

Documentazione del Progetto Spring Boot

Paolo Ardolino (215046)

4 febbraio 2025

Indice

1	Introduzione	3
1.1	Panoramica del Progetto	3
1.2	Pubblico Target	3
1.3	Link Utili	3
2	Tecnologie Utilizzate	4
3	Scopo del Progetto	4
4	Funzionalità	5
5	Architettura del Sistema	7
5.1	Entità	7
5.2	Diagramma dei Package	10
5.3	Descrizione delle Componenti Principali	14
5.4	Interazioni tra i Componenti	15
6	Design Patterns	15
6.1	Facade Pattern	15
6.2	Builder Pattern	15
6.3	Layered Pattern	17
7	Descrizione dei Controller	17
7.1	AdminController	17
7.2	ChatController	18
7.3	ClienteController	18
7.4	SuperAdminController	18

8 Servizi dell'Applicazione	19
8.1 ChatService	19
8.2 CustomSenderMessaggioService	19
8.3 GeneralService	19
8.4 ProdottoService	20
8.5 RigaOrdineService	21
8.6 UtenteService	21
9 Repository	22
9.1 ChatRepository	22
9.2 IndirizzoRepository	22
9.3 Repository degli Ordini	23
9.4 Repository degli Utenti	23
10 Configurazioni	24
10.1 FilterDiAutenticazione	24
10.2 Spring Security	24
10.3 Endpoint accessibili in base al ruolo	25
11 Gestione degli Errori	25
11.1 Gestione delle eccezioni	25
11.1.1 DatoNonValidoException	25
11.1.2 ResponseStatusException	26
11.2 Eccezioni personalizzate	26
12 Testing con JUnit e Spring Boot	27
12.1 Strumenti Utilizzati	27
12.2 Testing dei Controller	27
12.2.1 Test di Accesso alle Pagine	27
12.2.2 Test di Invio di Form	27
12.2.3 Test con Upload di File	28
12.3 Report sulla Coverage	29

1 Introduzione

1.1 Panoramica del Progetto

Il presente documento descrive lo sviluppo e le funzionalità di un'applicazione e-commerce per la compravendita di componenti usati per batterie, con l'obiettivo di promuovere la sostenibilità e facilitare l'accesso a soluzioni personalizzate. Il progetto è stato sviluppato per rispondere alla crescente esigenza di ridurre gli sprechi elettronici e incentivare il riutilizzo di materiali ancora funzionali.

1.2 Pubblico Target

Questa documentazione si rivolge a:

- **Sviluppatori e tecnici:** Per comprendere l'architettura e il design tecnico del progetto.
- **Utenti finali:** Per conoscere le funzionalità dell'applicazione e il modo in cui possono trarne vantaggio.

1.3 Link Utili

Per facilitare l'accesso a risorse rilevanti, ecco una lista di link utili:

- **Repository GitHub del progetto:**
<https://github.com/PaoloH4rd/DrumKit>
Include il codice sorgente, la documentazione tecnica

2 Tecnologie Utilizzate

Per la realizzazione del progetto sono state adottate le seguenti tecnologie principali:

- **Spring Boot:** Framework backend che facilita lo sviluppo di applicazioni web robuste e scalabili, fornendo un'integrazione nativa per API REST e gestione dei dati.
- **Database relazionale:** Utilizzo di un database SQL per memorizzare e gestire in modo strutturato le informazioni sui componenti disponibili, sugli utenti e sugli ordini effettuati.
- **WebSocket con RabbitMQ:** Implementazione di comunicazioni in tempo reale per aggiornare dinamicamente la disponibilità dei componenti e fornire notifiche agli utenti.
- **Thymeleaf:** Motore di template per il rendering lato server delle pagine web, consentendo un'integrazione fluida con i dati provenienti dal backend.

3 Scopo del Progetto

L'obiettivo principale del progetto è promuovere l'accessibilità e la sostenibilità nel settore delle batterie, riducendo la necessità di acquisti di prodotti nuovi e incoraggiando il riutilizzo dei componenti. La batteria è in questo modo messa insieme con tempo e cura, pezzo per pezzo invece di essere comprata in blocco. Questo garantisce anche grande modularità nel ricambio dei pezzi usurati.

Con questa applicazione, gli utenti possono:

- Acquistare componenti usati a prezzi accessibili.
- Costruire o riparare batterie personalizzate secondo le proprie esigenze.

