POLITECNICO
MILANO 1863

Politecnico di Milano
Facoltà di Ingegneria Industriale e dell'Informazione
Corso di Laurea Triennale in Ingegneria Informatica

EventGURU

Gateway to unforgettable adventures

Candidato:

Nome **Fontana Fabrizio**
Mat. **985114**
Cod. P. **10828904**

Relatore:

Prof. **Salnitri Mattia**

Anno Accademico 2023 – 2024

Indice

1 EventGURU

1.1 Panoramica

L'isolamento digitale è un fenomeno che si è diffuso in modo significativo con l'avvento delle tecnologie. Questo termine si riferisce alla condizione in cui individui o gruppi sociali si trovano sempre più soli e distanti dagli altri a causa dell'uso e dell'abuso di dispositivi e delle piattaforme online.

Le tecnologie digitali hanno rivoluzionato la comunicazione e la connettività globale, permettendo alle persone di connettersi e interagire in modi nuovi ed entusiasmanti. Tuttavia, questo stesso progresso ha portato con sé una serie di sfide legate all'isolamento.

1.2 Obiettivo

La visione di EventGURU consiste nell'utilizzare queste stesse tecnologie per riavvicinare le persone. L'obiettivo attuale è quello di creare uno spazio in cui gli organizzatori possono dare vita alle proprie passioni e agli eventi che amano, consentendo contemporaneamente alle persone di scoprire tali esperienze, partecipare attivamente e condividere la loro gioia.

1.3 Analisi dei requisiti

1.3.1 Requisiti funzionali

I requisiti funzionali sono una parte fondamentale del processo di sviluppo del software, poiché forniscono una base solida per la progettazione, lo sviluppo, la verifica e la validazione dello stesso. La documentazione dei requisiti funzionali aiuta a garantire che il prodotto software soddisfi le aspettative degli utenti e i bisogni dell'organizzazione. Il progetto supporta l'interazione di tre tipologie di utenti:

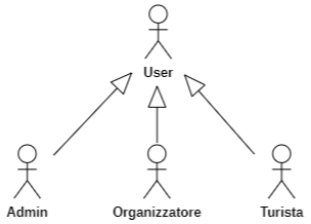
Admin – Può creare, modificare ed eliminare eventi, turisti e organizzatori, visualizzare e filtrare gli elenchi di eventi, turisti e organizzatori, nonché gestire il proprio account, inclusa la possibilità di recuperare la password tramite e-mail.

Turista – Può visualizzare, filtrare e iscriversi agli eventi, iscriversi alla newsletter, gestire i propri eventi, lasciare recensioni dopo gli eventi, seguire organizzatori, modificare i propri dati e gestire il proprio account, compresa la possibilità di recuperare la password tramite e-mail.

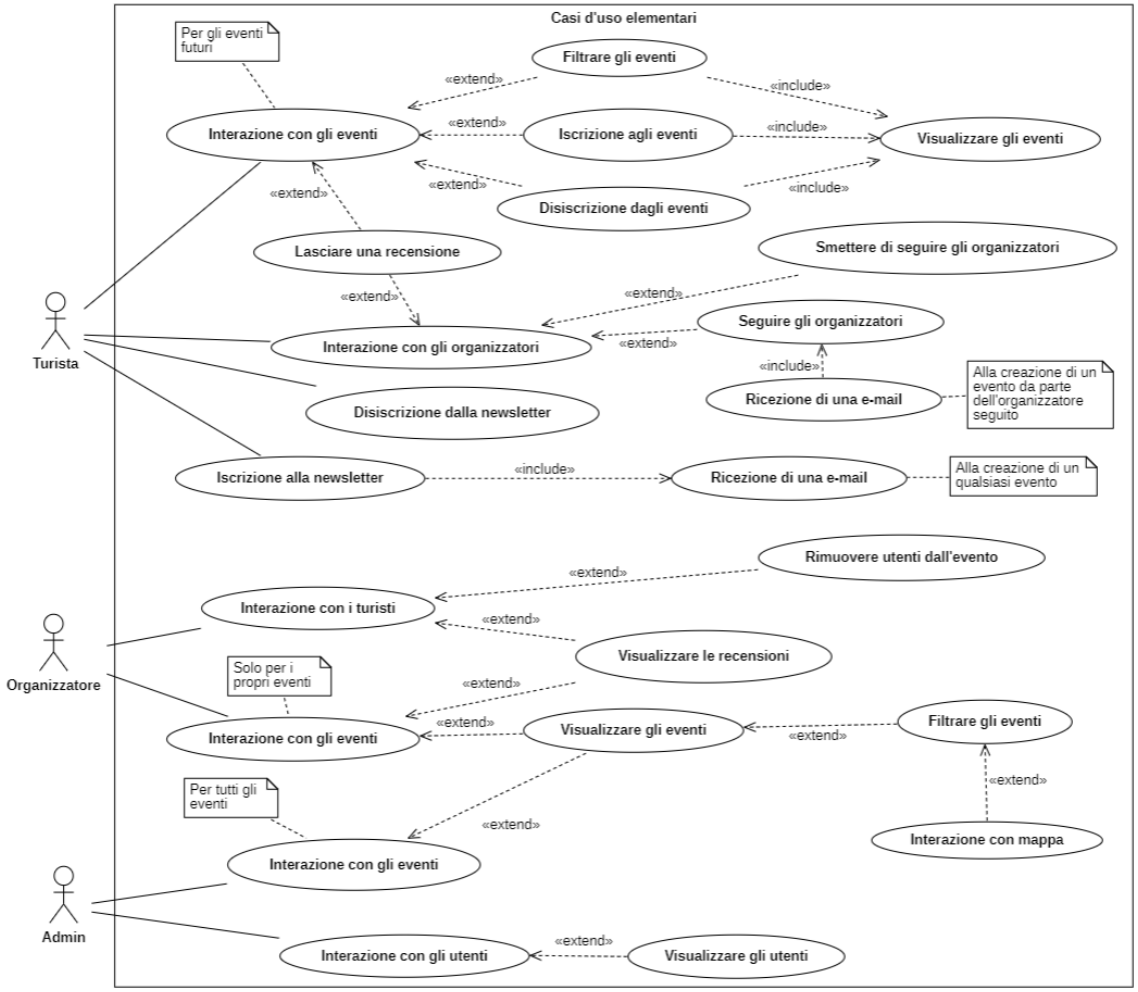
Organizzatore – Può creare, gestire e modificare eventi, visualizzare e filtrare gli eventi da loro creati, visualizzare recensioni e rating di eventi passati, essere seguito da turisti, gestire il proprio account, inclusa la possibilità di recuperare la password tramite e-mail.

1.3.1.1 **Use Case diagrams**

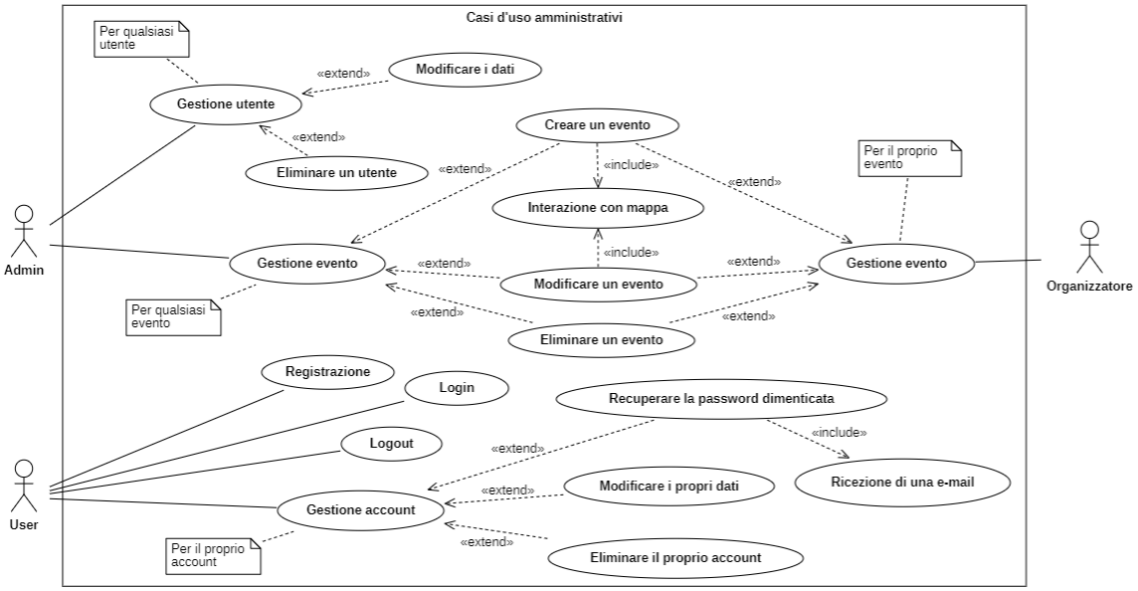
Di seguito due Use Case diagrams che sintetizzano chiaramente tutte le azioni tramite cui gli attori possono interagire con l'applicazione. Si tenga sempre conto della seguente generalizzazione, omessa dai due diagrammi per ovvie questioni di spazio e ordine:



Generalizzazione degli attori



Use Case diagram 1 – Schema dei casi d'uso elementari, cioè le principali azioni basate sul topic dell'applicazione. Comprende l'interazione tra gli utenti stessi e l'interazione tra utenti ed eventi.



Use Case diagram 2 – Schema dei casi d'uso d'amministrazione. Comprende creazione, eliminazione e modifica di eventi e utenti.

1.3.2 **Requisiti non funzionali**

I requisiti non funzionali in un'applicazione web si concentrano sulla qualità, sulle prestazioni e sulla sicurezza, definendo aspetti critici al di là delle funzionalità specifiche. Essi sono essenziali per garantire un'esperienza utente affidabile e soddisfacente, oltre a rispettare gli standard e le normative del settore. Tra le più rilevanti, implementate e curate all'interno del progetto, si hanno:

Manutenibilità – Caratteristica chiave del software che si riferisce alla facilità con cui un sistema può essere mantenuto, aggiornato, esteso o riparato nel tempo. In altre parole, indica quanto sia agevole e conveniente apportare modifiche ad un software senza causare effetti negativi o degradazione delle prestazioni. Ecco alcuni aspetti importanti della manutenibilità del software:

- **Leggibilità:** Codice sorgente scritto in modo chiaro e comprensibile, così che altri sviluppatori possano facilmente comprendere la sua logica e le sue funzionalità. Questo facilita la manutenzione futura.
- **Commenti e documentazione:** Codice ben commentato e documentato, in modo che altri sviluppatori possano comprendere lo scopo e il funzionamento di diverse parti del software.
- **Testing:** L'implementazione di test unitari serve per verificare che il software funzioni correttamente, soddisfi i requisiti e mantenga determinati standard di qualità.
- **Versionamento del codice:** L'uso di sistemi di controllo di versione, come *Git*, permette di tenere traccia delle modifiche e di tornare a versioni precedenti in caso di problemi.

Sicurezza – Concetto fondamentale che riguarda la protezione da minacce, pericoli o rischi in modo da garantire l'integrità, la confidenzialità e la disponibilità delle risorse e delle informazioni. È di fondamentale importanza per proteggere dati sensibili e la privacy degli utenti. Ecco alcuni aspetti chiave della sicurezza:

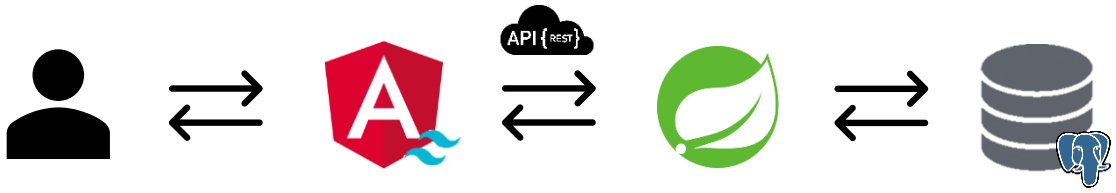
- **Integrità:** Crittografia dei dati, implementazione di controlli di accesso rigorosi e validazione dei dati in ingresso mirati a garantire che i dati o il sistema non siano alterati in modo non autorizzato.
- **Autenticazione e autorizzazione:** Per confermare l'identità di un utente e autorizzare l'accesso a risorse o funzionalità specifiche viene utilizzato il *JWT* (*JSON Web Token*). Si tratta di un token crittografato che contiene informazioni sull'utente autenticato e può essere verificato per garantire la sua legittimità. Questo approccio offre un modo efficiente e sicuro di gestire l'accesso utente, consentendo agli utenti di accedere a parti dell'applicazione solo se sono autenticati e autorizzati correttamente.

