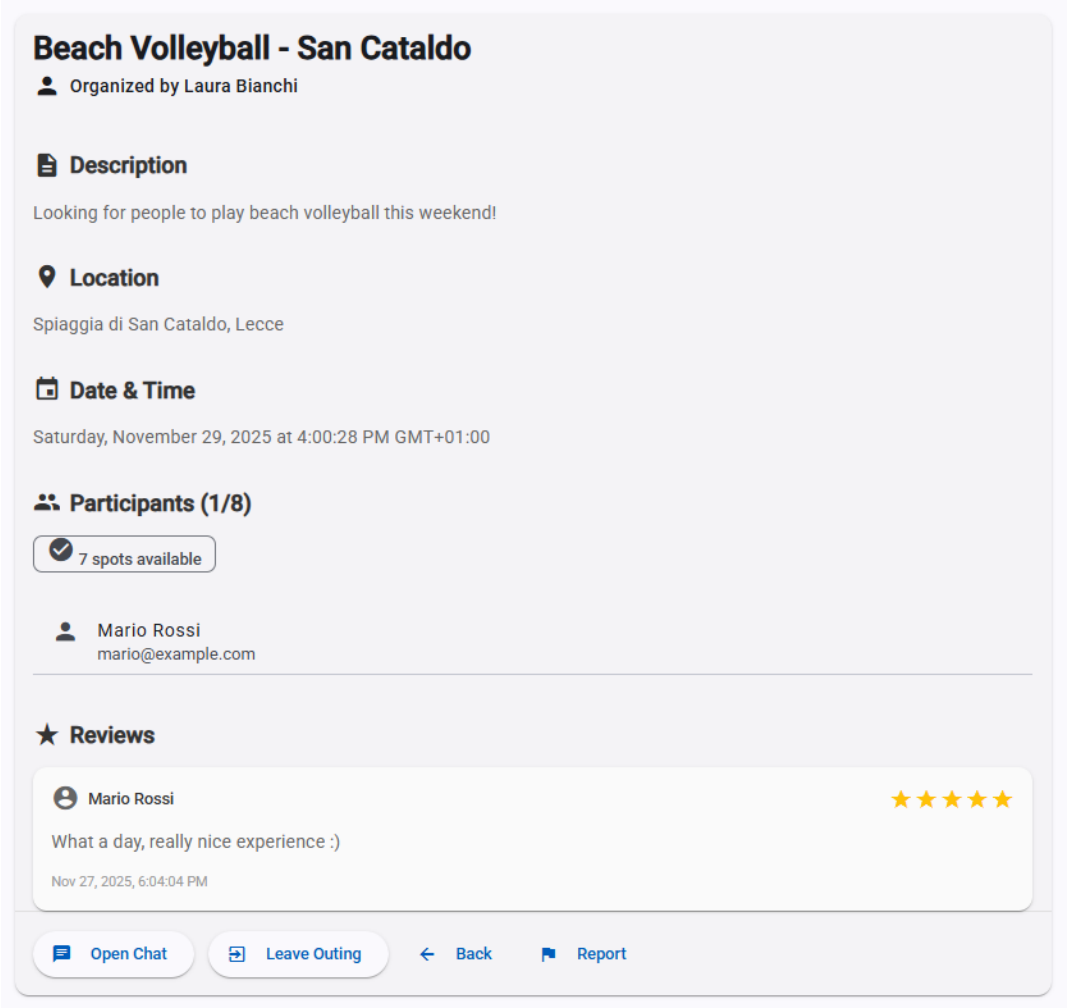


Figure 9: Dettaglio Uscita

**Descrizione:** Dettagli, lista partecipanti, accesso alla chat, recensioni. Lista partecipanti da approvare per organizzatore. Bottone "Partecipa", "Lascia". Possibilità di segnalazione.