4 Funzionalità

Super Admin

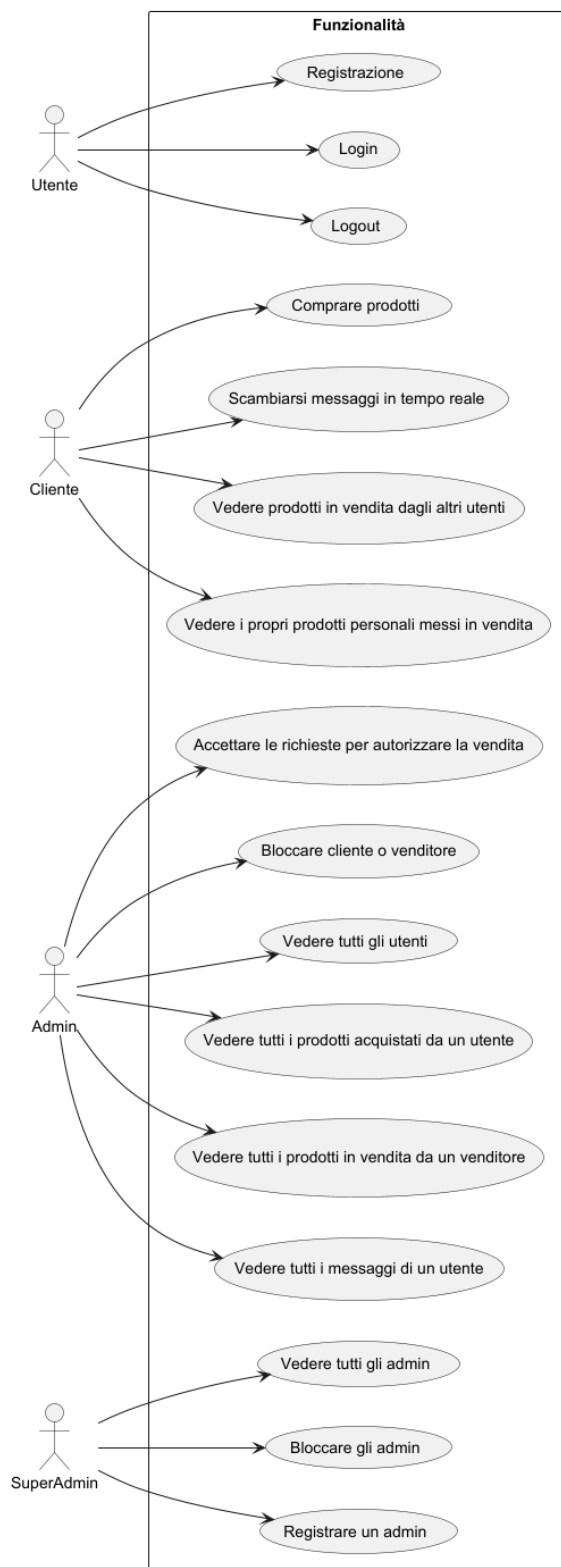
- Registrazione admin
- Vedere tutti gli admin
- Bloccare gli admin

Admin

- Accettare le richieste per autorizzare la vendita
- Bloccare venditore
- Vedere venditori
- Vedere prodotti messi in vendita da un utente

Venditore/Cliente

- Aggiungere prodotto in vendita
- Vedere i prodotti messi in vendita dagli altri utenti
- Acquistare prodotti
- Vedere tutti i propri messaggi
- Iniziare una chat in tempo reale



5 Architettura del Sistema

5.1 Entità

Utente (utente)

- Rappresenta gli utenti della piattaforma, identificati da un ID univoco.
- Contiene informazioni personali come nome, cognome, email, data di nascita e password.
- Ogni utente ha un ruolo (`admin`, `cliente`, `super_admin`).
- Gli utenti possono essere attivi o disattivati.

Chat (chat)

- Gestisce la comunicazione tra due utenti.
- Contiene gli ID dei due partecipanti e un ID univoco per la conversazione.

Messaggio (messaggio)

- Contiene i dettagli di ogni messaggio scambiato in una chat.
- Ogni messaggio è associato a una chat specifica.
- Contiene testo, timestamp e un indicatore per identificare quale utente lo ha inviato.

Ordine (ordine)

- Rappresenta un acquisto effettuato da un utente.
- Contiene lo stato dell'ordine (`confermato`, `consegnato`, `in attesa`, `in consegna`, `spedito`).
- Ogni ordine è associato a un utente.

Prodotto (prodotto)

- Rappresenta un articolo disponibile sulla piattaforma.
- Contiene informazioni come nome, descrizione, immagine, prezzo e disponibilità.
- È associato a un utente (probabilmente il venditore).

Riga Ordine (riga_ordine)

- Associa gli ordini ai prodotti acquistati.
- Specifica la quantità e il prezzo totale per ogni prodotto in un ordine.