Usabilità – Caratteristica chiave di un prodotto o di un sistema, che si riferisce a quanto sia facile e intuitivo per gli utenti interagire con esso al fine di raggiungere i loro obiettivi in modo efficace ed efficiente. Ecco alcuni aspetti importanti dell'usabilità:

- **Efficienza:** Abilità degli utenti di eseguire le attività in modo rapido e senza spreco di tempo o risorse.
- **UX/UI:** Interfaccia del prodotto intuitiva e basata su convenzioni comuni, in modo che gli utenti si sentano a proprio agio nell'usarla.

1.3.3 Design strutturale

L'applicazione è stata strutturata tramite un'**architettura a tre livelli**: un approccio comune alla progettazione di applicazioni web che mira a suddividere l'applicazione in tre componenti distinti per garantire una struttura ben organizzata e scalabile. Ogni livello svolge un ruolo specifico e si occupa di determinati aspetti dell'applicazione.



Client – Presenta le informazioni all'utente in modo intuitivo e interattivo, ottenute tramite richieste al server.

Server – Fornisce risorse o servizi quando richiesto dal client. Responsabile, inoltre, della comunicazione con il database.

Database – Sistema di archiviazione e gestione strutturato che consente di organizzare, memorizzare e recuperare dati in modo efficiente.

2 Database

2.1 Diagramma ER

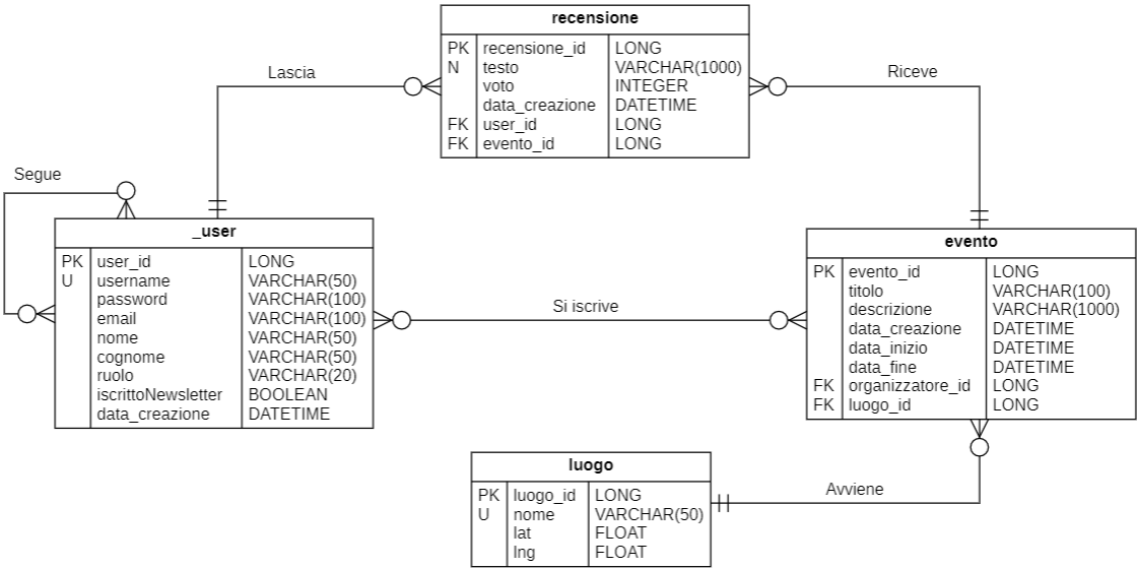


Diagramma ER – Contiene le entità (tabelle), gli attributi (campi) e le associazioni tra le tabelle stesse.

2.2 Note implementative

È stato scelto di affidarsi all'***RDBMS*** (*relational database management system*) *PostgreSQL*, spesso abbreviato come *Postgres*. Questa scelta può essere motivata da diverse ragioni: essendo **open source**, gode di una comunità attiva di sviluppatori che contribuiscono al suo miglioramento; offre robuste funzionalità di **sicurezza**, inclusi i controlli degli accessi e la crittografia dei dati.

Per la tabella dell'utente, è stato scelto il nome "***_user***" poiché "*user*" è una parola riservata in diversi DBMS, tra cui *Postgres*.

A differenza del back-end – dove il **ruolo** dell'utente è gestito come un enum per una maggiore struttura e gestione dei dati – nel database è memorizzato come una semplice stringa per garantire una maggiore leggibilità e flessibilità.

È presente un'**auto-referenziazione** nella tabella *_user* per realizzare un sistema di *following* tra turisti e organizzatori: un turista può seguire più organizzatori, un organizzatore può essere seguito da più turisti.

3 Back-end

3.1 Progettazione architetturale

3.1.1 Pattern MVCS

È stato deciso di implementare il pattern architetturale *MVCS* (*Model-View-Controller-Service*), ovvero una variante del più comune pattern *MVC*. È utilizzato per organizzare e separare le responsabilità all'interno di un'applicazione software complessa, consentendo una migliore **manutenibilità**, **scalabilità** e **riusabilità del codice**. Di seguito una breve descrizione di ciascun componente del pattern:

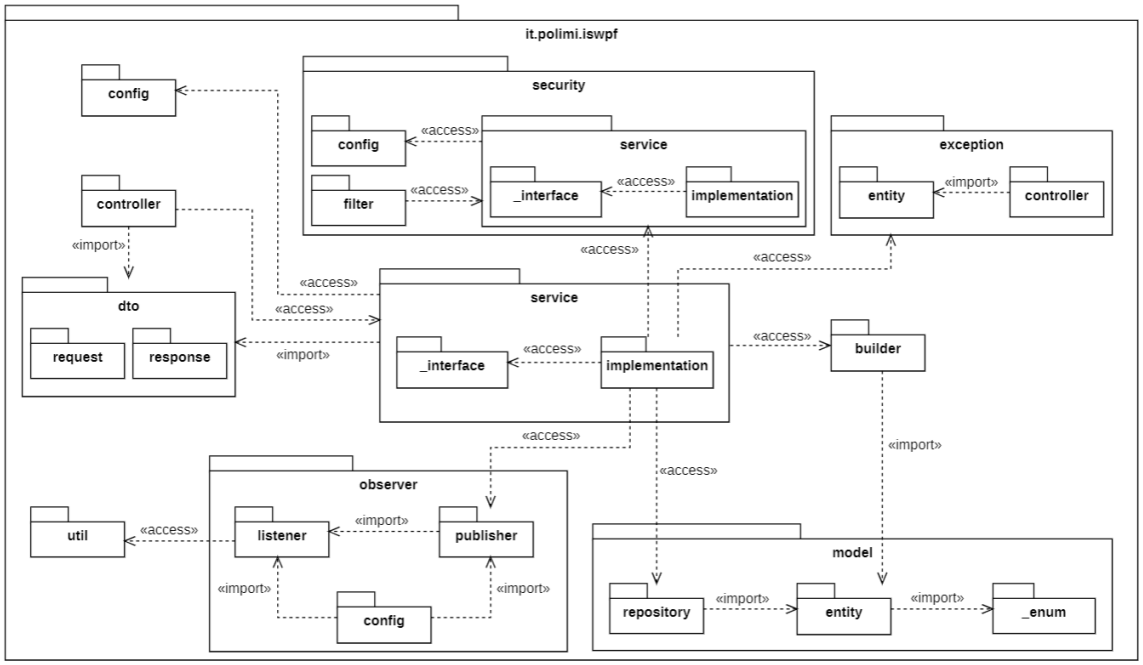
Model – Contiene le entità di dominio (opportunamente mappate sul database) e i metodi di accesso ai dati (rappresentati dai *repository*).

View – Responsabile dell'interfaccia utente dell'applicazione. È costituita principalmente dall'output generato dagli *endpoint*.

Controller – Classi che rappresentano l'interfaccia di accesso all'applicazione. Rimangono in ascolto e intercettano le richieste HTTP da parte del client e indirizzano la richiesta al service adatto.

Service – Insieme di classi contenente la *business logic*. Sollevano i controller da alcuni compiti, come flussi logici complessi o comunicazione con risorse esterne.

3.1.2 Packaging



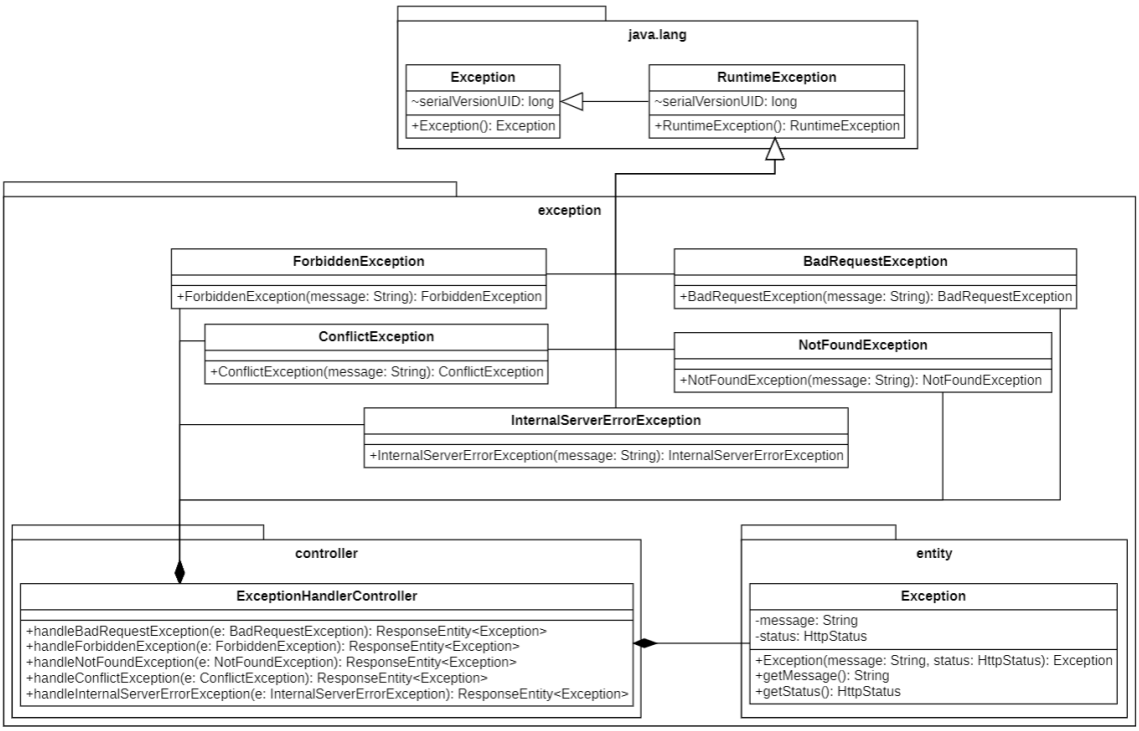
Package diagram – Sono riportati tutti i package del progetto. I collegamenti (rilevanti) rappresentano diversi tipi di dipendenze tra i package: un package importa le classi dell'altro oppure un package accede ai metodi delle classi dell'altro package.

3.2 **Exception**

Durante l'uso dell'applicazione, può avvenire che una richiesta HTTP vada incontro ad anomalie o errori. È una buona pratica *catturare* gli errori e notificare il client di quanto accaduto.

È stato, quindi, implementato un **sistema di gestione delle eccezioni** composto da un'entità con *messaggio* e *status* come attributi, un controller responsabile dell'invio della risposta al client e un insieme di classi figlie di *RuntimeException* fatte ad hoc in base al tipo di errore rilevato.

3.2.1 **Class diagram**



Class diagram del package exception. Si può notare la composizione del controller dall'entity e da tutte le classi figlie di RuntimeException.

