UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica e dell'Automazione

TESI DI LAUREA

Progettazione e implementazione in tecnologia Spring di un portale per la gestione di eventi

Design and implementation in Spring technology of a portal for handling events

Relatore

Prof. Domenico Ursino

Candidato

Michele Longo

ANNO ACCADEMICO 2022-2023

Ai mieni nonni Agostinelli.

Sommario

La tesi in oggetto si dedica allo sviluppo di un portale web per la gestione di eventi attraverso l'utilizzo del framework Spring. Tale scelta rappresenta l'approccio ideale nello sviluppo di applicazioni Java in ambito enterprise. La popolarità di Spring è attribuibile alle sue caratteristiche distintive di flessibilità, scalabilità e modularità, le quali semplificano lo sviluppo e la manutenzione di applicazioni complesse, avvalendosi, in particolare, delle funzionalità offerte da Spring MVC, Spring Boot e Spring Security. La tesi si addentrerà nelle diverse fasi che hanno condotto alla realizzazione del portale, iniziando con la presentazione dei vari framework utilizzati e concentrando l'attenzione sulle fasi di analisi dei requisiti, progettazione e implementazione del portale. Al termine, vengono tratte alcune conclusioni che evidenziano i contributi chiave del lavoro svolto e suggeriscono possibili sviuppi futuri.

Keyword: Spring Framework, Spring Data, Hibernate, Spring Security, Thymeleaf, Spring MVC, Spring Boot

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Negli ultimi decenni, l'evoluzione di Internet ha ridefinito radicalmente il modo in cui interagiamo con le teconologie digitali, facilitando la connessione e lo scambio globale di informazioni. Inizialmente, le pagine web erano statiche e limitate nelle loro funzionalità, ma con il tempo la crescente necessità di interattività e dinamicità ha guidato lo sviluppo di nuove tecnologie.

Con il progresso tecnologico, sono emersi i primi framework di sviluppo web che hanno semplificato e organizzato la creazione di applicazioni complesse. Questi strumenti hanno agevolato l'implementazione di pattern architetturali, come il Model-View-Controller (MVC), contribuendo a una migliore organizzazione del codice e della logica di business.

L'avvento di tecnologie come JavaScript e l'introduzione di AJAX (Asynchronous JavaScript and XML) hanno consentito alle applicazioni web di aggiornare dinamicamente i contenuti senza dover ricaricare l'intera pagina, garantendo un'esperienza utente più fluida e reattiva.

Negli anni successivi, l'attenzione si è focalizzata sulle Single Page Applications (SPA), in cui il caricamento iniziale della pagina è seguito da aggiornamenti asincroni. Framework front-end come Angular hanno dominato questa scena, offrendo potenti strumenti per la creazione di interfacce utente dinamiche e reattive.

Parallelamente, i framework di sviluppo back-end, come Spring per Java, hanno continuato a evolversi, fornendo strumenti sofisticati per la gestione di dati, autenticazione, sicurezza e integrazione con database.

La crescita di Internet ha portato a una maggiore complessità nelle esigenze delle applicazioni web, spingendo lo sviluppo di nuovi paradigmi e strumenti. Oggi, la tendenza è verso l'adozione di microservizi e la containerizzazione, come Docker e Kubernetes, garantendo una maggiore scalabilità, flessibilità e affidabilità delle applicazioni web.

Tra le tecnologie avanzate per il Web dinamico spiccano Spring, Hibernate e Spring Security, ampiamente utilizzati nello sviluppo di applicazioni enterprise. Spring, framework open source per Java, offre libertà di sviluppo insieme a soluzioni documentate per le problematiche comuni.

Hibernate è una piattaforma per lo sviluppo di applicazioni Java che fornisce un servizio di Object Relational Mapping, capace di garantire la rappresentazione e il mantenimento su database relazionale di un sistema di classi Java.

La crescente attenzione alla sicurezza in questi ultimi anni è andata di pari passo allo sviuppo di framework che si occupano di essa. Spring Security ne è un esempio; la sua evoluzione in simultanea con Spring ne fa un compagno ideale per gestire la sicurezza in ambiente Java.

Questa tesi si colloca in un contesto dinamico e in continua evoluzione, come quello descritto in precedenza. In particolare, essa illustra lo sviluppo di un portale dedicato alla gestione degli eventi, rispondendo efficacemente alle esigenze di organizzatori, partecipanti e amministratori.

Il nostro sistema offre una piattaforma completa e intuitiva per la pianificazione, la promozione e la partecipazione agli eventi di vario genere. Attraverso caratteristiche innovative e best practices nello sviluppo software, il nostro obiettivo è creare un ambiente virtuale che semplifichi ogni fase del processo.

In questo contesto, il nostro lavoro simula lo sviluppo di una piattaforma all'avanguardia, progettata per adattarsi alle mutevoli esigenze del settore degli eventi. Sebbene il portale non verrà mai utilizzato effettivamente, a scopo didattico miriamo a creare un ambiente completo. Con un focus sull'innovazione tecnologica, intendiamo fornire una simulazione di un nuovo standard di eccellenza nella gestione degli eventi.

Attraverso questa tesi, esploreremo in dettaglio il processo di progettazione, sviluppo e implementazione del nostro portale, mettendo in luce le scelte architetturali, le tecnologie adottate e le sfide superate. Ci concentreremo, anche, sulla simulazione delle interazioni e delle funzionalità offerte dalla piattaforma, fornendo una comprensione approfondita delle sue capacità e potenzialità.

La presente tesi è articolata in sei capitoli, di cui i primi due capitoli si concentrano sull'approfondimento dei framework, essenziali per una comprensione approfondita del lavoro. Nel Capitolo 1, esploreremo Spring Framework e Spring Boot, dedicando particolare attenzione alle peculiarità di Spring necessarie per sfruttare appieno le potenzialità del framework. Verrà illustrato, inoltre, il concetto di Dependency Injection che ha reso popolare Spring. Nel capitolo illustreremo anche Hibernate, che fornisce un servizio di Object Relational Mapping, facilitando l'iterazione con il database.

Nel Capitolo 2 approfondiamo il fondamentale aspetto della sicurezza con una panoramica su Spring Security. Questo framework rappresenta lo standard di riferimento per garantire la sicurezza nelle applicazioni basate su Spring, offrendo un robusto sistema di autenticazione e autorizzazione. Analizzando in dettaglio il funzionamento di Spring Security, il capitolo fornisce una comprensione approfondita delle sue caratteristiche e delle modalità con cui assicura la protezione dell'applicazione. Per quanto riguarda la parte relativa alla View, abbiamo adottato Thymeleaf come template engine. Ciò consente l'inclusione efficiente di espressioni dinamiche nei template, facilitando la presentazione flessibile dei dati e la creazione di interfacce utente interattive. L'associazione di Thymeleaf e Spring contribuisce a una sinergia armoniosa tra la gestione della sicurezza e la dinamicità delle pagine, garantendo un'esperienza coerente e intuitiva.

Conclusa l'illustrazione dei vari framework utilizzati, l'attenzione si sposta alla fase di progettazione. Nel Capitolo 3, focalizziamo l'attenzione sulla progettazione del sistema per la gestione degli eventi, partendo dall'analisi approfondità dei requisiti. Questo passo è essenziale per delineare chiaramente le funzionalità che il nostro portale dovrà offrire. Attraverso la definizione dei requisiti, miriamo a semplificare il processo di sviluppo, fornendo una guida chiara e strutturata per l'implementazione delle diverse funzionalità. Questa fase di progettazione è cruciale per garantire che il sistema sia in grado di soddisfare pienamente le esigenze degli organizzatori, dei partecipanti e degli amministratori, assicurando un'esperienza utente ottimale.

Nel Capitolo 4 approfondiamo l'architettura del sistema, la progettazione del database e la gestione della sicurezza, compresa la definizione dei ruoli degli utenti. La scelta di dedicare una sezione specifica a questi aspetti è motivata dalla centralità che essi rivestono nel garantire il corretto funzionamento e la sicurezza del portale. L'architettura del sistema è progettata in modo da assicurare la scalabilità e la modularità, elementi fondamentali per

adattarsi alle future esigenze e facilitare l'integrazione di nuove funzionalità. La progettazione del database è guidata dalla necessità di garantire una gestione efficiente dei dati, mentre la definizione dei ruoli degli utenti è cruciale per garantire un accesso appropriato alle diverse funzionalità del portale, contribuendo, così, alla sicurezza complessiva del sistema.

Nel Capitolo 5, ci immergiamo nella fase implementativa del sistema. Questa sezione è dedicata alla configurazione del sistema e alla stesura del manuale d'uso. La scelta di affrontare questi aspetti in dettaglio è guidata dalla volontà di garantire un'implementazione efficace e una fruizione agevole del nostro portale. La configurazione del sistema riveste un ruolo cruciale, poiché un'adeguata preparazione dell'ambiente di sviluppo contribuisce a garantire il corretto funzionamento dell'applicazione. Il manuale d'uso, invece, è essenziale per fornire agli utenti un riferimento chiaro su come interagire con il sistema, assicurando un'esperienza d'uso positiva e agevole.

Infine, nel Capitolo 6, traiamo le conclusioni dal lavoro svolto e delineamo alcune possibili direzioni per lo sviluppo futuro. Questa sezione è fondamentale per sintetizzare gli obiettivi raggiunti, le sfide superate e i risultati ottentuti con l'implementazione del nostro portale. L'analisi delle conclusioni fornisce un quadro completo sull'efficacia del progetto e sulle sue potenzialità. Inoltre, discutiamo di possibili sviluppi futuri, considerando l'evoluzione tecnologica e le esigenze in continua evoluzione degli utenti. Questa prospettiva orientata al futuro offre spunti per ulteriori miglioramenti e aggiornamenti, consentendo al nostro progetto di rimanere all'avanguardia nel campo della gestione degli eventi.

CAPITOLO 1

Introduzione a Spring

In questo capitolo si introducono i fondamenti del framework Spring, un pilastro essenziale per la costruzione di moderne applicazioni enterprise basate su Java. Verranno esplorate le caratteristiche chiave che hanno contribuito a rendere Spring ampiamente utilizzato e apprezzato nel panorama dello sviluppo software. Spring framework offre un modello completo di programmazione e configurazione, semplificando la creazione di applicazioni robuste e scalabili.

1.1 Spring Framework

Nel definire Spring è bene partire dalla definizione di framework in generale. Nel contesto informatico un framework, è un architettura logica di supporto per lo svilupo software, implementando design pattern specifici e agevolando il lavoro del programmatore. Questa strutture comprende classi astratte e le relative relazioni, si ottiene in queto modo un insieme di classi concrete con un insieme di relazioni tra classi. Oltre al contesto informatico, il terminte è utilizzato in ambito economico-gestionale per indicare una struttura pianificata che supporta prassi, metodologie o sistemi di gestione.

I framework sono accompagnati da una serie di librerie di codice per il linking con linguaggi di programmazione, insieme a strumenti di supporto allo sviluppo, come IDE e debugger. L'utilizzo di framework impone al programmatore una metodologia specifica nel processo di sviluppo del software. Lo scopo di un framework è quello di risparmiare al programmatore la riscrittura di codice già scritto in precedenza, ciò si presenta sempre più spesso con l'evolversi delle interfacce utente o con l'aumento della qualità di un software con funzionalità secondarie simili.

Il framework Spring, in particolare, semplifica la creazione di applicazioni enterprise in Java, fornendo tutti gli strumenti necessari per adottare il linguaggio Java in un ambiente enterprise. Supporta anche Groovy e Kotlin come linguaggi alternativi sulla JVM. A partire dalla Versione 6.0, Spring richiede Java 19 o versioni successive. Spring è un progetto open source e possiede una vasta community che contribuisce al suo sviluppo. Questo ha contribuito al successo continuo e alla sua evoluzione nel corso del tempo. Attualmente la versione di Spring è la 6.1 ed è quella su cui si basa il progetto relativo alla presente tesi.

Spring è organizzato in moduli, consentendo alle applicazioni di selezionare specifici moduli in base alle proprie esigenze. Nel suo nucleo si trovano i moduli del core container, che comprendono un modello di configurazine e un meccanismo di injection delle dipendenze. Oltre a ciò, il framework Spring fornisce un supporto fondamentale per diverse architetture

applicative, tra cui messaggistica, gestine transazionale dei dati e persistenza, nonché sviluppo web. Inoltre, include il framework basato su Servlet Spring MVC, e parallelamente, il framework web reattivo Spring WebFlux. Si è parlato di modularità del framework Spring; di seguito si elencano sia le funzionalità peculiari di Spring che i moduli utilizzati nello sviluppo della nostra web app.

- *Spring Core:* rappresenta l'essenza di Spring; attraverso l'interfaccia *ApplicationContext* consente la realizzazione dell'Inversion of Control (IoC), permettendo la gestione delle dipendenze e la creazione di oggetti gestiti dal framework.
- *Spring Data:* si occupa dell'accesso e dell'iterazione con i dati nel contesto di Spring. Fornisce un'astrazione semplificata per le operazioni di accesso ai dati, riducendo la complessità della gestione delle persistenze e fornendo supporto per le diverse tecnologie di memorizzazione dei dati.
- *Spring Web on Servlet*: si riferisce alla funzionalità di sviluppo web fornite da Spring utilizzando la tecnologia Servlet. Include la gestione delle richieste HTTP, la mappature degli URL e altre utilità per lo sviluppo di applicazioni web robuste.
- *Spring Security*: offre funzionalità di sicureza per le applicazioni Spring, gestendo l'autenticazione, l'autorizzazione e la protezione contro diverse minacce. Fornisce strumenti per implementare facilmente controlli di sicurezza avanzati all'interno di Spring. Garantisce una grande flessibilità e personalizzazione dei suoi componenti, rendendolo adattabile ad ogni contesto.

1.1.1 Spring Boot

Spring Boot, arrivato alla Versione 3, è un progetto facente parte dell'ecosistema Spring che introduce il concetto di *convention over configuration*, traducibile come convenzione anziché configurazione. L'idea è quella di offrire una configurazione predefinita che è possibile personalizzare secondo necessità.

Il risultato è quello di scrivere meno codice, perché si seguono convenzioni note, e differenziare le applicazioni con pochi e piccoli step. Quindi, anziché scrivere tutte le configurazioni per ogni singola applicazione, è più efficiente partire da una configurazione già predefinita e cambiare ciò che è diverso dalla convenzione.

Per creare un'applicazione web Spring, è necessario configurare un Servlet Container, creare un'istanza Servlet e assicurarsi di configurare correttamente quest'ultima in modo che Tomcat la chiami per qualsiasi richiesta del client. Con Spring Boot non è necessario scrivere tali configurazioni perché è tutto già scritto nella configurazione di default.

Le caratteristiche chiave di Spring Boot sono:

- *Creazione semplificata del progetto:* si può ottenere un'applicazione scheletro vuota ma già configurata attraverso un servizio di inizializzazione del progetto.
- *Starter di dipendenza:* Spring Boot raggruppa alcune dipendenze per uno scopo specifico, non è necessario capire tutte le dipendenze indispensabili da aggiungere al progetto, né le versioni da utilizzare per la compatibilità.
- *Autoconfigurazione basata sulle dipendenze:* in base alle dipendenze aggiunte al progetto, Spring Boot definisce alcune configurazioni predefinite.

1.2 Spring Core

Al cuore del framework Spring risiede la potente essenza di Spring Core, un elemento fondamentale che ha modellato l'architettura di numerose applicazioni enterprise basate su Java. Spring Core rappresenta il nucleo pulsante del framework, fornendo un modello di programmazione e configurazione che ha rivoluzionato l'approccio degli sviluppatori alla creazione di applicazioni solide e scalabili.

Questo nucleo incorpora principi chiave, come l'Aspect-Oriented Programming (AOP) e l'Inversion of Control (IoC) per decentralizzare la gestione delle dipendenze, consentendo un'architettura più flessibile e manutenibile. Si esaminano, ora, brevemente gli aspetti chiave, di Spring Core, per poi approfondirli nelle prossime sezioni:

- *Inversion of Controll (IoC):* un principio cardine di Spring Core è l'Inversione del Controllo (IoC), che ribalta la tradizionale gestione delle dipendenze. Invece di lasciare alle classi la responsabilità di creare e gestire le proprie dipendenze, Spring delega questo compito al suo contenitore, facilitando la creazione di applicazioni modulari e facilmente manutenibili.
- *Container:* il Contenitore Spring, uno degli aspetti più distintivi di Spring Core, offre un ambiente di runtime per le applicazioni Spring. Questo contenitore gestisce la creazione, l'inizializzazione e la distruzione degli oggetti all'interno dell'applicazione, garantendo un'elevata coesione e una bassa dipendenza tra i componenti. Ciò è reso possibile attraverso l'interfaccia Application Context.
- *Aspect-OrientedProgramming (AOP):* la Programmazione Orientata agli Aspetti (AOP) integra la Programmazione Orientata agli Oggetti (OOP). Nell'OOP, l'unità principale di modularità è la classe, mentre nell'AOP l'unità di modularità è l'aspetto. Gli aspetti catturano comportamenti trasversali a diverse classi, consentendo una separazione efficace delle responsabilità. Questi possono essere implementati mediante l'uso di classi regolari (approccio basato sullo schema) o, di classi regolari annotate con l'annotazione *@Aspect* (in stile *@AspectJ*).

Tra i benefici di Spring Core possiamo citare i seguenti:

- *Flessibilità e Modularità:* l'approccio IoC di Spring Core favorisce una progettazione più flessibile e modulare, permettendo alle applicazioni di adattarsi facilmente ai cambiamenti e di evitare accoppiamenti eccessivi tra i componenti.
- *Gestione delle Dipendenze Semplificata:* il Contenitore Spring semplifica notevolmente la gestione delle dipendenze all'interno delle applicazioni, riducendo la complessità nel configurare e mantenere le relazioni tra i diversi componenti.

In conclusione, questa sezione delinea il ruolo centrale di Spring Core nel promuovere una progettazione più pulita e una gestione efficiente delle dipendenze, elementi cruciali per lo sviluppo di applicazioni enterprise di successo.

1.2.1 Inversion of Control (IoC) e Beans

In questa sezione si analizza nel dettaglio il principio dell'Inversion of Control (IoC) , l'Injection of Dependency (DI) e si definisce il concetto di bean, peculiare in Spring.

L'Inversion of Control è un principio fondamentale che ribalta il tradizionale flusso di controllo dell'applicazione. Invece di gestire direttamente la creazione e il controllo di oggetti,

```java
public class Negozio {
    private Prodotto prodotto;

    public Negozio() {
        prodotto = new Prodotto();
    }

}
```

Figura 1.1: Costruttore classico

Spring sposta questa responsabilità al proprio IoC Container. Così è il Container che gestisce la configurazione e l'iniziazione degli oggetti. L'IoC di Spring è spesso realizzato attraverso l'uso delle Dependency Injection, consentendo agli oggetti di ricevere le loro dipendenze, invece di crearle internamente. La Dependency Injection (DI), quindi, rappresenta una modalità specializzata di IoC.

Il Contenitore IoC inietta tali dipendenze durante la creazione del bean. Tale processo sottolinea l'inversione di controllo, poiché è il contenitore stesso a gestire l'istanziazione o la collocazione delle dipendenze. Ciò che permette tale funzionalità è l'interfaccia *BeanFactory* che offre un meccanismo in grado di gestire diversi tipi di oggetti, nonché *ApplicationContext*, una sotto-interfaccia di *BeanFactory* che aggiunge:

- la gestione delle risorse dei messaggi (per l'implementare dell'internazionalizzazione)
- la pubblicazione di eventi
- contesti specifici dell'applicazione, come il *WebApplicationContext* per l'utilizzo in applicazioni web.

Quindi *BeanFactory* fornisce il framework di configurazione e le funzionalità base, mentre *ApplicationContext* estende queste funzionalità introducendo aspetti specifici per le applicazioni enterprise.

Nel contesto di Spring, gli oggetti che costituiscono la struttura portante dell'applicazione che sono gestiti dal Contenitore IoC di Spring vengono chiamati "beans".

Un bean non è altro che un oggetto istanziato, assemblato e gestito dal Contenitore IoC di Spring. L'interfaccia *ApplicationContext* rappresentea il contenitore di IoC; questa riceve istruzioni attraverso i metadati di configurazione. Nel nostro caso i metadati sono stati espressi attraverso Java, utilizzando l'annotazione *@Bean* si indica che un metodo istanzia, configura e inizializza un nuovo oggetto da gestire mediante il Contenitore IoC.

Per facilitare la comprensione sulla Dependency Injection (DI) in Spring, e per mostrare quanto sia immediata la sua implementazione, si fa di solito un esempio su due tipiche classi. Se una classe *Negozio* ha bisogno di un'istanza della classe *Prodotto* al suo interno, nella programmazione tradizionale, se vogliamo utilizzare la classe *Prodotto* in *Negozio* dovremmo istanziare una implementazione di *Prodotto* nella classe *Negozio*, come mostrato in Figura 1.1 .

Nella Figura 1.2, invece, si è utilizzato *ApplicationContext* per implementare un contenitore di Inversion of Control (IoC) e la nostra classe diventerà un bean. In questo caso, si è sfruttato la Dependency Injection tramite costruttore, dove l'argomento del costruttore rappresenta la dipendenza. Ciò rappresenta uno schema concettuale di come funziona l'Inversion of Control in Spring.

```java
public class Negozio {
    private Prodotto prodotto;

    public Negozio(Prodotto prodotto) {
        this.prodotto = prodotto;
    }
```

Figura 1.2: Dependency Injection

1.3 Spring Data

Il modulo di accesso ai dati (*Data Access*) all'interno del framework Spring costituisce un fondamentale strumento nell'ambito dello sviluppo software, offrendo un approccio consolidato e robusto per l'interazione con sorgenti di dati, specialmente con riferimento ai database relazionali. Tale modulo, attraverso l'implementazione di funzionalità e astrazioni avanzate, agevola la complessa operazione di manipolazione dei dati, contribuendo all'efficienza e alla manutenibilità delle applicazioni.

Le principali caratteristiche distintive di questo modulo includono un solido supporto transazionale, la presenza di template di accesso ai dati, come *JdbcTemplate*, *HibernateTemplate* e *JpaTemplate* per ridurre il boilerplate code, e un'astrazione del livello di persistenza, che permette di lavorare a un livello di astrazione superiore senza dover affrontare dettagli specifici delle tecnologie di persistenza.

Un aspetto di notevole rilevanza è l'integrazione trasparente con tecnologie ORM, come Hibernate e JPA, agevolando il mapping degli oggetti Java alle entità di database. La gestione delle eccezioni, mediante una gerarchia coerente di eccezioni come *DataAccessException*, rappresenta un ulteriore punto di forza.

L'implementazione delle transazioni, sia in modo dichiarativo, attraverso annotazioni, che mediante configurazioni XML, conferisce al modulo una flessibilità significativa, consentendo una gestione efficiente delle transazioni nell'ambito dell'accesso ai dati.

1.3.1 Gestione delle transazioni

La gestione delle transazioni in Spring Framework è un aspetto fondamentale che offre un supporto completo e flessibile per garantire l'integrità e la coerenza delle operazioni sui database in un'applicazione. Le transazioni sono essenziali per assicurare che un insieme di operazioni di accesso ai dati venga eseguito in modo atomico, garantendo che tutte le modifiche al database avvengano con successo o vengano, tutte annullate, in caso di errore.

Spring Gestisce le transazioni come di seguito specificato:

- *Astrazione Transazionale*: permette di gestire transazioni in modo uniforme su diverse tecnologie di persistenza, come JDBC, Hibernate o JPA. Ciò consente di scrivere codice senza preoccuparsi dei dettagli specifici della tecnologia di accesso ai dati utilizzata.
- *Gestione Dichiarativa*: è supportata attraverso annotazioni o configurazioni XML. Ciò consente di definire comportamenti transazionali direttamente nelle classi di servizio o nei metodi.
- *Gestione Programmatica:* è ovviamente possibile gestire il tutto avendo un maggiore controllo sulle transazioni attraverso il codice, se necessario.

1.3.2 Accesso dati con JDBC

La selezione di un approccio per l'accesso al database JDBC in un contesto Spring offre diverse alternative, ciascuna caratterizzata da attributi distintivi. Di seguito sono presentate alcune delle metodologie disponibili:

- *JdbcTemplate***:** rappresenta l'approccio classico e preminente di Spring JDBC, riconosciuto per il suo status di "livello più basso". Tutte le altre tecniche di accesso fanno uso di un oggetto *JdbcTemplate* come infrastruttura sottostante. Fa ricorso a placeholder JDBC convenzionali, quali `?`, per la specifica dei parametri nelle query SQL.
- *NamedParameterJdbcTemplate:* incapsula un oggetto *JdbcTemplate*, fornendo parametri nominati in luogo dei tradizionali placeholder JDBC. Questo tecnica promuove una migliore documentazione e facilità d'uso in situazioni in cui numerosi parametri sono inclusi in una dichiarazione SQL.
- *SimpleJdbcInsert e SimpleJdbcCall:* questi approcci ottimizzano i metadati del database, limitando la necessità di configurazione. La loro operatività è vincolata alla presenza di metadati sufficienti nel database.
- *Oggetti RDBMS (MappingSqlQuery, SqlUpdate e StoredProcedure):* richiedono la creazione di oggetti riutilizzabili e thread-safe durante la fase di inizializzazione del layer di accesso ai dati. Tale approccio consente la definizione della stringa di query, la dichiarazione dei parametri e la compilazione della query. Successivamente, i metodi *execute*(...), *update*(...) e *findObject*(...) possono essere invocati ripetutamente con differenti valori dei parametri.