**Screenshot 6.3.3: Form Creazione Uscita**

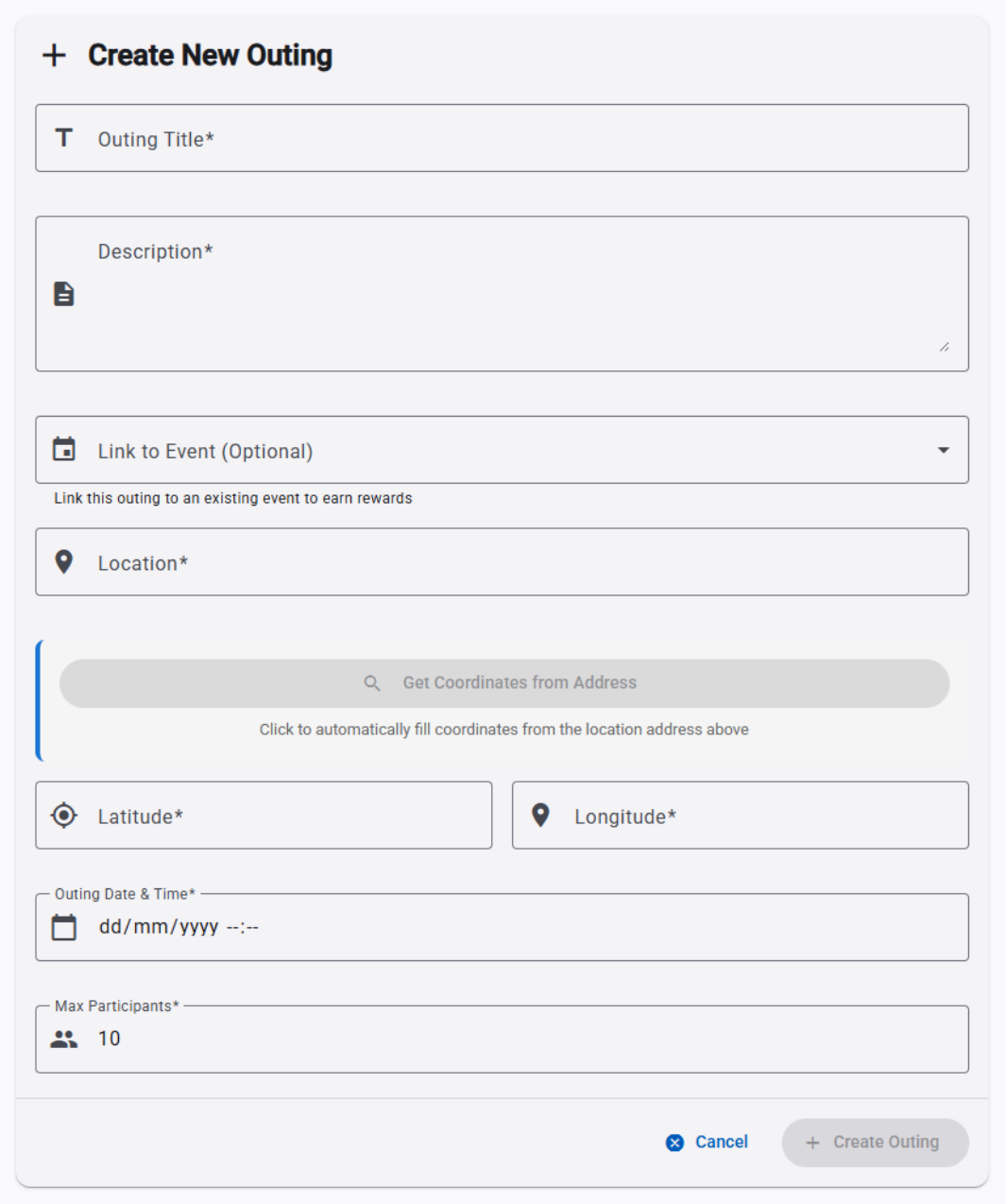


Figure 10: Creazione Uscita

**Descrizione:** Form con dropdown eventi da linkare, risoluzione automatica da indirizzo a coordinate, max partecipanti con limite per ruolo. Datepicker.