3.3 **Dominio**

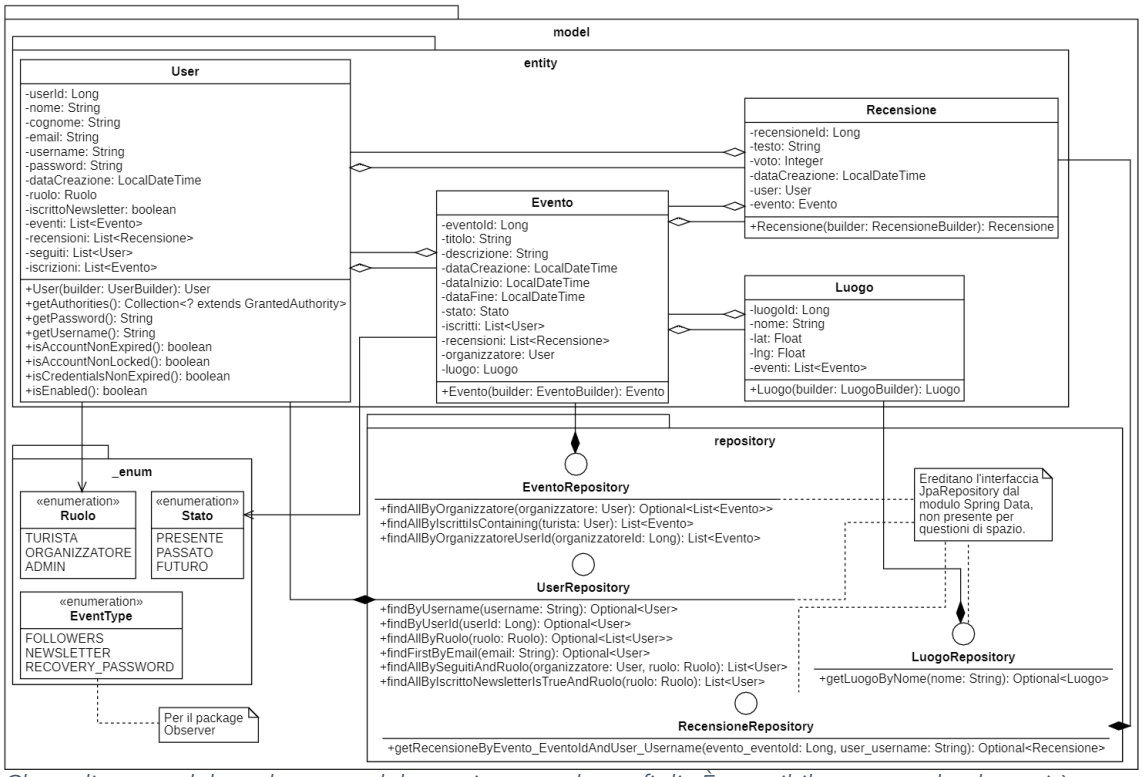
Rappresentato dal package ***model***. All'interno di questo package vi sono anche i *repository* perché, come da definizione di pattern MVC (quindi anche MVCS), **il model si occupa anche dell'interazione con il database**. Nello specifico si ha:

entity – Classi Java che rappresentano gli oggetti del dominio dell'applicazione. Grazie all'*ORM* ***Hibernate*** è possibile mapparle sul database sollevando gli sviluppatori dall'uso del linguaggio SQL.

repository – Insieme di interfacce che sfruttano (quindi ereditano) il modulo di ***Spring Data JPA*** per semplificare l'interazione con il database.

_enum – Tipi di dati che rappresentano un insieme predefinito di costanti.

3.3.1 **Class diagram**



*Class diagram del package model con i tre package figli. È possibile notare che le entità sono rappresentate come **composizioni** dei rispettivi repository (figli di JpaRepository) e possono essere associate ad un enum. Inoltre, le varie entità contengono alcune **aggregazioni** tra le stesse.*

3.4 **Design patterns**

L'uso dei design pattern aiuta a ridurre la complessità del software, migliorando la manutenibilità e l'efficienza. Tuttavia, è importante utilizzarli in modo appropriato, evitando l'implementazione quando non necessario. Di seguito la descrizione dei pattern applicati e i vantaggi di cui gode l'applicazione di conseguenza.

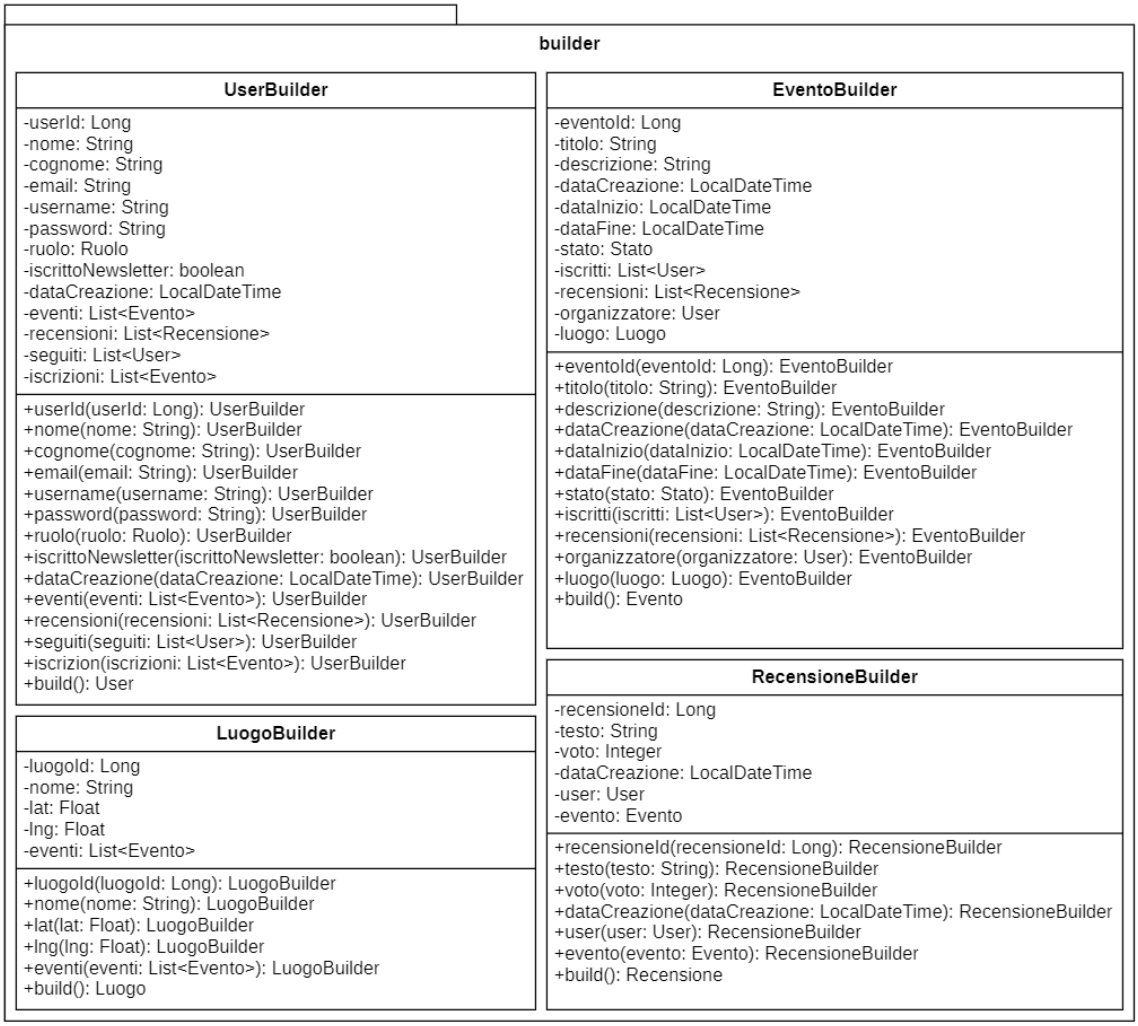
3.4.1 **Builder**

Pattern creazionale il cui scopo è creare oggetti, passo dopo passo. Nello specifico, invece di avere il singolo costruttore con molti parametri, il pattern Builder separa il processo di costruzione dell'oggetto in una serie di passi, consentendo di personalizzare l'oggetto in modo flessibile. I principali motivi per cui è stato scelto di implementarlo sono:

Testabilità – Poiché vengono separati chiaramente i passi di costruzione, è più semplice scrivere test unitari che coprano ciascun passo senza dover eseguire l'intero processo di costruzione.

Flessibilità – Il pattern Builder consente di costruire oggetti complessi con configurazioni flessibili. È possibile specificare solo le opzioni necessarie, evitando parametri opzionali confusi e valori nulli.

3.4.1.1 **Class diagram**



Class diagram del package builder. Sono presenti solo e tutti gli attributi privati delle entity.

```java
public Luogo(@NonNull LuogoBuilder builder) {
    this.luogoId = builder.getLuogoId();
    this.nome = builder.getNome();
    this.lat = builder.getLat();
    this.lng = builder.getLng();
    this.eventi = builder.getEventi();
}
```

Simil-costruttore dell'entity che riceve come parametro un'istanza del builder con i dati precedentemente settati. Ritorna un'istanza dell'entity "popolata".

```java
public LuogoBuilder nome(String nome) {
    this.nome = nome;
    return this;
}
```

Setter che ritorna l'istanza del builder stesso, così da poter concatenare altri metodi.

```java
public Luogo build() {
    return new Luogo(this);
}
```

Metodo build che si occupa del casting da builder a entity, chiamando quindi il simil-costruttore precedentemente citato.

3.4.2 **Singleton**

Pattern strutturale che mira a garantire che una classe abbia una sola istanza e fornisca un punto di accesso globale a tale istanza. Esistono alternative (soprattutto in *Spring*) per utilizzare il pattern, ma è stato scelto di implementarlo da zero **all'interno del pattern *Observer***. I principali vantaggi dell'utilizzo di questo pattern sono:

Ottimizzazione – La riduzione del numero di istanze di oggetti migliora le prestazioni dell'applicazione in termini di utilizzo della memoria e tempo di inizializzazione.

Consistenza – l'uso di una sola istanza di un oggetto è essenziale per garantire che i dati all'interno dell'applicazione siano sempre coerenti e sincronizzati.

```java
public class EventManager {
    //Unica istanza della classe, privata e statica.
    private static EventManager instance;

    //Costruttore privato per poterci accedere solamente tramite il metodo sottostante.
    private EventManager() { }

    //Metodo statico che ritorna l'unica istanza della classe.
    public static EventManager getInstance() {
        //Se l'istanza non esiste, viene creata tramite il costruttore privato.
        if(instance == null) {
            instance = new EventManager();
        }
        return instance;
    }
}
```

Implementazione del pattern Singleton su una classe dell'Observer. Sono presenti i commenti dove viene spiegato il funzionamento del codice. Data l'assenza di classi ausiliare per l'applicazione del pattern, risulta quindi superfluo un class diagram. Si può consultare, nel paragrafo successivo, il diagramma dell'Observer dove al suo interno vi è anche il Singleton.

3.4.3 **Observer**

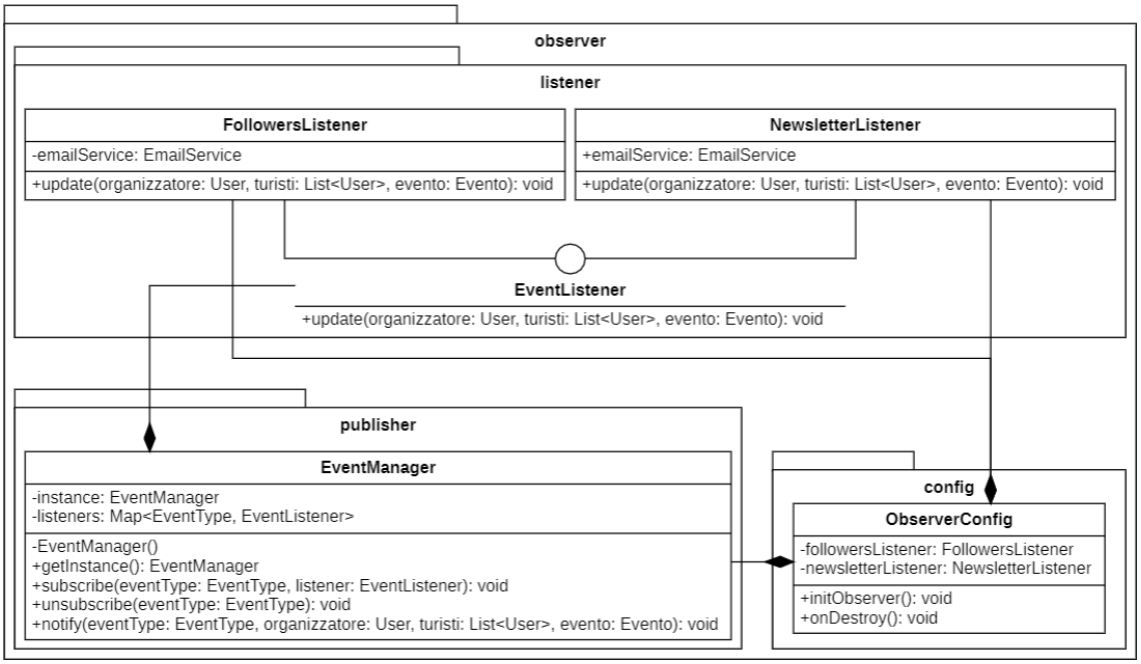
Pattern comportamentale ampiamente utilizzato nella programmazione ad oggetti. Risulta utile quando si desidera stabilire una relazione uno-a-molti tra oggetti in modo che, quando avviene un cambio di stato, tutti gli oggetti dipendenti da esso vengano notificati e aggiornati automaticamente. Di seguito i benefici riscontrati:

Scalabilità – È possibile aggiungere o rimuovere osservatori senza dover modificare il soggetto. Questo consente una facile estensione del sistema senza influire su altre parti del codice.