La scelta di uno di questi approcci offre la possibilità di combinare e adattare le varie metodologie al fine di includere funzionalità provenienti da tecniche diverse, in base alle esigenze specifiche del progetto. Per comprendere come il framework JDBC opera all'interno del framework Spring esaminiamo ora, la sua struttura a pacchetti, organizzata in quattro principali categorie.

1. *core:* comprende la classe *JdbcTemplate* e le relative interfacce di callback, oltre alle classi *SimpleJdbcInsert* e *SimpleJdbcCall*. Fornisce un'astrazione di livello basso per il controllo del processo JDBC di base, gestendo errori e semplificando le operazioni JDBC.
2. *datasource:* contiene classi di utilità per semplificare l'accesso a *DataSource*. Include anche supporto per la creazione di database embedded.
3. *object:* include classi che rappresentano query, aggiornamenti e stored procedure di RDBMS come oggetti riutilizzabili e thread-safe. Questo approccio orientato agli oggetti dipende dall'astrazione di livello inferiore nel pacchetto *core*.
4. *support:* fornisce funzionalità di traduzione delle eccezioni *SQLException* e alcune classi di utilità. Le eccezioni JDBC vengono tradotte in eccezioni definite nel pacchetto *org.springframework.dao*, semplificando la gestione degli errori senza richiedere implementazioni specifiche di JDBC.

Questa struttura organizzativa offre un'ampia gamma di funzionalità per semplificare l'accesso e la gestione delle risorse del database, fornendo, al contempo, un'astrazione efficace dai dettagli specifici di JDBC e dell'RDBMS.

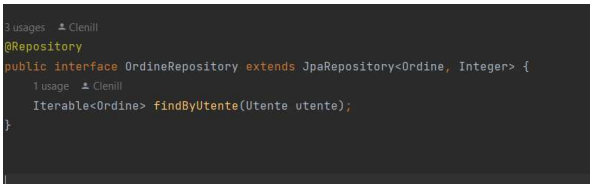

Figura 1.3: Repository Ordine

1.3.3 Spring Data JPA

Spring Data JPA semplifica notevolmente l'implementazione dei repository basati su JPA (Java Persistence API). La sua finalità è rendere più agevole la creazione di applicazioni Spring che utilizzano tecnologie di accesso ai dati. L'implementazione di uno strato di accesso ai dati per un'applicazione può risultare complessa, richiedendo la scrittura di un eccessivo codice ripetitivo anche per eseguire le query più elementari.

Spring Data JPA si propone di migliorare significativamente l'implementazione degli strati di accesso ai dati, riducendo lo sforzo allo stretto necessario. In qualità di sviluppatore, è possibile definire le interfacce dei propri repository utilizzando varie tecniche, e Spring le collegherà automaticamente. È, persino, possibile impiegare finder personalizzati o effettuare query tramite esempi, lasciando a Spring la scrittura della query al posto dello sviluppatore; ciò è stato sfruttato ampiamente nei nostri repository, evitando la scrittura di query SQL.

Il concetto centrale nell'astrazione del repository di Spring Data è l'interfaccia *Repository*. Essa riceve come argomenti di tipo la classe di dominio da gestire e il tipo di identificatore della classe di dominio. Nello specifico, dovendo gestire una tabella di un database, saranno passate la classe che gestisce tale entità e la sua chiave primaria, che corrisponde alla chiave primaria della tabella.

Questa interfaccia funge principalmente da interfaccia di marcatura per catturare i tipi con cui lavorare e per aiutare a scoprire le interfacce che la estendono. Le interfacce *CrudRepository* e *ListCrudRepository* forniscono una funzionalità avanzata di CRUD per la classe di entità che viene gestita. È anche possibile l'utilizzo di *Jparepository*, questa interfaccia estende *CrudRepository*. Per impostazione predefinita, i repository Spring Data JPA sono bean Spring predefiniti.

Definizione di Metodi di Query

Il repository offre due strategie per derivare una query specifica del database dal nome del metodo; in particolare, ciò può avvenire:

- *Derivando la query direttamente dal nome del metodo.*
- *Utilizzando una query definita manualmente.*

Il parsing dei nomi dei metodi di query è diviso in soggetto e predicato. La prima parte (*find…By*, *exists…By*) definisce il soggetto della query, mentre la seconda parte forma il predicato. La clausola introduttiva (soggetto) può contenere ulteriori espressioni. Qualsiasi testo tra *find* e *By* è considerato descrittivo a meno che non vengano utilizzate parole chiave di limitazione dei risultati, come *Distinct*, per impostare un flag distintivo sulla query da creare o *First,* per limitare i risultati della query. I metodi di query che restituiscono risultati multipli possono utilizzare le standard Java *Iterable*, *List* e *Set*. Un esempio di utilizzo di repository per effettuare una ricerca nella tabella *ordine* viene riportato in Figura 1.3:

Questa query restituisce un oggetto *Iterable* con tutti gli ordini effetuati dall'Utente passato come parametro; le due tabelle sono collegate tramite annotazione *@ManytoOne* e *@OneTo-*

Many. L'annotazione *@Repository* è essenziale per Spring; anche le tabelle hanno bisogno di annotazioni, come nome e colonne, che si riferiscono direttamente al database connesso.

1.3.4 Supporto DAO

In Spring, il DAO (Data Access Object) è un componente utilizzato per facilitare l'accesso e la manipolazione dei dati memorizzati in un sistema di persistenza dei dati, come un database relazionale o un sistema di archiviazione. Il DAO svolge un ruolo chiave nell'implementazione del livello di accesso ai dati di un'applicazione. Ecco alcuni concetti chiave relativi ai DAO in Spring:

- *Separazione delle responsabilità:* il DAO segue il principio di separazione delle responsabilità, consentendo di isolare il codice specifico del database dal resto dell'applicazione. Ciò favorisce una migliore manutenibilità e facilita il testing.
- *Interfaccia DAO:* un DAO è tipicamente definito tramite un'interfaccia che dichiara i metodi di base per le operazioni di accesso ai dati, come la lettura, la scrittura, l'aggiornamento e l'eliminazione. L'implementazione concreta di questa interfaccia gestirà le specificità del sistema di persistenza utilizzato.
- *Implementazione del DAO:* l'implementazione concreta del DAO si occupa dell'interazione diretta con il sistema di persistenza. In Spring, è comune utilizzare il supporto di Spring JDBC o Spring Data JPA per semplificare le operazioni di accesso ai dati.
- *Gestione delle Eccezioni:* i DAO devono gestire le eccezioni specifiche del sistema di persistenza e, se necessario, convertirle in eccezioni più generiche o specifiche dell'applicazione.
- *Transazioni:* Spring gestisce le transazioni in modo dichiarativo attraverso annotazioni o file XML di configurazione. I DAO possono partecipare alle transazioni, garantendo la coerenza dei dati e il rollback delle transazioni in caso di eccezioni.
- *Integrazione con Spring Framework:* i DAO vengono spesso integrati con il contesto di Spring come bean gestiti da Spring Container. Ciò consente l'iniezione delle dipendenze, la gestione del ciclo di vita e la gestione delle transazioni da parte del framework.

1.3.5 ORM con Spring

Il Framework Spring offre supporto all'integrazione con l'API Java Persistence (JPA) e fornisce un supporto nativo per Hibernate per la gestione delle risorse, l'implementazione degli oggetti di accesso ai dati (DAO) e le strategie di transazione.

Ad esempio, per quanto riguarda Hibernate, viene fornito un supporto notevole con numerose funzionalità di Inversion of Control (IoC) che affrontano molte delle tipiche problematiche di integrazione legate a Hibernate. È possibile configurare tutte le funzionalità supportate per gli strumenti di mappatura oggetto-relazionale (ORM) mediante l'iniezione delle dipendenze.

Questi strumenti possono partecipare alla gestione delle risorse e delle transazioni di Spring, conformandosi alle gerarchie di eccezioni generiche di transazione e DAO di Spring. L'approccio di integrazione consigliato consiste nel codificare i DAO utilizzando le API standard di Hibernate o JPA.

Spring aggiunge significativi miglioramenti al livello ORM prescelto quando si sviluppano applicazioni di accesso ai dati. È possibile utilizzare gran parte del supporto ORM come una libreria, indipendentemente dalla tecnologia adottata, poiché tutto è progettato come

un insieme di JavaBeans riutilizzabili. L'utilizzo di ORM in un contenitore IoC di Spring semplifica la configurazione e la distribuzione.

I vantaggi derivanti dall'uso del Framework Spring per la creazione dei propri DAO ORM includono:

- *Facilità di testing*: l'approccio IoC di Spring semplifica la sostituzione delle implementazioni e delle posizioni di configurazione delle istanze di Hibernate *SessionFactory*, delle istanze di JDBC *DataSource*, dei gestori di transazioni e delle implementazioni di oggetti mappati. Ciò semplifica notevolmente il testing di ciascuna parte del codice correlato alla persistenza in modo isolato.

- *Eccezioni comuni di accesso ai dati*: Spring è in grado di incapsulare le eccezioni provenienti dallo strumento ORM, convertendole da eccezioni proprietarie a una comune gerarchia di *DataAccessException* a runtime. Questa funzionalità consente di gestire la maggior parte delle eccezioni di persistenza, che sono non recuperabili, solo nei livelli appropriati, senza la necessità di fastidiosi blocchi di codice per la gestione, il lancio e la dichiarazione delle eccezioni.

- *Gestione generale delle risorse*: i contesti delle applicazioni Spring possono gestire la posizione e la configurazione delle istanze di Hibernate *SessionFactory*, delle istanze di JPA *EntityManagerFactory*, delle istanze di JDBC *DataSource* e di altre risorse correlate. Ciò semplifica la gestione e la modifica di questi valori.

- *Gestione integrata delle transazioni*: È possibile avvolgere il proprio codice ORM con un interceptor di metodo dichiarativo, basato su programmazione orientata agli aspetti (AOP), sia mediante l'annotazione *@Transactional* che configurando in un file di configurazione XML. In entrambi i casi, le semantiche delle transazioni e la gestione delle eccezioni (rollback e così via) sono gestite automaticamente.

L'obiettivo principale dell'integrazione ORM di Spring è la chiara stratificazione dell'applicazione (con qualsiasi tecnologia di accesso ai dati e transazione) e l'accoppiamento debole degli oggetti dell'applicazione. Ciò consente di avere un approccio semplice e coerente per collegare gli oggetti dell'applicazione, mantenendoli riutilizzabili e privi delle dipendenze del Contenitore il più possibile.

In una tipica applicazione Spring, molti oggetti importanti sono JavaBeans: modelli di accesso ai dati, oggetti di accesso ai dati, gestori di transazioni, servizi aziendali che utilizzano gli oggetti di accesso ai dati e i gestori di transazioni, risolutori di visualizzazione Web, controller Web che utilizzano i servizi aziendali, e così via.

1.3.6 Hibernate 6

Hibernate è una piattaforma middleware open source pensata per facilitare lo sviluppo di applicazioni Java. Questo framework fornisce un servizio di Object-Relational Mapping, ovvero gestisce la persistenza dei dati sul database attraverso la rappresentazione e il mantenimento su un sistema di oggetti Java. In sostanza, esso rende i dati relazionali visibili a un programma scritto in Java, in una forma naturale e sicura dal punto di vista dei tipi. Ciò avviene:

- facilitando la scrittura di query complesse e il lavoro con i loro risultati;

- permettendo al programma di sincronizzare facilmente i cambiamenti apportati in memoria con il database, rispettando le proprietà ACID delle transazioni;

- consentendo di effettuare ottimizzazioni delle prestazioni dopo che la logica di persistenza di base è già stata scritta.

I dati relazionali sono al centro dell'attenzione, insieme all'importanza della sicurezza dei tipi. L'obiettivo dell'ORM è di eliminare il codice fragile e non sicuro dal punto di vista dei tipi e rendere i programmi complessi più facili da mantenere nel lungo termine. Inoltre, l'ORM facilita molto l'ottimizzazione delle prestazioni successivamente, dopo che la logica di persistenza di base è già stata scritta.

Hibernate è nato nel 2001, ha subito diverse evoluzioni e la sua ultima versione stabile è la 6.2. È ampiamente utilizzato per lo sviluppo di applicazioni web. Agisce come uno strato software intermedio tra il livello logico di busines e il livello di persistenza dei dati sul database, noto come Data Access Layer. Questo strato svolge un ruolo cruciale nel gestire la comunicazione tra oggetti Java e il database relazionale, fornendo un meccanismo efficace di mapping e sincronizzazione.

L'utilizzo di Hibernate semplifica notevolmente la gestione della persistenza dei dati nelle applicazioni Java. Le sue funzionalità di ORM consentono di rappresentare oggetti Java come record nel database, e viceversa.

Hibernate 6 rappresenta una ristrutturazione significativa della soluzione ORM più diffusa e completa del mondo. Questo ridisegno ha coinvolto praticamente ogni sotto-sistema di Hibernate, compresi API, annotazioni di mappatura e il linguaggio di query. Il nuovo Hibernate è caratterizzato da maggiore potenza, robustezza e sicurezza.

Hibernate e JPA

Hibernate è stato l'ispirazione dietro la Java (ora Jakarta) Persistence API, o JPA, e include una completa implementazione dell'ultima revisione di questa specifica.

Il progetto Hibernate ha avuto inizio nel 2001, quando la frustrazione di Gavin King per Entity Beans in EJB 2 ha raggiunto il culmine. In breve tempo, ha superato altri concorrenti open source e commerciali per diventare la soluzione di persistenza più popolare per Java.

Nel corso degli ultimi due decenni, molte persone talentuose hanno contribuito allo sviluppo di Hibernate. Le componenti principali dell'API di Hibernate possono essere raggruppate in tre elementi:

- *Implementazione delle API definite da JPA*: include le interfacce *EntityManagerFactory* ed *EntityManager*, e le annotazioni di mapping O/R definite da JPA.
- *API nativa*: espone l'intero set di funzionalità disponibili, centrata sulle interfacce *SessionFactory* (che estende *EntityManagerFactory*) e *Session* (che estende *EntityManager*).
- Annotazioni di mapping: integrano le annotazioni di mapping O/R definite da JPA e possono essere utilizzate con le interfacce definite da JPA o con l'API nativa.

Scrittura del Codice Java con Hibernate

Per chi si avvicina per la prima volta a Hibernate e JPA, potrebbe sorgere la domanda su come sia strutturato il codice correlato alla persistenza. In genere, esso si articola in due livelli:

- *Rappresentazione del Modello di Dati in Java*: si tratta di un insieme di classi di entità annotate.
- *Funzioni per Interagire con le API di Hibernate*: queste funzioni interagiscono con le API di Hibernate per eseguire le operazioni di persistenza associate alle diverse transazioni.

La prima parte, il modello di dati o "dominio", è solitamente più agevole da implementare. Tuttavia, la realizzazione di un lavoro eccellente e pulito in questa fase avrà un impatto significativo sul successo della seconda parte.

Solitamente, il modello di dominio viene implementato come un insieme di "Plain Old Java Objects" (POJO), ossia semplici classi Java prive di dipendenze dirette da infrastrutture tecniche o da logiche applicative, che gestiscono l'elaborazione delle richieste, la gestione delle transazioni, le comunicazioni o l'interazione con il database.

La seconda parte del codice è molto più complessa da realizzare correttamente. Questo codice deve:

- gestire transazioni e sessioni;
- interagire con il database tramite la sessione di Hibernate;
- recuperare e preparare i dati necessari per l'interfaccia utente e gestire eventuali situazioni di errore.

Entità

Un'entità è una classe Java che rappresenta dati all'interno di una tabella di un database relazionale. L'entità è in stretta relazione con la tabella del database, e i suoi attributi corrispondono alle colonne della tabella. In particolare, ogni entità deve avere un identificatore (id) che si mappa sulla chiave primaria della tabella, consentendo l'associazione univoca tra una riga della tabella e un'istanza della classe Java. Le entità possono anche avere associazioni con altre entità, solitamente mappate attraverso chiavi esterne nelle tabelle del database.

Ogni entità deve essere annotata con *@Entity*, in alternativa si possono mappare classi con l'uso di XML. È possibile rendere non nulla una colonna; ciò può essere fatto tramite due tipi di annotazioni JPA, una appartiene al livello logico ed è *@Basic(optional=false)*, l'altra appartiene al livello di mapping ed è *@Column(nullable=false)*, è importante questa distinzione perché le informazioni possono essere dedotte dal livello logico al livello di mapping, e non viceversa. Oltre a queste due annotazioni si può utilizzare l'annotazione *@NotNull* da Bean Validation, che è considerata l'opzione migliore.

Le associazioni, che si ricorda sono relazioni tra entità, solitamente si classificano in base alla loro molteplicità; ciò è possibile attraverso annotazioni all'interno di classi definite come entità. Le associazioni possono essere di diversi tipi; considerando due classi entità A e B, si ha:

- *Uno a uno*, collega al massimo un'istanza unica di A con al massimo un'istanza unica di B.
- *Molti a uno*, collega zero o più istanze di A con un'istanza unica di B.
- *Molti a molti*, collega zero o più istanze di A con zero o più istanze di B.

L'associazione tra entità può essere bidirezionale o unidirezionale. Tutto ciò è svolto con annotazioni, rendendo molto semplice collegare tra loro classi entità e stabilire le regole di collegamento; un esempio di associazione con annotazioni è riportato in Figura 1.4.

Evitando di entrare troppo nel dettaglio ci sono diverse configurazioni comuni a queste annotazioni, in modo da poter configurare opportunamente ciò di cui si ha bisogno.

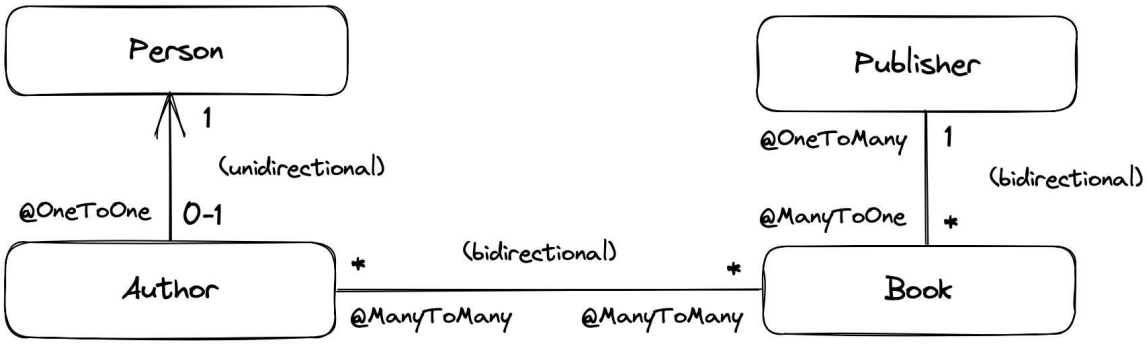

Figura 1.4: Schema tipi di associazioni

Mapping Object/Relational

Attraverso le annotazioni e lo schema visto precedentemente, Hibernate andrà a inferire uno schema relazionale completo, esso può essere anche esportato nel database, attraverso la creazione di codice SQL. Questo processo è chiamato (top-down mapping). Raramente le classi Java precedono lo schema relazionale; di solito si ha già uno schema e si costruisce il modello di dominio attorno allo schema, applicando una mappatura dal basso verso l'alto (bottom-up mapping).

Interazione con il Database

Per interagire con il database, cioè eseguire query o inserire, aggiornare o eliminare dati, è necessario impostare un'istanza di un'oggetto; questa può essere:

- un JPA *EntityManager*,
- una Hibernate *Session*, oppure
- una Hibernate *StatelessSession*.

Dato che l'interfaccia *Session* estende *EntityManager*, l'unica differenza tra loro è che *Session* offre qualche operazione aggiuntiva.

In Hibernate ogni *EntityManager* è una *Session*. Un'istanza di *Session* è una sessione con stato e media l'interazione tra il programma e il database tramite operazioni su un contesto di persistenza. Quest'ultimo è una sorta di cache, chiamato anche cache di primo livello. Per ogni istanza di entità letta dal database all'interno del contesto di persistenza e per ogni nuova entità resa persistente all'interno del contesto di persistenza, il contesto detiene una mappatura univoca dall'identificatore dell'istanza dell'entità all'istanza stessa.

Connessione al Database

Hibernate può integrarsi con un *javax.sql.DataSource* per ottenere connessioni JDBC. Le applicazioni informeranno Hibernate sulla DataSource tramite l'impostazione (obbligatoria) *hibernate.connection.datasource*. Per quanto riguarda la configurazione dei driver è possibile configurare il nome della classe del driver JDBC da utilizzare, così come l'url di connessione. È inoltre possibile fornire l'username e la pasword relativi al Datasource.

1.4 Spring Web MVC

Il pattern MVC (Model-View-Controller) è un design pattern architetturale ampiamente utilizzato nell'ambito dello sviluppo del software per separare le responsabilità di un'applicazione in tre componenti principali: Model, View e Controller. Questa separazione aiuta a organizzare il codice in modo modulare, facilitando la manutenzione e l'estensibilità del software. In Spring, l'implementazione del pattern MVC è fornita attraverso il modulo Spring MVC. Di seguito sono descritti i concetti principali su come Spring MVC implementa il pattern MVC:

- *Model:* rappresenta i dati e la logica di business dell'applicazione. Gestisce lo stato dell'applicazione e l'accesso ai dati. In Spring MVC, il Model può essere rappresentato da oggetti Java semplici (POJO), spesso detti "bean".
- *View:* rappresenta l'interfaccia utente dell'applicazione. Presenta i dati provenienti dal Model agli utenti e riceve input dagli utenti. È spesso rappresentata da pagine JSP (JavaServer Pages) o da template basati su tecnologie come Thymeleaf o FreeMarker.
- *Controller:* gestisce le interazioni tra utente e l'input. Agisce come intermediario tra la View e il Model. In Spring MVC l'implementazione è fatta con classi Java annotate con *@Controller*. Queste gestiscono richieste HTTP, interagiscono con il Model e selezionano la View. I metodi del Controller sono spesso annotati con *@RequestMapping* o altre annotazioni simili per mappare le richieste URL sui metodi appropriati.

Spring Web MVC è il framework web originale costruito su Servlet API ed è stato incluso nel framework Spring fin dall'inizio. Parallelamente a Spring Web MVC, il framework Spring 5.0 ha introdotto un framework web reattivo chiamato "Spring WebFlux".

Spring MVC, come molti altri framework web, è progettato seguendo il pattern del front controller, in cui un Servlet centrale, noto come *DispatcherServlet*, svolge un ruolo cruciale nell'elaborazione delle richieste.

Questo Servlet fornisce un algoritmo condiviso per la gestione delle richieste, mentre l'effettiva esecuzione delle operazioni è demandata a componenti delegati configurabili. Questo modello è caratterizzato da una notevole flessibilità e offre il supporto per flussi di lavoro diversificati.

Per rispettare la specifica Servlet, il *DispatcherServlet*, analogamente ad ogni altro Servlet, deve essere dichiarato e mappato utilizzando la configurazione Java o attraverso il file di descrizione *web.xml*.

Successivamente, il *DispatcherServlet* fa uso della configurazione fornita da Spring per individuare i componenti delegati necessari per il mapping delle richieste, la risoluzione delle viste, la gestione delle eccezioni e altre attività correlate.

L'uso del *DispatcherServlet* come punto centrale di controllo consente la gestione efficace delle richieste in un ambiente flessibile e adattabile alle esigenze specifiche dell'applicazione.

1.4.1 DispatcherServlet

Spring MVC, come molti framework web, è progettato attorno al pattern del front controller, dove un Servlet centrale, il *DispatcherServlet*, fornisce un algoritmo condiviso per l'elaborazione delle richieste, mentre il lavoro effettivo è delegato a componenti configurabili.

Il *DispatcherServlet* richiede la presenza di un *WebApplicationContext* per la propria configurazione. Il *WebApplicationContext* è collegato al *ServletContext* e il Servlet specifico con cui è associato.

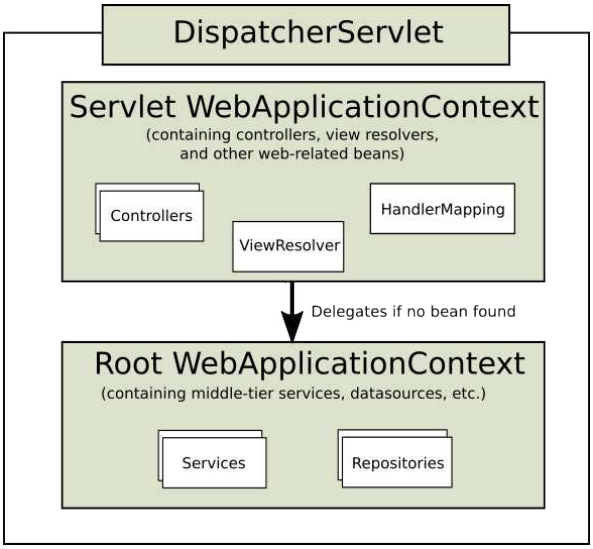

Figura 1.5: DispatcherServlet

WebApplicationContext è inoltre vincolato al *ServletContext* in modo che le applicazioni possano utilizzare i metodi statici forniti da *RequestContextUtils* per recuperare il *WebApplicationContext* nel caso in cui sia necessario accedervi (Figura 1.5).