**Screenshot 6.3.4: Gestione Partecipanti (Organizzatore)**

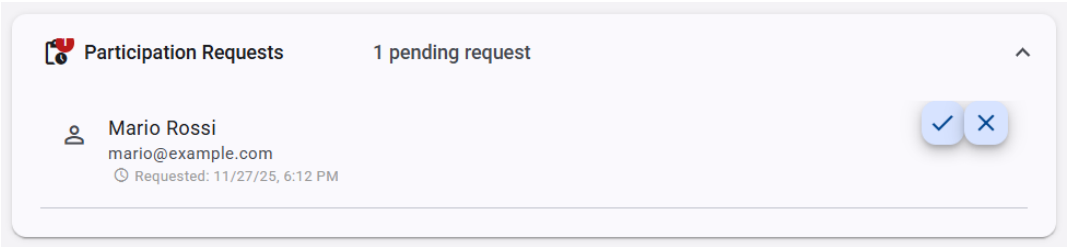


Figure 11: Gestione Partecipanti

**Descrizione:** Lista con partecipanti da approvare. Bottoni Approva/Rifiuta. Notifica real-time al partecipante.

## 6.4 Chat Real-Time

### Screenshot 6.4.1: Chat Attiva

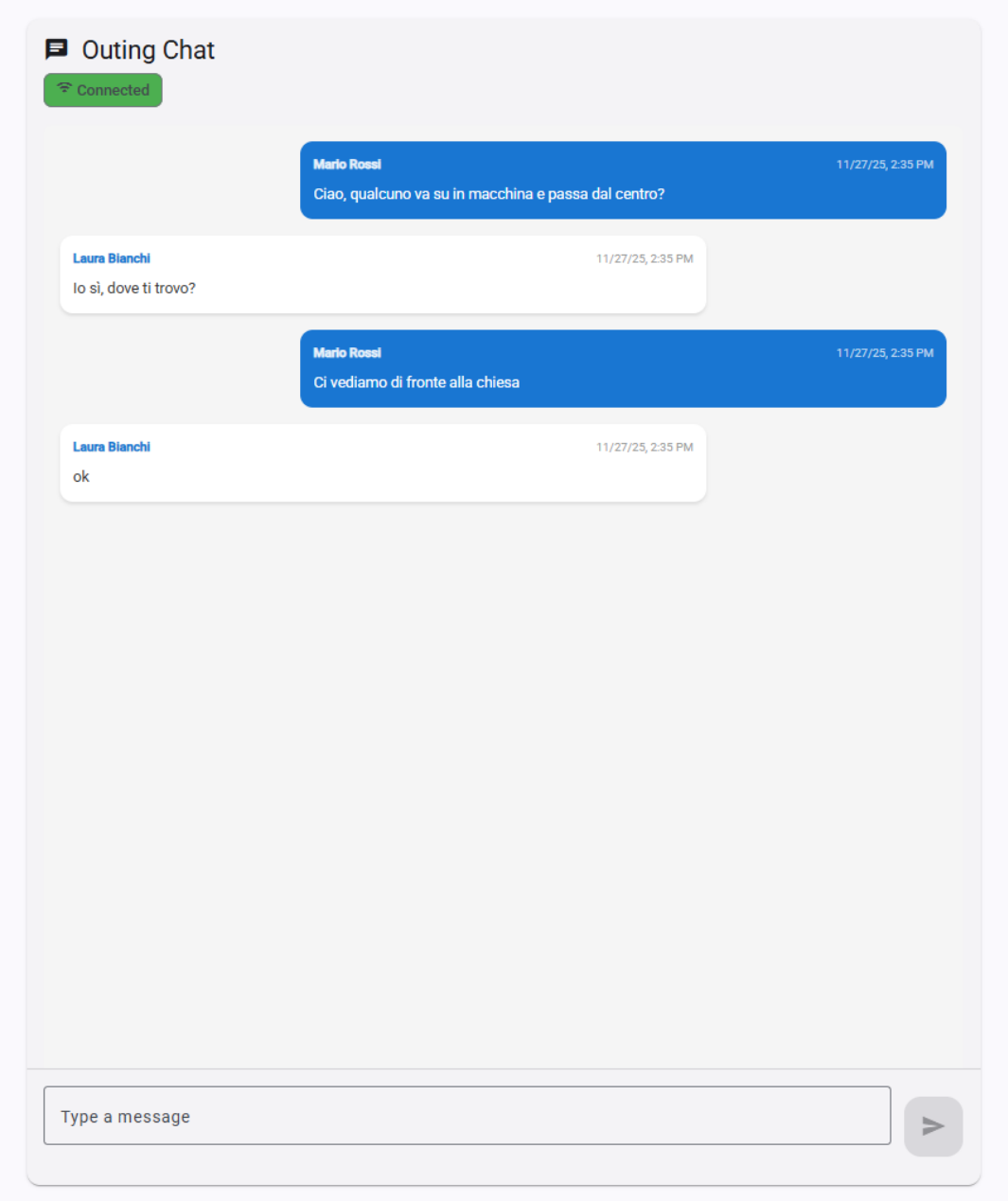


Figure 12: Chat Attiva

**Descrizione:** Chat con bubble differenziati, nome utente, timestamp. Indicatore digitazione "sta scrivendo…". Input testo. WebSocket real-time.