6

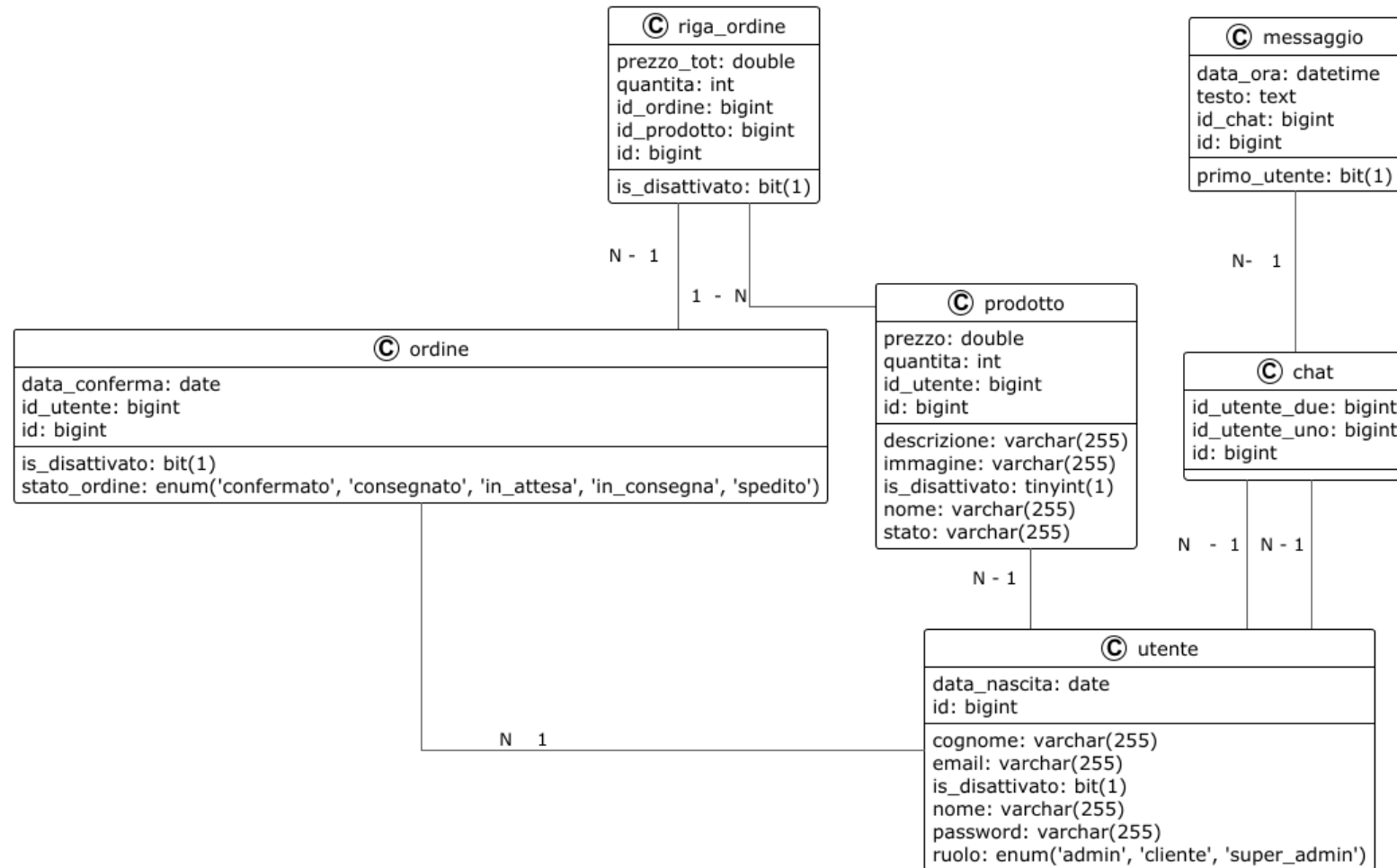


Figura 1: Coverage Report

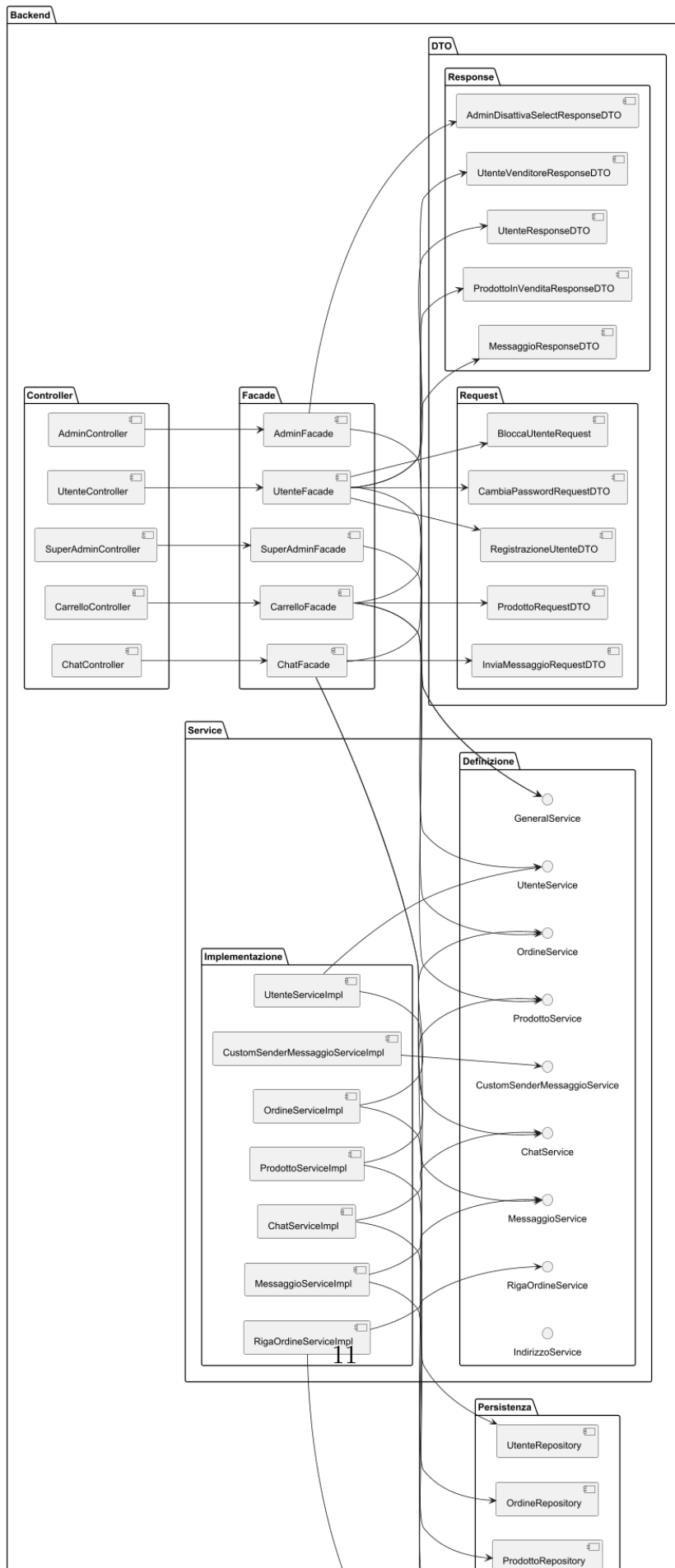
5.2 Diagramma dei Package

Il diagramma seguente illustra come le principali componenti del sistema interagiscono tra loro, organizzando le classi in pacchetti logici. Questa suddivisione facilita la comprensione dell'architettura del software e aiuta a separare le responsabilità tra le varie parti del sistema.