Il *DispatcherServlet* delega a particolari bean il compito di elaborare le richieste e generare le risposte appropriate. Per "bean speciali" si intendono istanze di oggetti gestite da Spring che implementano contratti del framework. Solitamente, questi bean sono dotati di contratti integrati, ma è possibile personalizzarne le proprietà, estenderli o sostituirli. Di seguito si descrivono alcuni di questi bean speciali:

- *HandlerMapping:* consente di mappare una richiesta verso un gestore (handler) insieme a un elenco di intercettori per la pre- e post-elaborazione. Le due principali implementazioni di *HandlerMapping* sono *RequestMappingHandlerMapping* e *SimpleUrlHandlerMapping*.
- *ViewResolver:* risolve i nomi delle viste basati su stringhe restituiti da un *handler* per ottenere effettivamente una vista con cui renderizzare la risposta.
- *LocaleResolver:* Spring fornisce meccanismi per risolvere automaticamente la lingua preferita del client. Questo permette all'applicazione di presentare contenuti e messaggi localizzati in modo dinamico, migliorando l'esperienza dell'utente in un contesto internazionale.
- *ThemeResolver:* il supporto per i temi in Spring consente di definire diversi aspetti visivi dell'interfaccia utente, come colori, stili e immagini, all'interno di temi distinti.

Il *DispatcherServlet* elabora le richieste mediante i seguenti passi:

1. *Ricerca e Associazione del WebApplicationContext:* ricerca del *WebApplicationContext*, che viene, poi, associato alla richiesta come attributo. Quest'ultimo è accessibile dal controller e da altri elementi nel processo.
2. *Associazione del Locale Resolver:* il risolutore della lingua viene associato alla richiesta per consentire agli elementi nel processo di risolvere la lingua da utilizzare durante l'elaborazione della richiesta.

3. *Associazione del Theme Resolver:* il risolutore del tema viene associato alla richiesta per consentire agli elementi, come le viste, di determinare quale tema utilizzare.

4. *Multipart Handling:* se viene specificato un risolutore di file multiparte, la richiesta viene ispezionata per verificare la presenza di parti multiple. In quest'ultimo caso, la richiesta viene incapsulata in una *MultipartHttpServletRequest* per ulteriori elaborazioni da parte di altri elementi nel processo.

5. *Ricerca di un Handler:* viene cercato un gestore appropriato. Se viene trovato un gestore, viene eseguita la catena di esecuzione associata ad esso (pre-elaboratori, post-elaboratori e controller) per preparare un modello per la renderizzazione.

6. *Renderizzazione della Vista:* se viene restituito un modello, viene eseguita la renderizzazione della vista. Se non viene restituito alcun modello (ad esempio, a causa di un pre-elaboratore o post-elaboratore che intercetta la richiesta per motivi di sicurezza), non viene eseguita alcuna renderizzazione della vista, poiché la richiesta potrebbe già essere stata soddisfatta.

Inoltre, i bean *HandlerExceptionResolver* dichiarati nel *WebApplicationContext* vengono utilizzati per risolvere le eccezioni generate durante l'elaborazione delle richieste.

1.4.2 Controller

Nel contesto del framework Spring MVC, il modello di programmazione basato su annotazioni consente l'implementazione di controller attraverso l'uso di annotazioni come *@Controller* e *@RestController*. Questi componenti utilizzano tali annotazioni per definire il mapping delle richieste, gestire l'input delle richieste, gestire eccezioni e altre funzionalità. I controller presentano firme di metodo flessibili e non richiedono l'estensione di classi di base né l'implementazione di interfacce specifiche.

Nel contesto del framework Spring MVC, è possibile definire i controller mediante una dichiarazione standard di bean Spring nel *WebApplicationContext* del Servlet. L'annotazione *@Controller* consente la rilevazione automatica, allineata al supporto generale di Spring per la rilevazione delle classi *@Component* nel classpath e la registrazione automatica delle definizioni di bean per esse. Essa funge anche da stereotipo per la classe annotata, indicando il suo ruolo come componente web.

Si può utilizzare l'annotazione *@RequestMapping* per mappare le richieste ai metodi dei controller nel framework Spring MVC. Questa annotazione dispone di vari attributi per effettuare il mapping in base all'URL, al metodo HTTP, ai parametri della richiesta, agli header e ai tipi di media. Può essere usata a livello di classe per esprimere mapping condivisi, o a livello di metodo per limitarsi a un mapping specifico per un endpoint. Sono a disposizione vari shortcut specifiche del metodo HTTP per l'annotazione *@RequestMapping*, come *@GetMapping* o *@PostMapping*.

Nella Figura 1.6 è riportato il codice del Controller quando è richiesta la pagina index; in questo caso si utilizza la classe *ModelAndView* alla quale si può assegnare sia il model che la view; anche se sono due entità distinte, con l'utilizzo di questa classe possono essere restituite insieme.

Quindi *ModelAndView* rappresenta un model e una vista restituiti da un gestore, da risolvere attraverso un *DispatcherServlet*. La vista può assumere la forma di un nome di vista String che dovrà essere risolto da un oggetto *ViewResolver*. Per quanto riguarda il model, si può semplicemente aggiungere attributi ad esso e renderizzarli nella vista, come, ad esempio, il nome dell'utente collegato per poterne usufruire lato front-ent. Un esempio è riportato nella Figura 1.7.

```java
@Controller
public class PublicController {

    @Autowired
    private EventoRepository eventoRepo;
    @Autowired
    private UtenteRepository utenteRepo;
    @Autowired
    private FaqRepository faqRepo;
    @Autowired
    private final EventoService eventoServ;
    @Autowired
    private TipologiaRepository tipologiaRepository;
    //-------------------------------------------
    Clenill
    public PublicController(EventoService eventoServ) { this.eventoServ = eventoServ; }

    Clenill *
    @GetMapping({"/public", "/public/index", "/public/", "/", "/index"})
    public  ModelAndView getAllEventPublic(Model model, RedirectAttributes redirectAttributes){



        ModelAndView gae = new ModelAndView( viewName: "public/index");

        Iterable<Evento> eventiperdata = eventoRepo.findAllByOrderByLocalDateAsc();
        gae.addObject( attributeName: "eventidata", eventiperdata);
        gae.addObject( attributeName: "tuttieventi", eventoRepo.findAll());
        return gae;
    }
```

Figura 1.6: PublicController pagina index

```
public PublicController(EventoService eventoServ) { this.eventoServ = eventoServ; }

 Clenill *
@GetMapping({"/public", "/public/index", "/public/", "/", "/index"})
public  ModelAndView getAllEventPublic(Model model, RedirectAttributes redirectAttributes){

    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

    if (authentication.isAuthenticated()){
        Object principal = authentication.getPrincipal();
        if(principal instanceof UserDetails){
             Clenill
            UserDetails userDetails = (UserDetails) principal;
            String username = userDetails.getUsername();
            Collection<? extends GrantedAuthority> collectionautorita = userDetails.getAuthorities();

            String primaAut = collectionautorita.iterator().next().getAuthority();
            model.addAttribute( attributeName: "autorita", primaAut);

            model.addAttribute( attributeName: "nomeutente", username);

        }
    }

    ModelAndView gae = new ModelAndView( viewName: "public/index");

    Iterable<Evento> eventiperdata = eventoRepo.findAllByOrderByLocalDateAsc();
    gae.addObject( attributeName: "eventidata", eventiperdata);
    gae.addObject( attributeName: "tuttieventi", eventoRepo.findAll());
```

Figura 1.7: Aggiunta di nome utente alla Vista

CAPITOLO 2

Altri Strumenti Utilizzati

In questo capitolo sono analizzati gli altri framework e software utilizzati nello sviluppo del progetto. Partiremo da Spring Security che è un framework che si occupa della sicurezza in ambiente Java. Nonostante l'integrazione con Spring mantiene la sua struttura indipendente; infatti per poter essere utilizzato è necessario integrarlo come dipendenza. Thymeleaf è un template engine che, data la sua completezza, è considerato alla stregua di un framework; con questo strumento è stato possibile dare dinamicità al sito. Anche questo è stato integrato tramite dipendenza nel file pom dell'applicazione. Infine viene proposta una panoramica sul DBMS utilizzato e viene illustrata la struttura del database utilizzata nell'applicazione.

2.1 Spring Security

Spring Security nasce come componente separato da Spring, infatti il progetto originale era chiamato in maniera diversa, ovvero *Acegi Security*. Nato come framework Java/Java EE, già nella prima versione si occupava di funzionalità come autenticazione, autorizzazione e altre caratteristiche per applicazioni enterprise. Succesivamente è stato incorporato in Spring e preso il nome di Spring Security con la Versione 2.0. Al momento, la Versione corrente è la 6.2 ed è in corso lo sviluppo della Versione 7; Spring Security è un progetto open source rilasciato sotto licenza Apache 2.0.

In questo decennio la sicurezza informatica è diventata sempre più preminente lato sviluppo software; infatti i programmatori stanno diventando sempre più consapevoli della necessità di software sicuro, e questo avviene già nelle prime fasi di sviluppo del software. Qualità come sicurezza, prestazioni, scalabilità e disponibilità possono avere un impatto nel lungo termine se non prese in considerazioni fin dall'inizio.

Spring Security è un framework che semplifica enormemente l'applicazione della sicurezza per le applicazioni Spring; esso rappresenta la scelta primaria per implementare la sicurezza a livello di applicazione in Sping. Lo scopo è offrire un modo altamente personalizzabile per l'implementazione di autenticazione, autorizzazione e protezione contro attacchi informatici.

2.1.1 Introduzione a Spring Security:

A livello di applicazione con Spring Security si può decidere se ad un utente è consentito eseguire un'azione o utilizzare una parte di dati; i componenti intercettano richieste e si assicurano che chiunque le faccia abbia il permesso di accedere alle risorse protette.

Altra responsabilità riguarda la memorizzazione e il transito di dati tra diverse parti dell'applicazione, intercettando le chiamate i componenti possono agire sui dati, ad esempio applicando algoritmi di crittografia ad essi. Lo scopo del framework è quello di permetterci di scrivere meno codice per implementare la funzionalità desiderata, ciò fa anche Spring Security, con funzionalità predefinite che possono essere personalizzate, garantendo grande flessibilità. La sicurezza è implementata a strati e con Spring Security si implementa al livello più alto.

Inserire Spring Security in un'applicazione Spring Boot è molto facile; basta introdurre la dipendenza nel file *pom.xml*. Di default viene applicata una protezione ad ogni endpoint, di tipo HTTP Basic, questo è un modo in cui un'applicazione web autentica un utente con credenziali che l'applicazione ottiene nell'intestazione della richiesta HTTP.

Non tutti gli endpoint hanno bisogno di essere messi in sicurezza, infatti nel nostro progetto c'è una sezione public, dove non è necessario essere autenticati per accedervi. In Figura 2.1 sono raffigurati i principali componenti di Spring Security e cosa accade quando è effettuata una richiesta dal client:

1. Il filtro di autenticazione delega la richiesta all'*Autentication Manager* e in base alla risposta configura il *Security Context*.
2. L'*Autentication Manager* usa il provider di autenticazione per elaborare quest'ultima attività.
3. Il provider di autenticazione implementa la logica di autenticazione.
4. Lo *User Details Service* implementa la gestione degli utenti, che il provider di autenticazione utilizza nella logica di autenticazione.
5. L'encoder della password implementa la gesione delle password, che il provider di autenticazione utilizza nella logica di autenticazione.
6. Il *Security Context* conserva i dati di autenticazione dopo il processo di autenticazione.

La funzione di *User Details Service* è quella di recuperare i dati dell'utente specifico tramite lo username fornito; è possibile aggiungere funzionalità estendendo User Details Service attraverso l'utilizzo di *User Details Manger*. L'ultimo tassello è *GrantedAuthority*, che conserva l'autorità concessa durante la fase di autenticazione, quindi il ruolo assegnato all'utente. In Figura 2.2 è possibile vedere il legame tra queste classi.

È necessario capire come viene implementato un utente in Spring Security. *User Details* serve al nostro scopo; questa interfaccia può essere espansa all'occorrenza ma, nella sua implementazione di default, c'è tutto ciò che occorre per rappresentare un utente autenticato.

2.1.2 Autenticazione

Spring Security ha bisogno della classe *User Details Service* per eseguire l'autenticazione. Questa classe può essere, a sua volta, espansa da *User Details Manager* aggiungendo ulteriori operazioni per la gestione degli utenti. L'implementazione di User Details Manager scelta è *JdbcUserDatailsManager*, questa classe gestisce gli utenti in un database *SQL*. La connessione al database avviene tramite JDBC, così *JdbcUserDatailsManager* è indipendente da qualsiasi altro framework o specifica legata alla connettività al database. In Figura 2.3 è possibile esaminare il diagramma di flusso di un'applicazione che utilizza *JdbcUserDatailsManager*.

JdbcUserDetailsManager è flessibile e può essere modificata all'occorrenza. Inoltre, essa necessita del *DataSource* per connettersi al database. Quest'ultimo può essere passato come parametro del metodo, oppure come un attributo della classe.

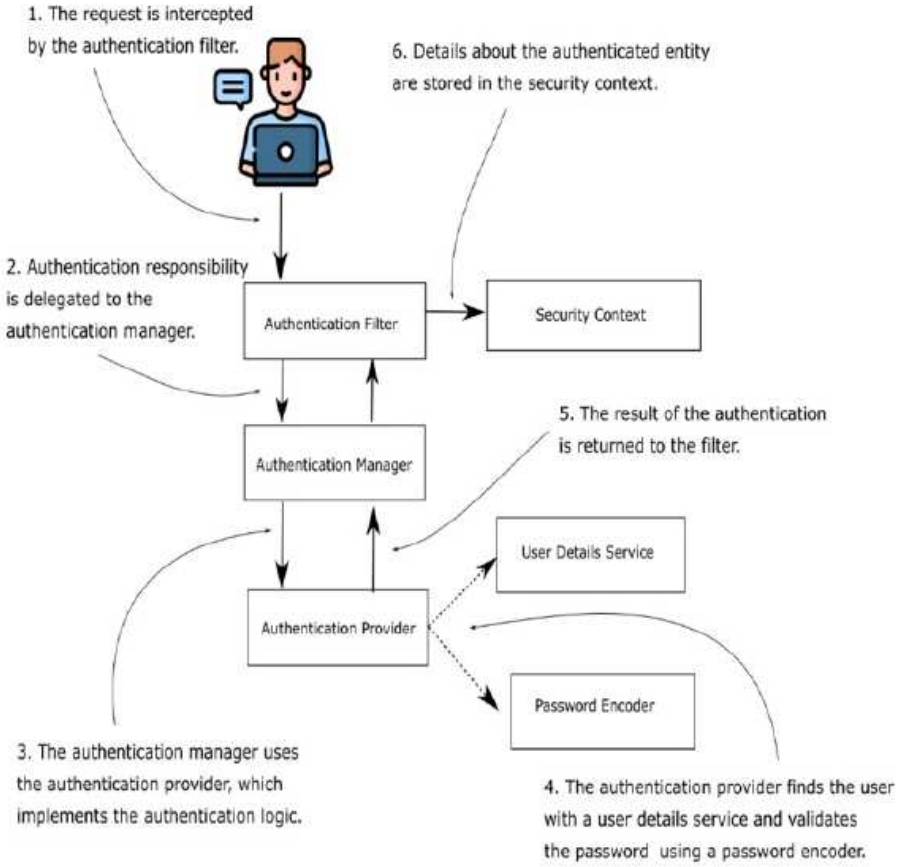

Figura 2.1: Flusso di autenticazione in Spring Security

È, inoltre, possibile configurare le *querie* utilizzate per prelevare dal database i dati necessari all'autenticazione, questo perché *JdbcUserDetailsManager* utilizza delle *querie* di default che devono essere adattate alla struttura del nostro database.

Per quanto riguarda la gestione della password, l'*Authentication Provider* usa un password encoder per validare la password dell'utente durante la fase di autenticazione. Spring Security fornisce già alcune implementazioni di password encoder, queste sono:

- *NoOp Password Encoder:* non decodifica la password, se ne sconsiglia l'utilizzo.
- *Standar Passwrod Encoder:* utilizza *SHA-256* per effettuare l'hashing della la password. Anche questa codifica è deprecata e se ne sconsiglia l'utilizzo.
- *Pbkdf2 Password Encoder:* utilizza una funzione di derivazione della chiave basata su *2(PBKDF2)*.
- *BCrypt Password Encoder:* utilizza una funzione di hashing *bcrypt* forte per la codifica.
- *SCrypt Password Encoder:* utilizza una funzione di scrypt per la codifica.

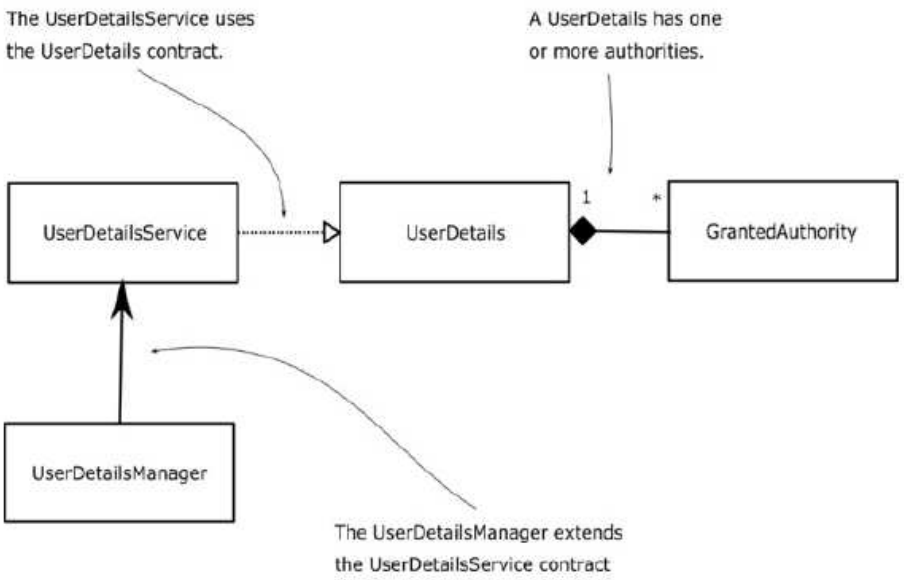

Figura 2.2: Legame che intercorre tra le classi associate alla fase di autenticazione.

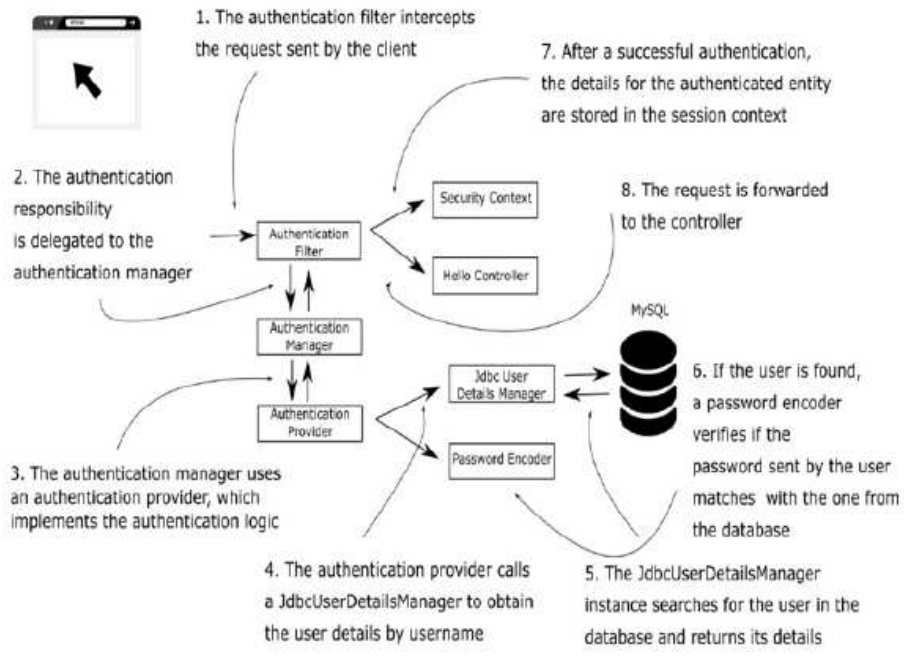

Figura 2.3: Autenticazione tramite Jdbc User Details Manager

Autenticazione in dettaglio

Spring Security permette di modellare le autorizzazioni a livello di richiesta. Quindi, è possibile stabilire che alcuni percorsi siano accessibili solo a determinate autorità. Per

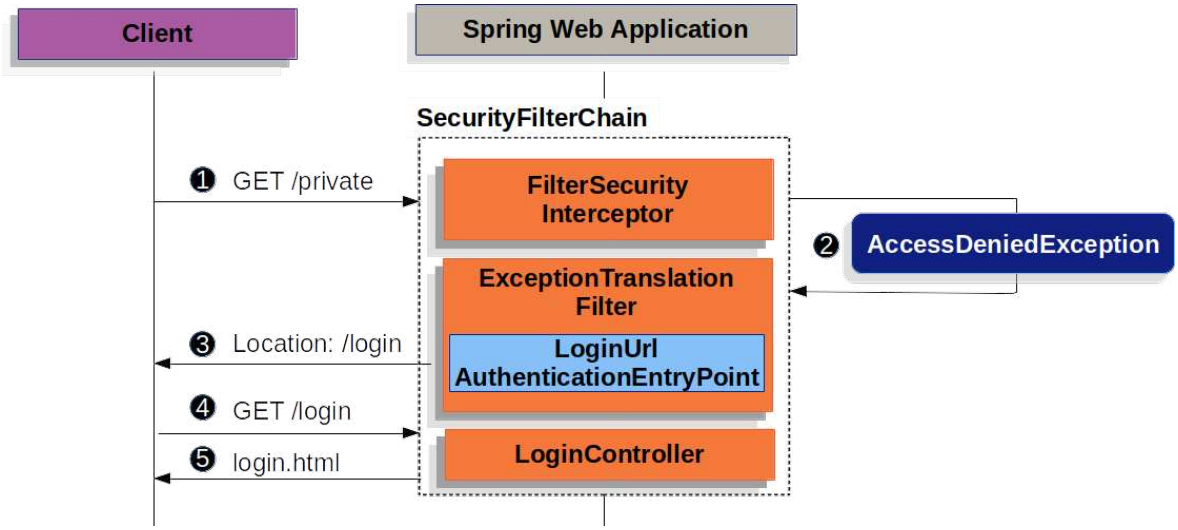

Figura 2.4: Come avviene il redirect alla pagina di login

impostazione predefinita tutte le richieste necessitano di un'autenticazione; è necessario avere un'istanza di *HttpSecurity* personalizzata e dichiarare le regole di autorizzazione.

L'*Authorization Filter* costruisce un *Supplier* (fornitore) che recupera un oggetto *Authentication* dal *Security Context Holder*. Il *Supplier* viene passato insieme alla *HttpServletRequest* all'*Authorization Manager*. Questo confronta la richiesta, personalizzata con regole specifiche di accesso, con i modelli presenti in *authorizeHttpRequet*, ed esegue la regola corrispondente.

L'*Authorization Filter* è l'ultimo filtro nella catena dei filtri in Spring Security; ciò sigifica che gli altri filtri, compresi filtri personalizzati, non richiedono autorizzazione. Vi è la possibilità di aggiungere filtri personalizzati dopo *Authorization Filter;* in tal caso richiederanno l'autorizzazione.

Tutto ciò diventa importante quando si aggiungono endpoint in *Spring MVC*; poiché vengono eseguiti dal *DispatcherServlet* e questo avviene dopo l'*Authorization Filter*, gli endpoint devono essere inclusi in *authorizeHttpRequest* per essere autorizzati.

Come detto in precedenza è possibile autorizzare specifici endpoint con opportune autorizzazioni, *Authorization Filter* elabora le coppie di modello/regola nell'ordine elencato; il linguaggio per configurare le regole in Spring Security è *Ant*.

Una volta che la richiesta è abbinata, si può utilizzare in diversi modi, come *permitAll*, *denyAll* e *hasAuthority* per legarla ad un'autorità specifica. Ci sono diversi modi di configurare una richiesta in Spring Security.

Uno dei modi più comuni per effettuare l'autenticazione è attraverso la validazione di username e password. Spring Security fornisce ovviamente supporto per questo tipo di autenticazione. Le credenziali di accesso vengono inserite in una form HTML; in Figura 2.4 è presente uno schema su come avviene il *redirect* alla form di login.