## 6.5 Mappa e Geolocalizzazione

**Screenshot 6.5.1: Vista Mappa con Marker**

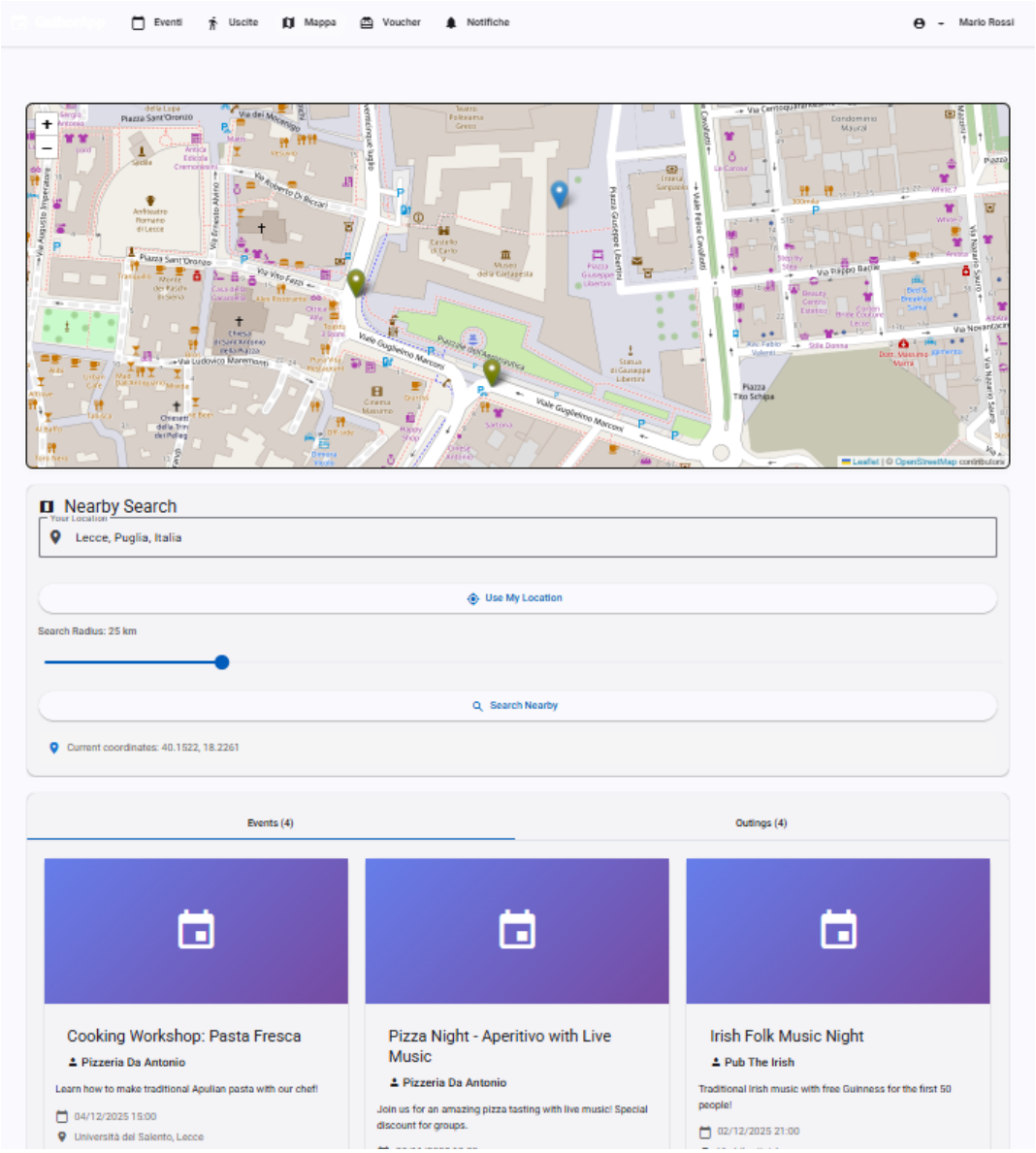


Figure 13: Mappa

**Descrizione:** Mappa Leaflet con segnaposto (colori differenziati tra eventi e ucite). Filtro raggio. Risoluzione automatica indirizzo. Possibilità di utilizzare posizione utente.