Ogni package rappresenta un'area funzionale distinta, raggruppando classi con ruoli simili. Ad esempio:

- Il **Controller** gestisce le richieste degli utenti e indirizza le operazioni al livello di servizio.
- Il **Facade** agisce come un intermediario tra il Controller e il livello di business, semplificando l'interazione con i servizi.
- Il **Service** è suddiviso in due sezioni:
 - **Definition**, che definisce le interfacce dei servizi.
 - **Implementation**, che fornisce le implementazioni concrete dei servizi.
- Il **DTO** (Data Transfer Object) contiene le strutture dati utilizzate per lo scambio di informazioni tra i vari livelli del sistema.
- Il **Persistence** include le classi responsabili dell'accesso ai dati, come i repository per la gestione del database.

Grazie a questa organizzazione, il sistema mantiene un'architettura modulare e facilmente estensibile, migliorando la manutenibilità e la scalabilità del codice.



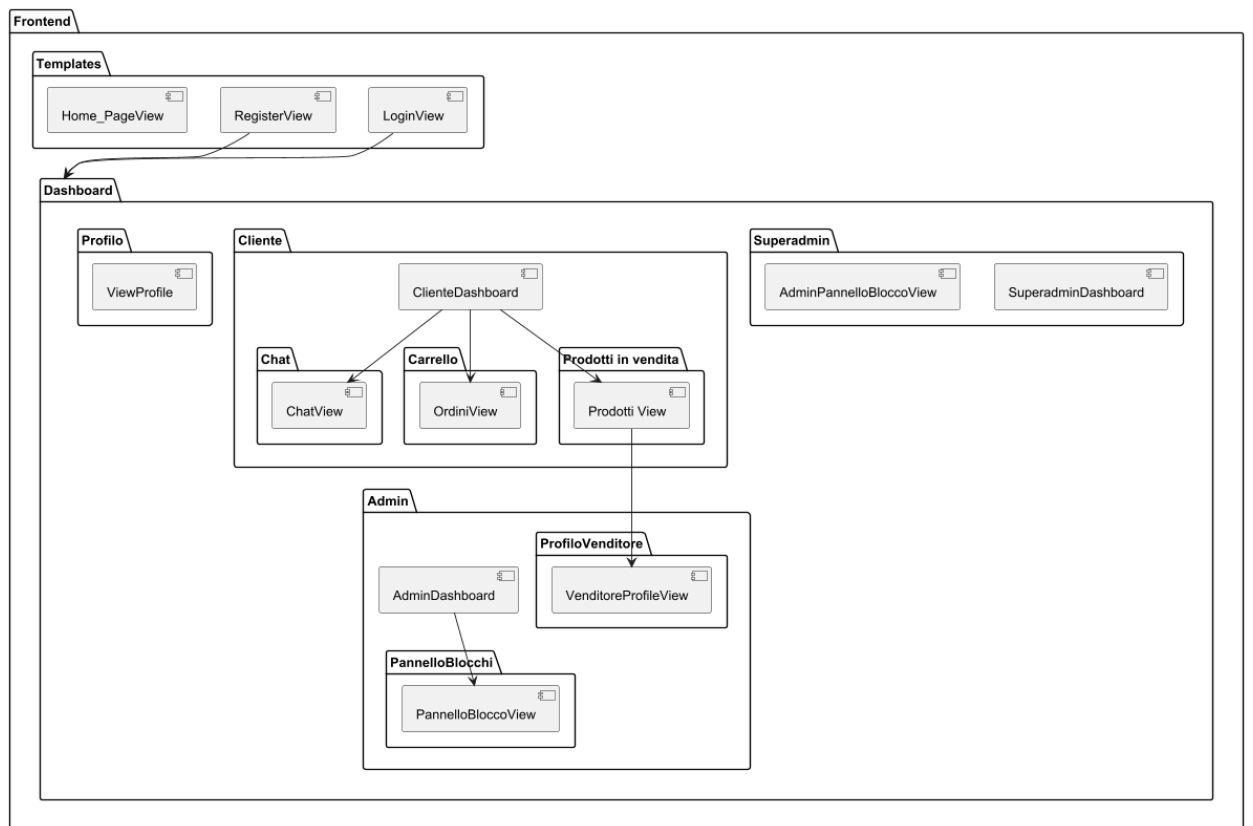


Figura 3: package diagram frontend

Il diagramma descrive l'architettura del frontend di un sistema e-commerce con gestione multi-ruolo. La struttura è organizzata in due macro-componenti principali:

Templates

- **LoginView**: Gestisce l'autenticazione degli utenti
- **RegisterView**: Responsabile della registrazione nuovi utenti
- **Home_PageView**: Vista principale pre-login

Dashboard (Post-Login)

Organizzata per tipologia di utente:

Admin

- **ProfiloVenditore**: Gestione profilo venditore
- **PannelloBloccoView**: Blocco/gestione utenti

Superadmin

- **AdminPannelloBloccoView**: Pannello avanzato di moderazione

Cliente

- **ChatView**: Interfaccia messaggistica
- **OrdiniView**: Gestione carrello e ordini
- **Prodotti View**: Catalogo prodotti
- Navigazione al profilo venditore tramite **VenditoreProfileView**