La Figura 2.4 è descritta nei seguenti 5 punti:

1. Il processo si innesca quando l'utente non autenticato fa richiesta di una risorsa per la quale non è autorizzato.
2. Il filtro di autorizzazione (*Authorization Filter*) indica che un richiesta non autenticata viene negata e lancia un'eccezione *Access Denied Exception*.
3. Poiché l'utente non è autenticato, viene inviato un reindirizzamento alla pagina di login.
4. Il browser richiede la pagina di login.

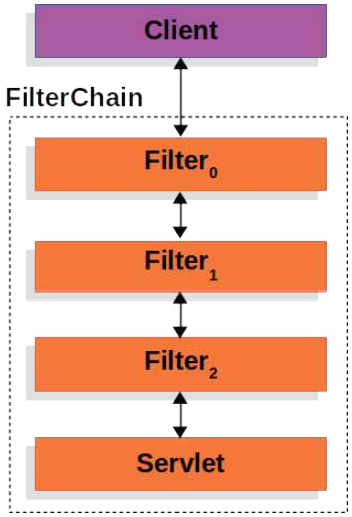

Figura 2.5: Catena di Filtri

5. Il Controller riceve la richiesta e la elabora, fornendo la pagina di login.

In Spring Security si può utilizzare la pagina di login fornita di default con il framework; in alternativa, è possibile fornire una pagina personalizzata di login, questo deve essere specificato nella configurazione HTTP della filter chain.

Una volta inviate le credenziali lo *Username Password Authentication Filter* crea un *Token*. Per estrarre username e password dall'istanza *HttpServletRequest*. È il momento dell'*Authentication Manager* che raccoglie l'istanza per essere autenticata. Se quest'ultima attività ha successo allora la *Session Authentication Strategy* è informata di un nuovo accesso. L'interfaccia *Authentication* è impostata su *Security Context Holder*. Infine, viene invocato l'*Authentication Success Handler*, che tipicamente contiene la richiesta effettuata per la quale non si era autorizzati.

2.1.3 Filtri

In Spring Security ai filtri HTTP si delegano le diverse responsabilità che si applicano ad una richiesta HTTP. In generale, i filtri HTTP gestiscono ciascuna responsabilità che deve essere applicata alla richesta. Essi formano una catena di responsabilità. Una volta che un filtro riceve una richiesta, esegue la sua logica e alla fine delega la richesta al filtro successivo nella catena.

Il supporto *Servlet* di Spring Security si basa sui filtri servlet. In Figura 2.5 è possibile avere una panoramica di alto livello dei filtri e di come sono disposti i gestori per una singola richiesta HTTP.

Quando il client invia una richiesta, il contenitore crea la catena di filtri, contenente le istanze di filtro, e la *Servlet* che dovranno elaborare la *HttpServletRequest*, in base all'URI della richiesta. In un'applicazione Spring la *Servlet* può gestire una singola *HttpServletRequet* e *HttpServletResponse*, ma è consentito l'uso di più di un filtro. Poiché un filtro influisce sulle istanze di filtro successive e sul *Servlet*, l'ordine in cui è invocato un filtro è molto importante.

Una *ServletRequest* rappresenta la richesta HTTP; si utilizza l'oggetto *ServletRequest* per recuperare i dettagli sulla richiesta. Invece, una *ServletResponse* rappresenta la risposta HTTP e viene utilizzata per modificare la risposta prima di inviarla al client o lungo la catena di filtri.

Spring fornisce un'implementazione di filtro chiamata *Delegating Filter Proxy* che consente di fare da ponte tra il contenitore *Servlet* e l'*Application Context*. Il contenitore può registrare le istanze di filtro ma non è consapevole dei bean definiti da Spring.

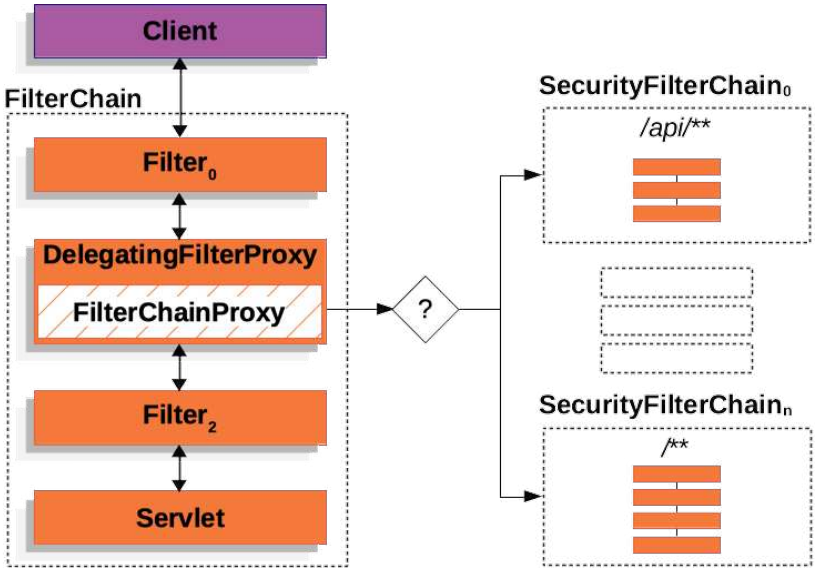

Figura 2.6: Istanze multiple di Security Filter Chain

Security Filter Chain è utilizzato da *Filter Chain Proxy*; questo è un filtro speciale fornito da Spring Security ed è incapsulato in *Delegating Filter Proxy*. Quindi *Security Filter Chain* determina quale istanza filtro di Spring Security invocare per la richiesta corrente.

I filtri di sicurezza in *Security Filter Chain* sono tipicamente Beans ma sono registrati con *Filter Chain Proxy*. Questo fornisce numerosi vantaggi. In primo luogo è un punto di partenza per il supporto *Servlet* di Spring Security. Inoltre, poiché *Filter Chain Proxy* è centrale in Spring Security, può svolgere compiti che non sono opzionali, come eliminare il *Security Context* per evitare memory leak, ed altre operazioni relative alla sicurezza. La Figura 2.6 mostra istanze multiple di *Security Filter Chain*; il *Filter Chain Proxy* decide quale istanza può essere utilizzata.

I filtri di sicurezza vengono inseriti nel *Filter Chain Proxy;* questi possono essere usati per scopi quali autenticazione, autorizzazione e protezione da exploit. Essi vengono eseguiti in un ordine specifico per garantire l'invocazione al momento giusto; ad esempio, il filtro che esegue l'autenticazione viene eseguito prima del filtro che esegue l'autorizzazione.

Quando una richiesta non ha autenticazione ed è relativa ad una risorsa che richiede l'autenticazione, è necessario salvare la richiesta per la risorsa autenticata in modo da richiederla nuovamente dopo che l'autenticazione ha avuto successo. In Spring Security ciò è fatto salvando l'*HttpServletRequet* utilizzando un'implementazione di *Request Cache*.

2.1.4 Security Context Holder

Nel cuore del modello di autenticazione di Spring Security c'è il *Security Context Holder*, che contiene il *Security Context*. In esso Spring Security salva i dettagli su chi è autenticato. Se esso contiene un valore, questo può essere utilizzato come l'utente attualmente autenticato. In Figura 2.7 viene riportato un listato su come ottenere l'utente attualmente autenticato e le principali informazioni su di esso, come username e autorità, in modo che possono essere utilizzate nell'applicazione.

Il *Security Context* contiene un oggetto di *Authentication*. Quest'ultima svolge due scopi principali in Spring Security, ovvero:

- Funge da input per *Authentication Manager*, verificando le credenziali che un utente ha fornito per l'autenticazione.

```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();

if (authentication.isAuthenticated()){
    Object principal = authentication.getPrincipal();
    if(principal instanceof UserDetails){
        UserDetails userDetails = (UserDetails) principal;
        String username = userDetails.getUsername();
        Collection<? extends GrantedAuthority> collectionautorita = userDetails.getAuthorities();

        String primaAut = collectionautorita.iterator().next().getAuthority();
        model.addAttribute( attributeName: "autorita", primaAut);

        model.addAttribute( attributeName: "nomeutente", username);

    }
}
```

Figura 2.7: Come reperire il Security Context Holder all'interno del controller

- Rappresenta l'utente attualmente autenticato.

L'oggetto *Authentication* è composto da tre campi:

1. *Principal:* identifica l'utente; spesso è un istanza di *User Details*.
2. *Credentials:* può essere una password; questa è cancellata dopo l'autentiazione.
3. *Authorities:* è un'istanza di *Granted Authority* e rappresenta le autorizzazioni di alto livello concesse all'utente. L'oggetto *Granted Autority* è una *Collection*; quindi un utente può avere più ruoli.

2.1.5 Authentication Manager

È l'*API* che definisce come i filtri di Spring Security effettuano l'autenticazione. L'oggetto *Authentication* restituito viene impostato sul *Security Context Holder* dalla istanza filtro di Spring Security che ha invocato l'*Authentication Manager*.

Pubblicare un *Authentication Manager* come *@Bean* è una cosa comune per consentire autenticazioni personalizzate. Spring Security fornisce un *Authentication Manager* di default composto da un *DaoAuthenticationProvider* per l'autenticazione con username e password.

L'implementazione più comune di un *Authentication Manager* è *Provider Manager*; questo delega l'autenticazione a una lista di istanze di *Authentication Provider* (Figura 2.8). Ogni *Authentication Provider* può indicare che l'autenticazione dovrebbe avere successo o fallire; esso può anche specificare che non può prendere una decisione e permettere all'*Authentication Provider* successivo di decidere. Se nessuna istanza di *Authentication Provider* può autenticare, l'autenticazione fallisce con un'eccezione.

Ogni *Authentication Provider* sa come effettuare un tipo specifico di autenticazione; ciò consente di supportare più tipi di autenticazione esponendo un solo bean di *Authentication Manager*. Si può, inoltre, definire qualsiasi logica di autenticazione personalizzata, definendo un *Authentication Provider*.

Si possono iniettare più istanze di *Authenticatin Provider* in *Provider Manager*; in questo caso ogni *Authentication Provider* esegue un tipo specifico di autenticazione. Ad esempio, *Dao Authentication Provider* supporta l'autenticazione con username e password, mentre *Jwt Authentication Provider* supporta l'autenticazione di un token JWT.

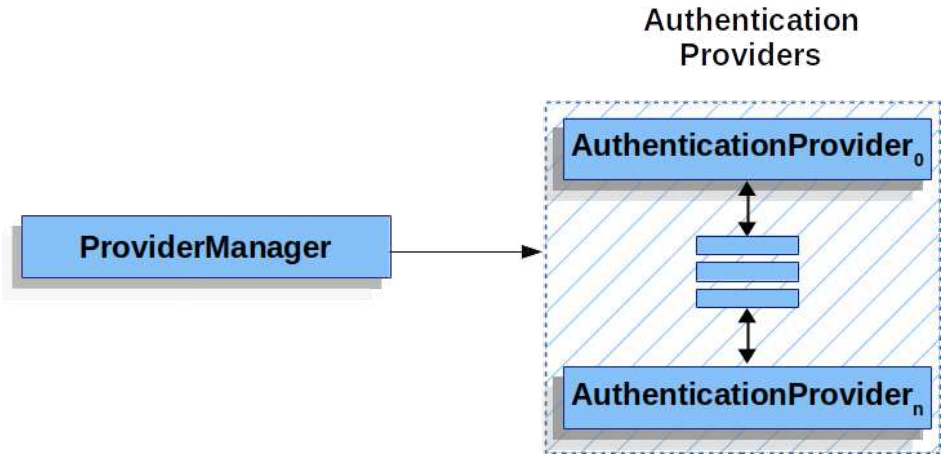

Figura 2.8: Lista di Provider Manager; ognuno di essi esegue un tipo diverso di autenticazione

Come comportamento predefinito, *Provider Manager* cerca di cancellare qualsiasi informazione sensibile sulle credenziali dell'oggetto *Authentication*, impedendo che informazioni come la password siano conservate più a lungo del necessario nella *Http Session*.

Come visto in Figura 2.1, l'*Authentication Provider* è responsabile della logica di autenticazione, ciò gli viene delegato dall'*Authentication Manager*. Il processo di autenticazione può avere solo due risultati:

1. L'utente che effettua la richiesta non è autenticato; di solito lo stato di risposta inviato al client è una HTTP 401 (non autorizzato).

2. L'utente è autenticato; in questo caso, i dettagli sul richiedente sono memorizzati in modo che l'applicazione li possa utilizzare per l'autorizzazione. *Security Context* è responsabile dei dettagli sulla richiesta autenticata corrente.

Se lo username e la password corrispondono, l'*Authentication Provider* restituisce un'istanza di *Authentication* contrassegnata come autenticata, che contiene i dettagli sulla richiesta.

Una volta completato il processo di autenticazione da parte di *Authentication Manager,* esso memorizza l'istanza di *Authentication*; questa istanza è chiamata *Security Context*.

2.1.6 Implementazione Autorità

Nella fase di autenticazione l'applicazione stabilisce e identifica chi sta cercando di accedere ad una risorsa. L'autorizzazione è il processo durante il quale il sistema decide se un client identificato ha il permesso di accedere alla risorsa richiesta.

In Spring Security, una volta che l'applicazione completa il processo di autenticazione, delega la richiesta ad un filtro di autorizzazione. Il filtro permette o rifiuta la richiesta in base alle regole configurate. Se la richiesta è autorizzata, questa è passata al *Controller* per la gestione.

La restrizione all'accesso è basata su autorizzazoni e ruoli, un concetto chiave nella sicurezza delle applicazioni. *User Details* descrive l'utente in Spring Security; esso ha una collezione di istanze di *Granted Authority*, un utente può avere uno o più privilegi. Il metodo *getAutorities* restituisce tale collezione.

Spring Security include un'implementazone di *Granted Authority*: *SimpleGrantedAuthority*. Questa consente di convertire qualsiasi stringa fornita al metodo in una *Granted Authority*. Tutte le istanze di *Authentication Provider* utilizzano *Simple Granted Authority* per popolare l'oggetto *Authentication*.

2.2 Thymeleaf

Thymeleaf è un moderno template engine Java lato server sia per ambienti web che autonomi.

L'obiettivo principale è quello di portare eleganti template nel workflow di sviluppo, creando HTML che può essere correttamente visualizzato nei browser, . I template HTML scritti in Thymeleaf hanno ancora l'aspetto e le funzionalità dell'HTML.

2.2.1 Cos'è e come integrarlo con Spring

I template engine svolgono un ruolo chiave nei framework web; essi sono uno dei loro componenti più importanti poiché responsabili della produzione dell'interfaccia utente.

Thymeleaf è in grado di elaborare diversi tipi di template, ovvero:

- *HTML* (incluso HTML 5, XHTML, HTML 4);
- *XML*;
- *TEXT* (testo normale);
- *JAVASCRIPT* (file .js);
- *CSS* (file .css).

Essendo il template engine estremamente estensibile, esso consente di definire e personalizzare in maniera dettagliatata il modo in cui i template vengono elaborati.

Un oggetto che applica una certa logica ad un artefatto di markup è chiamato processore, e un insieme di questi processori costituisce normalmente un dialect. Di default la libreria di Thymeleaf fornisce un dialetto chiamato *Standard Dialect*, che dovrebbe essere sufficiente per lo sviluppo della maggior parte dei progetti.

Includere Thymeleaf in una pagina HTML5 si rileva particolarmente semplice; basta aggiungere l'attributo *xmlns:th* nell'intestazione del nostro file *html*. Questa aggiunta, che dovrebbe rendere il documento prodotto non valido dal punto di vista HTML 5, in realtà non ha alcuna influenza nell'elaborazione del template perché i browser ignorano tutti gli attributi che non comprendono.

2.2.2 Internazionalizzazione

L'internazionalizzazione è fondamentale per poter raggiungere un pubblico globale, consentendo la personalizzazione del contenuto in base alla lingua. Ciò migliora l'accessibilità e l'esperienza dell'utente nella fruizione del contenuto di una pagina web.

Questo processo implica l'esternalizzazione del testo, ovvero estrarre frammenti di codice dai file di template per consentirne la conservazione in file separati. Questi file possono essere facilmente sostituiti con testi equivalenti scritti in diverse lingue. Questo processo è chiamato internazionalizzazione o *i18n*. I frammenti di testo esternalizzati sono noti come *messaggi*.

Thymeleaf semplifica questo processo attraverso la definizione di *messaggi* tramite la sintassi #..., consentendo un'efficiente pratica di internazionalizzazione. I messaggi sono identificati da una chiave e questa chiave deve essere specificata all'interno della sintassi del *messaggio*, in modo da poter sostituire la porzione di testo corrispondente con un frammento di testo esterno.

La posizione del testo esternalizzato è completamente configurabile, normalmente è utilizzata un'implementazione basata su file *.properties*, ma è possibile personalizzare quest'ultima utilizzando dati da un database. Il risolutore di messaggi predefinito si aspetta di trovare i messaggi in file nella stessa cartella e con lo stesso nome del template.

```
<form action="#" th:object="${nuovafaq}" th:action="@{/admin/salvaaddfaq}" method="post">



    <p>Setta Domanda:</p>
    <textarea type="text" name="domanda" value="" id="domanda" placeholder="aggiungi domanda"
              required="required" autofocus="autofocus" th:value="*{domanda}"></textarea>
    <span th:if="${#fields.hasErrors('domanda')}" th:errors="*{domanda}" style="...">Minimo 3 caratteri, massimo 20</span>

    <p>Setta Risposta:</p>
    <textarea type="text" name="risposta" value="" id="risposta" placeholder="aggiungi risposta"
           required="required" autofocus="autofocus" th:value="*{risposta}"></textarea>
    <span th:if="${#fields.hasErrors('risposta')}" th:errors="*{risposta}" style="...">La mail deve esser formattata bene</span>

    <div class="pulsanti-container">
        <button type="submit" class="pulsanteformok" name="addRow">Salva</button>
        <a th:href="@{/admin/faq}" class="bottoneform-indietro">Torna indietro</a>
    </div>



</form>
```

Figura 2.9: Espressione sulle selezioni

2.2.3 Sintassi e iterazione

In questa sezione di analizza lo *Standard Dialect* di Thymeleaf, chiamato *Thymeleaf Standard Expression syntax*. Oltre ai *messaggi* descritti nella sezione precedente, possono essere utilizzate tutte le espressioni OGNL (Object-Graph Navigation Language), come, ad esempio, le variabili tramite *$...*. Nelle applicazioni per Spring MVC, OGNL verrà sostituito con SpringEL, la cui sintassi è molto simile a quella di OGNL.

Vi è la possibilità di utilizzare espressioni sulle selezioni; le variabili possono essere scritte non solo con *$...* ma anche con **...*. Tuttavia c'è una differenza importante: la sintassi con l'asterisco valuta espressioni sugli oggetti selezionati invece che valutare l'intero contesto (Figura 2.9). L'oggetto selezionato è il risultato di un'espressione che utilizza l'attributo *th:object*.

Gli URL sono considerati elementi fondamentali nei modelli delle applicazioni web, determinando la navigazione tra le diverse pagine. Thymeleaf adotta una sintassi speciale per gli URL, ovvero *@...*. Ci sono essenzialmente due tipi di URL utilizzabili in Thymeelaf, ovvero:

- *URL assoluti*;
- *URL relativi:* essi sono preceduti da / e verranno automaticamente preceduti dal nome del contesto dell'applicazione.

L'elaborazione di queste espressioni e la conversione in URL è effettuata dall'interfaccia *ILinkBuilder*. Per utilizzare i link con Thymeleaf è necessario utilizzare l'attributo *th:href*. Esso è un attributo modificatore, una volta elaborato calcolerà l'URL del collegamento da utilizzare e imposterà tale valore come attributo href del tag <a>.

Un altro elemento di grande importanza sono gli iteratori; essi permettono di iterare gli elementi di una collezione. Lo Standard Dialect offre un attributo per questo compito denominato *th:each*.

Quindi il nostro Controller recupera l'oggetto da iterare attraverso il livello di servizio e lo aggiunge al model. Con questa operazione possiamo facilmente accedere nella nostra pagina Thymeleaf all'oggetto (Figura 2.9).

2.2.4 Template Layout

Thymeleaf consente l'inclusione di frammenti di template; questi possono essere header, menù, piè di pagina ed altro. L'utilizzo di frammenti consente di separare diverse parti della pagina, organizzandola in moduli riutilizzabili in modo da facilitare la manutenzione e da migliorare la lettura del codice.

I frammenti possono includere qualsiasi attributo *th:**. Questi ultimi saranno valutati una volta che il frammento è incluso nel template di destinazione; essi potranno fare riferimento a qualsiasi variabile di contesto definita nel template di destinazione.

Un grande vantaggio derivante nell'approccio ai frammenti è la possibilità di scriverli in pagine che sono perfettamente visualizzabili da un browser, con una struttura di markup completa e valida, pur conservando la capacità di essere inclusi da Thymeleaf in altri template.

2.3 Database

Un database è una collezione di dati immagazzinata e accessibile per via elettronica. La progettazione di database si basa su tecniche formali e considerazioni pratiche, come la modellazione dei dati, la rappresentazione efficiente di essi, il linguaggio di interrogazione, la sicurezza e la privacy.

Un sistema di gestione noto come DBMS (Database Management System) è il software che interagisce con gli utenti finali, le applicazioni e il database. Questo sistema di gestione comprende anche funzionalità di base necessarie all'amministrazione del database.

È possibile classificare i sistemi di gestione in termini del modello di database che essi supportano. I database relazionali sono diventati il modello predominante; essi utilizzano righe e colonne in una serie di tabelle.

2.3.1 Database relazionali

La struttura relazionale favorisce la coerenza e l'integrità dei dati attraverso vincoli, garantendo che le relazioni tra le tabelle siano ben definite. I database relazionali utilizzano linguaggi di interrogazione come SQL (Structured Query Language) per eseguire operazioni sulla base di dati. Esistono database non relazionali, come i database NoSQL, che adottano linguaggi diversi dall'SQL.

Le transazioni sono un aspetto fondamentale nei database relazionali per garantire la consistenza dei dati. Esse forniscono l'esecuzione di più operazioni in maniera atomica, assicurando che o tutte le operazioni vengano completate con successo o che il database torni allo stato precedente.

2.3.2 MariaDB e diagramma del Database

MariaDB è un sistema di gestione di database relazionale (RDBMS) open source nato nel 2009 come fork di MySQL. Esso offre un'alternativa compatibile con MySQL, cerca di introdurre miglioramenti, nuove funzionalità e ottimizzazioni delle prestazioni. Con una crescente base di utenti e sviluppatori continua ad evolversi, mantenendo un equilibrio tra compatibilità e innovazione.

La scelta di MariaDB è stata guidata da diversi fattori chiave. In primo luogo, esso offre una compatibilità diretta con MySQL. Questa interoperabilità consente di capitalizzare sulle conoscenze esistenti e sfruttare le risorse della vasta community che si è sviluppata attorno a MySQL.

La natura open source è un altro elemento chiave nella nostra decisione. La community attiva e impegnata offre supporto, soluzioni e aggiornamenti regolari.

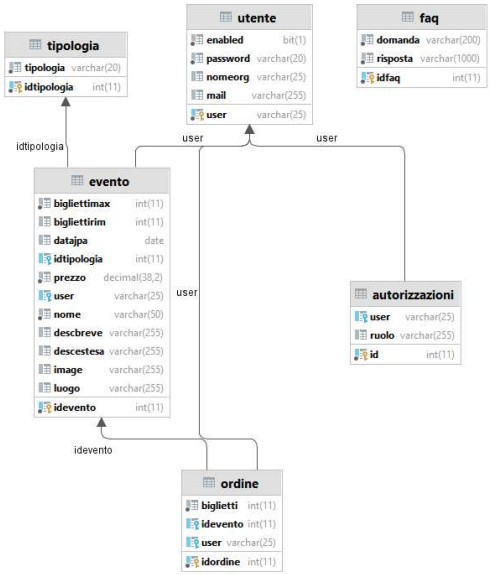

Figura 2.10: Struttura del database utilizzata

Per avere una visione più approfondita del nostro schema di database e delle relazioni tra entità, esploreremo il diagramma del database utilizzato nel nostro progetto. Questo ci darà una panoramica visiva della struttura dati, facilitando la comprensione del sistema (Figura 2.10).

CAPITOLO 3

Analisi dei Requisiti e diagramma dei casi d'uso

Nel contesto dello sviluppo di un qualsiasi progetto, la definizione dei requisiti gioca un ruolo chiave nel determinare il successo e la soddisfazione degli utenti finali. Questa è un'attività preliminare allo sviluppo di un sistema software con lo scopo di definire le funzionalità che il sistema deve avere.

3.1 Premessa

La realizzazione di un sito web dedicato alla pubblicazione e all'acquisto di eventi richiede un'attenta analisi dei requisiti per garantire un'esperienza utente intuitiva e soddisfacente. La fase di analisi comincia con la comprensione delle esigenze degli utenti, identificando le funzionalità chiave e definendo i requisiti tecnici che saranno alla base del sistema.

Le funzionalità di pubblicazione degli eventi devono consentire agli organizzatori di caricare dettagli completi, incluse date, luoghi e descrizioni. D'altra parte, gli acquirenti devono poter navigare agevolmente tra gli eventi, cercare in base a criteri specifici ed effettuare acquisti in modo efficiente.