Immediatezza – Gli osservatori ricevono notifiche istantanee dei cambiamenti di stato, consentendo agli utenti di vedere i dati in tempo reale.

Flessibilità – È possibile creare numerose implementazioni con comportamenti diversi sfruttando un'interfaccia comune con il metodo *update*. Questo consente di personalizzare la logica senza dover modificare il soggetto o gli altri osservatori.

3.4.3.1 **Class diagram**



Class diagram del package observer con i tre package figli. Il publisher, come da definizione, è responsabile della gestione delle iscrizioni e della notifica dei listeners. Ogni listener contiene una propria concretizzazione del metodo update con la quale notificherà gli iscritti.

```
EventManager.getInstance().notify(EventType.FOLLOWERS, organizzatoreExists.get(), followers, evento);
EventManager.getInstance().notify(EventType.NEWSLETTER, organizzatoreExists.get(), iscrittiNewsletter, evento);
```

Chiamata del metodo notify nel service specificando il tipo di evento e l'elenco di utenti da notificare.

```
public void notify(EventType eventType, User organizzatore, List<User> turisti, Evento evento) {
    listeners.get(eventType).update(organizzatore, turisti, evento);
}
```

Notifica degli listeners. Vengono prima prelevati tutti i listeners del tipo richiesto (tramite il metodo get) per poi eseguire il metodo update, concretizzato nelle implementazioni dell'interfaccia. (N. B. in questo snippet sono stati omessi i controlli così da visualizzare solo le parti rilevanti del pattern).

```
@Component
@RequiredArgsConstructor //Per iniettare le dipendenze (bean).
public class FollowersListener implements EventListener {
    private final EmailService emailService;
    //Override del metodo update dichiarato nell'interfaccia, contenente una propria business logic.
    @Override
    public void update(User organizzatore, List<User> turisti, Evento evento) {
        emailService.inviaEmail(new InviaEmailRequest(
            turista.getEmail(), //Indirizzo email del destinatario.
            "Novità in EventGURU", //Oggetto della email.
            dynamicData, //Corpo della email.
            EventType.FOLLOWERS //Tipo di listener.
        ));
    }
}
```

Implementazione custom del metodo update. Avendo omesso tutti gli opportuni controlli di sicurezza, in questo snippet è presente solo la parte rilevante del metodo: l'invio dell'email al turista con i dati dell'evento appena creato da parte di un organizzatore che segue.

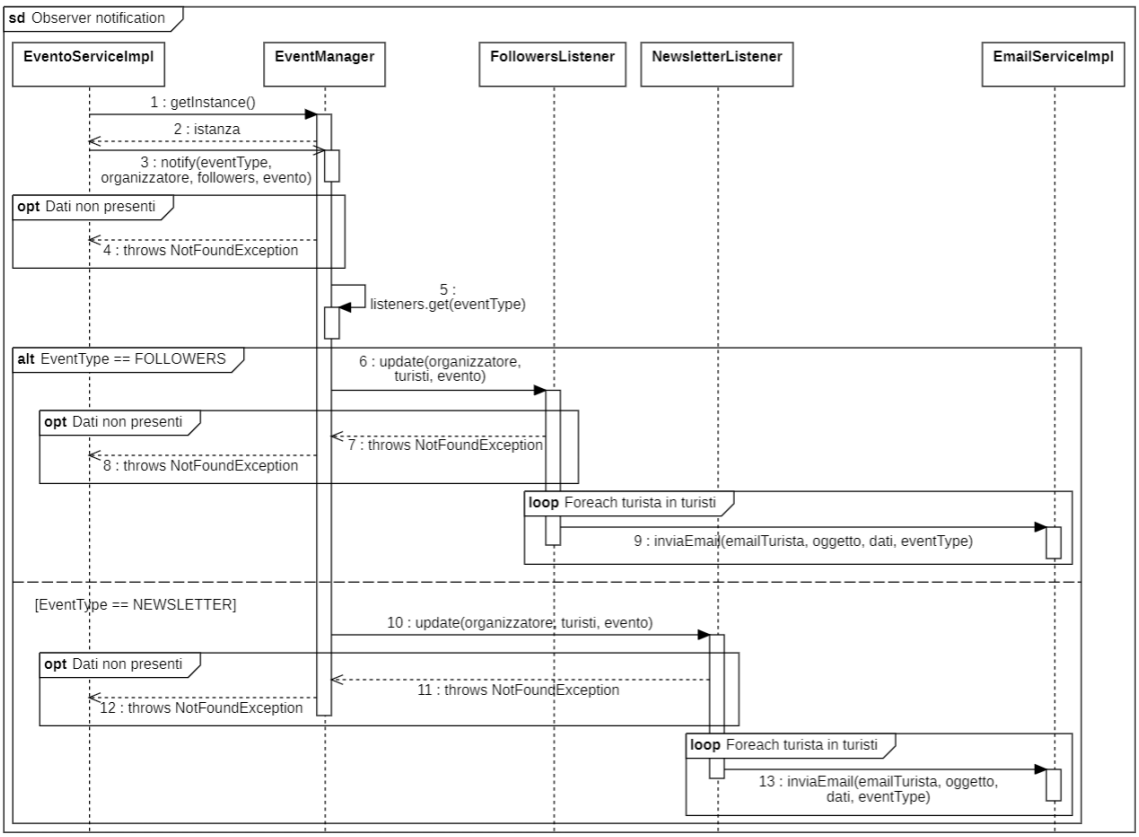
```java
public void subscribe(EventType eventType, EventListener listener) {
    listeners.put(eventType, listener);
}
```

Iscrizione di un listener ad un determinato tipo di evento tramite l'aggiunta all'hashmap.

```java
public void unsubscribe(EventType eventType) {
    listeners.remove(eventType);
}
```

Disiscrizione di un listener da un determinato tipo di evento tramite la rimozione dall'hashmap.

3.4.3.2 **Sequence diagram**



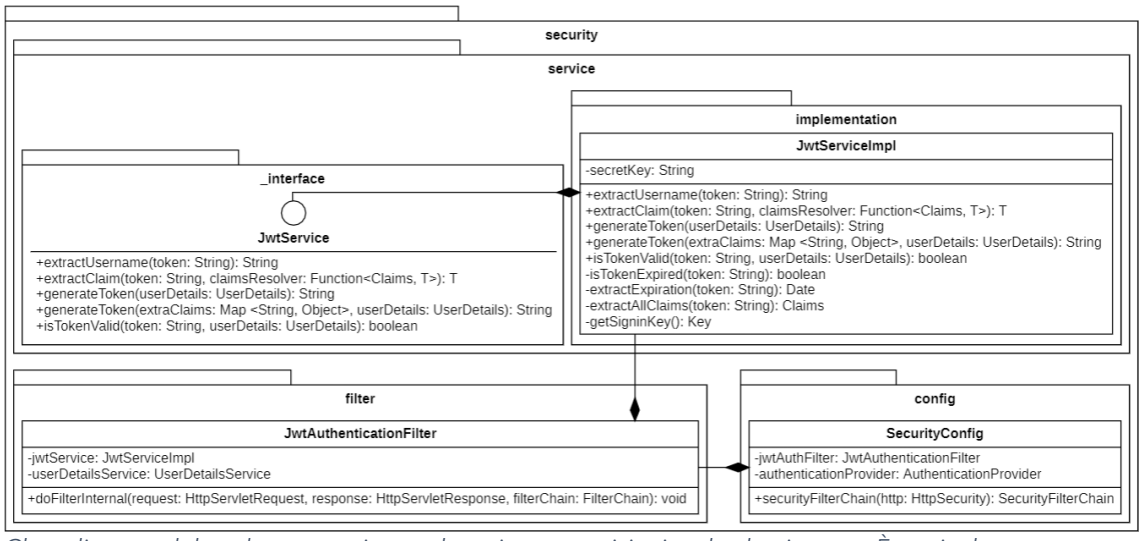
*Sequence diagram del sistema di notifica email quando viene creato un evento. Sono presenti tutti i diversi casi in cui può essere eseguito il metodo, definiti dai **fragments** opt e alt.*

3.5 **Sicurezza e autorizzazione**

Al fine di assicurare la sicurezza delle applicazioni web, si è potenziato il processo di autenticazione mediante l'impiego del modulo **Spring Security**. Durante la fase di accesso, gli utenti vengono sottoposti a un processo di autenticazione e autorizzazione per consentire loro di accedere alle risorse protette.

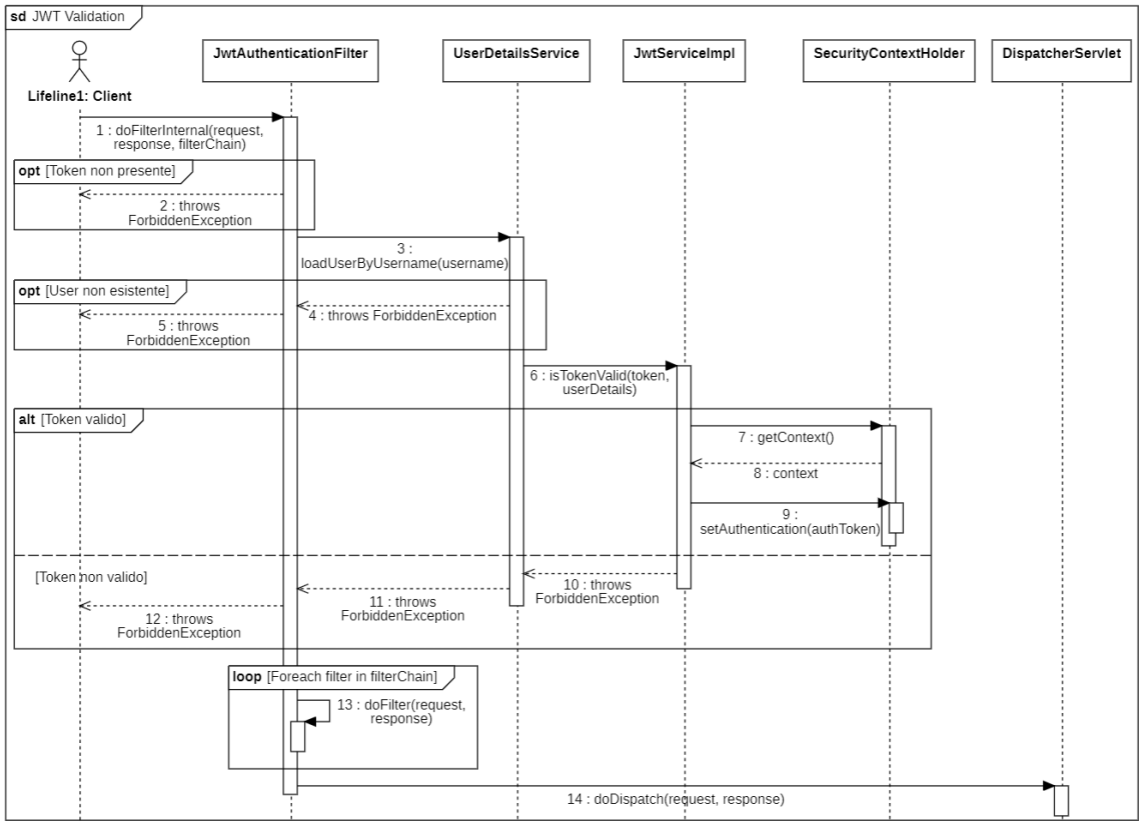
Nel dettaglio, quando un utente cerca di accedere all'applicazione inviando al server una coppia di credenziali *{username, password}*, il sistema, una volta decifrata la password e confermata l'identità dell'utente, genera un token contenente le informazioni dell'utente stesso. Questo particolare token, noto come *JSON Web Token* (***JWT***), viene trasmesso al client e svolge un ruolo fondamentale nell'autenticazione durante le successive richieste al server.