Profilo Utente

- **ViewProfile**: Gestione profilo personale

Flusso di Navigazione

- Accesso iniziale tramite `LoginView` o `RegisterView`
- Dashboard specifiche per ruolo dopo l'autenticazione
- Interconnessioni tra:
 - Catalogo prodotti e profili venditori
 - Dashboard cliente e funzionalità core (chat, ordini)
 - Strumenti di amministrazione per gestione utenti

Osservazioni Architetture

- Struttura modulare con separazione dei concern
- Pattern MVC evidenziato da:
 - View dedicate per ogni funzionalità
 - Routing chiaro tra componenti
 - Separazione logica per ruoli utente
- Componentizzazione riutilizzabile (es. `VenditoreProfileView` usato in contesti diversi)

5.3 Descrizione delle Componenti Principali

- **Controller:** I controller gestiscono le richieste HTTP e invocano i servizi per elaborare i dati. Restituiscono risposte al client sotto forma di JSON o HTML.
- **Service:** I servizi contengono la logica aziendale e orchestrano le operazioni tra il controller e il repository.
- **Repository:** I repository sono responsabili della gestione della persistenza e dell'interazione con il database. Utilizzano JPA per le operazioni CRUD.
- **Configurazioni:** Le configurazioni, definite nei file `application.yml` o `application.properties`, includono parametri relativi al database, alla sicurezza e ad altre proprietà del progetto.
- **Filtri:** Questi componenti gestiscono aspetti trasversali come l'autenticazione, la gestione delle autorizzazioni e il logging.

- **Gestione degli errori:** Utilizziamo annotazioni come `@ControllerAdvice` e `@ExceptionHandler` per intercettare e gestire errori comuni, restituendo messaggi comprensibili al client.

5.4 Interazioni tra i Componenti

Il sistema segue l'architettura a livelli:

1. Il **Controller** riceve le richieste HTTP e le invia al **Service**.
2. Il **Service** applica la logica aziendale e comunica con il **Repository**.
3. Il **Repository** interagisce con il database per eseguire operazioni di lettura e scrittura.
4. I risultati vengono restituiti dal **Repository** al **Service**, che li elabora ulteriormente prima di inviarli al **Controller**.
5. Il **Controller** restituisce la risposta al client.

6 Design Patterns

6.1 Facade Pattern

Il pattern Facade fornisce un'interfaccia semplificata a un insieme di interfacce in un sistema complesso. Questo pattern definisce un'interfaccia di alto livello che rende il sistema più facile da usare. Il Facade riduce la complessità del sistema e disaccoppia il codice client dal sistema sottostante. È particolarmente utile quando si ha a che fare con librerie o framework complessi, poiché nasconde la complessità e rende più semplice l'interazione con le funzionalità del sistema.

6.2 Builder Pattern

Il pattern Builder è un pattern creazionale che fornisce un modo per costruire oggetti complessi passo dopo passo. Invece di avere un costruttore che accetta tutti i parametri richiesti per creare un oggetto, il Builder consente di creare oggetti in modo incrementale, utilizzando metodi specifici per impostare i vari attributi. Questo pattern è utile quando la costruzione di un oggetto richiede molte fasi o quando è necessario creare varianti diverse di un oggetto. Il Builder migliora la leggibilità del codice e facilita la creazione di oggetti complessi.



Figura 4: Facade e Controllers

- Gestione degli utenti bloccati con la visualizzazione del pannello di blocco (`bloccaUtente`), il blocco (`bloccaUtente`) e lo sblocco (`sbloccaUtente`) degli utenti.
- Visualizzazione del profilo venditore (`profiloVenditore`) e dei relativi prodotti.

7.2 ChatController

Il `ChatController` gestisce le operazioni legate alla messaggistica e alle chat. Le sue principali funzionalità includono:

- Visualizzazione della dashboard delle chat (`chats`) con l'elenco delle conversazioni disponibili.
- Gestione dei messaggi in tempo reale tramite WebSocket (`handleDmMessage`).
- Apertura di una chat specifica (`getChatRabbit`) e invio di messaggi (`inviaMessaggioRabbit`) tramite RabbitMQ.

7.3 ClienteController

Il `ClienteController` è responsabile delle operazioni dell'area clienti. Include:

- Visualizzazione del pannello cliente (`pannelloCliente`), con la lista dei prodotti approvati e disponibili.
- Visualizzazione del profilo venditore (`profiloVenditore`) e dei prodotti in vendita associati.
- Aggiunta di un nuovo prodotto in vendita (`aggiungiProdottoVendita`), con supporto per il caricamento di immagini.

7.4 SuperAdminController

Il `SuperAdminController` gestisce le funzionalità riservate ai super amministratori. Le operazioni principali sono:

- Visualizzazione del pannello super admin (`pannelloAdmin`), che mostra gli amministratori attivi.
- Disattivazione di un amministratore esistente (`disattivaAdmin`).
- Registrazione di un nuovo amministratore (`aggiungiAdmin`) con validazione dei dati.

8 Servizi dell'Applicazione

8.1 ChatService

Descrizione:

L'interfaccia `ChatService` gestisce le operazioni relative alle chat tra utenti. Fornisce metodi per recuperare, creare e salvare le chat.

Metodi:

- `List<Chat> getAllByUsername(String username):`
Restituisce tutte le chat associate all'utente specificato.
- `Chat getByUsernameAndAltroNome(String username, String secondoUsername):`
Recupera la chat tra due utenti specificati.
- `void creaChat(Utente utenteUno, Utente utenteDue):`
Crea una nuova chat tra i due utenti forniti.
- `Chat salva(Chat c):`
Salva l'istanza della chat nel repository.