3.2 Requisiti Progetto

Prima di svolgere la fase di progettazione, è necessario stabilire cosa il sistema dovrebbe fare. L'obiettivo è creare delle specifiche di alto livello del sistema, coinvolgendo le varie parti interessate (stakeholder).

L'ingegneria dei requisiti è la disciplina incaricata della raccolta dei requisiti provenienti dalle diverse parti interessate per il sistema. Di solito è un processo di negoziazione che tiene conto delle diverse esigenze degli stakeholder.

Un requisito è una specifica che dovrebbe essere implementata, esso definisce ciò che il sistema dovrebbe fare, non come dovrebbe farlo. Per formulare i requisiti abbiamo adottato un formato semplice:

- *Id:* un identificatore univoco per ciascun requisito;
- *Funzione:* una breve dichiarazione che esplicita l'obiettivo che il requisito mira a soddisfare nell'applicazione.

3.2.1 Requisiti Funzionali

In questa sezione vengono delineati i requisiti funzionali; ognuno di essi rappresenta una funzione svolta dall'applicazione. Inizialmente si individuano macro categorie, espresse come funzionalità relative ad ogni possibile utilizzatore finale; successivamente ogni categoria è espansa con una descrizione. In Figura 3.1 sono rappresentati i requisiti funzionali; di seguito invece le macro categorie:

- *Pubblicazione di eventi:* rappresenta ciò che un visitatore del sito web non autenticato può fare all'interno della sezione pubblica.
- *Gestione utenti:* rappresenta le attività che possono essere condotte da un utente autenticato; egli può procedere all'acquisto di biglietti e visualizzare lo storico dei suoi ordini.
- *Gestione organizzatori:* rappresenta ciò che può essere fatto da un utente collegato con autorizzazione organizzatore; egli ha una serie di privilegi relativi alla pubblicazione e gestione degli eventi.
- *Gestione amministrazione:* rappresenta le attività che possono essere effettuate da un utente che ha i privilegi di amministratore; in particolare, egli ha accesso a tutte le funzionalità dell'applicazione.

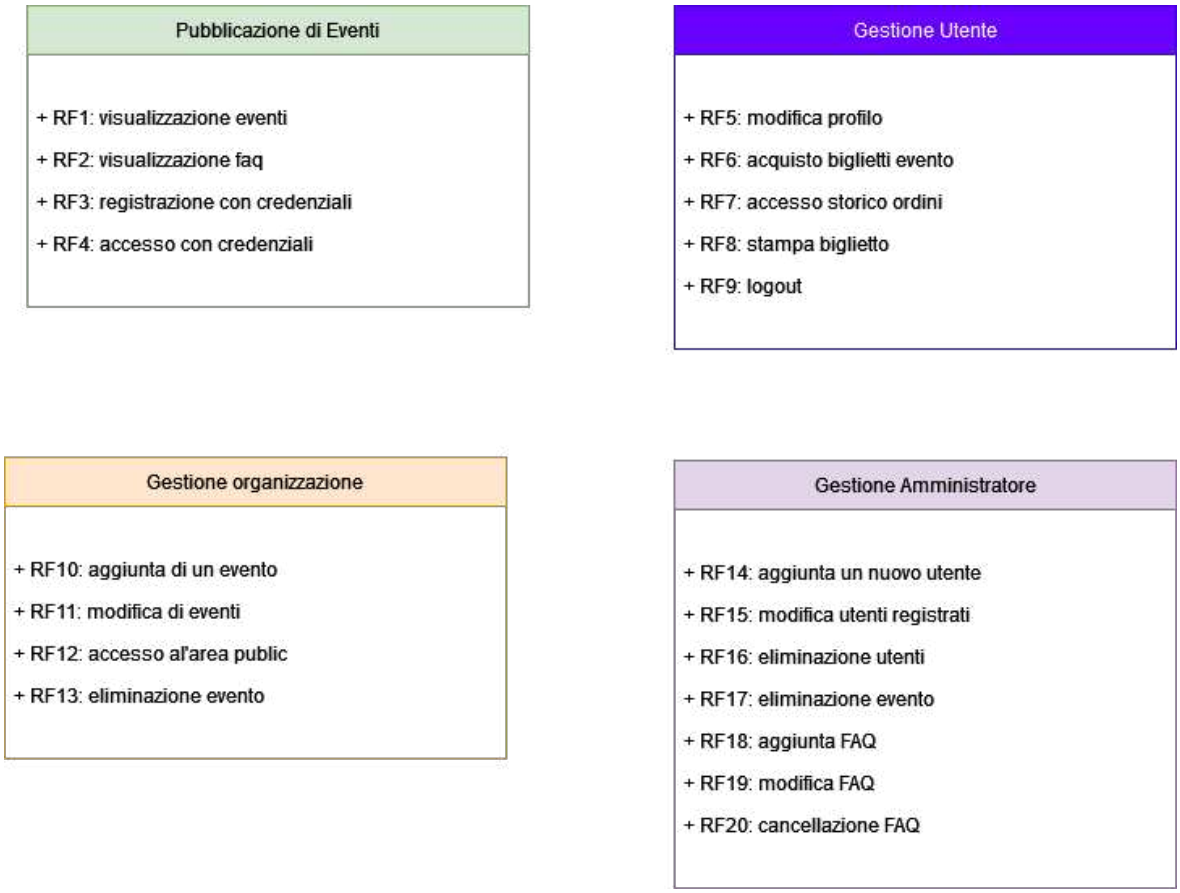

Figura 3.1: Requisiti Funzionali del sistema

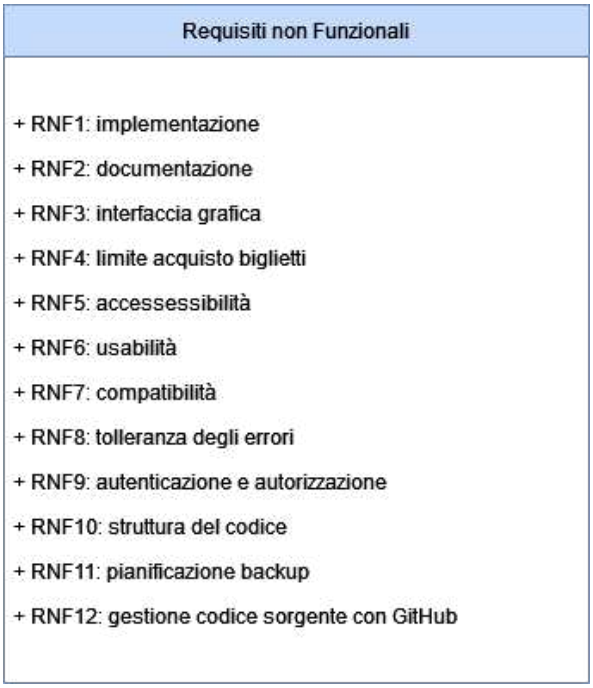

Figura 3.2: Requisiti non Funzionali del sistema

Descrizione dei requisiti funzionali

I requisiti funzionali sono descritti nella relativa Tabella 3.1; in essa ad ogni requisito è associata una descrizione per comprenderne la funzione.

3.2.2 Requisiti non Funzionali

I requisiti non funzionali riguardano aspetti relativi alla qualità del sistema, invece che a funzionalità specifiche. Questi requisiti sono cruciali per garantire un'esperieza utente ottimale e prestazioni efficienti. I requisiti non funzionali sono elencati in Figura 3.2 .

Descrizione dei requisiti non funzionali

Nella Tabella 3.2 sono delineati i requisiti non funzionali, ognuno accompagnato da una descrizione dettagliata al fine di chiarirne la natura e il contesto operativo. Tali requisiti sono accuratamente definiti, in modo da fornire una comprensione approfondita delle loro funzioni e del loro impatto.

3.3 Casi d'uso

La modellazione dei casi d'uso è una tecnica diversa per raccogliere e documentare i requisiti. Questa metodologia comprende i seguenti passaggi:

1. *Individuare il confine del sistema:* cominciamo con una valutazione iniziale per definire chiaramente il confine del sistema, delimitando, così, l'ambito dell'attività di modellazione. La precisione nel definire il confine è cruciale per evitare problemi.

2. *Individuare gli attori:* identifichiamo gli attori, essi rappresentano entità esterne al sistema che interagiscono direttamente con esso. Agli attori sono associati dei ruoli che tali entità assumono durante le iterazioni con il sistema.

Requisito	**Descrizione**
RF1: visualizzazione eventi	L'applicazione consente l'accesso alla home page a tutti gli utenti.
RF2: visualizzazione faq	L'applicazione consente di consultare a tutti le faq in caso di bisogno.
RF3: registrazione con credenziali	La registrazione come utente normale è possibile a chiunque voglia registrarsi.
RF4: accesso con credenziali	La pagina di login è permessa in ogni parte del sito web.
RF5: modifica profilo	L'applicazione permette la modifica di alcuni dati utente.
RF6: acquisto biglietti evento	L'applicazine permette l'acquisto di biglietti per gli utenti registrati.
RF7: accesso strorico ordini	Un utente registrato può accedere allo storico dei propri ordini.
RF8: stampa biglietto	Una volta effettuato l'acquisto è possibile stampare il relativo biglietto.
RF9: logout	Ogni utente connesso può fare logout dal proprio profilo.
RF10: aggiunta di un evento	Un organizzatore può pubblicare un proprio evento.
RF11: modifica di un evento	Un organizzatore può modifiacare un proprio evento.
RF12: accesso all'area public	Un utente registrato può accedere alla home page per avere una panoramica di tutti gli eventi
RF13: eliminazione evento	Un organizzatore può cancellare un proprio evento.
RF14: aggiunta nuovo utente	All'amministratore è concesso di registrare vari tipi di utenti.
RF15: modifica utente registrato	All'amministratore è concesso di modificare gli utenti.
RF16: eliminazione utenti	All'amministratore è concesso di eliminare utenti.
RF17: eliminazione evento	L'amministatore può canellare un evento qualsiasi.
RF18: aggiunta faq	L'amministatore può aggiungere faq.
RF19: modifica faq	L'amministatore può modificare faq.
RF20: cancellazione faq	L'amministratore può cancellare faq.

Tabella 3.1: Tabella di descrizione dei requisiti funzionali

Requisito	**Descrizione**
RNF1: implementazione	Lo sviluppo dell'applicazione dovrà essere basato sul framework Spring.
RNF2: documentazione	Verrà fornito un manuale utente per l'utilizzo della web app.
RNF3: interfaccia grafica	L'applicazione sarà fruibile attraverso browser web.
RNF4: limite acquisto biglietti	L'applicazione sarà strutturata per controllare che nella fase di acquisto ci siano biglietti disponibili.
RNF5: accettabilità	L'applicazione dovrà essere accettabile per gli utenti per i quali è progettata.
RNF6: usabilità	Progettare un'interfaccia utente chiara e intuitiva per agevolare la navigazione.
RNF7: compatibilità	Assicurarsi che l'applicazione sia compatibile con una vasta gamma di browser e dispositivi.
RNF8: tolleranza degli errori	Gestire in modo robusto eventuali errori o fallimenti senza compromettere l'integrità del sistema.
RNF9: autenticazione e autorizzazione	Implementare un sistema sicuro di autenticazione e autorizzazione per proteggere le informazioni sensibili.
RNF10: struttura del codice	Mantenere una struttura del codice pulita e documentata per agevolare manutenzioni future.
RF11: pianificazione backup	Implementare un sistema di backup regolare per proteggere i dati e garatire un rapido ripristino.
RF12: gestione codice sorgente con GitHub	L'intero codice sorgente deve essere gestito utilizzando GitHub.

Tabella 3.2: Tabella di descrizione dei requisiti non funzionali

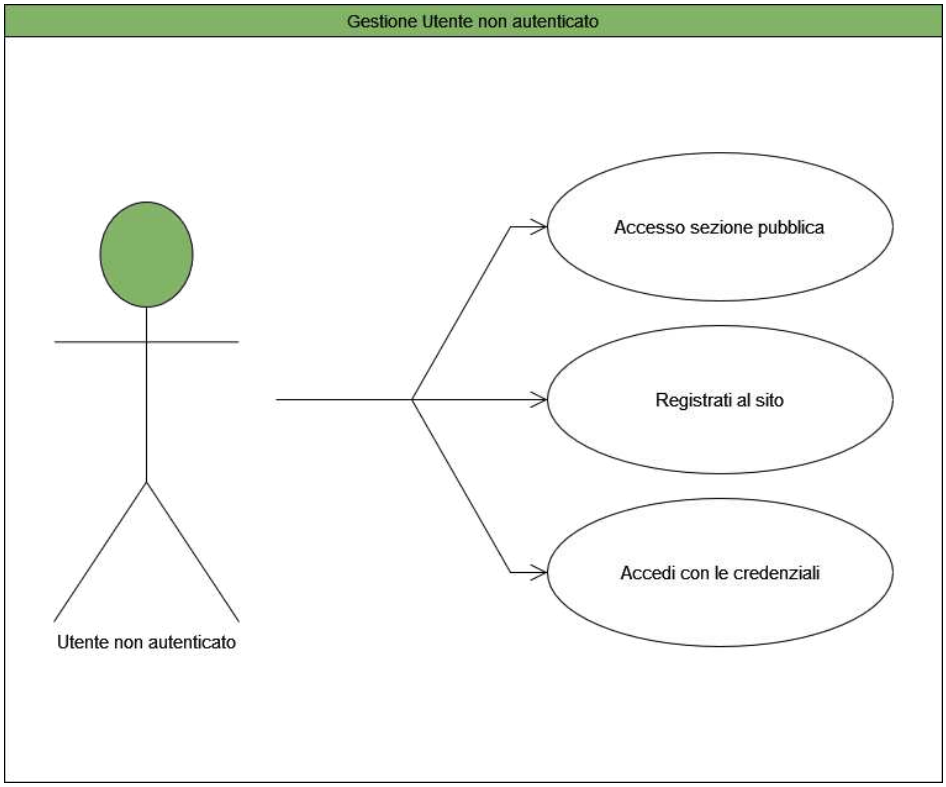

Figura 3.3: Diagramma dei casi d'uso per il sottosistema relativo all'utente non autenticato

3. *Individuare i casi d'uso:* passiamo alla fase di specifica dei casi d'uso che rappresentano ciò che un attore desidera che il sistema faccia. I casi d'uso son sempre avviati da un attore e vengono descritti dal loro punto di vista.

Gli attori che interagiscono con la web app TicketLove sono:

- *Utente non autenticato*: rappresenta l'utente appena connesso al sito o un utene che non ha effettuato il login;
- *Utente autenticato*: rappresenta un utente che ha effettuato il login con autorizzazione user;
- *Organizzazione*: rappresenta un utente che ha effettuato il login con autorizzazione org;
- *Amministratore*: è un solo utente che ha autorizzazione admin.

3.3.1 Caso d'uso: utente non autenticato

Questo caso si verifica quando un utente accede al sistema per la prima volta, quindi non è autenticato, oppure quando un utente non è registrato ed è un visitatore nella sezione public del sito (Figura 3.3).

Caso d'uso: accesso sezione pubblica

- *Descrizione*: questo caso si verifica quando un utente accede al sito come utente non registrato o ancora non effettua l'autenticazione.
- *Pre-condizioni*: nessuna.

- *Post-condizioni*: l'utente è collegato alla home page nella sezione public.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando un utente non autenticato accede al sito;
  - l'applicazione restituisce la home page del sito;
  - l'utente collegato può navigare tra le varie pagine della sezione public.

Caso d'uso: registrazione al sito

- *Descrizione*: questo caso d'uso si verifica quando un utente intende registrarsi al sito e accede alla sezione sign up nella barra di navigazione.
- *Pre-condizioni*: nessuna.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando un utente non autenticato intende registrarsi al sito;
  - l'utente accede alla sezione sign up;
  - il sistema fornisce il modulo di registrazione;
  - l'utente inserisce nel modulo il nome utente e la password;
  - il sistema registra l'utente con lo username e password fornite e reindirizza alla home page;
  - il sistema restituisce un messaggio di successo dell'operazione.

Caso d'uso: accesso con le credenziali

- *Descrizione*: questo caso si verifica quando un utente che ha le credenziali di accesso vuole effettuare il login.
- *Pre-condizioni*: l'utente è registrato al sistema.
- *Post-condizioni*: il sistema reindirizza l'utente alla sezione dedicata al proprio livello di autorizzazione.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando un utente vuole accedere con le credenziali;
  - l'utente accede alla sezione login;
  - il sistema fornisce il modulo di login;
  - l'utente inserisce nel modulo le credeziali di accesso;
  - il sistema analizza le credenziali e, se queste sono corrette, effettua un reindirizzamento in altra sezione, altrimenti itera il processo di validazione delle credenziali.

3.3.2 Caso d'uso: utente autenticato

Questo caso d'uso si verifica quando un utente effettua il login ed essendo un autorizzazione di tipo user viene reindirizzato alla sezione user del sistema. Da questa sezione ha accesso ad una serie di privilegi relativi al ruolo *user* (Figura 3.4).

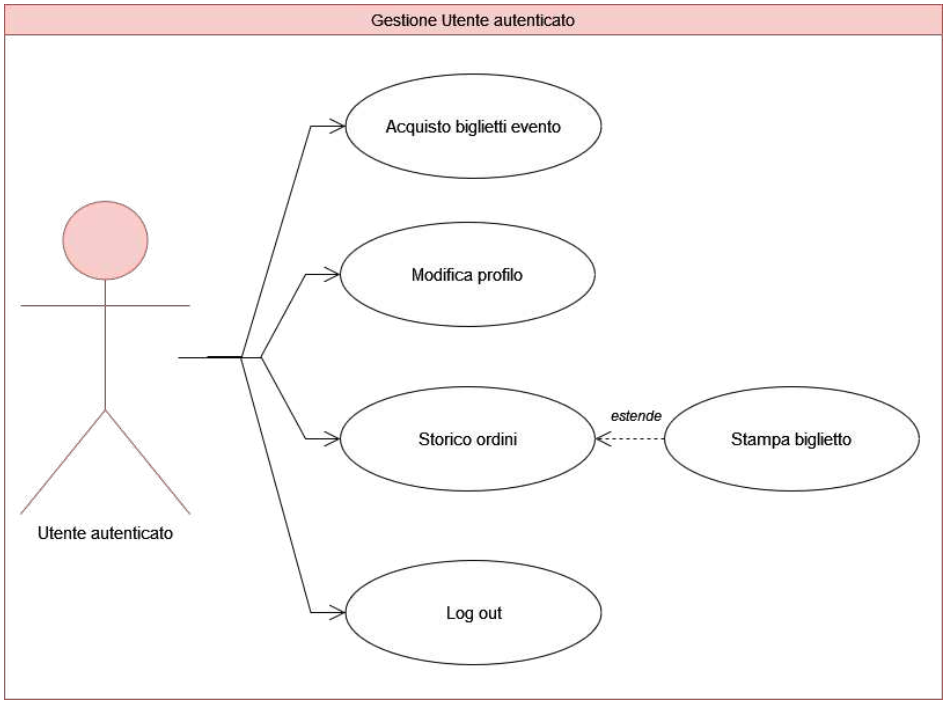

Figura 3.4: Diagramma dei casi d'uso per il sottosistema relativo all'utente autenticato

Caso d'uso: acquisto biglietti evento

- *Descrizione*: questo caso si verifica quando un utente vuole acquistare uno o più biglietti per un evento.
- *Pre-condizioni*: l'utente è registrato nel sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando un utente vuole acquistare un biglietto;
  - l'utente avvia la procedura di acquisto per un dato evento;
  - il sistema fornisce il modulo di acquisto;
  - l'utente seleziona il numero di biglietti da acquistare;
  - l'utente invia il modulo di acquisto;
  - il sistema informa l'utente dell'avvenuto acquisto.

Caso d'uso: modifica profilo

- *Descrizione*: questo caso si verifica quando un utente vuole modificare i propri dati personali.
- *Pre-condizioni*: l'utente è registrato nel sistema.
- *Post-condizioni*: i dati vengono modificati se rispettano i vincoli del sistema, altrimenti restano invariati.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando un utente vuole modificare i propri dati personali;

  - l'utente avvia la procedura di modifica dei dati;
  - il sistema fornisce il modulo di modifica dei dati personali;
  - l'utente può modificare la propria password rispettando il vincolo di tre caratteri minimi e venti massimi;
  - l'utente procede a modificare i propri dati;
  - il sistema controlla che siano rispettati i vincoli;
  - il sistema aggiorna la password;
  - il sistema informa l'utente delle modifiche avvenute con successo.

Caso d'uso: storico ordini

- *Descrizione*: questo caso si verifica quando un utente vuole accedere allo storico dei suoi ordini.
- *Pre-condizioni*: l'utente è registrato nel sistema, l'ordine esiste a sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando un utente vuole visualizzare lo storico dei suoi ordini;
  - l'utente accede alla sezione relativa allo storico degli ordini;
  - il sistema reperisce gli ordini effettuati dall'utente;
  - il sistema restituisce gli ordini effettuati dall'utente.

Caso d'uso: stampa biglietto

- *Descrizione*: questo caso si verifica quando un utente vuole stampare il pdf del biglietto acquistato.
- *Pre-condizioni*: l'utente è registrato nel sistema, l'ordine esiste nel sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando un utente vuole stampare il pdf del biglietto acquistato;
  - l'utente avvia la procedura di download;
  - il sistema avvia la procedura di download sul dispositivo dell'utente.

Caso d'uso: log out

- *Descrizione*: questo caso si verifica quando un utente vuole effettuare la disconnessione.
- *Pre-condizioni*: l'utente è registrato nel sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando un utente vuole disconnettersi;
  - l'utente avvia la procedura di log out;
  - il sistema disconnette l'utente;
  - il sistema ritorna alla sezione pubblica del sito.

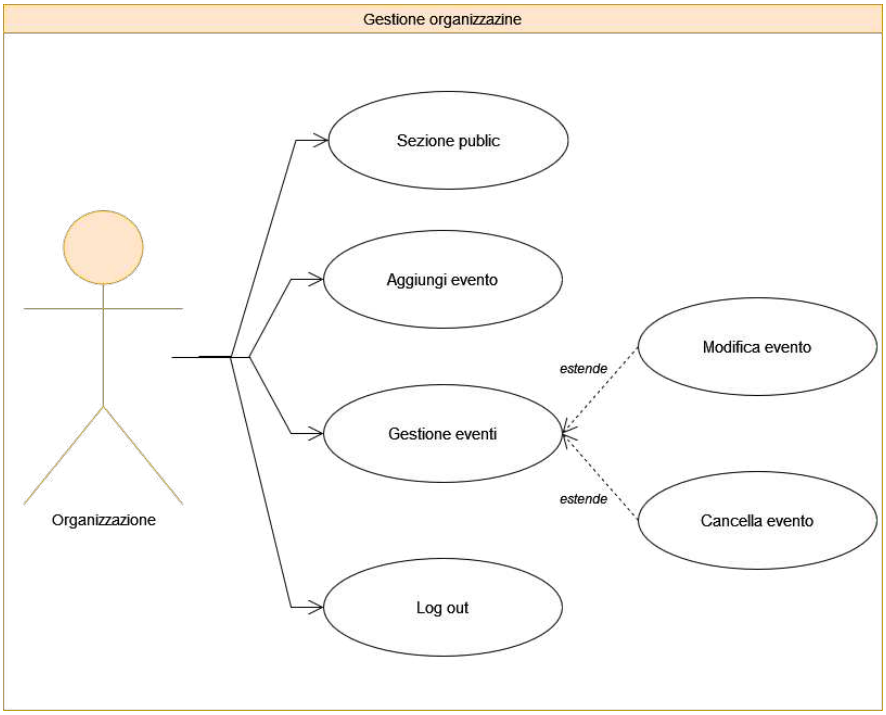

Figura 3.5: Diagramma dei casi d'uso per il sottosistema relativo all'organizzatore

3.3.3 Caso d'uso: organizzazione

Questo caso d'uso si verifica quando un utente si autentica con autorizzazione di tipo *organizzatore*; in questo caso, egli viene reindirizzato alla sezione *organizzazione* del sistema dove dispone di diversi privilegi (Figura 3.5).

Caso d'uso: accesso sezione pubblica

- *Descrizione*: questo caso si verifica quando l'utente vuole accedere alla sezione public.
- *Pre-condizioni*: nessuna.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'utente vuole accedere alla sezione public dalla sezione organizzatore;
  - l'utente accede alla sezione pubblica;
  - il sistema fornisce la sezione pubblica mantenendo i privilegi dell'organizzatore.

Caso d'uso: aggiungi evento

- *Descrizione*: questo caso si verifica quando l'utente vuole aggiungere un evento.
- *Pre-condizioni*: l'utente è registrato nel sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:

  - il caso d'uso inizia quando l'utente vuole aggiungere un nuovo evento;
  - l'utente avvia la procedura di aggiunta di un nuovo evento;
  - il sistema fornisce il modulo di aggiunta di un nuovo evento;
  - l'utente compila il modulo con i campi per il nuovo evento;
  - l'utente procede ad aggiungere l'evento;
  - il sistema effettua la validazione dei vari campi;
  - il sistema procede ad aggiungere il nuovo evento se i vincoli sono rispettati;
  - il sistema informa l'utente del successo dell'operazione.