3.5.1 **Class diagram**



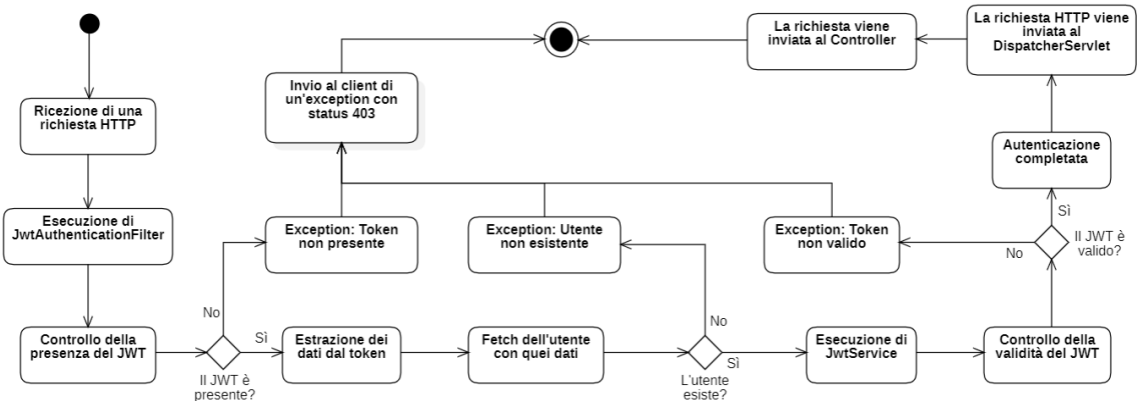
*Class diagram del package security con le varie composizioni tra le classi stesse. È particolarmente evidente il concetto di **polimorfismo** applicato ai service, tramite la presenza di un'**interfaccia** e successivamente la propria implementazione.*

3.5.2 **Sequence diagram**



*Sequence diagram del processo attuato da Spring Security all'arrivo di ogni richiesta HTTP. Il diagramma è noto per basarsi sull'interazione tra oggetti di una determinata sequenza di eventi. Si noti la presenza di un **loop** tramite il quale viene delegato il controllo agli altri filtri. Si tratta del **Chain of Responsibility pattern**.*

3.5.3 **Activity diagram**



Activity diagram che rappresenta l'esecuzione di Spring Security all'arrivo di ogni richiesta HTTP. Questo tipo di diagramma si focalizza maggiormente su attività, decisioni e flussi di controllo.

3.6 **Mailing**

Si è optato per l'implementazione di un sistema di messaggistica al fine di comunicare con gli utenti in occasione di situazioni specifiche. Tra gli esempi rientrano la creazione di notifiche via email quando un organizzatore crea un evento seguito, il reset della password o l'erogazione di servizi di newsletter.

```java
@Configuration //Classe eseguita all'avvio del server
public class EmailConfig {
    @Bean
    public JavaMailSender getJavaMailSender() {
        JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
        mailSender.setHost("smtp.gmail.com");
        mailSender.setPort(587);
        mailSender.setUsername("eventguru.service@gmail.com");
        mailSender.setPassword(""); //Passowrd omessa per ovvi motivi.
        return mailSender;
    }
}
```

Metodo che fornisce un bean configurato per l'invio di email attraverso il server SMTP di Gmail.

3.6.1 **FreeMarker**

Al fine di creare contenuti dinamici all'interno del corpo delle email, si è fatto ricorso a ***FreeMarker***, un motore di templating ampiamente adottato in contesti simili. Sono stati sviluppati vari file *.ftl*, simili all'*HTML*, contenenti elementi statici specifici per vari contesti. Successivamente, all'interno del servizio responsabile dell'invio delle email, è stata implementata una logica che consente di collegare dinamicamente i dati presenti in una mappa con i contenuti dinamici in uno dei file menzionati in precedenza.

3.6.2 **Class diagram**

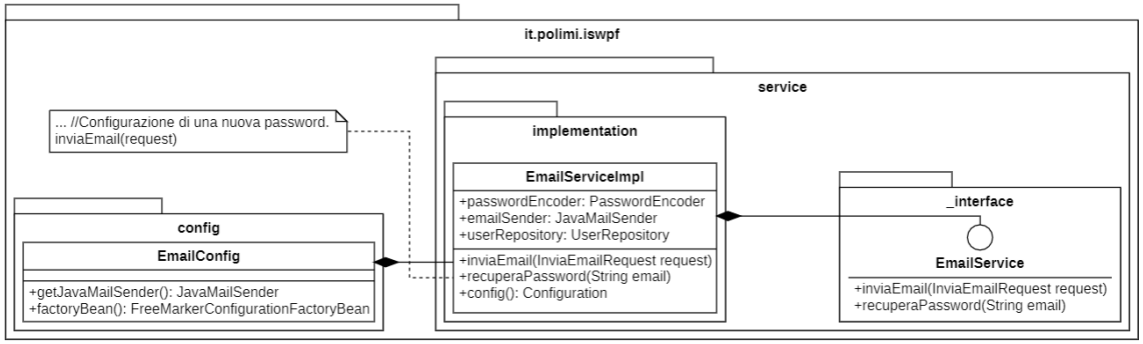
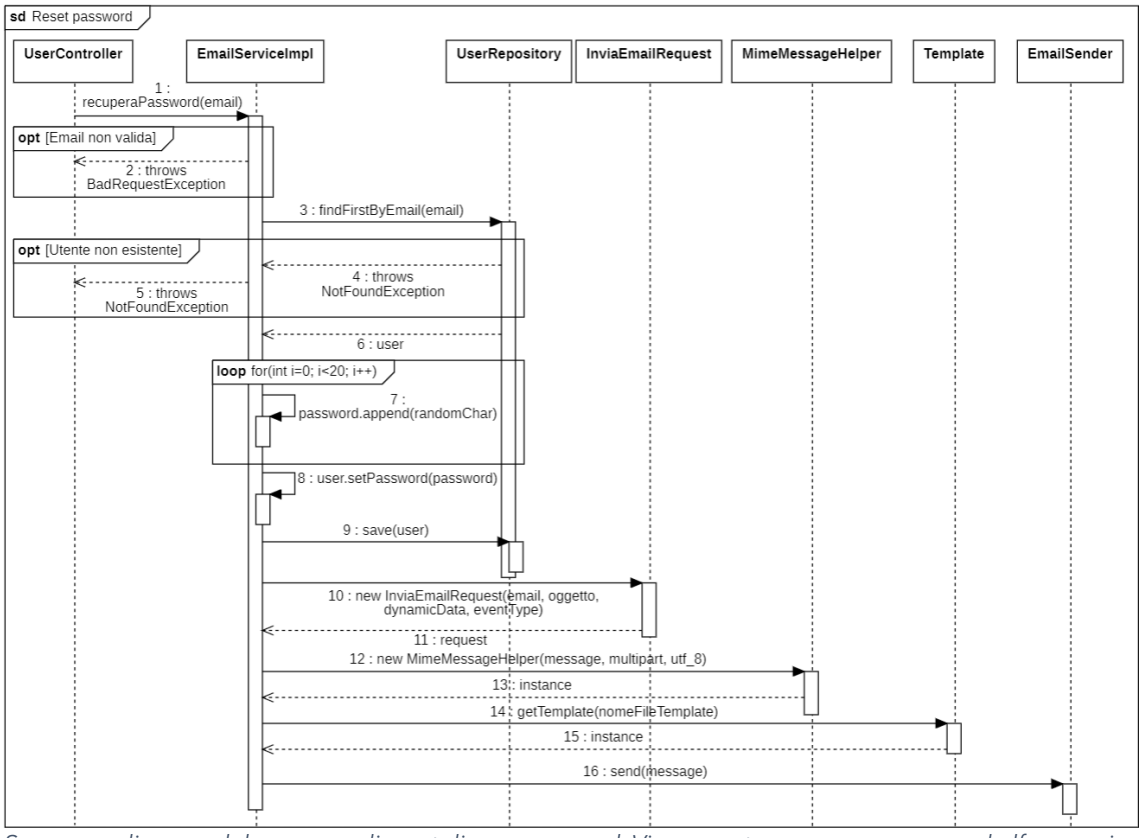


Diagramma delle classi rilevanti per la configurazione e la gestione del servizio di mailing.

3.6.3 **Sequence diagram**



Sequence diagram del processo di reset di una password. Viene creata una nuova password alfanumerica da 20 caratteri, successivamente l'utente viene aggiornato sul database e infine arriverà all'utente interessato una email con la nuova password.

3.7 **Sistema di poligoni e coordinate**

Nel *front-end*, che offre **mappe interattive**, è stata inclusa la possibilità di associare un **marker** (composto da una coppia di coordinate latitudine e longitudine insieme a un nome) a ciascun evento durante la sua creazione. Di conseguenza, sono state implementate funzionalità che consentono agli utenti di effettuare ricerche e visualizzare i markers situati all'interno di un'area geografica definita da un poligono o da una circonferenza da loro stessi disegnati.

3.7.1 **Coordinate dentro un poligono**

Mediante la ricezione di una lista di coordinate nel corpo della richiesta, che corrispondono ai vertici del poligono disegnato, è possibile restituire esclusivamente i punti situati all'interno di tale poligono.

In dettaglio, viene effettuata una verifica sulla validità delle coordinate dei vertici del poligono. Successivamente, si recuperano dal database le coordinate di ciascun evento, e si utilizza l'**algoritmo Ray Casting** per verificare, per ciascun marcatore, se si trova all'interno del poligono.

3.7.1.1 **Algoritmo Ray Casting**

```
1 private boolean isMarkerInsidePolygon(float lat, float lng, List<PuntoPoligono> request) {
2     boolean isInside = false;
3     for(int i = 0, j = request.size() - 1; i < request.size(); j = i++) {
4         float latI = request.get(i).getLat();
5         float lngI = request.get(i).getLng();
6         float latJ = request.get(j).getLat();
7         float lngJ = request.get(j).getLng();
8 
9         boolean intersect =
10            lngI > lng != lngJ > lng &&
11            lat < ((latJ - latI) * (lng - lngI)) / (lngJ - lngI) + latI;
12 
13        if(intersect) {
14            isInside = !isInside;
15        }
16    }
17    return isInside;
18 }
```

Nell'algoritmo seguente, per ciascun segmento del poligono, viene calcolato un valore booleano *intersect* che verifica se il raggio proveniente dal punto in esame e diretto da est a ovest interseca il segmento stesso. Se *intersect* è true, il valore di *isInside* viene invertito. Questa fase costituisce il nucleo dell'algoritmo Ray Casting: se il raggio attraversa un numero dispari di segmenti del poligono, il punto viene considerato all'interno del poligono. In caso contrario, viene considerato all'esterno.

3.7.2 **Coordinate dentro una circonferenza**

In modo simile a quanto descritto in precedenza, quando si ricevono i dati relativi a una circonferenza, ovvero le coordinate del centro e il raggio, è possibile restituire solo i punti situati all'interno della circonferenza disegnata. Dopo aver verificato la validità dei dati, viene utilizzato un metodo per calcolare la distanza in metri tra due coordinate terrestri.