## 6.6 Voucher

### Screenshot 6.6.1: Lista Voucher

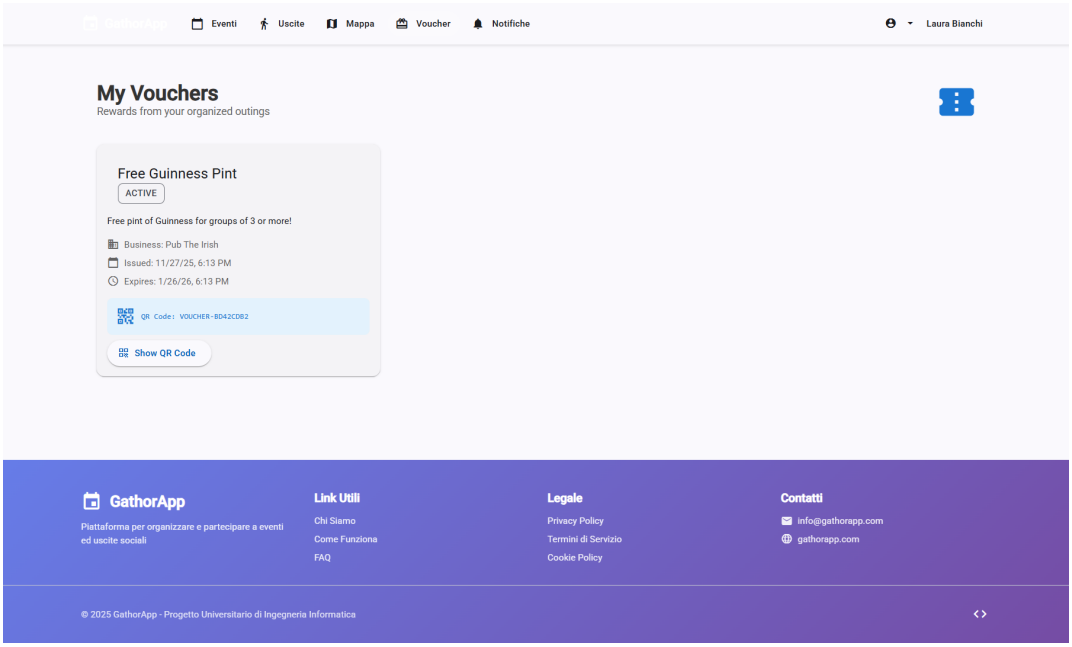


Figure 14: Lista Voucher

**Descrizione:** Griglia voucher con QR code, premio, entità rilascio, status, data rilascio e data scadenza.

## 6.7 Profilo Utente

**Screenshot 6.7.1: Profilo - Tab Informazioni**

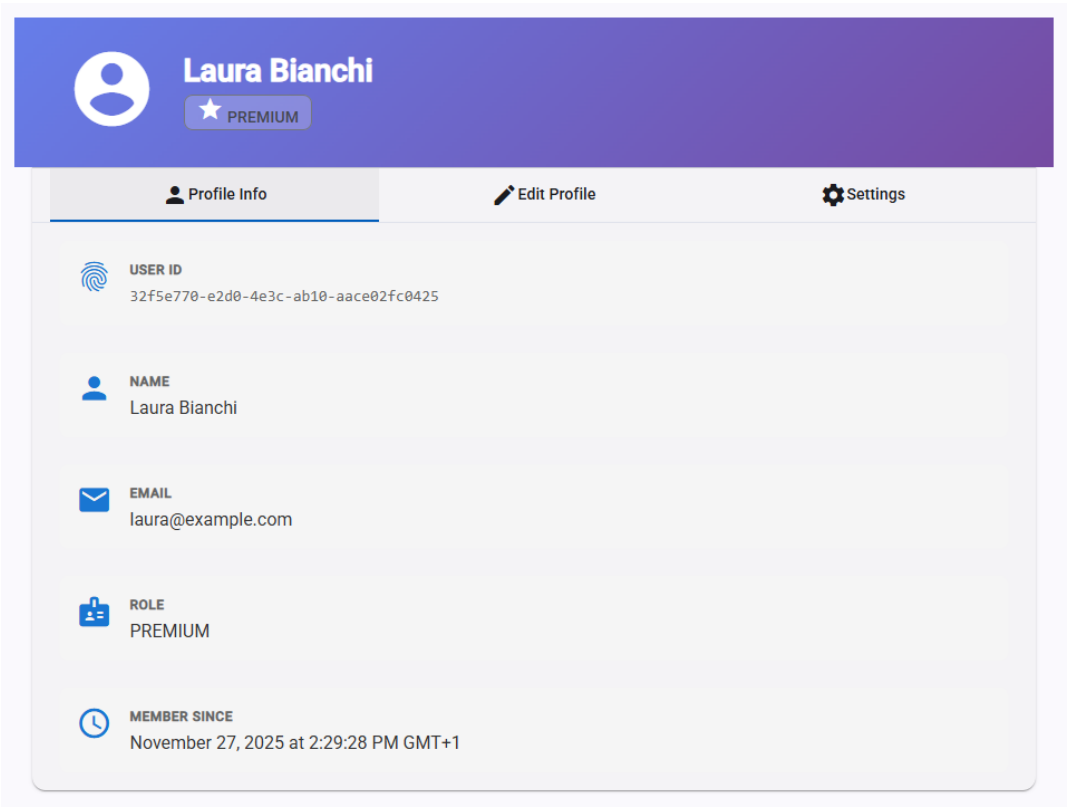


Figure 15: Profilo Info

**Descrizione:** Header con nome, ruolo. Tab Informazioni: dettagli account.

**Screenshot 6.7.2: Profilo - Tab Modifica**

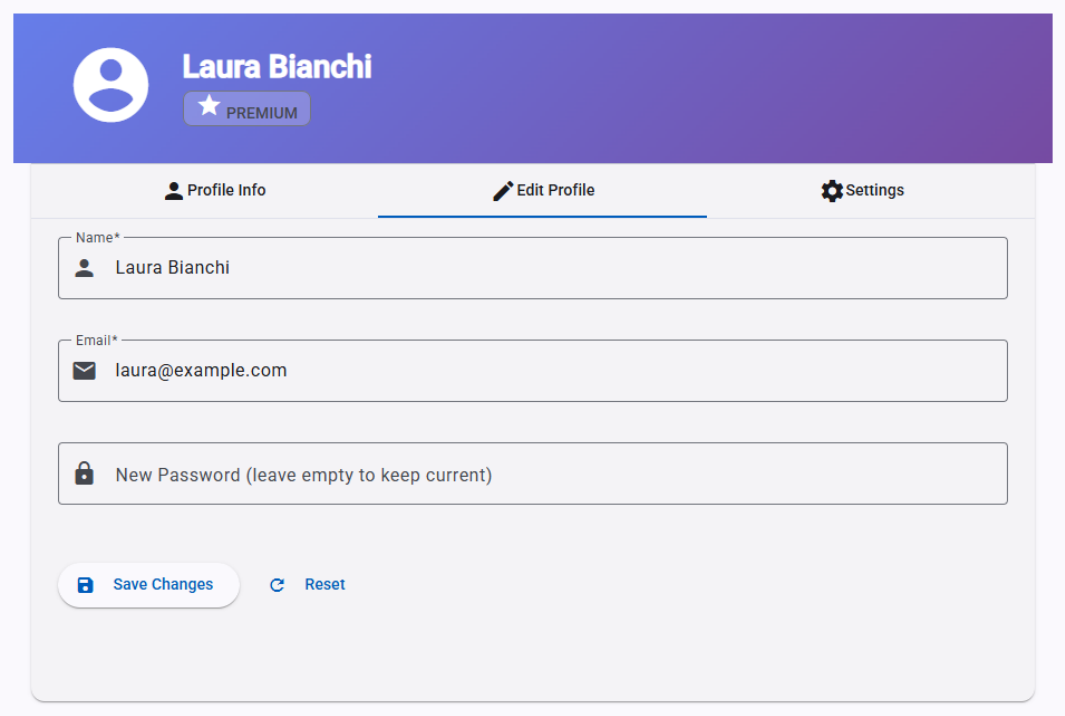


Figure 16: Profilo Edit