Caso d'uso: gestione eventi

- *Descrizione*: questo caso si verifica quando l'utente accede alla lista dei suoi eventi pubblicati per visualizzarli, modificarli o gestirli.
- *Pre-condizioni*: l'utente è registrato nel sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'utente vuole accedere alla sezione relativa alla gestione degli eventi;
  - l'utente accede alla sezione di gestione degli eventi;
  - il sistema reperisce e fornisce la lista degli eventi pubblicati dall'utente.

Caso d'uso: modifica evento

- *Descrizione*: questo caso si verifica quando l'utente vuole modificare un evento.
- *Pre-condizioni*: l'utente è registrato nel sistema e l'evento è presente nel sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'utente vuole modificare un evento;
  - l'utente avvia la procedura di modifica per l'evento scelto;
  - il sistema fornisce la pagina di modifica di un evento;
  - l'utente compila il modulo con i campi per modificare l'evento;
  - l'utente procede a modificare l'evento;
  - il sistema effettua la validazione dei vari campi;
  - il sistema procede con la modifica se i vincoli sono rispettati;
  - il sistema informa l'utente del successo dell'operazione.

Caso d'uso: cancellazione evento

- *Descrizione*: questo caso si verifica quando l'utente vuole cancellare un evento.
- *Pre-condizioni*: l'utente è registrato nel sistema, l'evento è presente nel sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'utente vuole cancellare un proprio evento;
  - l'utente avvia la procedura di cancellazione per l'evento scelto;
  - il sistema avvia la procedura di cancellazione dell'evento;
  - l'utente procede con la cancellazione dell'evento;
  - il sistema procede ad eliminare l'evento;
  - il sistema informa l'utente del successo dell'operazione.

Caso d'uso: log out

- *Descrizione*: questo caso si verifica quando un utente vuole effettuare la disconnessione.
- *Pre-condizioni*: l'utente è registrato nel sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando un utente vuole disconnettersi;
  - l'utente avvia la procedura di log out;
  - il sistema disconnette l'utente;
  - il sistema ritorna alla sezione pubblica del sito.

3.3.4 Caso d'uso: amministratore

Questo caso d'uso si verifica quando un utente si autentica con autorizzazione di tipo *amministratore*; in questo caso egli viene reindirizzato alla sezione *amministratore* del sistema dove dispone di diversi privilegi (Figura 3.6).

Caso d'uso: accesso sezione pubblica

- *Descrizione*: questo caso si verifica quando l'amministratore vuole accedere alla sezione public.
- *Pre-condizioni*: l'utente è connesso al sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'amministratore vuole accedere alla sezione public ed ha un'autorizzazione come amministratore;
  - l'utente accede alla sezione pubblica;
  - il sistema fornisce la sezione pubblica mantendo i privilegi dell'amministratore.

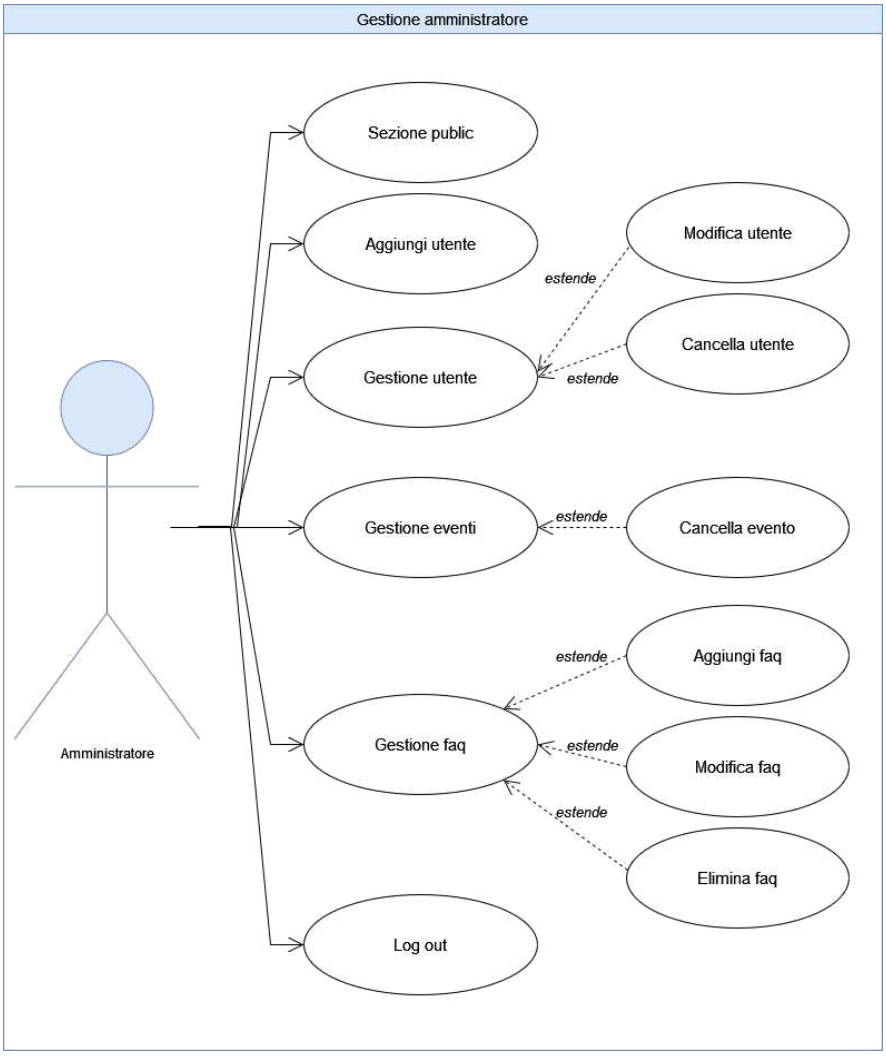

Figura 3.6: Diagramma dei caso d'uso per il sottosistema relativo all'amministratore

Caso d'uso: aggiungi utente

- *Descrizione*: questo caso si verifica quando l'amministratore vuole aggiungere un utente nel sistema.
- *Pre-condizioni*: l'utente è l'amministratore.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'amministratore vuole aggiungere un nuovo utente;
  - l'amministratore avvia la procedura di aggiunta di un nuovo utente;
  - il sistema fornisce il modulo di aggiunta di un nuovo utente;
  - l'amministratore compila il modulo per aggiungere l'utente;
  - l'amministratore procede ad inviare il modulo di aggiunta;
  - il sistema effettua la validazione dei vari campi;
  - il sistema procede ad aggiungere il nuovo utente se è rispettato il vincolo di unicità dello username altrimenti annulla l'operazione;
  - il sistema informa l'amministratore del successo dell'operazione.

Caso d'uso: gestione utente

- *Descrizione*: questo caso si verifica quando l'amministratore vuole visionare la lista di tutti gli utenti registrati nel sistema.
- *Pre-condizioni*: l'utente è l'amministratore.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'utente vuole accedere alla sezione di gestione di un utente;
  - l'amministratore accede alla sezione di gestione di un utente;
  - il sistema fornisce la lista di tutti gli utenti registrati nel sistema.

Caso d'uso: modifica utente

- *Descrizione*: questo caso si verifica quando l'amministratore vuole modificare un utente.
- *Pre-condizioni*: l'utente è registrato nel sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'amministratore vuole modificare un utente esistente;
  - l'amministratore avvia la procedura di modifica;
  - il sistema fornisce la pagina di modifica di un utente;
  - l'amministratore compila il modulo con i campi per modificare l'utente;
  - l'amministratore procede a modificare l'utente;
  - il sistema effettua la validazione dei vari campi;
  - il sistema procede con la modifica;
  - il sistema informa l'amministratore del successo dell'operazione.

Caso d'uso: cancellazione utente

- *Descrizione*: questo caso si verifica quando l'amministratore vuole cancellare un utente.
- *Pre-condizioni*: l'utente è registrato nel sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'amministratore vuole cancellare un utente;
  - l'amministratore avvia la procedura di cancellazione per l'utente;
  - il sistema avvia la procedura di cancellazione dell'utente;
  - l'amministratore procede a cancellare l'utente;
  - il sistema procede ad eliminare l'utente se è rispettato il vincolo di nessun evento pubblicato per l'utente;
  - il sistema informa l'amministratore del successo dell'operazione.

Caso d'uso: gestione eventi

- *Descrizione*: questo caso si verifica quando l'amministratore vuole visionare la lista di tutti gli eventi registrati nel sistema.
- *Pre-condizioni*: l'utente è l'amministratore.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'amministratore vuole accedere alla sezione di gestione degli eventi;
  - l'amministratore accede alla sezione gestione degli eventi;
  - il sistema fornisce la lista di tutti gli eventi registrati nel sistema.

Caso d'uso: cancellazione evento

- *Descrizione*: questo caso si verifica quando l'amministratore vuole cancellare un evento.
- *Pre-condizioni*: l'evento è registrato nel sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'amministratore vuole cancellare un evento;
  - l'amministratore avvia la procedura di cancellazione dell'evento;
  - il sistema avvia la procedura di cancellazione dell'evento;
  - l'amministratore procede a cancellare l'evento;
  - il sistema procede ad eliminare l'evento;
  - il sistema informa l'amministratore del successo dell'operazione.

Caso d'uso: gestione faq

- *Descrizione*: questo caso si verifica quando l'amministratore vuole accedere alla sezione di gestione delle faq registrate nel sistema.
- *Pre-condizioni*: l'utente è l'amministratore.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'amministratore vuole accedere alla sezione di gestione delle faq.
  - l'amministratore accede alla sezione di gestione delle faq;
  - il sistema fornisce la lista di tutte le faq registrate nel sistema.

Caso d'uso: aggiungi faq

- *Descrizione*: questo caso d'uso si verifica quando l'amministratore vuole aggiungere una nuova faq nel sistema.
- *Pre-condizioni*: l'utente è l'amministratore.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'amministratore vuole aggiungere una faq;
  - l'amministratore avvia la procedura di aggiunta di una nuova faq;
  - il sistema fornisce il modulo di aggiunta di una faq;
  - l'amministratore compila il modulo per aggiungere una faq;
  - l'amministratore procede ad inviare il modulo di aggiunta;
  - il sistema effettua la validazione dei vari campi;
  - il sistema procede ad aggiungere la faq;
  - il sistema informa l'amministrazione del successo dell'operazione.

Caso d'uso: modifica faq

- *Descrizione*: questo caso si verifica quando l'amministratore vuole modificare una faq esistente nel sistema.
- *Pre-condizioni*: l'utente è l'amministratore.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'amministratore vuole modificare una faq;
  - l'amministratore avvia la procedura di modifica della faq;
  - il sistema fornisce il modulo di modifica delle faq;
  - l'amministratore compila il modulo per modificare la faq;
  - l'amministratore procede ad inviare il modulo di modifica;
  - il sistema effettua la validazione dei vari campi;
  - il sistema procede a modificare la faq;
  - il sistema informa l'amministrazione del successo dell'operazione.

Caso d'uso: elimina faq

- *Descrizione*: questo caso si verifica quando l'amministratore vuole eliminare una faq esistente nel sistema.
- *Pre-condizioni*: l'utente è l'amministratore.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando l'amministratore vuole eliminare una faq;

  - l'amministratore procede alla cancellazione della faq;
  - il sistema fornisce l'interfaccia di cancellazione relativa alla faq;
  - l'amministratore conferma la cancellazione della faq;
  - il sistema procede a cancellare la faq;
  - il sistema informa l'amministrazione del successo dell'operazione.

Caso d'uso: log out

- *Descrizione*: questo caso si verifica quando l'amministratore vuole effettuare la disconnessione.
- *Pre-condizioni*: l'amministratore è registrato nel sistema.
- *Post-condizioni*: nessuna.
- *Sequenza degli eventi principale*:
  - il caso d'uso inizia quando un l'amministratore vuole disconnettersi;
  - l'amministratore avvia la procedura di log out;
  - il sistema disconnette l'amministratore;
  - il sistema ritorna alla sezione pubblica del sito.

CAPITOLO 4

Progettazione

La fase di progettazione svolge un ruolo cruciale nell'andamento complessivo del progetto, influenzando la scalabilità, la sicurezza, l'esperienza utente e la manutenibilità dell'applicazione.

4.1 Architettura sistema

Nella fase di progettazione di un sistema, assume un ruolo di fondametale importanza la comprensione dell'architettura che il sistema dovrà adottare. Questo processo non solo determina la struttura del sistema, ma anche come le varie componenti interagiranno tra loro per garantire il corretto funzionamento dell'applicazione.

Dopo aver identificato dettagliatamente i requisiti, il passo successivo è tradurre questi ultimi in una struttura organizzata e coerente, comunemente realizzata attraverso l'implementazione di un'architettura a tre livelli.

La scelta di un'architettura a tre livelli si basa sulla necessità di organizzare il sistema in maniera modulare e scalabile. I tre livelli fondamentali sono generalmente identificati come il livello di presentazione, il livello di logica applicativa e il livello di persistenza dei dati. Ogni livello svolge un ruolo specifico nel sistema, contribuendo a una separazione chiara delle responsabilità e una manutenzione più agevole.

La Figura 4.1 rappresenta la struttura a strati per una tipica applicazione Java (lato server) ed un client che effettua una richiesta HTTP:

- il client effettua una richiesta HTTP.
- il *Web Server Tomcat* che è in ascolto, gestisce la richiesta.
- Il *Web Server* inoltra la richiesta al *Web Container (Tomcat)* che è responsabile dell'esecuzione della Servlet Java.
- Il *Web Container* determina quale servlet deve essere eseguita e crea gli oggetti *HttpServletRequest*, *HttpServletResponse* e *HttpSession* per interagire con la Servlet.
- La Servlet genera dinamicamente la risposta; questa può essere un documento HTML, dati JSON o altro tipo di contenuto.
- Una volta che la Servlet genera la risposta, il *Web Container* la invia al *Web Server*.
- Il *Web Server* la invia come risposta HTTP al client che ha inizialmente effettuato la richiesta.

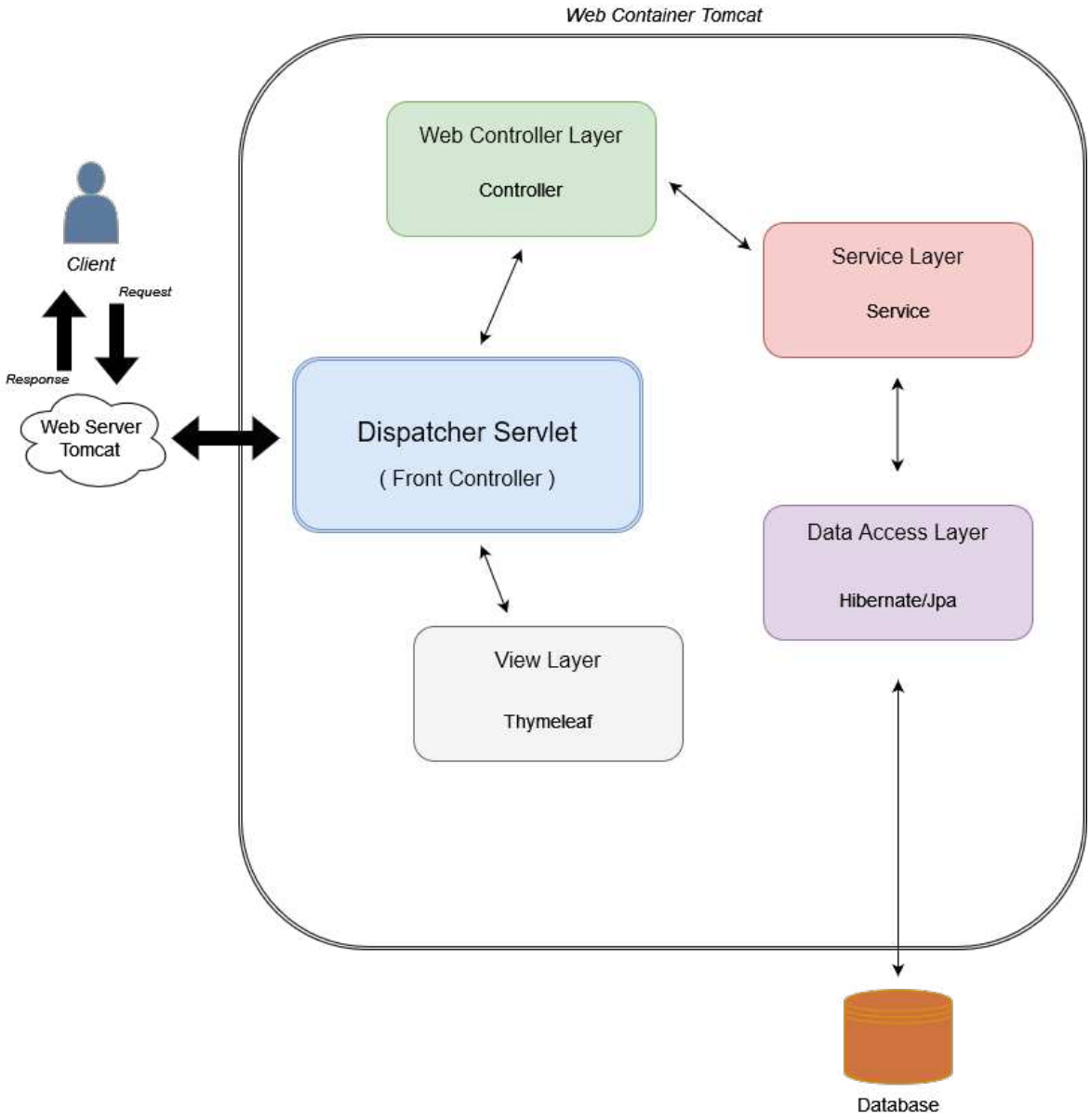

Figura 4.1: Struttura a strati per una richiesta HTTP da parte del client

4.1.1 Architettura a strati

Come detto in precedenza, la stratificazione delle applicazioni porta numerosi vantaggi; di seguito gli strati menzionati in precedenza verranno analizzati in maniera più dettagliata e specifica per un'applicazione Spring:

- *Web Controller Layer (o Presentation Layer):* Spring MVC usa il *Dispatcher Servlet* per gestire le richieste in ingresso. Questo servlet funge da front controller, e utilizza uno o più Handler mapping per mappare la richiesta in arrivo ad un Controller. Il mapping è basato su annotazioni *@RequestMapping* o configurazioni definite in modo esplicito. I controller si possono definire tramite annotazioni *@Controller* o *@RestController*. Una volta effettuato il mapping della richiesta al controller corretto, il *Dispatcher Servlet* chiama il metodo del controller associato alla richiesta. Il controller elabora la richiesta, esegue la logica di business e restituisce la vista o il risultato. Durante l'esecuzione

del controller il modello può essere popolato con dati da usare nella vista, infatti il controller può restituire un oggetto di tipo Model and View.

- *Service Layer (o Business Layer):* in un'applicazione Spring, questo strato è comunemente implementato utilizzando i servizi. I servizi contengono metodi che eseguono la logica di business e coordinano l'accesso ai dati; essi possono essere annotati con *@Service* per essere riconosciuti e gestiti da Spring.

- *Data Access Layer (o Data Access Object):* si occupa di accedere e manipolare i dati nello storage persistente, tipicamente un database relazionale. Questo strato è responsabile di separare la logica di accesso ai dati dalla logica di business e di fornire un'interfaccia uniforme per interagire con il database. Questo può essere implementato con Spring Data JPA, JDBC Template o Hibernate/JPA.

- *View Layer:* è responsabile della presentazione dei dati agli utenti e della gestione dell'iterazione utente. In Spring MVC è spesso composta da pagine HTML. Si utilizzano template engine per creare dinamicamente pagine HTML; Thymeleaf è il template engine più popolare in ambito Spring. Il *Dispatcher Servlet* usa un *viewResolver* per risolvere il nome della vista restituito dal controller in un oggetto concreto di tipo view. La vista viene processata per incorporare i dati nel modello; i tag Thymeleaf nel template vengono interpretati e inseriti nella pagina HTML risultante.

4.2 Progettazione del database

La progettazione del database costituisce una fase fondamentale nello sviluppo di sistemi informativi efficienti e affidabili. In questa sezione ci concentreremo sulla metodologia e sugli strumenti utilizzati per la progettazione di un database. Una metodologia di progettazione del database consiste nella decomposizione dell'attività in singoli passi, indipendenti tra loro, e nell'adozione di una strategia nelle scelte e di un modello di descrizione dei dati. Negli anni si è diffusa una metodologia che suddivide la progettazione del database in tre fasi:

- *Progettazione concettuale*: ha come compito quello di ricevere in input le specifiche informali della realtà di riferimento, per produrre una descrizione formale e completa. In sostanza si vuole rappresentare il contenuto informativo della base di dati senza preoccuparsi dell'effettiva implementazione. Durante questa fase si produce lo schema concettuale, o diagramma E/R, che descrive i dati in modo astratto in una rappresentazione di alto livello.

- *Progettazione logica*: essa riceve in input lo schema concettuale prodotto nella fase precedente ed effettua una traduzione in uno schema logico che fa riferimento al tipo di database utilizzato. Quindi la progettazione logica dipende dal modello di rappresentazione dei dati che si vuole utilizzare; i dati sono descritti e organizzati sotto forma di tabelle.

- *Progettazione fisica*: in questa fase si parte dallo schema logico e lo si completa con la specifica di parametri di memorizzazione dei dati, che riguardano principalmente l'organizzazione dei file e degli indici. Il modello fisico dipende dal DBMS utilizzato, e comprende lo schema concettuale e lo schema logico da cui dipende, sotto forma di documentazione della base di dati.

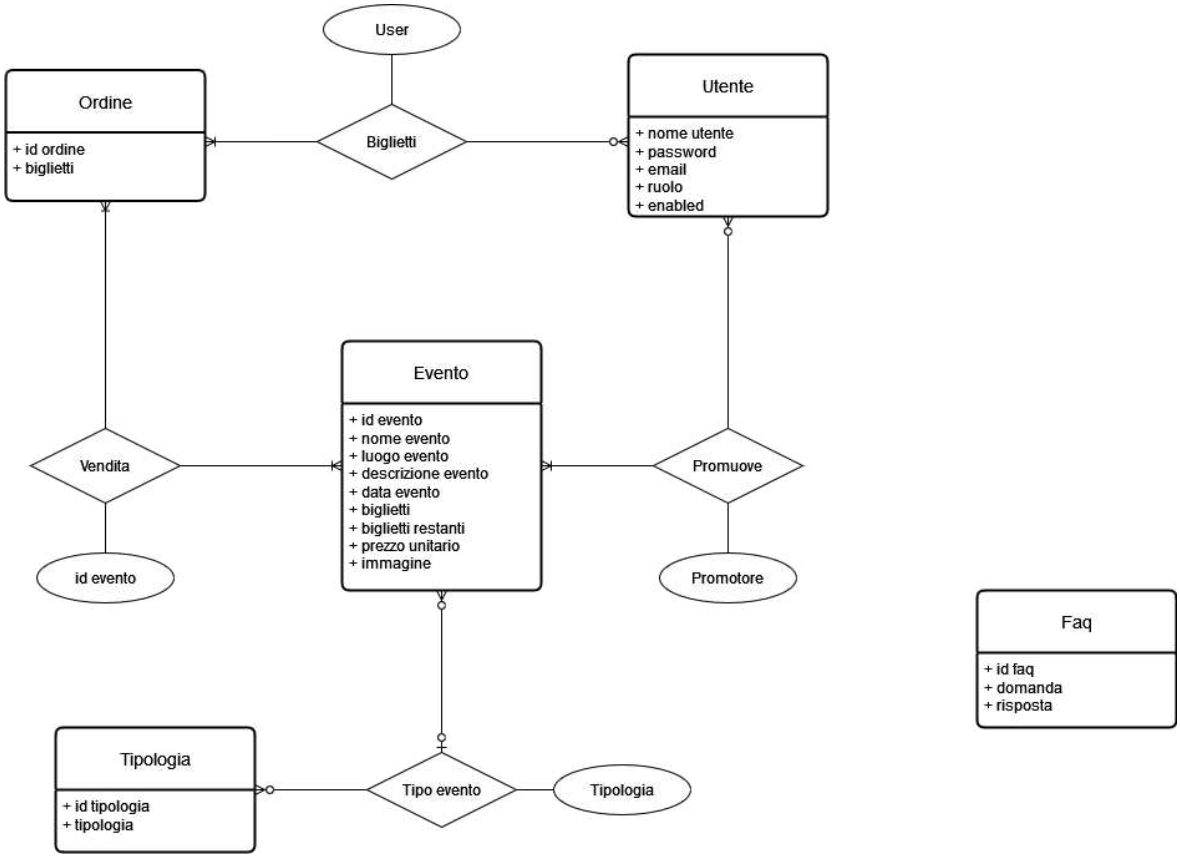

Figura 4.2: Diagramma Entità/Relazione

4.2.1 Identificazione entità e relazioni

L'identificazione accurata delle entità e delle relazioni rappresenta il primo passo cruciale nella progettazione del database. Durante questa fase è fondamentale comprendere a fondo il dominio dell'applicazione al fine di identificare e definire le entità coinvolte. In Figura 4.2 è riportato il diagramma Entità/Relazione ottenuto nella fase di progettazione concettuale. Esso è stato ottenuto mediante le seguenti due fasi:

1. *Identficazione delle Entità*. Le entità rappresentano oggetti o concetti del mondo reale che devono essere memorizzati nel database. È essenziale considerare attentamente le caratteristiche uniche di ciascuna entità, identificando correttamente le chiavi primarie e gli attributi che le definiscono.