3.7.2.1 **Formula di Haversine**

```java
private float distanceInMetersBetweenEarthCoordinates(float markerLat, float markerLng, float centroLat, float centroLng) {
    final float RAGGIO_TERRA_METRI = 6371000;

    float differenzaLat = degreesToRadians(centroLat - markerLat);
    float differenzaLng = degreesToRadians(centroLng - markerLng);

    markerLat = degreesToRadians(markerLat);
    centroLat = degreesToRadians(centroLat);

    float harv = (float)
            (Math.sin(differenzaLat / 2) * Math.sin(differenzaLat / 2) +
            Math.sin(differenzaLng / 2) * Math.sin(differenzaLng / 2) *
            Math.cos(markerLat) * Math.cos(centroLat));

    float c = (float) (2 * Math.atan2(Math.sqrt(harv), Math.sqrt(1 - harv)));

    return RAGGIO_TERRA_METRI * c;
}
```

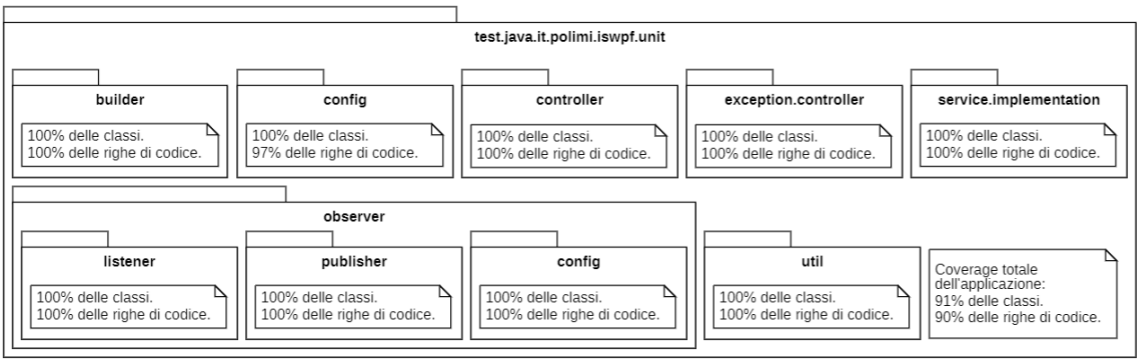
Nelle prime fasi dell'algoritmo, viene invocato un metodo di conversione da gradi a radianti. Successivamente, viene calcolato il valore *harv*, che rappresenta la *haversine* dell'angolo centrale tra due punti sulla superficie terrestre, utilizzando la *formula di Haversine* (o dell'*emisenoverso*). Questo valore coinvolge le funzioni trigonometriche seno e coseno degli angoli. Infine, con tutti gli strumenti necessari per calcolare la distanza tra due punti sulla superficie terrestre, si verifica semplicemente se questa distanza è inferiore al raggio specificato. In tal caso, il punto è considerato all'interno della circonferenza.

3.8 **Testing**

Il **testing unitario** rappresenta una pratica essenziale nel contesto dello sviluppo software, finalizzata a garantire che singole unità di codice, come classi o metodi, operino in modo accurato e isolato.

Nell'attuazione di questa pratica, vengono spesso impiegati framework come ***JUnit*** e ***Mockito*** per semplificare la creazione e l'esecuzione dei test. *JUnit* mette a disposizione utili annotazioni, ad esempio *@Test*, per identificare i metodi di test, mentre *Mockito* si occupa della gestione dei *mock*, cioè delle simulazioni dei comportamenti di oggetti e dipendenze reali esistenti.

3.8.1 **Package diagram con coverage**



Package diagram della sezione di testing dell'applicazione, con le apposite percentuali di coverage per ogni package figlio.

A seguito di quanto mostrato, è possibile notare che, grazie allo sviluppo di **219 metodi di test**, è stata raggiunta una **copertura del 90%** delle righe di codice (1309/1441) dell'intera applicazione.

3.8.2 **Reflection**

Di seguito viene riportata una porzione di un metodo di test che necessita dell'utilizzo della classe ***ReflectionTestUtils*** di *Spring*.

```
1 @Test
2 void creaEventoSuccessful() {
3     ReflectionTestUtils.setField(EventManager.getInstance(), "instance", eventManager);
4     //...
5 }
```

Tramite una reflection si sta accedendo all'oggetto singleton-observer EventManager.

La problematica deriva dal fatto che alcune variabili di istanza degli oggetti sono dichiarate come private per garantire l'**incapsulamento**. Tuttavia, in un contesto di testing, potrebbe essere necessario accedere a tali campi privati al fine di effettuare asserzioni o modificarli. *ReflectionTestUtils* fornisce la possibilità di farlo.

4 Front-end

4.1 Tecnologie utilizzate

4.1.1 Angular

Angular è un **framework open-source** sviluppato principalmente da *Google*. Implementato con *TypeScript*, adotta un approccio modulare, opzionalmente orientato agli oggetti e si basa sull'utilizzo di **componenti** per la creazione di applicazioni web sofisticate e dinamiche. Tra le sue caratteristiche distintive spiccano la gestione del **routing**, l'iniezione delle **dipendenze**, la manipolazione avanzata del *DOM*, la gestione agile degli eventi e un sistema di **binding dei dati bidirezionale**.

4.1.1.1 Components

Rappresentano blocchi fondamentali che integrano la struttura *HTML*, la logica di programmazione e i metadati per formare elementi **modulari** e **riutilizzabili** nelle interfacce utente delle applicazioni web. Questa concezione architetturale mira a potenziare la manutenibilità e la scalabilità del codice, promuovendo un approccio che agevola la gestione e l'estensione del sistema nel tempo.

```
1 #Struttura del comando: ng generate component [folder]/[nome-component]
2 ng generate component components/login
```

Creazione di un componente tramite CLI. Se non esiste già, verrà generata automaticamente una cartella di nome "components" con tre files all'interno: .html, .css, .ts.

4.1.1.2 Services

Classi che offrono funzionalità specifiche che possono essere condivise tra vari componenti all'interno di un'applicazione. In questo contesto, sono implementati diversi services per gestire la comunicazione con il *back-end*. Grazie alla loro **iniezione** nei componenti, i servizi favoriscono la **modularità** e la **riutilizzabilità** del codice, consentendo una gestione più efficiente e snodata delle diverse parti dell'applicazione.

```
1 #Struttura del comando: ng generate service [folder]/[nome-service]
2 ng generate service services/authentication
```

Creazione di un service tramite CLI. Se non esiste già, verrà generata automaticamente una cartella di nome "services" con all'interno il file .ts.

4.1.1.3 Routes

Strumenti essenziali per dirigere la **navigazione** all'interno di una ***Single Page Application* (*SPA*)**. La loro funzione principale è definire percorsi specifici all'interno dell'applicazione e associare ognuno di essi a un componente particolare. Questi percorsi vengono poi riflessi nella barra degli indirizzi del browser quando l'utente naviga attraverso l'applicazione.

L'interazione tra percorsi e componenti semplifica la gestione della navigazione all'interno della SPA, poiché consiste di identificare in modo chiaro quale componente deve essere caricato per ciascun percorso, consentendo una navigazione fluida e intuitiva all'interno dell'applicazione.

```
const routes: Routes = [
  { path: "homepage", component: HomepageComponent },
  { path: "login", component: LoginComponent },
  { path: "register", component: RegisterComponent },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Vengono definite tre rotte: "homepage", "login" e "register", ognuna associata ad un componente specifico.

4.1.2 **TailwindCSS**

TailwindCSS è un framework *CSS* progettato per semplificare lo sviluppo e la manutenzione del codice *CSS*. In contrasto con molti altri framework che forniscono componenti stilizzati predefiniti, *Tailwind* si distingue per la fornitura di classi di utilità *CSS* **basate su singole proprietà**. Di seguito, alcune delle sue caratteristiche chiave:

Leggibilità – Le classi di utilità sono spesso nominate in modo intuitivo, aumentando quindi la facilità di lettura.

```
<!-- Imposta il colore del testo su una tonalità di blu specifica. -->
<p class="text-blue-500">Paragrafo di prova</p>
```

Esempio di utilizzo di una classe Tailwind in un elemento HTML.

Basso livello di astrazione – A differenza di alcuni framework che astraggono completamente lo stile attraverso componenti, *Tailwind* si trova a un livello molto più basso, consentendo un controllo più dettagliato e diretto degli stili *CSS* applicati.

Elasticità – È possibile personalizzare il set di classi nel file di configurazione per adattarlo a esigenze specifiche.

```
.my-custom-card {
    @apply bg-blue-500 text-white p-4 rounded-md shadow-md;
}
```

*Esempio di creazione di una classe custom dove, tramite la keyword **@apply**, vengono applicate classi tailwind già esistenti. Quindi, con la classe **.my-custom-card** verranno applicate tutte le classi sottostanti.*

4.1.3 **Leaflet**

Libreria *TypeScript open-source* che fornisce un set di strumenti per la creazione di mappe interattive all'interno di applicazioni web. È stata progettata per

essere leggera, flessibile e facile da utilizzare. Alcune delle sue caratteristiche principali includono:

Leggerezza – *Leaflet* è noto per essere leggero, con un file principale *TypeScript* di dimensioni ridotte. Ciò contribuisce a tempi di caricamento più veloci delle mappe integrate.

Estendibilità – La libreria è estendibile attraverso l'uso di *plugin*, che permettono di aggiungere funzionalità specifiche alle mappe.

```html
<div class="map-container"> <!-- Contenitore per la mappa. -->
    <div class="map-frame"> <!-- Frame con gli stili della mappa. -->
        <div id="map"></div> <!-- Grazie all'id viene definita la mappa nel file .ts. -->
    </div>
</div>
```

Struttura HTML per la visualizzazione della mappa.

```typescript
initMap(map: L.Map): L.Map {
    //Visualizzazione iniziale della mappa.
    map = L.map('map', {
        center: [41.9027835, 12.4963655], //Coordinate di Roma.
        zoom: 10,
    });

    //Leaflet si affida a OpenStreetMap, una mappa open-source.
    const tiles: L.TileLayer = L.tileLayer(
        'https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', //URL del servizio di tile di OpenStreetMap.
        {
            maxZoom: 18, //Livello di zoom massimo consentito.
            minZoom: 3, //Livello di zoom minimo consentito.
            //Attribuzione dei dati della mappa.
            attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>',
        }
    );
    tiles.addTo(map);

    const drawFeatures: L.FeatureGroup = new L.FeatureGroup();
    map.addLayer(drawFeatures); //Aggiungo le features per disegnare i poligoni.

    return map;
}
```

Metodo per la configurazione di base di una mappa generica.

4.1.3.1 **Leaflet-draw**

Rifacendoci al precedente concetto di **estendibilità**, è stato integrato un plugin che fornisce un'interfaccia utente per il disegno e l'editing di figure geometriche sulla mappa, con lo scopo di **filtrare** e visualizzare solo gli eventi all'interno della figura disegnata.

```typescript
const drawControl: L.Control.Draw = new L.Control.Draw({
    draw: { //Viene abilitato solo il disegno di poligoni e circonferenze.
        polygon: true,
        circle: true,
        rectangle: false,
        circlemarker: false,
        polyline: false,
        marker: false
    },
});
map.addControl(drawControl);

//Alla creazione di una figura, viene catturato l'evento.
map.on('draw:created', (e: DrawCreatedEvent) => {
    drawFeatures.addLayer(e.layer); //La figura viene visualizzata sulla mappa.
}
```

*Configurazione base del plugin leaflet-draw, da inserire nel metodo **initMap** visto precedentemente.*

4.1.4 Toastr

Libreria utilizzata per visualizzare **notifiche toast** sulla pagina web. Le notifiche toast sono piccoli messaggi che appaiono temporaneamente nella parte superiore dello schermo per informare gli utenti di eventi o azioni importanti **senza interrompere l'esperienza utente**.

```
npm install ngx-toastr
```

Comando per l'installazione della libreria.

```typescript
@NgModule({
    imports: [
        BrowserAnimationsModule,
        ToastrModule.forRoot()
    ],
    //Altre configurazioni.
})
export class AppModule { }
```

Configurazione del modulo all'interno dell'applicazione.