**Descrizione:** Form editabile: nome, email password. Validazione modifiche. Possibilità reset prima di salvare.

**Screenshot 6.7.3: Profilo - Tab Impostazioni**

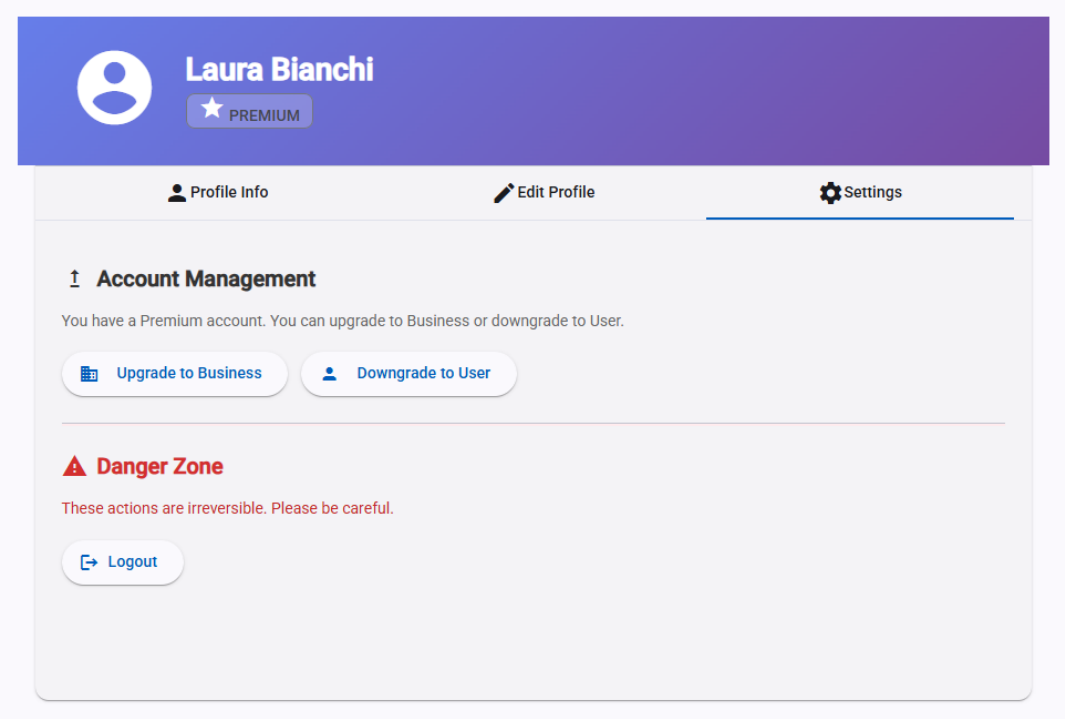


Figure 17: Profilo Settings

**Descrizione:** Upgrade / downgrade profilo. Logout.

## 6.8 Amministrazione

**Screenshot 6.8.1: Dashboard Admin**

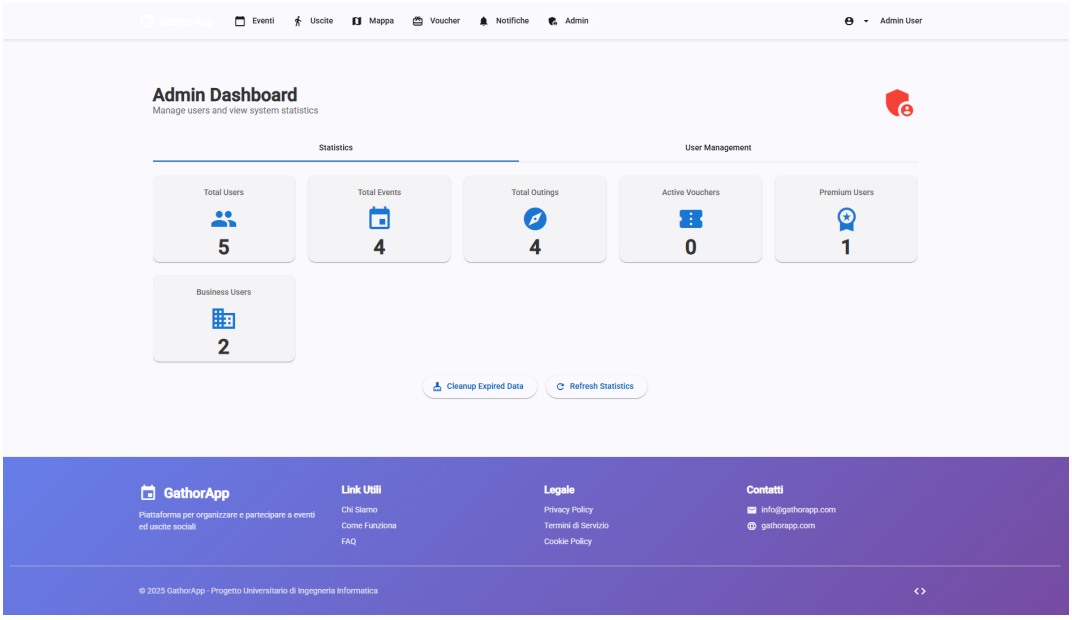


Figure 18: Admin Dashboard

**Descrizione:** Dashboard con card statistiche. Nella tab user management é possibile cambiare ruolo agli utenti e bannarli/sbannarli.