2. *Definizione delle Relazioni.* Dopo aver individuato le entità, è fondamentale determinare se esistono connessioni tra di esse. Ci si dedica all'identificazione delle relazioni presenti tra le entità e alla definizione del tipo di cardinalità associato.

4.2.2 Progettazione logica

La fase di progettazione logica del database riguarda la traduzione del diagramma Entità/Relazione ottenuto al termine della fase concettuale, in modo da ottenere uno schema logico che possa descrivere il database in opportune strutture dati. Durante questa fase è necessario prendere in considerazione le specifiche del DBMS che sarà utilizzato per l'implementazione.

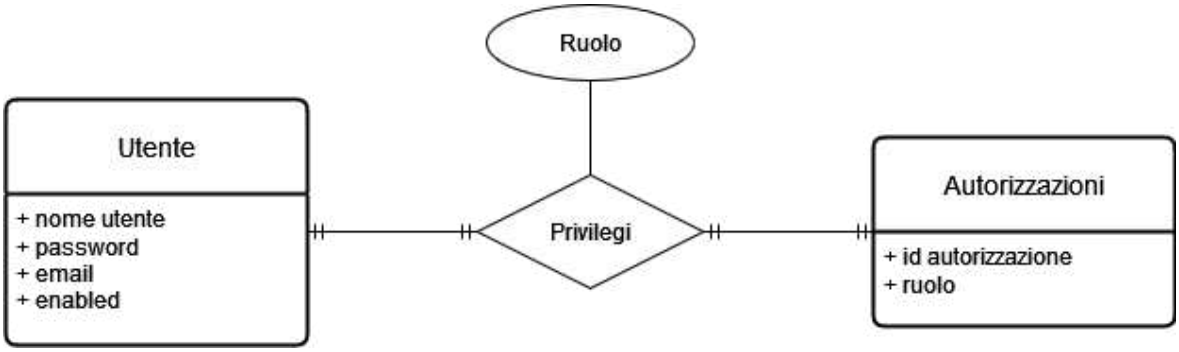

Figura 4.3: Normalizzazione entità *Utente* in due entità più specifiche

La ristrutturazione del diagramma E/R è un processo che coinvolge la modifica della progettazione concettuale del database per migliorare la qualità del modello. Questa attività può essere necessaria per risolvere problemi di progettazione e ottimizzare lo schema in modo che possa essere adattato ad uno schema relazionale. Di seguito sono descritte alcune tecniche comuni utilizzate nella ristrutturazione del diagramma E/R:

1. *Normalizzazione:* è una tecnica che mira a ridurre le ridondanze dei dati e a migliorare l'integrità. Può comportare la suddivisione di entità o relazioni complesse in entità più piccole, riducendo il numero di attributi in ciascuna entità.

2. *Aggiunta o rimozione di attributi:* il modello relazionale non supporta attributi composti o multivalore, pertanto è necessario sostituirli, rispettivamente, con attributi più semplici o nuove entità.

3. *Scelta delle chiavi primarie:* bisogna stabilire quali attributi presenti nel diagramma E/R possono diventare chiavi primarie della rispettiva entità.

Analizzando il diagramma ci siamo resi conto di una generalizzazione relativa all'entità *Utente*. Infatti questa contiene il ruolo dell'utente all'interno del sistema; si è reso, quindi, necessario creare un'ulteriore entità *Autorizzazioni* per garantire una suddivisione più specifica dei ruoli nel sistema (Figura 4.3).

Un'altra generalizzazione riguarda l'attributo *descrizione evento* nell'entità *Utente*, si è proceduto ad eliminare il campo e sostituirlo con due attributi: *descrizione breve* e *descrizione dettagliata*, in modo da avere attributi più specifici all'interno dell'entità.

La traduzione del diagramma E/R rappresenta il passaggio dalla progettazione concettuale del database alla progettazione logica, che coinvolge il modello relazionale. Le entità, le relazioni e gli attributi vengono mappati in tabelle, colonne e vincoli del database relazionale. Di seguito è riportata la traduzione nel modello relazionale:

- *EVENTO* (idevento, nome, luogo, bigliettirim, bigliettimax, datajpa, image, prezzo, user, idtipologia);
- *UTENTE* (user, password, mail, enabled);
- *AUTORIZZAZIONI* (id, ruolo, user);
- *FAQ* (idfaq, domanda, risposta);
- *ORDINE* (idordine, biglietti, idevento, user);
- *TIPOLOGIA* (idtipologia, tipologia).

Attributo vincolato	Associazione vincolante
EVENTI.idtipologia	TipoEvento
EVENTI.user	Promotore
ORDINE.idevento	Vendita
ORDINE.user	Acquisto
AUTORIZZAZIONI.user	Privilegi

Tabella 4.1: Tabella dei vincoli di integrità referenziale

Nello schema logico è necessario stabilire i vincoli di integrità, questi aiutano a garantire la coerenza e l'integrità dei dati. In Tabella 4.1 sono riportati i vincoli di integrità referenziale; questi vincoli assicurano il mantenimento della congruenza tra gli elementi di diverse tabelle.

4.3 Gestione della sicurezza

La gestione della sicurezza in un'applicazione è di fondamentale importanza per garantire la protezione dei dati e delle risorse. Spring Security è il modulo di Spring Framework che fornisce supporto completo per l'implementazione di funzionalità di sicurezza. L'utilizzo di un framework come Spring Security è sicuramente un vantaggio, poiché consente di sfruttare tutte le soluzioni proposte dal framework ed evita di dover reinventare soluzioni a problemi già studiati e risolti dalla community degli sviluppatori. Nonostante l'utilizzo di Spring Security, bisogna stabilire alcuni passi fondamentali per la gestione della sicurezza:

1. *Autenticazione*: Spring Security fornisce meccanismi di autenticazione flessibili, che consentono di gestire e personalizzare l'accesso agli utenti.
2. *Autorizzazione*: la gestione dell'autorizzazione è molto importante; è necessario definire regole di autorizzazione per i diversi utenti all'interno dell'applicazione.
3. *Gestione delle Sessioni*: è possibile personalizzare le sessioni, configurare la durata, gestire le sessioni utente e prevenire attacchi CSRF (Cross-Site Request Forgery).
4. *Protezione della Passoword*: è necessario utilizzare tecniche sicure per la memorizzazione delle password.
5. *Gestione degli errori di sicurezza*: è necessario configurare un gestore degli errori personalizzato per gestire gli errori di sicurezza.

La fase di autenticazione, come visto precedentemente, può essere modificata secondo le esigenze, data la progettazione per l'archiviazione dei dati; si è deciso di sfruttare l'autenticazione tramite username e password attraverso la validazione di Spring Security, utilizzando il provider di autenticazione dedicato a questo scopo.

La gestione delle autorizzazioni è fondamentale per ogni applicazione moderna, al fine di controllare l'accesso e le autorizzazioni degli utenti. I ruoli definiscono le diverse categorie di utenti e specificano quali azioni o risorse un utente di un determinato ruolo può eseguire o accedere.

4.3.1 Suddivisione in ruoli

La progettazione dei ruoli e delle autorizzazioni deve essere pensata attentamente per soddisfare i requisiti specifici dell'applicazione e garantire che gli utenti ottengano solo l'accesso e le autorizzazioni necessarie per svolgere le loro attività. I ruoli individuati nell'applicazione sono :

- *Utente non registrato*: rappresenta un visitatore dell'applicazione che non è registrato nel sistema o che non ha effettuato l'accesso con credenziali.
- *Utente registrato*: rappresenta un utente registrato nel sistema, che vuole acquistare dei biglietti per un evento.
- *Organizzatore*: rappresenta un utente che può pubblicare eventi nel portale.
- *Amministratore*: rappresenta un unico utente che ha i privilegi di gestione di alto livello dell'applicazione.

Una volta definiti i ruoli dei diversi utenti che l'applicazione può ospitare, è necessario delineare le regole di accesso a cui queste tipologie di utenti hanno accesso. Spring Security offre la possibilità di definire un insieme di autorizzazioni, consentendo di specificare le pagine che possono essere visualizzate solo se l'utente possiede le autorizzazioni necessarie.

CAPITOLO 5

Implementazione e manuale utente

In questa sezione verranno analizzate le scelte implementative effettuate per la realizzazione del sistema; successivamente presenteremo un manuale utente completo, al fine di garantire un approccio integrato e user-friendly alla navigazione e alla gestione delle funzionalità offerte.

5.1 Implementazione

5.1.1 Configurazione del framework

La configurazione accurata di Spring Boot è il primo passo cruciale pima di avviare l'implementazione del nostro sistema. Esso semplifica notevolmente questo processo fornendo configurazioni predefinite che riducono il lavoro di setup. Per garantire un ambiente operativo efficiente, sarà essenziale aggiungere i moduli necessari per plasmare l'applicazione desiderata.

Inizieremo incorporando le dipendenze specifiche richiesti dal nostro progetto, sfruttando la struttura modulare di Spring Boot per integrare le funzionalità desiderate. L'obiettivo è creare un ambiente di sviluppo agilmente configurato, pronto per accogliere la logica di business e le iterazioni utente. Le configurazioni vengono effettuate attraverso due file:

- *pom.xml*: in questo file sono dichiarate le dipendenze che verranno inglobate e configurate nel progetto da Spring Boot.
- *application.properties*: in questo file sono dichiarate le impostazioni relative alla nostra specifica applicazione.

Nella Figura 5.1 è presentata una porzione del file *pom.xml* utilizzato per definire le dipendenze utilizzate nell'applicazione.

Le applicazioni basate su Spring Boot necessitano di un entry point, questo contiene il metodo *main* ed è la classe che avvia l'applicazione. Questa classe è molto importante poiché contiene l'annotazione *@SpringBootApplication*, utilizzata per identificare una classe di configurazione principale di Spring Boot. Questa è un'annotazione composta che include tre annotazioni diverse, di seguito analizzate:

- *@Configuration*: questa annotazione indica che la classe è una fonte di configurazione per il contesto dell'applicazione Spring.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.1.5</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.tirociniotriennale</groupId>
    <artifactId>sitoeventi</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>ticketlove</name>
    <description>Sito web per pubblicazione e acquisto eventi</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jdbc</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.thymeleaf.extras</groupId>
            <artifactId>thymeleaf-extras-springsecurity6</artifactId>
        </dependency>
```

Figura 5.1: Configurazione del file *pom.xml*

```
debug=true
spring.thymeleaf.cache=false
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mariadb://localhost:3306/dbticketlove
spring.datasource.username=dbticketlove
spring.datasource.password=
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
server.servlet.context-path=/
logging.level.org.springframework.security=DEBUG

```

Figura 5.2: Configurazione di *application.properties*

- *@EnableAutoConfiguration*: questa annotazione abilita le configurazioni automatiche di Spring Boot.
- *@ComponentScan*: questa annotazione abilita la scansione automatica dei componenti all'interno del package dell'applicazione e dei suoi subpackage. In questo modo Spring rileva automaticamente i componenti, come i controller, i servizi e i repository, senza richedere dichiarazioni esplicite.

I parametri di connessione al database, come il driver da utilizzare, il nome utente, la password e l'URL, vengono, invece, definiti in *application.properties*. La configurazione adottata è presentata nella Figura 5.2. È possibile abilitare diverse impostazioni di debug all'interno di questo file, ottimizzando, così, l'ambiente di sviluppo.

```java
package com.tirociniotriennale.sitoeventi.model;

import java.util.Set;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToMany;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
import jakarta.persistence.CascadeType;

 Clenill
@Entity
@Table(name = "utente")
public class Utente {

    @Id
    @Column(name = "user", length = 25)
    private String user;

    3 usages
    @Column(name = "password", length = 20, nullable= false)
    @NotBlank
    @Size(min=3, max = 20)
    private String password;

    3 usages
    @Column(name = "enabled", nullable= false)
    private boolean enabled;

    3 usages
    @Column(name= "nomeorg", length = 25)
    private String nomeorg;

    3 usages
    @Column(name = "mail")
    private String mail;
```

Figura 5.3: Porzione della classe *Utente*

5.1.2 Gestione degli utenti

In questa sezione illustreremo come la gestione degli utenti è implementata nella nostra applicazione, evidenziando le diverse fasi coinvolte. Questo processo include la persistenza dei dati nel database, la validazione dei campi durante la fase di registrazione e, infine, la modalità in cui il sistema interpreta tali dati per indirizzare le richieste in base al ruolo specifico di ciascun utente.

La registrazione nel portale è possibile attraverso due procedure prestabilite. La prima è dedicata agli utenti non registrati e si realizza compilando il modulo standard di registrazione. Questa procedura concede esclusivamente i privilegi di utente base, consentendo agli utenti di effettuare acquisti. D'altra parte, la seconda procedura è riservata all'amministratore. Attraverso di essa è possibile registrare nel sistema sia utenti con privilegi di organizzatore che utenti con privilegi base.

La fase di persistenza viene attuata mediante la definizione della classe POJO. Nella Figura 5.3 è presentata una sezione di questa classe, arricchita dalle opportune annotazioni JPA e di validazione. Invece, in Figura 5.4 sono riportate le associazioni che questa classe ha con le altre entità. Tali annotazioni servono a collegare la classe al database e a garantire la validazione dei campi prima di procedere con il salvataggio dei dati. Abbiamo scelto di utilizzare le annotazioni per mappare le classi, evitando, così, la necessità di scrivere numerosi file XML, che rappresentano un'alternativa possibile per il mapping.

La classe Utente, tramite Hibernate, genera automaticamente la tabella corrispondente per rendere persistente la registrazione dell'utente. Attraverso il repository specifico per l'utente, Hibernate implementa i metodi necessari per accesso al database. È da notare che

```
58      public String getPassword() { return password; }
61
62      public void setPassword(String password) { this.password=password; }
65
66      public boolean getEnabled() { return enabled; }
69
70      public String getMail(){return mail;}
71
72      public void setMail(String mail){this.mail = mail;}
73
74      public void setEnebled(boolean enabled) { this.enabled = enabled; }
77
78      public String getNomeorg(){return nomeorg;}
79
80      public void setNomeorg(String nomeorg){this.nomeorg = nomeorg;}
81
82      @OneToMany(mappedBy = "utente",
83              fetch = FetchType.EAGER)
84      private Set<Ordine> ordini;
85      public Set<Ordine> getOrdini() { return ordini; }
88
89      @OneToOne(mappedBy = "utenteAut", cascade = CascadeType.ALL)
90      private Autorizzazioni autorizzazioni;
91
92      public Autorizzazioni getAutorizzazioni() { return autorizzazioni; }
95
96      @OneToMany(mappedBy = "utenteevento",
97              fetch = FetchType.EAGER)
98      private Set<Evento> eventi;
99      public Set<Evento> getEventi() { return eventi; }
102
103 }
104
```

Figura 5.4: Associazioni della classe *Utente*

```
@Controller
public class SignupController {
    @Autowired
    private UtenteRepository utenteRepository;
    @Autowired
    private AutorizzazioniRepository autorizzazioniRepository;
    @Autowired
    private UserService userService;

    @GetMapping("/public/registrati")
    public ModelAndView registraUtente(Model model) {
        ModelAndView nur = new ModelAndView( viewName: "public/registrati");
        Utente nuovoUtente = new Utente();
        nur.addObject( attributeName: "nuovoutente", nuovoUtente);
        return nur;
    }

    @RequestMapping(value = "/public/salvautente", method= RequestMethod.POST)
    public String salvaUtente(@Valid @ModelAttribute Utente utente, BindingResult bindingResult, Model model, RedirectAttributes redirectAttributes) {
        Boolean registrato = false;

        if(bindingResult.hasErrors()){
            redirectAttributes.addFlashAttribute( attributeName: "messaggiored",  attributeValue: "La password deve avere almeno 3 caratteri");
            return "redirect:/public/registrati";
        }

        registrato = userService.salvaNuovoUtente(utente);

        if(!registrato){
            redirectAttributes.addFlashAttribute( attributeName: "messaggiored", attributeValue: "Nome utente già esistente," +
                    "impossibile completare l'operazione");
            return "redirect:/public/registrati";
        }