```typescript
@Component({
    //Altre configurazioni del componente.
})
export class ProvaToastrComponent {
    constructor(private toastr: ToastrService) {} //Iniezione della dipendenza richiesta.

    provaToastr() {
        this.toastr.success("Messaggio di successo", "Titolo");
        this.toastr.error("Messaggio di errore", "Titolo");
        this.toastr.warning("Messaggio di allerta", "Titolo");
        this.toastr.info("Messaggio di informazione", "Titolo");
    }
}
```

Utilizzo metodi offerti dalla libreria.

*Output visivo del codice scritto in precedenza alla chiamata del metodo **provaToastr**.*

4.2 **Progettazione architetturale**

4.2.1 **Pattern MVVM**

In *Angular*, uno dei pattern architetturali più efficaci è il ***Model View View-Model* (*MVVM*)**. Usato per separare la logica di presentazione dall'implementazione sottostante, offre notevoli vantaggi: **legame bidirezionale dei dati**, **aggiornamento automatico della UI**, **riutilizzabilità del codice**. I componenti del pattern sono:

Model – Rappresenta i dati dell'applicazione, recuperati da ***API RESTful***. In questo contesto, è stato delegato ai *service* il compito di interagire con il server.

View – Composta dai **template *HTML***. Definisce come i dati del model devono essere visualizzati e interagiti dall'utente. La separazione della Vista consente di mantenere la logica di presentazione distinta dalla logica di business, rispettando il principio di separazione delle responsabilità.

View-Model – I **componenti** Angular fungono da View-Model, in quanto contengono la logica che connette il modello alla vista. Attraverso il ***two-way data binding*** e le **direttive** di Angular, i componenti mantengono sincronizzati il modello e la vista.

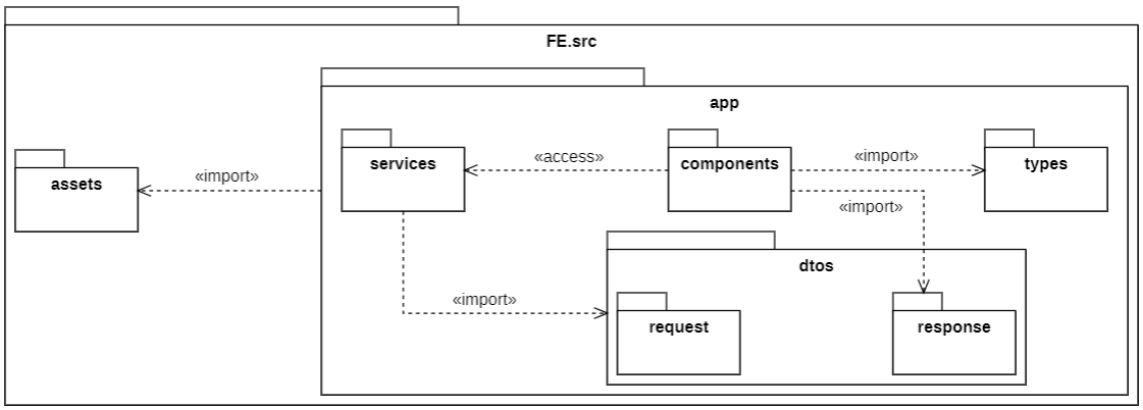
```
1 <input [(ngModel)]="titolo" name="titolo" type="text" class="input-form" required />
```

*Esempio di two-way binding. Tramite la keyword **ngModel** di Angular, viene **sincronizzata** la variabile con l'apposito componente.*

```
1 protected titolo: string = '';
```

*Viene dichiarata e inizializzata una variabile nel componente con lo stesso nome presente nell'ngModel. Si noti che ogni variabile che rispetti il pattern è stata definita **protected** dato che template html e business logic appartengono alla stessa cartella.*

4.2.2 **Foldering**



Package diagram delle cartelle del progetto Angular. Si noti che nel front-end, convenzionalmente, si parla di "folders" e non di "packages", di conseguenza i nomi delle cartelle sono al plurale.

4.3 Design patterns

4.3.1 DTO

Il pattern *DTO* (*Data Transfer Object*) si riferisce all'uso di oggetti specializzati per trasferire dati tra i componenti di un'applicazione. Questi oggetti contengono solo dati e non includono logica di business o metodi. L'obiettivo principale del pattern DTO è semplificare la comunicazione e lo scambio di dati all'interno dell'applicazione, come ad esempio tra il *front-end* e il *back-end*. All'interno della cartella dedicata sono presenti due sottocartelle per differenziare in base al tipo: *request* o *response*.

```
1 export interface LoginRequest {
2     username: string;
3     password: string;
4 }
```

Dichiarazione, tramite un'interfaccia, del DTO di richiesta con i dati del login.

```
1 login(request: LoginRequest): Observable<LoginResponse> {
2     return this.http.post<LoginResponse>('http://localhost:8080/api/v1/auth/login', request);
3 }
```

Esempio di applicazione di DTO request e response. Nello specifico, viene chiamato il back-end per eseguire il login dato un DTO con le credenziali. Verrà restituito l'analogo DTO contenente il JWT.

4.3.2 Dependency Injection (DI) & Singleton

La *DI* è un pattern fondamentale in *Angular* che consiste nel fornire alle classi le dipendenze di cui hanno bisogno durante l'istanziazione. Di seguito una serie di passaggi chiave per applicare efficacemente il pattern:

Provider – Le dipendenze vengono definite come *provider*. Un *provider* è un oggetto che dice al sistema di *DI* come ottenere o creare una dipendenza.

Iniezione – Le dipendenze vengono iniettate nei costruttori delle classi di cui ne hanno bisogno. Quando *Angular* crea un'istanza di una classe, controlla il costruttore per vedere quali dipendenze sono richieste e le fornisce automaticamente.

Configurazione – I provider possono essere configurati a livello di componenti grazie all'annotazione ***@Injectable***. L'attributo ***providedIn: "root"***, che inietta il servizio a livello globale, indica implicitamente l'utilizzo in **singleton**, ovvero esiste una sola istanza di questi provider che viene condivisa nell'applicazione.

```
1 @Injectable({
2     providedIn: "root"
3 })
4 export class AuthService {
5     constructor(private http: HttpClient) { }
6     //...
7 }
```

Esempio di DI. Il modulo per le richieste http viene iniettato in un service per comunicare col back-end.

4.3.3 **Observer**

Modello di programmazione asincrona, noto per gestire sequenze di eventi o dati in modo reattivo, ampiamente impiegato per gestire richieste *HTTP*. Forniscono una gestione avanzata rispetto alla classe *Promise* di *TypeScript*. Il pattern si basa su due semplici passaggi, convenzionalmente implementati nei *services* e nei *components*. Il primo si occupa di interfacciarsi con classi esterne per richiedere dati o servizi, il secondo comunica con il *service* stesso per **iscriversi** all'evento offerto.

```
getAllEventi(): Observable<GetAllEventiResponse[]> {
    return this.http.get<GetAllEventiResponse[]>('http://localhost:8080/api/v1/evento/getAll');
}
```

Creazione di una richiesta HTTP. In questo esempio vengono richiesti tutti gli eventi presenti sul database.

```
this.eventService.getAllEventi().subscribe({
    next: (res: GetAllEventiResponse[]) => {
        //Gestione della risposta.
    },
    error: (err: HttpErrorResponse) => {
        //Gestione dell'errore.
    }
});
```

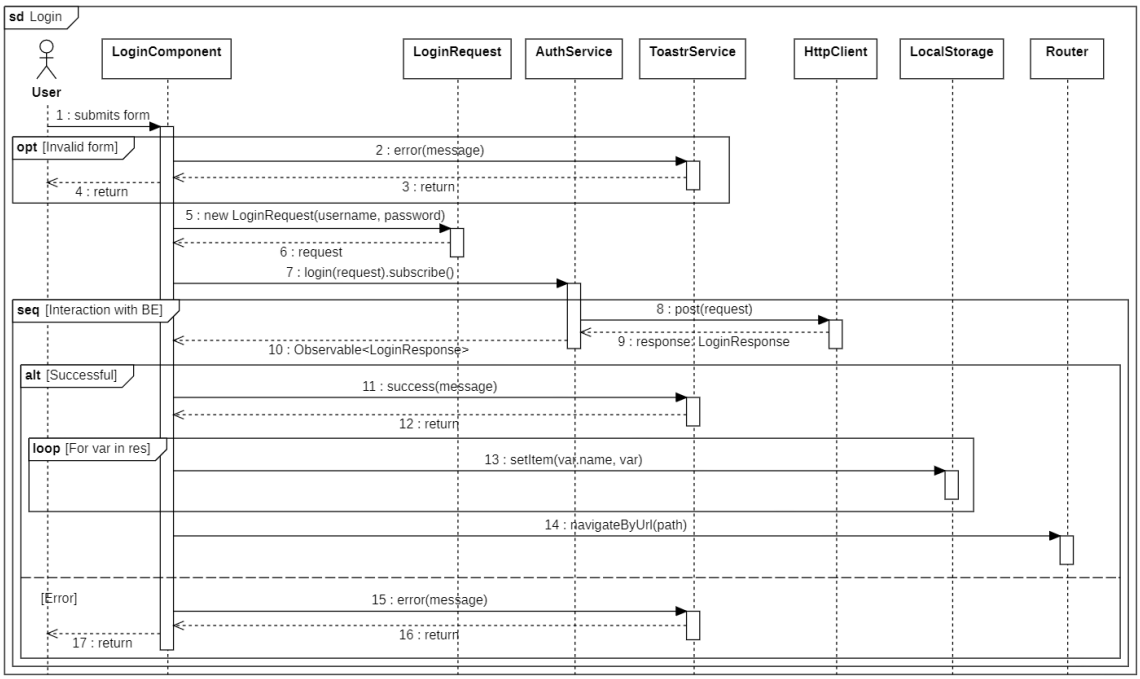
*Tipica gestione delle richieste HTTP da parte dei components. Questi ultimi si iscrivono all'evento e attendono una risposta che verrà successivamente gestita nell'apposita **arrow function**.*

4.4 **Login**

Il processo di login in un'applicazione riveste un ruolo fondamentale, poiché consente agli utenti di accedere ai propri account in modo sicuro.

Dettagliatamente, l'utente compila un semplice form inserendo una coppia di credenziali e, tramite l'azione di un pulsante, invia i dati al componente adatto. Il *LoginComponent*, dopo aver controllato la validità dei dati, crea una variabile *request* tramite il **pattern *DTO*** e la invia al service, che si occuperà di gestire l'interazione con il *back-end*. Una volta ricevuta una qualsiasi risposta (sia essa positiva o negativa), viene visualizzato un messaggio tramite il **toastr** adatto e, in caso di successo, l'utente viene indirizzato alla homepage e i suoi dati salvati nel **localStorage**.

4.4.1 **Sequence diagram**



Sequence diagram del servizio di autenticazione descritto precedentemente.

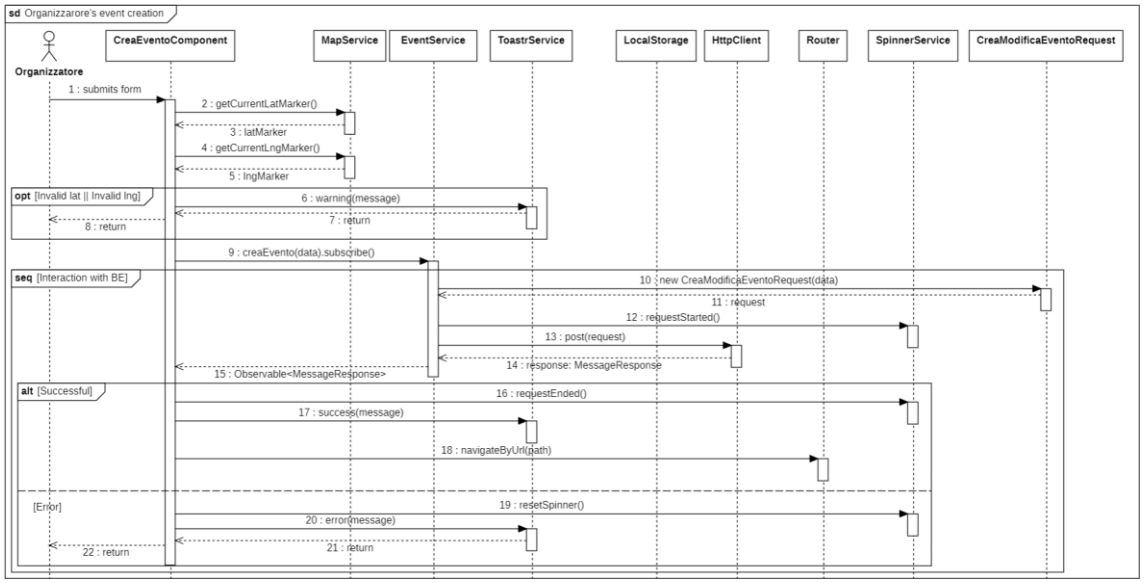
4.5 **Creazione di un evento**

Funzionalità cruciale dell'intera applicazione è indubbiamente la creazione di un evento. A tale scopo vi è stata dedicata una pagina specifica dotata di un **form** dove inserire i dati necessari e di una **mappa interattiva** tramite la quale è possibile selezionare il luogo dell'evento.

Dopo aver verificato la validità dei dati, viene delegato al service il compito di interagire con il *back-end* mediante una chiamata *POST* con il *DTO* della richiesta.

È sostanziale sottolineare che, in fase di creazione di un evento, di solito, vengono inviate diverse e-mail. Questo può comportare un aumento significativo dei tempi necessari per ricevere una risposta. Pertanto, al fine di garantire un'esperienza utente più fluida, è stato implementato uno ***spinner*** che segnala all'utente l'esistenza di un processo in corso.

4.5.1 **Sequence diagram**



Sequence diagram della creazione di un evento descritta precedente.

5 Git

È un **sistema di controllo versione** utilizzato per gestire progetti software e tracciare le modifiche al codice sorgente nel corso del tempo. Consente agli sviluppatori di collaborare efficacemente tra loro e mantenere un registro storico delle modifiche apportate al codice. I vari componenti fondamentali da conoscere sono:

Repository (repo) – Archivio che contiene tutti i file, le cartelle e la cronologia delle modifiche di un progetto. Può essere **locale** (sulla macchina) o **remoto** (su un server).

Branching – Un ramo (branch) è una "copia" separata del codice che consente di lavorare su nuove funzionalità senza influenzare il ramo principale (solitamente chiamato **master** o **main**).

GitHub – Servizi che offrono hosting remoto per i repository Git. Fornisce funzionalità di collaborazione, richieste di pull e altro ancora.

5.1 Comandi principali

```
1 git init
```

Inizializzazione di una nuova repository nella directory corrente.