---

# 7. PIANO DI TEST

## 7.1 Test di Unità (Unit Tests)

**Ubicazione:** *backend/src/test/java/com/alfano/gathorapp/*

**Copertura richiesta:** >= 80%

**Test principali:**

| Package | Test Class | Count | Oggetto |
|---|---|---|---|
| pattern/strategy | UserLimitationStrategyTest | 30 | Limiti per ruolo |
| pattern/observer | NotificationObserverTest | 20 | Observer + parallelizz. |
| participation | ParticipationServiceTest | 20 | Race condition, locking |
| outing | OutingServiceTest | 26 | CRUD e validazioni |
| event | EventServiceTest | 14 | Gestione eventi |
| voucher | VoucherServiceTest | 19 | Generazione voucher |
| chat | ChatServiceTest | 24 | Invio messaggi |
| chat | ChatDeactivationSchedulerTest | 8 | Auto-disattivazione |
| user | UserServiceTest | 19 | Gestione utenti |
| auth | AuthServiceTest | 11 | JWT Authentication |

| | | | |
|---|---|---|---|
| notification | NotificationServiceTest | 15 | Creazione notifiche |
| review | ReviewServiceTest | 15 | Gestione recensioni |

**Framework:** JUnit 5 + Mockito

**Totale test:** 585 test eseguiti (di cui 221 principali elencati sopra)

**Comando esecuzione:**

```
cd backend
gradle test
gradle jacocoTestReport
```

### 7.2 Test di Integrazione (Integration Tests)

Test di controller con MockMvc per verificare l'integrazione tra controller, service e repository:

- *OutingControllerIntegrationTest* - Test endpoint outings (7 test)
- *ParticipationControllerIntegrationTest* - Test endpoint partecipazioni (8 test)
- *ChatControllerIntegrationTest* - Test endpoint chat (6 test)
- *VoucherControllerIntegrationTest* - Test endpoint voucher (9 test)
- *UserControllerIntegrationTest* - Test endpoint utenti (3 test)
- *ReviewControllerIntegrationTest* - Test endpoint recensioni (3 test)

### 7.3 Test Manuale - Chat WebSocket

Procedura di test manuale:

1. Connessione WebSocket con JWT (verificare autenticazione)
2. Invio e ricezione messaggi tra utenti
3. Indicatore di digitazione in tempo reale
4. Auto-disattivazione chat dopo 7 giorni dall'uscita

### 7.4 Report Copertura JaCoCo

**Ubicazione:** *backend/build/reports/jacoco/test/html/index.html*

**Copertura raggiunta:** 88% (verificata con JaCoCo)

**Verifica minima 80%:**

```
gradle jacocoTestCoverageVerification
```

**Report:** Analizza la copertura per ogni package, classe e metodo. Le classi critiche (ParticipationService, NotificationManager, VoucherService) hanno una copertura superiore all'85%.

**Code Coverage Report**

| Pkg | Missed Instr. (Lost/Cov.) | Cov. | Missed Branches (Lost/Cov.) | Cov. | Missed (Cxty) | Cxty | Missed (Lns) | Lns | Missed (Mthds) | Mthds | Missed (Clss) | Clss |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Total** | **986 of 8,619** | **88%** | **55 of 347** | **84%** | **113** | **627** | **239** | **2,008** | **71** | **452** | **1** | **94** |
| *event* | 111 / 322 | 74% | 4 / 10 | 71% | 12 | 32 | 29 | 109 | 8 | 25 | 0 | 4 |
| *admin* | 110 / 339 | 75% | - | n/a | 9 | 23 | 28 | 104 | 9 | 23 | 0 | 2 |
| *websock.* | 102 / 64 | 38% | 12 / 0 | 0% | 8 | 12 | 25 | 36 | 2 | 6 | 0 | 2 |
| *notific.* | 92 / 292 | 76% | 0 / 6 | 100% | 6 | 26 | 24 | 100 | 6 | 23 | 0 | 4 |
| *review* | 91 / 353 | 79% | 2 / 38 | 95% | 8 | 46 | 25 | 107 | 7 | 26 | 0 | 6 |
| *except.* | 80 / 306 | 79% | - | n/a | 5 | 19 | 18 | 80 | 5 | 19 | 0 | 4 |
| *auth* | 77 / 425 | 84% | 2 / 12 | 85% | 8 | 33 | 18 | 139 | 6 | 26 | 0 | 4 |
| *reward* | 70 / 127 | 64% | 1 / 5 | 83% | 4 | 14 | 21 | 52 | 3 | 11 | 0 | 4 |
| *user* | 46 / 449 | 90% | 11 / 31 | 73% | 9 | 47 | 8 | 108 | 1 | 25 | 0 | 5 |
| *report* | 44 / 489 | 91% | 1 / 3 | 75% | 9 | 34 | 4 | 121 | 8 | 32 | 1 | 7 |
| *outing* | 36 / 734 | 95% | 4 / 24 | 85% | 6 | 57 | 7 | 180 | 3 | 43 | 0 | 4 |
| *voucher* | 35 / 574 | 94% | 6 / 34 | 85% | 8 | 48 | 8 | 149 | 2 | 28 | 0 | 7 |
| *partic.* | 23 / 605 | 96% | 5 / 19 | 79% | 5 | 38 | 3 | 140 | 2 | 26 | 0 | 5 |
| *config* | 18 / 699 | 97% | 1 / 39 | 97% | 1 | 41 | 7 | 163 | 0 | 21 | 0 | 4 |
| *secur.* | 18 / 140 | 88% | 0 / 8 | 100% | 6 | 21 | 5 | 39 | 6 | 17 | 0 | 4 |
| *config seed* | 17 / 138 | 89% | 4 / 4 | 50% | 4 | 8 | 3 | 26 | 0 | 4 | 0 | 1 |
| *chat* | 11 / 686 | 98% | 2 / 28 | 93% | 4 | 50 | 4 | 170 | 2 | 35 | 0 | 12 |
| *root* | - | 37% | - | n/a | 1 | 2 | 2 | 3 | 1 | 2 | 0 | 1 |
| *map* | 0 / 560 | 100% | 0 / 14 | 100% | 0 | 31 | 0 | 87 | 0 | 24 | 0 | 6 |
| *patt. obs.* | 0 / 211 | 100% | 0 / 4 | 100% | 0 | 18 | 0 | 62 | 0 | 16 | 0 | 4 |
| *patt. str.* | 0 / 117 | 100% | 0 / 13 | 100% | 0 | 27 | 0 | 33 | 0 | 20 | 0 | 4 |