8.2 CustomSenderMessaggioService

Descrizione:

L'interfaccia `CustomSenderMessaggioService` gestisce l'invio di notifiche e messaggi privati agli utenti.

Metodi:

- `void inviaNotifica(MessaggioResponseDTO m, String topic):`
Invia una notifica relativa al messaggio specificato al topic indicato.
- `void sendPrivateMessage(String idUtente, String message, Long idChat):`
Invia un messaggio privato all'utente specificato all'interno della chat indicata.

8.3 GeneralService

Descrizione:

L'interfaccia generica `GeneralService<T>` definisce operazioni comuni per la gestione di entità di tipo generico.

Metodi:

- `void add(T t):`
Aggiunge una nuova entità.
- `void update(T t):`
Aggiorna l'entità esistente.
- `List<T> getAll() :`
Restituisce tutte le entità.
- `T getById(long id):`
Recupera l'entità con l'ID specificato.
- `void setIsDisattivatoTrue(long id):`
Imposta lo stato di disattivazione dell'entità con l'ID specificato.

8.4 ProdottoService

Descrizione:

L'interfaccia `ProdottoService` estende `GeneralService<Prodotto>` e gestisce le operazioni specifiche relative ai prodotti.

Metodi:

- `void creaProdotto(String nome, String descrizione, double prezzo, int quantita, Utente venditore, String immagine):`
Crea un nuovo prodotto con le informazioni fornite.
- `Utente getProprietario(Long id):`
Recupera il proprietario del prodotto con l'ID specificato.
- `void setStatoProdottoRifiutato(Long idProdotto):`
Imposta lo stato del prodotto come rifiutato.
- `List<Prodotto> getAllProdottiDaApprovare():`
Restituisce tutti i prodotti in attesa di approvazione.
- `List<Prodotto> getAllProdottiApprovatiNonDiUtenteLoggato(Long idUtente):`
Restituisce tutti i prodotti approvati non appartenenti all'utente loggato.
- `void setStatoProdottoApprovato(Long idProdotto):`
Imposta lo stato del prodotto come approvato.
- `List<Prodotto> getAllProdottiApprovareById(Long idVenditore):`
Restituisce tutti i prodotti approvati del venditore specificato.

8.5 RigaOrdineService

Descrizione:

L'interfaccia `RigaOrdineService` estende `GeneralService<RigaOrdine>` e gestisce le operazioni specifiche relative alle righe d'ordine.

Metodi:

- `void aggiungiProdottoAlCarrello(Prodotto prodotto, int quantita):`
Aggiunge un prodotto al carrello con la quantità specificata.
- `List<RigaOrdine> getAllRigheOrdine(Long idUtenteLoggato):`
Restituisce tutte le righe d'ordine associate all'utente loggato.

8.6 UtenteService

La classe `UtenteServiceImpl` implementa l'interfaccia `UtenteService` e fornisce i seguenti servizi per la gestione degli utenti:

- **Autenticazione:** verifica credenziali con `loginCheck()` e recupera utenti per email o ID.
- **Registrazione:** crea nuovi clienti e amministratori con password criptata tramite SHA-256.
- **Gestione utenti:** disattiva utenti e amministra ruoli, impedendo la disattivazione degli admin da parte di utenti normali.
- **Gestione password:** modifica e aggiorna la password di un utente.
- **Recupero informazioni:** ottiene utenti attivi, amministratori e utenti bloccati.



Figura 6: Repository class diagram

9 Repository

9.1 ChatRepository

`ChatRepository` estende `JpaRepository` e fornisce i metodi per interagire con la tabella delle chat nel database. I metodi principali sono:

- `findAllByIdUtente(long id)`: Trova tutte le chat di un utente tramite il suo ID.
- `findAllByEmail(String email)`: Trova tutte le chat di un utente tramite la sua email.
- `findAllByEmail(String emailUno, String emailDue)`: Trova una chat tra due utenti tramite le loro email.

9.2 IndirizzoRepository

`IndirizzoRepository` estende `JpaRepository` e permette di eseguire operazioni sui record degli indirizzi. I metodi principali sono:

- `findAllByUtente_IdAndIsDisattivatoIsFalse(long id_utente)`: Trova tutti gli indirizzi di un utente non disattivato.
- `findAllByIsDefaultIsFalse()`: Trova tutti gli indirizzi che non sono impostati come predefiniti.
- `findByIdAndIsDisattivatoIsFalse(long id)`: Trova un indirizzo tramite il suo ID che non sia disattivato.

9.3 Repository degli Ordini

`OrdineRepository` estende `JpaRepository` e fornisce i metodi per lavorare con gli ordini nel database. I metodi principali sono:

- `findAllByUtente_IdAndIsDisattivatoIsFalse(long idUtente)`: Trova tutti gli ordini di un utente non disattivato.
- `findAllByDataConfermaAndIsDisattivatoIsFalse(LocalDate dataConferma)`: Trova gli ordini confermati in una data specifica.
- `findAllByStatoOrdineAndIsDisattivatoIsFalse(StatoOrdine statoOrdine)`: Trova gli ordini con un determinato stato.
- `findAllByIsDisattivatoIsFalse()`: Trova tutti gli ordini non disattivati.
- `findByIdAndIsDisattivatoIsFalse(long id_ordine)`: Trova un ordine tramite il suo ID che non sia disattivato.