```
1 git remote add origin https://github.com/nomeAccount/nomeRepository.git
```

Collegamento della repo locale ad una repo remota, raggiunta tramite il link.

```
1 git add .
```

*Aggiunta di **tutti** i files alla repository locale.*

```
1 git commit -m "Messaggio"
```

Snapshot delle modifiche apportate ai file con associato un messaggio.

```
1 git push origin nomeBranch
```

Caricamento del commit locale nella repository remota.

I primi due comandi vengono utilizzati solamente prima della creazione del progetto, mentre gli ultimi 3, in sequenza, ogni qual volta si ritiene siano state apportate modifiche significative al progetto.

5.2 Codice sorgente

L'intero progetto è disponibile al seguente link. La repo è pubblica, quindi il progetto è **open source**. Il vantaggio principale è dato da una **comunità attiva** di sviluppatori e utenti che possono **contribuire alle migliorie**.

6 Docker

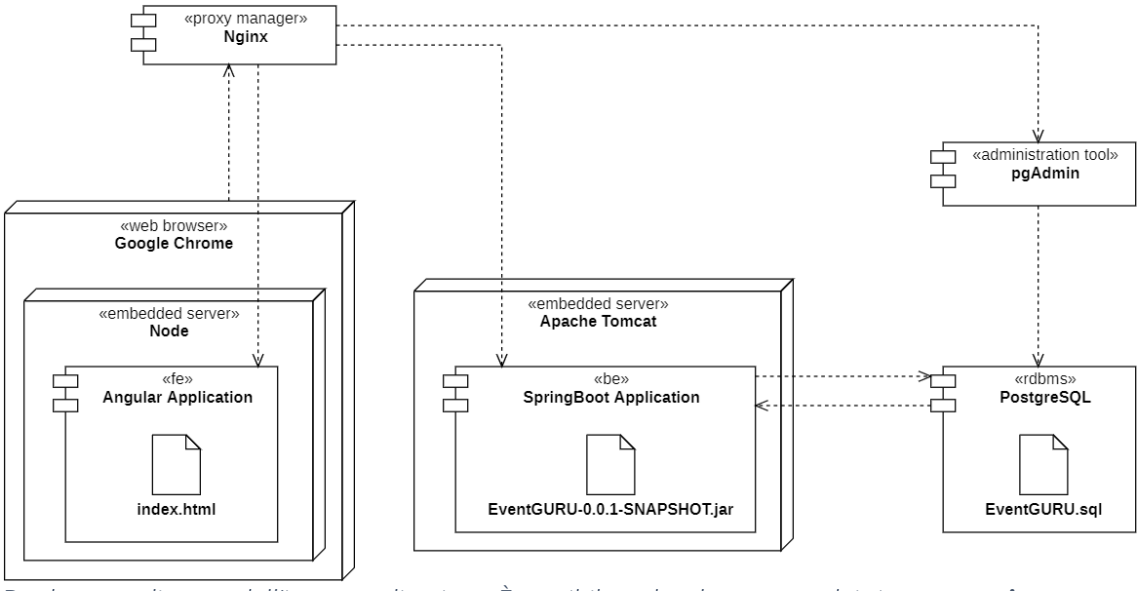
Piattaforma di virtualizzazione leggera che consente di creare, distribuire ed eseguire applicazioni in **containers**. I container *Docker* contengono tutto il necessario per eseguire un'applicazione, inclusi il codice, le librerie e le dipendenze, isolati dal sistema operativo sottostante. Questa tecnologia è utilizzata per semplificare il processo di sviluppo, distribuzione e gestione delle applicazioni. Di seguito alcune nomenclature utili per comprendere l'architettura:

Immagine – Pacchetto leggero che contiene tutto il necessario per eseguire un'applicazione, inclusi il codice, le librerie, le dipendenze e le configurazioni. Le immagini sono utilizzate per creare containers.

Container – Istanza eseguibile di un'immagine *Docker*. Sono leggeri, portabili e possono essere eseguiti in qualsiasi ambiente *Docker* compatibile.

Dockerfile – File di configurazione che contiene le istruzioni passo-passo per la creazione di un'immagine Docker. Specifica le dipendenze, le configurazioni e le azioni da eseguire durante la costruzione dell'immagine.

6.1 Deployment diagram



*Deployment diagram dell'intera applicazione. È possibile vedere la presenza dei cinque **container** e come sono relazionati tra loro. All'interno di alcuni è presente l'**artifact**, ovvero un file eseguibile che racchiude l'intero progetto. Viene usato il proxy manager Nginx per gestire il routing del traffico HTTP e HTTPS.*

6.2 Guida all'installazione e all'esecuzione

Prerequisiti:

- Java >= 17.0.7
- Apache Maven >= 3.8.7
- Docker >= 24.0.6
- Git >= 2.40.0
- Node >= 19.7.0
- Npm >= 9.5.0

I seguenti comandi devono essere eseguiti dopo aver scelto un path specifico nel file system e aver creato un'apposita cartella vuota.

```
1 git clone https://github.com/ffonti/EventGURU.git
```

Download dell'intero progetto in locale.

```
1 npm install
2 npm run build
```

*Comandi da eseguire **all'interno della cartella "FE"**.*

```
1 mvn clean install -DskipTest
```

*Script maven da lanciare nell'apposita sezione all'interno di Intellij, **dentro la cartella "BE"**. Crea il jar BE/target/EventGURU-0.0.1-SNAPSHOT.jar.*

```
1 docker compose build
2 docker compose up -d
```

*Costruisce le immagini e avvia il container, sempre **dentro la cartella "BE"**.*

Il servizio è raggiungibile da *localhost:81*. Per autenticarsi al servizio utilizzare:
username: admin@example.com
password: changeme

Una volta autenticato è necessario configurare nuova e-mail e password.
Successivamente, tramite la *navbar,* raggiungere la sezione **Dashboard** quindi **Proxy Hosts**. Aggiungere un nuovo proxy host specificando i dati del front-end, cioè:
Domain Names: localhost
Forward Hostname / IP: EventGURU_FE
Forward Port: 80

Successivamente aggiungere una **custom location** per il back-end:
Define location: /api
Forward Hostname / IP: EventGURU_BE
Forward Port: 8080

Salvare le modifiche e aprire l'applicazione all'indirizzo localhost.

N.B. È consigliato l'utilizzo del browser *Google Chrome*.

7 Conclusioni

7.1 Scelte implementative

Questa sezione mira a fornire una chiara comprensione delle decisioni di progettazione chiave adottate durante lo sviluppo, evidenziando le ragioni che ne hanno guidato l'implementazione. Ogni scelta è stata ponderata attentamente per garantire un equilibrio tra efficienza, manutenibilità e chiarezza nell'esposizione dell'architettura dell'applicazione.

Class diagrams (BE) – La decisione di non includere **alcuni** class diagrams è stata guidata dal principio di fornire, all'interno di questo documento, informazioni utili e significative. Nei package in questione, le classi, giustamente, mostrano un livello di interazione limitato o nullo tra di loro. L'inclusione di class diagrams in tali contesti avrebbe aggiunto complessità senza apportare alcun valore sostanziale nella comprensione dell'architettura complessiva.

Class diagrams (FE) – La mancanza di class diagrams nel *front-end* è giustificata dalla natura dei progetti stessi, che spesso seguono un'architettura basata su componenti fortemente modularizzati. La struttura tipica di questi progetti, indipendentemente dalla tecnologia utilizzata, tende a ridurre le interazioni dirette tra classi, poiché i componenti sono progettati per essere autonomi e comunicare attraverso interfacce ben definite. La presenza di class diagrams avrebbe non avrebbe offerto alcuna rappresentazione delle interazioni tra classi, che rimangono minimali e focalizzate sulla comunicazione tra componenti specifici.

Polimorfismo – Nel package *service* del *back-end*, è stata adottata una strategia di progettazione basata sul polimorfismo per favorire la flessibilità e la manutenibilità del codice. Nello specifico, sono state definite delle interfacce contenenti la firma dei metodi correlati alle funzionalità specifiche del service stesso. Successivamente, ciascuna interfaccia è implementata da una classe concreta all'interno del package. Queste classi forniscono l'implementazione effettiva dei metodi dichiarati nell'interfaccia, dettagliando il comportamento corrispondente. Non è presente il class diagram del package per il motivo spiegato in precedenza, ma è comunque possibile consultare il class diagram del package *security* (3.5.1), con all'interno il proprio service.

7.2 Sviluppi futuri

Data l'intenzione di esplorare attivamente diverse aree dell'informatica, al fine di ampliare e approfondire le mie competenze, di seguito verranno esposte alcune direzioni chiave che s'intendono intraprendere per migliorare e arricchire l'esperienza degli utenti e la funzionalità dell'applicazione:

Analisi dei dati – Pianificazione di soluzioni avanzate di analisi dei dati per rivelare *insight* profondi sul comportamento degli utenti con lo scopo di prevedere le preferenze dei singoli utenti.

Intelligenza Artificiale – Implementazione di algoritmi di intelligenza artificiale che, avendo come *dataset* i dati forniti dal punto precedente, mirano all'integrazione di un servizio di suggerimento di eventi adatti all'utente in base alle sue precedenti iscrizioni.

Partecipazione della community – Dato che l'applicazione è open source, si incoraggia attivamente la partecipazione della community di sviluppatori esterni. La community può proporre nuove funzionalità, identificare e risolvere bug, e offrire ottimizzazioni delle prestazioni. La presenza di un repository su una piattaforma di hosting, come GitHub, facilita la gestione dei pull request e delle segnalazioni di problemi, rendendo il processo di collaborazione trasparente e accessibile.

7.3 **Considerazioni personali**

Il percorso di sviluppo di EventGURU è stato un viaggio **stimolante** e **formativo** attraverso il mondo complesso delle tecnologie e dello sviluppo software.

Durante questo progetto, ho affrontato delle **sfide** legate alla creazione di un'applicazione completa, utilizzando metodologie di sviluppo come il testing, la progettazione sia *back-end* che *front-end*, l'integrazione e distribuzione continua (*CI/CD*). L'utilizzo sistematico di *Git* ha garantito una gestione efficace del codice, mentre la modellazione *UML* ha aiutato a definire in modo chiaro la struttura dell'applicazione.

Nel processo di sviluppo, ho applicato i principi chiave della **OOP**, incorporando con successo il polimorfismo, l'ereditarietà e l'incapsulamento. Questi paradigmi hanno fornito solidità al design dell'applicazione e migliorato la manutenibilità del codice.

Questo progetto ha rappresentato non solo un'opportunità di creare una soluzione significativa per il problema dell'isolamento digitale, ma anche una palestra per l'apprendimento e la **crescita personale**. Sono grato per le sfide affrontate, i successi ottenuti e le lezioni apprese durante questa esperienza di sviluppo.

EventGURU è molto più di un'applicazione; è un capitolo significativo del mio percorso professionale che continuerà a influenzare positivamente le mie future avventure nel mondo dello sviluppo software.