---

# 8. DEPLOYMENT E ESECUZIONE

## 8.1 Avvio Backend

**Prerequisiti:** java JDK 17 e gradle installati

```
cd backend

# Build
gradle clean build

# Esecuzione con H2
gradle bootRun
```

**Backend disponibile:** *http://localhost:8080*
**Swagger UI:** *http://localhost:8080/swagger-ui.html*

## 8.2 Avvio Frontend

**Prerequisiti:** NodeJS installato

```
cd frontend
npm install
npm start
```

**Frontend disponibile:** *http://localhost:4200*

### 8.3 Database Seeding

Il *DataSeeder* crea automaticamente:

- Admin user (admin@gathorapp.com / admin123)
- Utenti business con eventi e premi
- Utenti premium
- Utenti standard
- Uscite

---

## 9. CONSIDERAZIONI SULLA PROGETTAZIONE

### 9.1 Scelta del linguaggio

- Richiesta del corso: Java è il linguaggio consigliato
- Programmazione a oggetti: sfrutta appieno le potenzialità OOP di Java
- Framework maturo: Spring Boot è ben documentato e largamente utilizzato
- Performance: Java è performante per applicazioni web
- Type safety: Type system forte previene bug

### 9.2 Gestione della Concorrenza

La gestione della concorrenza è fondamentale nel sistema di partecipazioni:

**Scenario critico:** 100 utenti cercano di unirsi a un'uscita con 10 posti disponibili.

**Soluzione implementata:**

1. **SERIALIZABLE isolation:** Impedisce "phantom reads"
2. **Synchronized method:** Lock a livello di thread JVM
3. **Pessimistic locking:** Database-level lock (SELECT FOR UPDATE)

Questo triplo livello garantisce che solo 10 transazioni avranno successo.

### 9.3 Pattern Observer e Prestazioni

Con *parallelStream()*, le notifiche vengono inviate in parallelo:

- Osservatore database: salva su DB (~10ms)
- Osservatore WebSocket: invia messaggio real-time (~50ms)

Senza parallelizzazione: ~60ms sequenziale
Con parallelizzazione: ~50ms parallelo (tempo del più lento)

### 9.4 Scelta delle Tecnologie Frontend

Angular 20 è stato scelto per:

- Componenti standalone (nessun NgModule)
- Type safety con TypeScript
- OpenAPI Generator per client auto-generato
- Supporto WebSocket nativo

---

## 10. QUALITÀ DEL CODICE

- **Javadoc completo:** Ogni classe e metodo documentato
- **Logging:** SLF4J con livelli DEBUG, INFO, WARN, ERROR
- **Exception handling:** Eccezioni custom e messaggi specifici
- **Naming conventions:** Seguono Java standard
- **DRY principle:** Riduzione della duplicazione di codice
- **SOLID principles:** Applicati nel design

---

## 11. CONFORMITÀ AI REQUISITI DELL'ESAME

 **Architettura client/server:** Backend Spring + Frontend Angular
 **Pattern MVC:** Controller → Service → Repository
 **Pattern aggiuntivi:** Strategy + Observer (documentati e testati)
 **OOP:** Uso estensivo di classi, ereditarietà, polimorfismo, incapsulamento
 **Unit test >= 80%:** JUnit 5 + JaCoCo (88% copertura raggiunta)
 **Interfaccia grafica:** Angular 20 con componenti standalone
 **Multithreading:** Observer pattern con *parallelStream()* + pessimistic locking con *synchronized*
 **Documentazione:** UML completo + Javadoc + README
 **Resource condivisa:** *Participation* entity con controllo concorrenza
 **Lock/Semafori:** SERIALIZABLE isolation + pessimistic locking + synchronized

---

## RIFERIMENTI


- Repository GitHub: https://github.com/lysandre995/gathorapp
- Spring Boot Documentation: https://docs.spring.io/spring-boot/
- Angular Documentation: https://angular.io
- OpenAPI/Swagger: http://localhost:8080/swagger-ui.html
- Design Patterns: https://refactoring.guru/design-patterns
- Java Concurrency: https://docs.oracle.com/javase/tutorial/essential/concurrency/

## ALLEGATI

- Class Diagram (class-diagram.pdf)
- Use Case Diagram: Eventi e Uscite (use-case-diagram-1-events-outings.pdf)
- Use Case Diagram: Comunicazione e Ricompense (use-case-diagram-2-communication-rewards.pdf)
- Use Case Diagram: Amministrazione (use-case-diagram-3-administration.pdf)
- Sequence Diagram: Outing Participation Flow (sequence-diagram.pdf)
- Sequence Diagram: Chat Websocket (sequence-diagram-chat-websocket.pdf)
- Sequence Diagram: Voucher Redemption (sequence-diagram-voucher-redemption.pdf)