9.4 Repository degli Utenti

`UtenteRepository` estende `JpaRepository` e fornisce i metodi per gestire gli utenti nel database. I metodi principali sono:

- `findByEmailAndPasswordAndIsDisattivatoIsFalse(String email, String password)`: Trova un utente tramite email e password se non è disattivato.
- `findAllByIsDisattivatoIsFalse()`: Trova tutti gli utenti attivi.
- `findByEmailAndIsDisattivatoIsFalse(String email)`: Trova un utente tramite email che non sia disattivato.
- `findAllByRuoloAndIsDisattivatoIsFalse(Ruolo ruolo)`: Trova tutti gli utenti con un determinato ruolo che non siano disattivati.
- `findByEmail(String email)`: Trova un utente tramite la sua email.
- `findAllByIsDisattivatoIsTrue()`: Trova tutti gli utenti disattivati.

10 Configurazioni

10.1 FilterDiAutenticazione

Il filtro `Filter Di Autenticazione` estende `OncePerRequestFilter` ed è responsabile della gestione dell'autenticazione degli utenti nelle richieste HTTP.

- Recupera l'email dell'utente dalla sessione.
- Controlla se la richiesta è per una pagina pubblica (login, registrazione, ecc.) e, in tal caso, continua con la catena di filtri senza ulteriori controlli.
- Se l'email è `null`, l'utente non è autenticato e viene reindirizzato alla pagina di login.
- Se l'utente è autenticato, carica i dettagli dell'utente e imposta l'autenticazione nel `SecurityContext`.

10.2 Spring Security

La classe `GestoreFilterChain` è un componente fondamentale per la gestione della sicurezza nell'applicazione. Essa definisce come le richieste HTTP devono essere filtrate e autorizzate, stabilendo regole chiare per l'accesso alle diverse sezioni dell'applicazione.

All'interno di questa classe, viene utilizzato un filtro di autenticazione personalizzato, che svolge un ruolo cruciale nel processo di autenticazione degli utenti. Questo filtro verifica se un utente è autenticato e gestisce la navigazione verso le pagine appropriate in base allo stato di autenticazione.

La configurazione della sicurezza prevede che alcune richieste, come quelle destinate a risorse come `/ws/**` e `/immagine/**`, siano accessibili a tutti, senza alcuna restrizione. Ciò consente una facile interazione con le funzionalità pubbliche dell'applicazione.

In particolare, sono state stabilite regole di accesso per le aree riservate agli amministratori. Solo gli utenti con i ruoli di `ADMIN` o `SUPER_ADMIN` possono accedere a risorse specifiche come `/pannelloAdmin/**`. Inoltre, le richieste per `/pannelloSuperAdmin/**` sono limitate esclusivamente agli utenti con il ruolo di `SUPER_ADMIN`.

Per le pagine che richiedono autenticazione, come quelle nella sezione `/dashboard/**`, viene effettuato un controllo dell'autenticazione. Se un utente non è autenticato, verrà reindirizzato alla pagina di login.

In sintesi, `GestoreFilterChain` implementa una gestione della sicurezza flessibile ed efficace, permettendo di garantire che solo gli utenti autorizzati

possano accedere alle sezioni protette dell'applicazione, mentre le risorse pubbliche rimangono facilmente accessibili a tutti. La classe `GestoreFilterChain` si occupa della configurazione della sicurezza dell'applicazione. Essa stabilisce quali ruoli possono accedere a determinati endpoint e gestisce il flusso di autenticazione degli utenti.

10.3 Endpoint accessibili in base al ruolo

- **Endpoint accessibili a tutti:**
 - `/ws/**`
 - `/ws`
 - `/immagine/**`
- **Endpoint accessibili solo agli amministratori:**
 - `/pannelloAdmin/**`
- **Endpoint accessibili solo ai super amministratori:**
 - `/pannelloSuperAdmin/**`
- **Endpoint accessibili solo agli utenti autenticati:**
 - `/dashboard/**`

11 Gestione degli Errori

La classe `Handler` gestisce le eccezioni che possono verificarsi durante l'esecuzione dell'applicazione. Utilizza l'annotazione `@ControllerAdvice` per catturare le eccezioni a livello globale e fornire risposte appropriate agli utenti.

11.1 Gestione delle eccezioni

11.1.1 `DatoNonValidoException`

Quando si verifica un'eccezione di tipo `DatoNonValidoException`, viene invocato il metodo `handleDatoNonValido`. Questo metodo restituisce una vista che mostra un messaggio di errore specifico per il caso in cui i dati forniti non siano validi. Il messaggio di errore viene passato come oggetto al modello per la visualizzazione.

11.1.2 `ResponseStatusException`

Per le eccezioni di tipo `ResponseStatusException`, viene utilizzato il metodo `handleResponseStatusException`. Questo metodo mappa il codice di stato della risposta a un messaggio di errore significativo. A seconda del codice di stato, vengono restituiti messaggi di errore predefiniti, come ad esempio:

- 409 `CONFLICT`: "Account già esistente"
- 404 `NOT_FOUND`: "Account non esistente"
- 403 `FORBIDDEN`: "Accesso non consentito"
- 401 `UNAUTHORIZED`: "Accesso non autorizzato"
- 400 `BAD_REQUEST`: "Richiesta non valida"
- `default`: "Errore generico"

In tutti i casi, viene restituita la stessa vista per visualizzare il messaggio di errore.

11.2 Eccezioni personalizzate

Sono definite due eccezioni personalizzate:

- `DatoNonValidoException`: utilizzata per indicare che i dati forniti non sono validi.
- `UtenteDisattivatoException`: utilizzata per segnalare che l'utente è disattivato.