        redirectAttributes.addFlashAttribute( attributeName: "messaggio",  attributeValue: "Utente Salvato!");
        return "redirect:/public/index";
    }
}
```

Figura 5.5: Metodo del Controller che mappa la richiesta di registrazione per un nuovo utente

questo repository estende *JpaRepository*, il quale offre tutte le operazioni CRUD necessarie per la manipolazione dei dati dell'utente nel database.

I metodi del repository possono essere utilizzati nel Controller o nei servizi implementati per effettuare le relative operazioni CRUD di cui si ha bisogno. In Figura 5.5 è riportato il listato del Controller associato alla registrazione. Il Controller richiama lo strato di servizio che implementa i controlli ed esegue il salvataggio nel database del nuovo utente.

La validazione nel sistema è attuata mediante l'integrazione della dipendenza *validation*, la quale può essere inclusa nel file *pom.xml*. Concretamente, facciamo uso di annotazioni nella classe POJO, le quali variano in base ai diversi campi a cui sono applicate. Durante l'elaborazione nel Controller, i parametri di validazione vengono richiamati tramite l'utilizzo dell'annotazione *@Valid*, posizionata prima dell'oggetto da validare. Entrambi vengono passati come attributi al metodo del Controller per poter effettuare la validazione.

Gestire la sicurezza di un'applicazione implica assicurarsi che essa sia in grado di valutare le informazioni di accesso fornite dall'utente e, in base a queste, decidere se consentire o meno l'ingresso. Inoltre, l'applicazione deve essere in grado di determinare le autorizzazioni dell'utente per l'accesso alle diverse risorse del sistema.

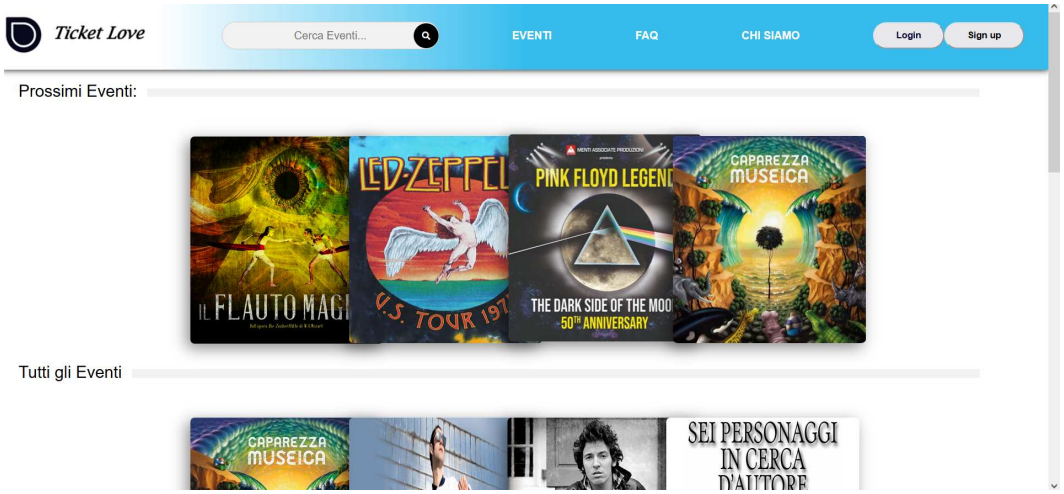

Figura 5.6: Vista della Home page

Gli utenti all'interno della nostra applicazione vengono gestiti in base ai permessi associati a ciascuno di essi. L'interpretazione di tali permessi comporta la suddivisione degli utenti in diverse categorie in base ai ruoli che ricoprono. Per implementare questa funzionalità, è stato necessario apportare modifiche alle funzionalità base di Spring Security, intervenendo sul *successHandler*. L'accesso alla propria sezione richiede il login; una volta autenticati, Spring Security memorizza le informazioni dell'utente, consentendo l'utilizzo del nome utente e della autorizzazioni assegnate. Questo meccanismo risulta particolarmente utile nel corretto smistamento e nella corretta gestione delle diverse categorie di utenti.

5.2 Manuale Utente

Questa sezione offre un documento guida per l'utilizzo del sistema. In dettaglio, mira a spiegare le procedure che gli utenti devono seguire per assicurare un suo corretto funzionamento. Vengono, inoltre, descritti i servizi fruibili in ciascuna sezione dell'applicazione.

5.2.1 Home Page del sistema

La pagina a cui si viene reindirizzati una volta effettuato l'accesso all'applicazione è illustrata nella Figura 5.6. Questa è suddivisa in un'intestazione orizzontale, che consente la navigazione nel sistema informativo, una sezione in primo piano, che presenta gli eventi ordinati per data, e un'ulteriore sezione, che elenca tutti gli eventi.

5.2.2 Autenticazione al sistema

Per accedere alle varie sezioni riservate è necessario effettuare il login; il modulo di login è illustrato in Figura 5.7. Dopo aver inserito le credenziali, l'utente viene automaticamente indirizzato in base ai propri privilegi.

Ciascuna sezione, destinata agli utenti collegati, include un menù a tendina posizionato nella barra di navigazione, offrendo, così, una serie di operazioni disponibili per l'utente.

5.2.3 Acquisto biglietti e storico degli ordini

I biglietti per un evento sono disponibili per l'acquisto esclusivamente per gli utenti registrati con autorizzazione base. Per effettuare l'acquisto, è necessario essere autenticati e

Figura 5.7: Vista relativa al Login

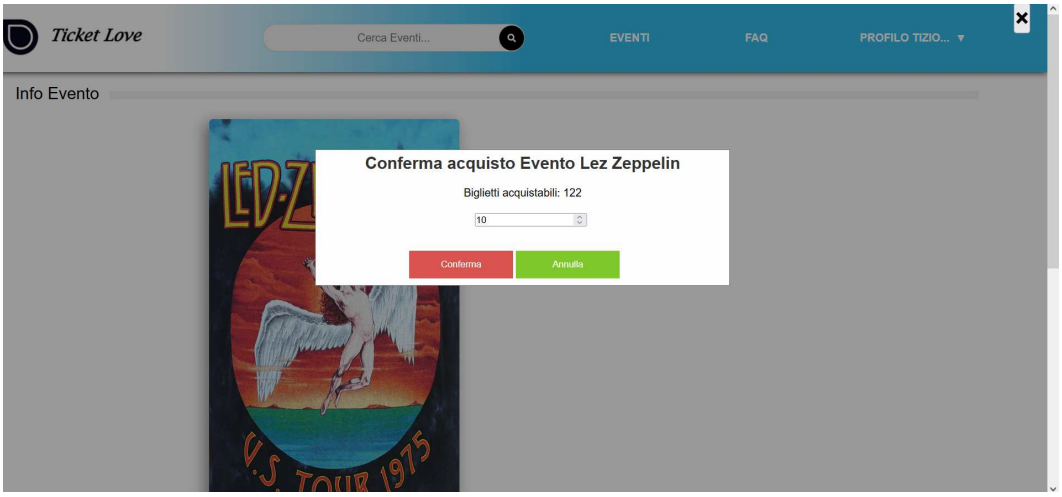

Figura 5.8: Acquisto dei biglietti relativi ad un evento

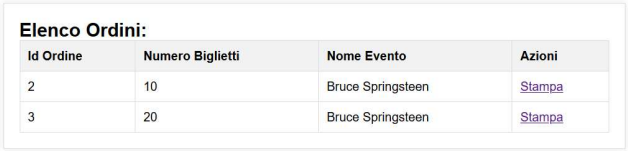

Figura 5.9: Storico degli ordini di un utente

selezionare l'evento di desiderato. Durante la procedura di acquisto per l'evento scelto, sarà possibile specificare la quantità desiderata di biglietti. Un esempio di questa procedura è illustrato nella Figura 5.8.

La cronologia degli ordini relativa all'utente può essere visualizzata tramite il menù a tendina presente nella barra di navigazione del sistema informativo. All'interno di questa sezione è, inoltre, possibile generare il PDF dell'acquisto associato ad un determinato evento. Un esempio di questa funzione è illustrato nella Figura 5.9. Il PDF verrà scaricato tramite il browser sul dispositivo dell'utente.

5.2.4 Aggiunta di un nuovo evento

La funzione di aggiunta di un nuovo evento è accessibile attraverso il menù a tendina dedicato agli utenti con privilegi di organizzatore. Una volta selezionata questa opzione,

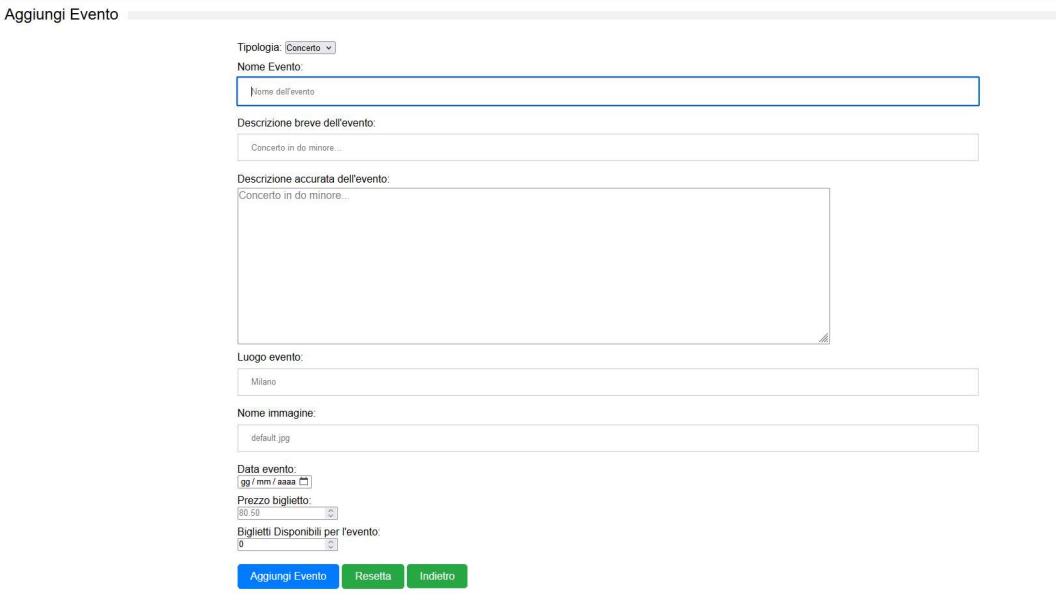

Figura 5.10: Modulo per l'aggiunta di un nuovo evento

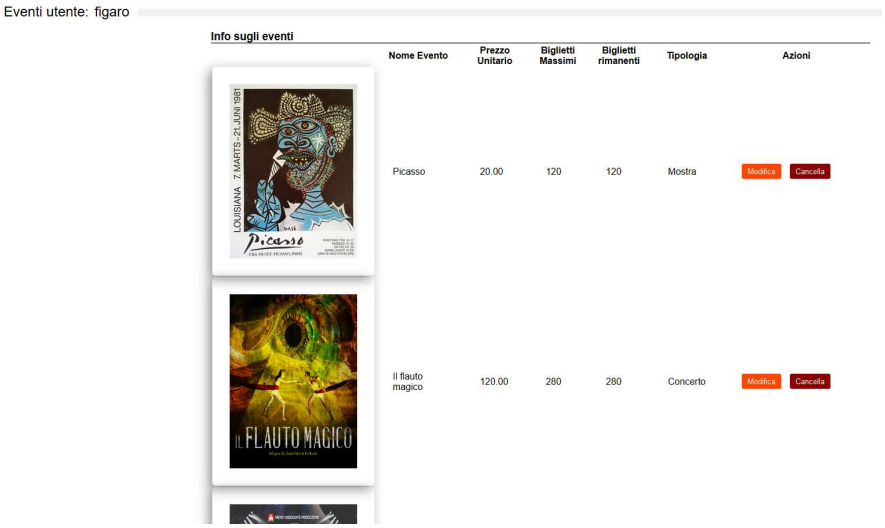

Figura 5.11: Vista di gestione degli eventi

si accede a una sezione che contiene il modulo da compilare. Tra i vari campi da inserire, troviamo la categoria alla quale l'evento appartiene. Questo attributo è configurabile attraverso un menù a tendina, permettendo di assegnare l'evento a categorie predefinite. Dopo aver completato il modulo, è necessario inviarlo per la validazione dei campi e l'inserimento effettivo in caso di successo. La Figura 5.10 mostra un esempio del modulo di inserimento relativo a un nuovo evento.

5.2.5 Modifica di un evento

La modifica di un evento esistente è consentita sia per l'organizzatore dell'evento che per l'amministratore. L'organizzatore può accedere facilmente alla lista dei propri eventi selezionando l'opzione corrispondente nel proprio menù a tendina. La Figura 5.11 illustra tale sezione. Una volta scelto l'evento desiderato, è possibile interagire con esso tramite la sezione "Azioni", contenente i pulsanti modifica e cancella.

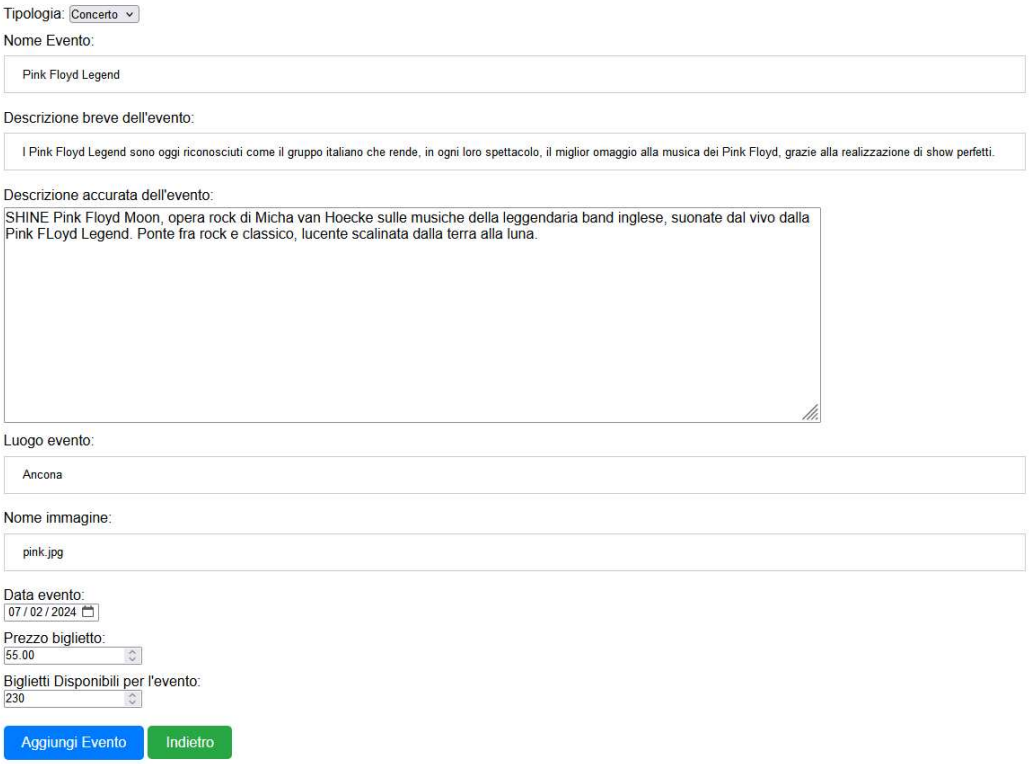

Figura 5.12: Modulo di modifica relativo ad un evento pubblicato

Figura 5.13: Cancellazione di un evento

La modifica di un evento da parte dell'amministratore è realizzabile attraverso una sezione simile, in cui sono elencati tutti gli eventi pubblicati. In entrambi i casi, gli utenti verranno reindirizzati al modulo di modifica dell'evento. La Figura 5.12 mostra il modulo corrispondente.

5.2.6 Cancellazione di un evento

La cancellazione di un evento può essere effettuata interagendo con la sezione "Azione", tramite l'apposito pulsante "Cancella". Prima della cancellazione verrà chiesta una conferma aggiuntiva. Questa funzionalità è abilitata per gli utenti organizzatori e per l'amministrazione. Un esempio di questa procedura è illustrato nella Figura 5.13.

5.2.7 Gestione degli utenti

La gestione degli utenti è una funzione riservata all'amministratore del sistema; si può accedere a tale sezione tramite il menù a tendina nella barra di navigazione. Una volta selezionata la voce corrispondente, la vista per la gestione degli utenti viene caricata, permettendo di

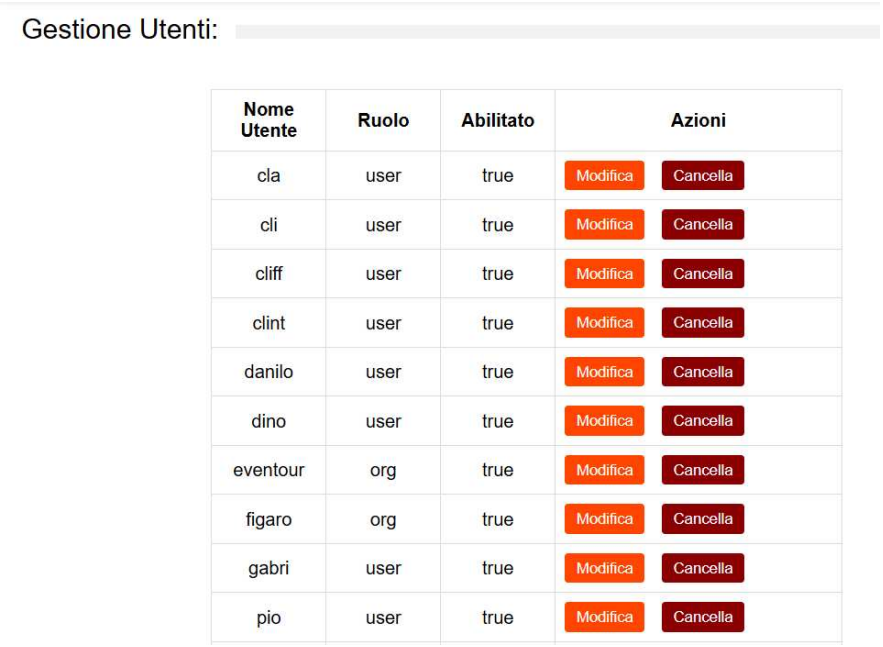

Figura 5.14: Gestione degli utenti

Figura 5.15: Conferma della cancellazione di un utente dal sistema

interagire con ciascun utente attraverso la sezione "Azioni", che include i pulsanti di modifica e cancellazione. La vista relativa è mostrata nella Figura 5.14.

Cancellazione di un utente

La procedura di cancellazione di un utente opera in modo molto simile a quella di cencellazione di un evento. Questa somiglianza è sfruttata per migliorare il livello di confidenza che l'utente ha nei confronti delle varie funzionalità offerte. La vista corrispondente è presentata nella Figura 5.15. È importante notare che un organizzatore non può essere cancellato se ha eventi in corso; è comunque possibile cancellare i suoi eventi e poi procedere con la cancellazione dell'utente. Al contrario, è possibile cancellare un utente che ha effettuato un ordine; tuttavia, tali ordini non saranno inclusi nei biglietti rimanenti per l'evento considerato, poiché l'acquisto è andato a buon fine.

Modifica di un utente

La modifica di un utente da parte dell'amministratore è accessibile tramite la pagina di gestione degli utenti illustrata nella Figura 5.14. Una volta selezionato l'utente che si desidera modificare, viene caricato il modulo di modifica corrispondente. Un esempio di questa funzionalità è mostrato nella Figura 5.16.

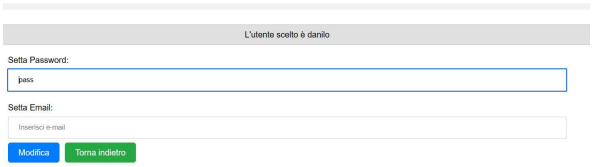

Figura 5.16: Modulo per la modifica dei dati di un utente

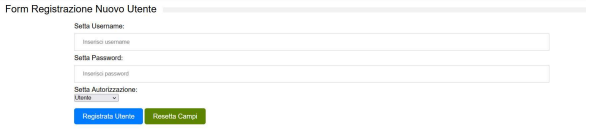

Figura 5.17: Modulo per l'aggiunta di un nuovo utente

Figura 5.18: Vista relativa alla gestione delle FAQ

Aggiunta di utenti al sistema

Questa sezione è accessibile attraverso il menù nella barra di navigazione. L'amministratore ha la possibilità di aggiungere nuovi utenti con diversi tipi di autorizzazione; questo rappresenta l'unico modo per inserire utenti con privilegi di organizzatore. Nella Figura 5.17 è illustrato il modulo per l'aggiunta di un nuovo utente.

5.2.8 Gestione FAQ

La gesione delle FAQ è accessibile tramite il menù a tendina nella barra di navigazione. Una volta selezionata, la sezione corrispondente viene caricata, consentendo di aggiungere nuove FAQ, modificare una FAQ esistente o cancellare una FAQ. Il modulo per l'aggiunta di una nuova FAQ e quello per la sua modifica sono molto simili, differendo solo nei campi compilati nel caso di una modifica. La procedura di cancellazione segue il noto processo, richiedendo la conferma all'amministratore prima di procedere con l'eliminazione. La Figura 5.18 illustra la pagina di gestione delle FAQ.

CAPITOLO 6

Conclusioni

In questa sezione sono presentate le conclusioni al nostro lavoro di tesi e alcuni possibili sviluppi futuri che potrebbero essere implementati all'interno dell'applicazione.

6.1 Conclusioni

Le conclusioni di questo progetto testimoniano il successo nell'implementazione di un portale dedicato alla gestione degli eventi, sviluppato utilizzando il framework Spring.

Attraverso il percorso di sviluppo, ci siamo posti l'obiettivo di creare una piattaforma versatile, intuitiva e robusta, capace di soddisfare le esigenze di utenti con differenti ruoli e necessità.

Il sistema permette l'accesso a quattro categorie di utenti: quelli non registrati, quelli registrati con autorizzazione base, quelli organizzatori di eventi e l'amministratore.

Nella realizzazione del sistema sono stati rispettati tutti i passi previsti in fase di progettazione. Il punto di partenza è stato l'analisi dei requisiti, durante la quale abbiamo stabilito le funzionalità necessarie. Successivamente, abbiamo proceduto con la costruzione dell'architettura del sistema, adottando una struttura basata su quattro layer: il *Web Controller Layer*, il *Service Layer*, il *Data Access Layer* e il *View Layer*.

Durante lo sviuppo, abbiamo adottato le best practices di Spring, sfruttando appieno le sue funzionalità per garantire una gestione efficiente e sicura dei dati, oltre a fornire un'esperienza utente ottimale. La realizzazione del sistema è stata facilitata dall'utilizzo di framework che hanno semplificato la scrittura del codice, contribuendo alla coerenza e alla qualità complessiva del progetto.

Un elemento chiave del nostro progetto è stata l'adozione di Spring Boot per la gestione dell'infrastruttura e la configurazione dell'applicazione. Esso ci ha fornito una soluzione completa e conveniente per avviare rapidamente il nostro progetto, semplificando la configurazione e la gestione delle dipendenze.

La principale caratteristica di Spring Boot è la sua capacità di incorporare le migliori pratiche di Spring Framework, offrendo, allo stesso tempo, un approccio *convention over configuration*. La sua struttura di default ci ha permesso di concentrarci maggiormente sulla logica del nostro progetto piuttosto che sulle configurazioni complesse.

Inoltre, la facilità con cui Spring Boot gestisce l'iniezione delle dipendenze e la configurazione automatica ha semplificato notevolmente lo sviluppo e l'integrazione di nuove funzionalità nel nostro portale. L'utilizzo di Spring Boot ha garantito anche la compatibi-

lità con molte delle funzionalità di Spring, offrendo un ambiente di sviluppo coerente e omogeneo.

Un aspetto importante del progetto è stato il focus sull'usabilità, progettando un'interfaccia intuitiva che semplificasse l'esperienza degli utenti nell'accesso alle diverse funzionalità del portale. La gestione degli eventi, l'acquisto di biglietti e la visualizzazione delle informazioni correlate sono state ottimizzate per massimizzare l'iterazione utente e migliorare l'accessibilità.

Questo è stato possibile mediante l'utilizzo di Thymelaf come motore di template. Thymeleaf ha dimostrato di essere una scelta eccellente per la creazione di pagine web dinamiche, permettendoci di interagire facilmente con i dati nelle nostre view. La sua sintassi leggibile e l'integrazione nativa con Spring hanno semplificato la gestione e la creazione di pagine web interattive e reattive.

La flessibilità del portale è stata ulteriormente potenziata con l'integrazione di funzionalità per la gestione degli utenti, consentendo agli organizzatori di eventi di pubblicare, modificare e cancellare eventi, mentre gli utenti registrati possono navigare, acquistare biglietti e interagire con il sistema in modo intuitivo.

L'adozione di Hibernate ha permesso una mappatura oggetto-relazione (ORM) potente e flessibile, semplificando notevolmente le operazioni di accesso al database. L'utilizzo di Hibernate ha reso la gestione dei dati più efficiente e conforme agli standard, migliorando la manutenibilità e la scalabilità del nostro sistema.

Per garantire un ambiente sicuro e protetto abbiamo deciso di integrare Spring Security. Esso ha assicurato una corretta gestione dell'autenticazione e autorizzazione, garantendo che l'accesso alle risorse del'applicazione fosse controllato e sicuro.

Spring Security ci ha fornito un sistema di autenticazione robusto, consentendo di accedere al portale attraverso una procedura di login sicura e affidabile. Abbiamo implementato un sistema di autenticazione basato su username e password, garantendo agli utenti registrati un accesso personalizzato alle funzionalità del portale.

Per gestire le diverse tipologie di utenti con ruoli specifici, abbiamo sfruttato le funzionalità di autorizzazione offerte da Spring Security. Ciò ci ha permesso di definire chiaramente quali risorse e funzionalità erano accessibili per ciascun ruolo. Gli organizzatori di eventi sono stati dotati di privilegi distinti rispetto agli utenti base, mentre il ruolo dell'amministratore della piattaforma è stato definito in modo che possa gestire tutte le funzionalità disponibili. Similmente, agli utenti non autenticati è concesso l'accesso al portale, limitato alle funzionalità base.

La combinazione di Spring Security, Thymeleaf e Hibernate ha fornito una base solida per il nostro portale, consentendoci di realizzare una piattaforma dinamica e performante. L'adozione di queste teconologie ha contribuito significativamente al successo complessivo del progetto, fornendo strumenti potenti e ben integrati per affrontare sfide specifiche legate alla gestione degli eventi e all'interazione con gli utenti.

Il progetto ha dimostrato di rispondere in maniera positiva alle sfide iniziali, superandole con successo. Tuttavia, è importante sottolineare che la tecnologia evolve rapidamente e le esigenze degli utenti possono cambiare nel tempo. Pertanto, il portale è stato progettato in modo modulare e scalabile, permettendo future espansioni e aggiornamenti.

In conclusione, questo progetto ci ha fornito una panoramica approfondita sulla progettazione e implementazione di un portale per la gestione degli eventi con Spring. L'adozione di Spring Boot è stata determinante per il successo del nostro progetto, consentendoci di realizzare una piattaforma stabile, efficiente e pronta per il rilascio in modo rapido. L'approcio user-friendly di Spring Boot ha reso l'intero processo di sviluppo più agevole e ha contribuito notevolmente al conseguimento dei nostri obiettivi di realizzazione del nostro portale.

6.2 Sviluppi futuri

Il sistema che abbiamo realizzato costituisce una solida base per soddisfare le diverse esigenza degli utenti, ma ci sono opportunità di arricchire l'applicazione con nuove funzionalità per renderla ancora più completa e allineata alle tendenze attuali del mondo informatico.

Un ambito di miglioramento potrebbe riguardare l'ottimizzazione della ricerca degli eventi all'interno del portale. Considerando l'importanza dell'immediatezza delle informazioni, potrebbe essere interessante rendere la ricerca più dinamica. Questo obiettivo potrebbe essere raggiunto mediane l'implementazione di tecnologie come AJAX e JavaScript, consentendo agli utenti di ottentere risultati in tempo reale durante la digitazione.

Un'altra area di sviluppo potrebbe concentrarsi sulla gestione della disabilitazione di eventi e utenti. Per quanto riguarda gli utenti, Spring Security offre un meccanismo agevole per la disabilitazione. Tuttavia, per gli eventi, sarebbe necessario apportare modifiche alla struttura della tabella e concedere tale privilegio all'amministratore. Questa funzionalità consentirebbe di disabilitare un evento mantenendo lo storico degli ordini associati, offrendo, così, una maggiore flessibilità gestionale.

Ulteriori miglioramenti potrebbero coinvolgere l'implementazione di funzionalità di supporto agli utenti, andando oltre la sezione FAQ che attualmente fornisce risorse statiche. Introdurre un servizio clienti in tempo reale potrebbe offrire un'assistenza dinamica, aumentando la fiducia degli utenti nell'applicazione.

Infine, un'idea interessante potrebbe essere l'implementazione di un carrello per gli utenti, arricchito con un timer per completare l'acquisto. Questa funzionalità permetterebbe l'annullamento automatico dell'ordine nel caso in cui il timer raggiunga la scadenza prevista. Allo stesso tempo, fornirebbe agli utenti un elenco dettagliato degli ordini in attesa di pagamento, contribuendo in modo significativo a ottimizzare e rendere più trasparente l'intero processo di acquisto, migliorando l'esperienza complessiva degli utenti all'interno dell'applicazione.

Bibliografia

FELIPE GUTIERREZ, J. B. O. (2022), *Introducing Spring Framework 6: Learning and Building Java-based Applications With Spring*, Apress.

IULIANA COSMINA, C. S. C. H., ROB HARROP (2023), *Pro Spring 6: An In-Depth Guide to the Spring Framework*, Apress.

J. ARLOW, I. N. (2007), *UML 2 e Unified Process*, McGraw Hill.

LEONARD, A. (2020), *Spring Boot Persistence Best Practices*, Apress.

MARTEN DEINUM, J. L., DANIEL RUBIO (2023), *Spring 6 Recipes: A Problem-Solution Approach to Spring*, Apress.

MASSIMO NARDONE, C. S. (2023), *Pro Spring Security: Securing Spring Framework 6 and Boot 3-based Java Applications*, Apress.

MUSIB, S. (2022), *Spring Boot in Practice*, Manning.

PAOLO ATZENI, P. F., STEFANO CERI (2023), *Basi di dati*, McGraw-Hill.

SACCO, A. (2022), *Beginning Spring Data: Data Access and Persistence for Spring Framework 6 and Boot 3*, Apress.

SLVA PRASED REDDY, S. U. (2022), *Beginning Spring Boot 3*, Apress.

SPILCA, L. (2023), *Spring Security in Action, Second Edition*, Manning.

SPLICA, L. (2021), *Spring Start Here*, Manning.

TURNQUIST, G. L. (2022), *Learning Spring Boot 3.0, Third Edition*, Packt.

ULLENBOOM, C. (2024), *Spring Boot 3 and Spring Framework 6*, SAP PRESS.

Siti web consultati

- Documentazione Spring Framework – `https://spring.io/projects/spring-framework`

- Documentazione Java – `https://docs.oracle.com/en/`
- Hibernate ORM User Guide – `https://docs.jboss.org/hibernate/orm/6.4/userguide/html_single/Hibernate_User_Guide.html`
- Documentazione Thymeleaf – `https://www.thymeleaf.org/`
- MariaDB Foundation – `https://mariadb.org/`
- Introduction to Hibernate 6 – `https://docs.jboss.org/hibernate/orm/6.4/introduction/html_single/Hibernate_Introduction.html#introduction`
- Wikipedia – `www.wikipedia.org`

Ringraziamenti

Non mi sembra quasi vero di essere giunto alla fine del mio percorso di studi, percorso un po' travagliato e con qualche intoppo. Ho iniziato da giurisprudenza, dove non troppo brevemente ho realizzato che quella strada non mi si addiceva a pieno. In quei primi anni ho continuato a sviluppare personalmene la mia passione per l'informatica, fornendo supporto ad amici e parenti con la costruzione di computer e risolvendo le solite problematiche che si appioppano agli informatici, o presunti tali.

In quel periodo ho maturato la consapevolezza che la mia bravura nell'ambito informatico, emersa già alle scuole superiori, sarebbe potuta essere un qualcosa su cui investire. Grazie al supporto fornito dalla mia famigia ho potuto intraprendere gli studi di ingegneria.

I primi anni di università non sono stati facili; riprendere gli studi scientifici dopo anni di abbandono ha reso necessaria una svolta mentale senza la quale non sarei potuto arrivare a questo punto.

Un aiuto mi è stato fornito dalle persone incontrate in questi anni; sopratutto ringrazio Erica per il suo continuo supporto; mi ha regalato momenti unici. Durante un viaggio ci si ritrova a guadagnare qualcosa e dover lasciare qualcos'altro indietro, volendo o meno. Ringrazio i miei nonni, sopratutto nonno Antonio, per esserci sempre stato. Ringrazio i miei genitori, che mi hanno inculcato fin da bambino la necessità di una laurea, essenziale per potersi realizzare. Ringrazio il prof. D'Amico, che mi insegnava informatica alle superiori, per avermi spinto a credere in me stesso.

Ringrazio il professore Domenico Ursino che mi ha permesso di terminare gli studi nel modo migliore possibile, indicandomi la strada da seguire per lo svolgimento del tirocinio e della tesi. Una persona disponibile, professionale, con molta pazienza e voglia di fare. Il suo esempio di professionalità mi ha spinto a dare il meglio in queste fasi finali.