Queste eccezioni estendono `RuntimeException` e consentono di gestire situazioni specifiche in modo più efficace all'interno dell'applicazione.

12 Testing con JUnit e Spring Boot

Il processo di testing automatico di un'applicazione basata su Spring Boot viene eseguito utilizzando il framework **JUnit 5** in combinazione con **Spring Boot Test** e **MockMvc**. Questa metodologia consente di testare il comportamento dei controller senza la necessità di avviare un server reale.

12.1 Strumenti Utilizzati

- **JUnit 5**: framework di testing per eseguire e validare test automatici.
- **Spring Boot Test**: fornisce strumenti per l'integrazione dei test con Spring Boot.
- **MockMvc**: consente di simulare richieste HTTP per testare i controller.
- **MockHttpSession**: permette di simulare sessioni utente per testare pagine protette.
- **MockMultipartFile**: utilizzato per testare l'upload di file tramite richieste HTTP multipart.

12.2 Testing dei Controller

I test sui controller vengono eseguiti per verificare il corretto funzionamento delle operazioni fornite dall'applicazione.

12.2.1 Test di Accesso alle Pagine

Per verificare che un utente autenticato possa accedere a una determinata area, si utilizza il seguente approccio:

Questo test verifica che:

- L'utente autenticato ottiene una risposta con codice di stato 200 (OK).
- La vista restituita sia quella attesa.

12.2.2 Test di Invio di Form

Per validare il funzionamento dell'invio di form, si utilizza una richiesta HTTP POST con parametri:

```

1  @Test
2  public void testInvioForm() throws Exception {
3      MockHttpSession session = new MockHttpSession();
4      session.setAttribute("email", "utente@mail.it");
5
6      mockMvc.perform(MockMvcRequestBuilders.post("/areaCliente
           /aggiungiProdotto")
7              .session(session)
8              .param("nome", "Prodotto_Test")
9              .param("descrizione", "Descrizione_del_
               prodotto")
10             .param("prezzo", "9.99")
11             .param("quantita", "1")
12             .contentType(MediaType.
                APPLICATION_FORM_URL_ENCODED))
13         .andExpect(status().is3xxRedirection())
14         .andExpect(MockMvcResultMatchers.redirectedUrl("/
           areaCliente/aggiungiProdotto"));
15 }

```

Listing 1: Esempio di test per l'invio di un form

Questo test verifica che:

- Il form viene inviato correttamente.
- L'applicazione risponde con un reindirizzamento alla pagina attesa.

12.2.3 Test con Upload di File

Per testare il caricamento di file tramite una richiesta multipart, si utilizza `MockMultipartFile`:

```

1  @Test
2  public void testUploadFile() throws Exception {
3      MockMultipartFile file = new MockMultipartFile(
4          "immagine",
5          "test.jpg",
6          MediaType.IMAGE_JPEG_VALUE,
7          "contenuto_dell'immagine".getBytes()
8      );
9
10     MockHttpSession session = new MockHttpSession();
11     session.setAttribute("email", "utente@mail.it");
12
13     mockMvc.perform(MockMvcRequestBuilders.multipart("/upload
           ")
14             .file(file)
15             .session(session))
16         .andExpect(status().is3xxRedirection())

```

```
.andExpect(MockMvcResultMatchers.redirectedUrl("/  
    areaCliente"));
```

Listing 2: Esempio di test con upload di file

Questo test verifica che:

- Il file venga correttamente incluso nella richiesta.
- La richiesta venga gestita con successo.
- L'utente venga reindirizzato alla pagina corretta dopo l'upload.

12.3 Report sulla Coverage

Il report sulla coverage fornisce una panoramica dell'efficacia dei test eseguiti sul codice sorgente, suddividendo la copertura in tre categorie principali: linee di codice, metodi e classi. Di seguito vengono commentate le percentuali ottenute:


- **Copertura delle linee di codice (91%):**
- **Copertura dei metodi (91%):**
- **Copertura delle classi (97%):**

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Branch, %	Line, %
all classes	97,9% (47/48)	91,8% (180/196)	68,2% (133/195)	91,3% (654/716)

Coverage Breakdown

Package 	Class, %	Method, %	Branch, %	Line, %
org.paolo.drumkit_	100% (1/1)	100% (2/2)		100% (2/2)
org.paolo.drumkit_.configuration	100% (3/3)	100% (10/10)		100% (32/32)
org.paolo.drumkit_.controller	100% (9/9)	97,1% (33/34)	82,4% (28/34)	93,9% (169/180)
org.paolo.drumkit_.dto.request	100% (1/1)	100% (6/6)		100% (6/6)
org.paolo.drumkit_.dto.response	100% (10/10)	97,6% (40/41)	50% (13/26)	92,2% (95/103)
org.paolo.drumkit_.exception	66,7% (2/3)	80% (4/5)	20% (1/5)	68,8% (11/16)
org.paolo.drumkit_.facade	100% (4/4)	93,5% (29/31)	70,3% (52/74)	93,8% (137/146)
org.paolo.drumkit_.mapper	100% (3/3)	100% (10/10)	58,3% (7/12)	100% (61/61)
org.paolo.drumkit_.model	100% (4/4)	88,9% (8/9)		73,9% (17/23)
org.paolo.drumkit_.security	100% (3/3)	100% (7/7)	90,9% (20/22)	100% (42/42)
org.paolo.drumkit_.service.impl	100% (7/7)	75,6% (31/41)	54,5% (12/22)	78,1% (82/105)

generated on 2025-02-03 16:00

Figura 7: Coverage Report